PG - ENDBERICHT

# PG 490 - Automatisiertes Management von Web-Service-Systemen

**Lehrstuhl IV**

**Teilnehmer:**
Ralf Bochon, Hülya Cetin, Oliver Dohndorf, Christian Fisseler, Falk Freudenau, Yassir Hammadi, Felix Holland, Christian Kaltepoth, Jens Kattwinkel, Daniar Mamytov, Fabian Prümm, Iryna Sokhnych

**Betreuer:**
Prof. Dr. Heiko Krumm
Andre Pohl

**Zeitraum:**
SS 2006 - WS 2006/2007

# Inhaltsverzeichnis

# 1. Einleitung

## 1.1. Motivation

Wer heutzutage vor der Aufgabe steht, eine Architektur für ein verteiltes Softwaresystem zu entwickeln, wird früher oder später mit dem Begriff *Serviceorientierte Architektur (SOA)* in Kontakt kommen. Die *Serviceorientierte Architektur* sieht lose gekoppelte und unabhängige Dienste vor, die in temporären Beziehungen zueinander stehen. Dadurch sollen das Softwaresystem und seine Komponenten dynamisch, flexibel, wiederverwendbar und anpassbar werden.

Als Plattform für eine Serviceorientierte Architektur benötigt man eine Middleware, welche die SOA-Ziele optimal umsetzt. Die am weitesten verbreitete Plattform mit den zur Zeit besten Zukunftsaussichten ist die *Webservice*-Plattform. *Webservice*s basieren auf plattformunabhängigen Technologien und sind damit besonders für heterogene Umgebungen geeignet.

Das Management solcher dynamischer Systeme ist eine große Herausforderung für die Informatik. Besonders aufgrund der Komplexität der Systeme besteht der Bedarf nach automatisiertem Management, um einen stabilen und effizienten Betrieb zu ermöglichen. Doch gerade die Dynamik solcher Systeme macht eine Automatisierung äußerst schwierig. Trotzdem gibt es heute bereits einige vielversprechende Ansätze für die technische Administration solcher Softwaresysteme.

## 1.2. Ziele und Ergebnisse

Die Projektgruppenbeschreibung aus dem Projektgruppenantrag nennt einige Minimalziele, die durch unsere Arbeit als Teilnehmer der Projektgruppe erreicht werden sollten. Zum einen wird die Entwicklung eines Laufzeit-Managementsystems genannt, das grundlegende Funktionen des Konfigurationsmanagements sowie automatisierte Managementfunktionen zum Fehlermanagement anbietet. Zum anderen wird der Entwurf einer geeigneten Policy-Hierarchie und Policy-Darstellung nebst Metamodell-Entwicklung und Integration in das Tool *Mobasec* genannt. Außerdem soll zu den ersten beiden Punkten ein Beispielsystem entwickelt und das Managementsystem daran erprobt werden.

Im Verlauf der Projektgruppenarbeit wurde wie in folgenden Kapiteln beschrieben zunächst eine Seminar- und eine Praktikumsphase veranstaltet, die in die Thematik einführen und einarbeiten sollten. Unter verschiedenen Alternativen wählten wir den DPWS-Stack als Web-Service-Plattform aus. Als Szenario für das automatisierte Management entwickelte sich nach einiger Zeit neben anderen Vorschlägen (Kapitel 5.3) die Idee des *Digital Home*, welches als Basis für die zur Verfügung stehenden Geräte und deren Management dienen sollte. Zusätzlich entschieden wir uns, zu einer Standort-Erkennung oder Identifikation der potentiellen Bewohner des *Digital Home*, Bluetooth-Technik (Bluetooth-Adapter oder PDAs oder Handys mit Bluetooth-Unterstützung) zu verwenden. Dazu stand uns auch ein Foxboard (Kapitel 4.3) zur Verfügung, das man mit Bluetooth ausstatten konnte. Neben einem Bluetooth-Handy war das Foxboard die einzige „echte“ Hardware, die für das eigentliche *Digital-Home*-Szenario zu Verfügung stand, die anderen Geräte sollten in ihrem Verhalten auf Rechnern simuliert werden. Dazu entwickelten wir Gerätekategorien und Spezifikationen von Geräten und schließlich die simulierten Geräte selbst, die im *Digital Home* zur Verfügung stehen sollten. Für die Entwicklung erschien uns die Nutzung des Tools Mobasec nicht zielgerichtet, sondern wir implementierten das System unabhängig hiervon.

Letztendlich wurde ein auf Policies basierendes Management-System entwickelt, das die zur Verfügung stehenden Geräte in Bezug auf verschiedene Faktoren verwalten kann. Zur Kom-

munikation zwischen den Geräten werden DPWS-Nachrichten verwendet und auf den Geräten laufen Services, in denen mehrere Properties registriert sind, die den Gerätezustand darstellen können. Ändert sich dieser oder treffen zum Beispiel Events vom DPWS-Stack ein, kann das Management aktiv werden und eine durch Policies bestimmte Reaktion veranlassen. So lassen sich beispielsweise durch Policies Maßnahmen zur Fehlerbehandlung erzielen. Bei der Nutzung verschiedener Geräte können außerdem Konfigurationen auf diese geladen werden, die die Properties der angebotenen Services nach den angegebenen Vorgaben ändern. Die verschiedenen Typen von Policies werden im Kapitel 10 beschrieben. Unter Nutzung von Bluetooth-Hardware wurden Funktionen implementiert, die eine drahtlose Gerätekommunikation ermöglichen. So soll eine Aufenthaltsbestimmung durchführbar sein, so dass man weiß, welche Person sich in welchem Raum befindet. Dies funktioniert noch etwas langsam und nicht ganz zuverlässig. Um das System und die Integration der einzelnen Komponenten demonstrieren zu können, wurde ein Szenario entwickelt, das einen möglichen Ablauf im *Digital Home* darstellen kann. Ein wichtiger Bestandteil dazu ist auch das Personal Device (Kapitel 8.2), das jeweils einem Benutzer im Haus zugeordnet ist und eine Interaktion – auch unter Managementanwendung – mit den unterschiedlichen Geräten und damit auch eine Steuerungsfunktion ermöglicht.

## 1.3. Kapitelübersicht

Dieser Projektgruppenendbericht besteht aus zwei Teilen. Im ersten Teil (Kapitel 2 bis 4) werden fremde Werkzeuge, Programmiersprachen und Software-Komponenten vorgestellt. Der zweite Teil dokumentiert die Arbeit, die von der Projektgruppe geleistet wurde.

Kapitel 2 stellt das Konzept der Webservices vor, welches Basis für das gesamte Projekt war. Die Realisierung dieses Konzeptes stellt das Web-Service-Framework DPWS (*Device Profile for Web Services*) dar, welches in Kapitel 3 beschrieben wird. Die eingesetzen Programmiersprachen und die zur Programmierung verwendeten Umgebungen werden in Kapitel 4 erläutert.

Der Anfang der Projektgruppenarbeit wird in Kapitel 5 ausgeführt. Er bestand aus einer Einarbeitungsphase mit anschließender Konkretisierung des Projektes. Die daraus entstandene Projektidee – das *Digital Home* – wird in Kapitel 6 geschildert. Um eine Übersicht über die Geräte, die im *Digital Home* zur Verfügung stehen sollten, zu bekommen, wurden Kategorien entworfen. Diese sind in Kapitel 7 aufgelistet. In Kapitel 8 werden alle implementierten Geräte erklärt. Für einige von diesen sollte Mobilität gewährleistet und Lokalisierung ermöglicht werden. Dazu wurde von der Projektgruppe *Bluetooth* (Funkvernetzung von Geräten gemäß [IEE02]) ausgewählt. Details über diese Funkschnittstelle und ihre Verwendung sind in Kapitel 9 zu finden. Die Umsetzung des automatisierten Managements, welches ein Kernthema für diese Projektgruppe war, wird in Kapitel 10 vorgestellt. Zu Test- und Präsentationszwecken wurde ein Szenario entworfen, welches in Kapitel 11 dokumentiert ist. Das *Fazit* als letztes Kapitel (12) reflektiert abschließend die Arbeit der Projektgruppe.

# 2. Webservices

Es gibt viele Definitionen des Begriffes Webservice, jedoch keine allgemein anerkannte. Das W3C definiert Webservices in [ABFG04] wie folgt:

"*A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.*"

Ein Webservice basiert demnach auf dem Prinzip einer Serviceorientierten Architektur (SOA). Er kann durch andere Softwaresysteme über eine URI gefunden werden und besitzt eine XML-basierte Schnittstellenbeschreibung, in der die Art der Kommunikation anderer Systeme mit dem Webservice beschrieben wird.

An anderer Stelle findet man eine weitere Definition des W3C:

"*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*"[HB06]

Die Definition besagt, dass Webservices in automatisierter Weise Daten über ein Netzwerk senden. Softwaresysteme können eigenständig miteinander kommunizieren und die Daten austauschen. Zum Beschreiben der Schnittstellen eines Webservice wird WSDL benutzt und SOAP als Nachrichtenformat zum Austausch der Daten.

Durch die Darstellung und Übertragung der Daten in XML ist ein Webservice sowohl unabhängig vom Betriebssystem, der Programmiersprache sowie vom Transport-Protokoll. Webservices haben noch einige andere Eigenschaften, die in diesen beiden Definitionen keine Erwähnung finden, jedoch auch von fundamentaler Bedeutung sind.

Sie sind lose gekoppelt [KL04], d. h. die einzelnen Systeme sind autonom und kommunizieren nur über Nachrichten miteinander. Dadurch ergibt sich der Vorteil, dass einzelne Systeme unabhängig voneinander weiterentwickelt werden können. Webservices sind zudem komponentenbasiert. Ihre Funktionalität kann also wie eine "Black-Box" in der Programmierung verwendet werden.

Als letzter Punkt sei die Sprach- und Plattformunabhängigkeit von Webservices angeführt.

Um Webservices einheitlich zu entwickeln und die Interoperabilität zwischen den Software-Herstellern zu erreichen, hat das W3C ein Konzept für Webservices vorgeschlagen. Dieses ist in Abbildung 1 [BRS03] zu sehen.

Das Konzept beinhaltet folgende Komponenten [DGH03]:

- **Dienstanbieter**
  Der Dienstanbieter stellt Webservices zur Verfügung. Dazu muss der Provider einen standardisierten Zugriff ermöglichen und eine Beschreibung von diesem Service vorbereiten.

- **Benutzer**
  Ein Benutzer, meist eine Anwendung, auch WS-Client genannt, nimmt Dienste eines

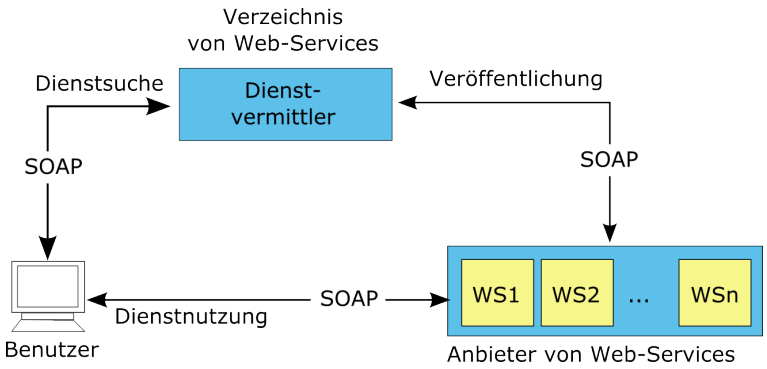


**Abbildung 1:** Konzept der Webservices

Dienstanbieters in Anspruch. Er lokalisiert einen passenden Dienst mittels des Dienstvermittlers und interagiert dann durch die vorgegebene Dienstschnittstelle mit dem Dienst.

- **Dienstvermittler**
  Der Dienstvermittler bietet dem Dienstgeber einen Service-Veröffentlichungsdienst, durch den die Dienste weltweit bekannt gemacht werden, und dem Dienstnehmer einen Service-Lokalisierungsdienst an.

## 2.1. Architektur

Um verteilte Anwendungen über das Internet zu verbinden, bieten Webservices eine eigene Architektur an [DGH03]. Die nachfolgende Abbildung 2 zeigt den Protokoll-Stack eines Webservice und die vier unterschiedlichen Schichten mit den jeweils eingesetzten Technologien.

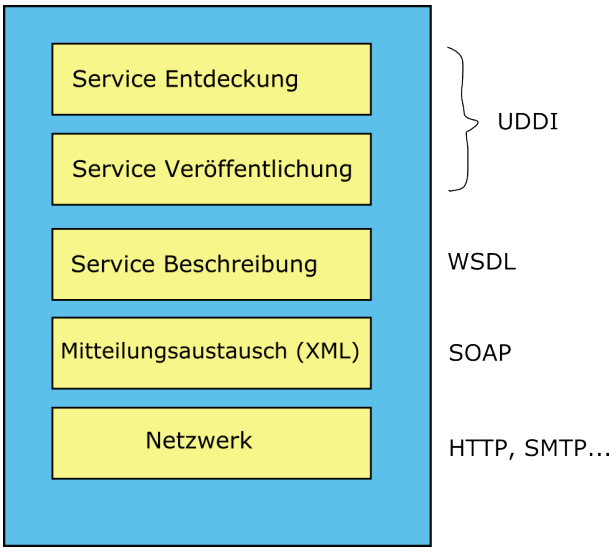


**Abbildung 2:** Der Webservice Stack

- Transportschicht
  Diese Schicht ist für den Transport der Nachrichten über ein Netzwerk verantwortlich. Da

der darüberliegende Webservice von dieser Schicht abstrahiert, können unterschiedliche Protokolle, wie z.B. HTTP, FTP, SMTP usw., zum Einsatz kommen.

- XML-Mitteilungsaustausch
  Diese Schicht verpackt Nachrichten in ein von allen Teilnehmern verstandenes XML-Format. Die Nachrichten werden z. B. mit SOAP, einem Framework zum Austausch von XML-Nachrichten, verschickt. Eine ausführliche Abhandlung von SOAP folgt in Kapitel 2.2.

- Service-Beschreibung
  Diese Schicht beschreibt mit Hilfe einer Beschreibungssprache, WSDL, den Webservice und dessen Schnittstelle (Interface). Somit kann von der darunterliegenden Implementierung abstrahiert werden.

- Service-Entdeckung und -Veröffentlichung
  Service-Anbieter können mit Hilfe von UDDI ihren Service beschreiben und in einem öffentlichen Verzeichnis publizieren. Der Anwender benutzt dieses Verzeichnis, um vorhandene Webservices zu suchen und danach in seine Verarbeitung und Systeme einzubinden.

Im weiteren Verlauf dieses Dokuments werden der Mitteilungsaustausch (SOAP) und die Service-Beschreibung (WSDL) genauer betrachtet.

## 2.2. SOAP - Simple object access protocol

### 2.2.1. Überblick

SOAP ist ein Kommunikationsprotokoll für den Austausch von strukturierten und typisierten Daten mittels XML-Dokumenten für verteilte Anwendungen.

Der Standard definiert allgemeine Regeln zum Austausch von Nachrichten zwischen Applikationen und eignet sich besonders für RPC-Nachrichten. Die SOAP-Spezifikation beinhaltet im wesentlichen folgende 3 Teile [BRS03]:

1. Das SOAP-**envelope**-Tag:
   Es beschreibt die übergeordnete Struktur: Welche Daten eine Nachricht enthält, an wen sie adressiert ist und ob diese behandelt werden muss oder ob sie nur optional ist.

2. Die SOAP-**encoding**-Regeln:
   Sie beschreiben den Serialisierungsmechanismus zum Austausch von benutzerdefinierten Datentypen.

3. Das SOAP-**binding-framework**:
   Es spezifiziert den Austausch der SOAP-envelopes über ein zugrunde liegendes Transportprotokoll.

### 2.2.2. Aufbau einer SOAP-Nachricht

Abbildung 3 zeigt den Aufbau einer SOAP Nachricht [Mit03]. Jede SOAP-Nachricht beinhaltet ein XML-Dokument, in dem ein *SOAP-Envelope*, ein *SOAP-Header* und ein *SOAP-Body* enthalten sein können.

Quellcode 1 zeigt ein entsprechendes XML-Dokument.

```
<soap:Envelope soap:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header>
<!--Der Header ist optional -->
</soap:Header>
<soap:Body>
<!-- Serialisierte Objektdaten -->
</soap:Body>
</soap:Envelope>
```

**Quellcode 1:** XML-Struktur einer SOAP-Nachricht

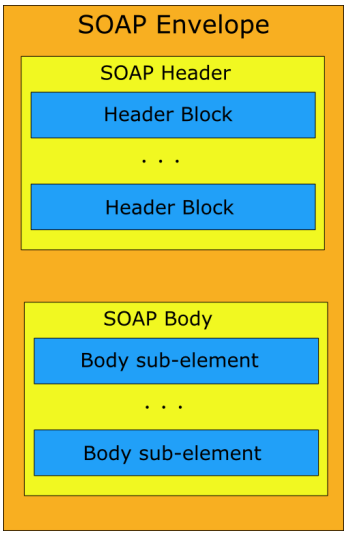


**Abbildung 3:** Aufbau einer SOAP-Nachricht

Im Folgenden werden die einzelnen Elemente einer SOAP-Nachricht näher erläutert.

### 2.2.3. SOAP-Envelope

Ein Envelope, der das Wurzelelement ist und die ganze Nachricht somit umschließt, muss in jeder SOAP-Nachricht vorhanden sein.

Die kommunizierenden SOAP-Applikationen müssen, um sich zu verstehen, das gleiche "Vokabular" benutzen. Das heißt, sie müssen die gleichen Namensräume verwenden. Daher wird ein Namensraum für alle Elemente und Attribute festgelegt. SOAP definiert zwei Namensräume [STK02], die abgerufen werden können:

- Namensraum für SOAP-Encoding:
  Das *encodingStyle*-Attribut gibt an, wie Objekte zu XML serialisiert worden sind, also die Umwandlung einer SOAP-Nachricht in eine Folge einzelner Bytes zur Übermittlung. Es ist zwingend nötig, diesen Namensraum anzugeben.

- Namensraum für Envelope:
  Das soap-Attribut definiert den SOAP-Namensraum.

Beide Attribute können in Quellcode 1 nachvollzogen werden.

### 2.2.4. SOAP-Header

Der Header einer SOAP-Nachricht ist optional und enthält konkrete Informationen wie z. B. Nachrichtennummern oder die Art der Nachricht (Antwort, Anfrage).

Ein Header durchläuft auf dem Weg vom Client zum Service eine Menge von Knoten, die als Nachrichtenpfad definiert wird [DGH03]. Der Header kann von den SOAP-Knoten auf diesem Pfad ausgelesen, verändert, gelöscht oder hinzugefügt werden. Er wird von den Knoten zum Austausch von Kontrollinformationen genutzt. Nachdem ein Header von einem Knoten bearbeitet wurde, muss dieser aus der Nachricht entfernt werden. Allerdings darf ein Knoten selbstständig entscheiden, ob der Header wieder hinzugefügt, geändert oder gar neu erstellt wird.

Jeder Knoten nimmt innerhalb der Übertragungskette eine bestimmte Rolle ein. Die verschiedenen Rollen werden in Tabelle 1 [Mit03] dargestellt.

| **Abkürzung** | **Vollständiger Name** | **Beschreibung** |
|---|---|---|
| next | http://www.w3.org/2003/05/soap-envelope/role/next | Jeder Zwischenknoten und der endgültige Empfänger müssen in dieser Rolle handeln. |
| none | http://www.w3.org/2003/05/soap-envelope/role/none | Kein Knoten darf einen Header mit dieser Rolle behandeln. |
| ultimateReceiver | http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver | Nur der endgültige Empfänger darf in dieser Rolle handeln. |

**Tabelle 1:** SOAP-Rollennamen

Ein Header kann ebenfalls ein solches Rollenattribut besitzen. Sind die Rolle im Header und die des Knotens identisch, muss der Knoten die Nachricht bearbeiten. An dieser Stelle sei zudem noch das optionale Attribut *mustUnderstand* erwähnt. Ist dieses auf true gesetzt, muss ein Knoten mit der entsprechenden Rolle den Header **vollständig** interpretieren können. Wenn dies nicht der Fall ist, wird der Header von dem entsprechenden Knoten gelöscht. Wie die Bearbeitung einer SOAP-Nachricht aussieht, liegt außerhalb der SOAP-Spezifikation und wird somit von der Anwendung festgelegt. Wenn ein Header nicht bearbeitet wird, muss er entfernt werden. Das *relay*-Attribut kann dieses Verhalten außer Kraft setzen. (Siehe Tabelle 2 [Mit03])

Header enthalten Elementblöcke, die als Aufgabenbeschreibung der Bearbeitung der einzelnen Knoten dienen. Blöcke werden durch URIs identifiziert und können so den einzelnen Knoten zugeordnet werden. Header können so durch beliebig viele Blöcke erweitert werden,

| Rolle | | Headerblock | |
|---|---|---|---|
| Kurzname | Eingenommen | Verstanden & bearbeitet | weitergeleitet |
| next | Ja | Ja | Nein, außer wieder eingefügt |
| | | Nein | Nein, außer relay=true |
| user-defined | Ja | Ja | Nein, außer wiedereingefügt |
| | | Nein | Nein, außer relay=true |
| | Nein | N/A | Ja |
| ultimateReceiver | Ja | Ja | N/A |
| | | Nein | N/A |
| none | Nein | N/A | Ja |

**Tabelle 2:** relay-Attribut

die eigentlich außerhalb der SOAP-Spezifikation liegen. Beispielhaft können an dieser Stelle WS-Security [ADLH+02] oder WS-Transaction angebracht werden.

### 2.2.5. SOAP-Body

Der Body muss in jeder SOAP-Nachricht enthalten sein. Er enthält die Nutzdaten der Nachricht. Aufbau und Inhalt ist vom Nachrichtentyp abhängig.

- Methodenaufruf
  Methodenname und Eingabeparameter werden per XML-Dokument übermittelt
- Antwort
  Rückgabedaten werden per XML-Dokument übermittelt
- Fehlermeldung
  Fehlerinformationen werden übermittelt

Grundsätzlich schreibt die SOAP-Spezifikation keine Struktur für den Body vor. Einzig die Fehlernachrichten sind genau spezifiziert [STK02]. Abbildung 4 beschreibt den Aufbau einer solchen Fehlermeldung.

#### Code

Das Code-Element enthält einen Errorcode, welcher in einem weiteren Subelement Value enthalten sein muss. Das Code-Element stellt die maschinenlesbare Form des Fehlers dar.

#### Reason

Das Reason-Element enthält einen Text und ist die menschenlesbare Form des Fehlers.

#### Detail

Enthält anwendungsspezifische Informationen und ist optional.

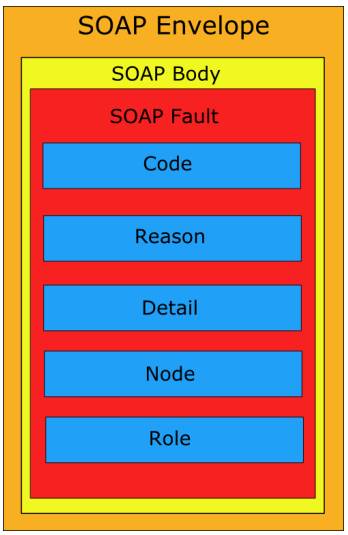


**Abbildung 4:** SOAP-Nachricht mit Fehlermeldungen

#### Node

Identifiziert den Knoten, der die Fehlermeldung generiert. Dieses Element ist optional, bei Fehlen wird automatisch der endgültige Empfänger als Verursacher angenommen.

#### Role

Gibt die Rolle an, die der Knoten innehat, wenn er die Fehlermeldung generiert. Dieses Element ist ebenfalls optional.

### 2.2.6. Kommunikation mit SOAP

Der SOAP-Standard legt nicht fest, mit welchem Protokoll Nachrichten versendet werden sollen, gibt aber als standardisierte Möglichkeit die Einbettung von SOAP in HTTP an.

Möglich wäre aber auch eine Übermittlung der Nachrichten mit Hilfe eines Mailprotokolls (SMTP) oder eines Datenübertragungsprotokolls (FTP). Diese Möglichkeiten sind aber bisher nicht standardisiert.

Die HTTP-Einbettung von SOAP nutzt die vielfältigen Möglichkeiten von HTTP, ohne dabei die Semantik zu ändern. Vielmehr wird der bestehende Standard benutzt, um SOAP-Nachrichten als Nutzlast zu transportieren.

SOAP definiert zwei Modi von Kommunikationsformen [BRS03]. Zur Unterscheidung werden folgende Begriffe eingeführt:

- style="document"
  In diesem Modus wird im Body ein komplettes Dokument übermittelt. Die Kommunikation verläuft meist asynchron.

- style="rpc"
  Beschreibt den *RPC-Modus*, der den entfernten Methodenaufruf beschreibt. In diesem Fall funktioniert SOAP nach dem Request/Response-Prinzip.

Abbildung 5 [BRS03] illustriert die Übermittlung von SOAP-Nachrichten mittels HTTP im "RPC-Modus".

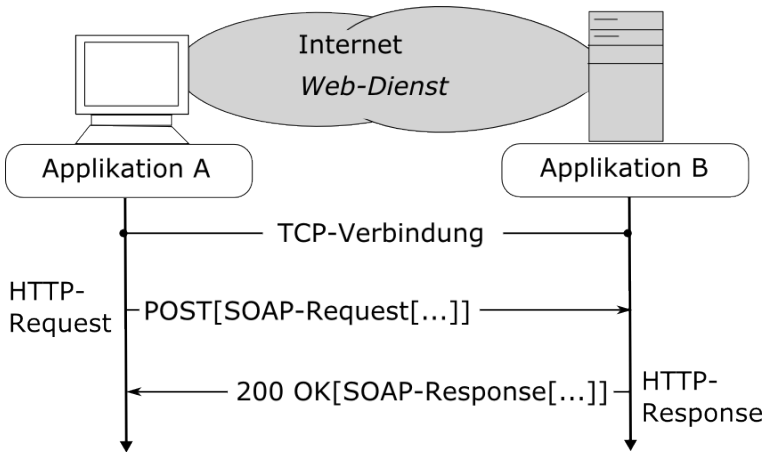


**Abbildung 5:** SOAP-Nachrichtenübermittlung mit Hilfe von HTTP

Da HTTP das verbindungsorientierte Transportprotokoll TCP nutzt, muss zunächst eine Verbindung zwischen den beiden Applikationen aufgebaut werden. Applikation A übermittelt einen SOAP-Request in einem HTTP-Request *POST* an die Applikation B. Diese antwortet mit einem SOAP-Response in einem HTTP-Response *200 OK*, falls es sich um eine "positive" Antwort handelt.

Quellcode 2 gibt ein Beispiel einer solchen Nachricht [KKSS03]. Es wird eine Anfrage an einen Temperaturdienst gestellt.

`Adresse des Dienstes: http://wetter.fmi.uni-passau.de:8004/service/Temperaturdienst`

```
POST /service/Temperaturdienst HTTP/1.1
HOST: wetter.fmi.uni-passau.de:8004
Content-Type: text/xml; charset=utf-8
Content-Length: nnnn <-- Gibt die Laenge der POST-Anforderung an -->
SOAPAction: temperaturAbfrage

<soap:Envelope  soap:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/"
                xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soap:Body>
<n:temperaturAbfrage xmlns:n="http://wetter.fmi.uni-passau.de/
    Temperaturdienst">
<n:datum xsi:type="xsd:dateTime">2002-02-22T12:00:00Z</n:datum>
</n:temperaturAbfrage>
</soap:Body>
</soap:Envelope>
```

**Quellcode 2:** SOAP-Nachricht im HTTP-Request POST

Quellcode 2 beschreibt den Aufbau eines SOAP-Dokuments inklusive der Einbettung in eine HTTP-Nachricht.

Das Envelope-Element spezifiziert die Namensräume und die verwendete Serialisierung für das Dokument. Das Body-Element enthält ein Unterelement, das genauso heißen muss wie die Methode, die aufgerufen werden soll. Innerhalb dieses Elements werden alle Parameter in derselben Reihenfolge und mit demselben Namen aufgezählt, wie in der Methodendeklaration angegeben.

Das Beispiel stellt also eine Anfrage an einen Temperaturdienst dar, die am 22.02.2002 um 12:00 Uhr gemessene Temperatur zu liefern.

Die Antwort des Dienstes ist in Quellcode 3 dargestellt.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: nnnn

<soap:Envelope  soap:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/"
                xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soap:Body>
<n:temperaturAbfrageResponse xmlns:n="http://wetter.fmi.uni-passau.de/
    Temperaturdienst">
<Result xsi:type="ns2:TemperaturdienstAntwort"
xmlns:ns2="http://wetter.fmi.uni-passau.de/Temperaturdienst.xsd">
<temperatur xsi:type="xsd:double">4.4</temperatur>
<einheit xsi:type="xsd:string">Celsius</einheit>
<messdatum xsi:type="xsd:dateTime">2002-02-22T12:00:00Z</messdatum>
</Result>
</n:temperaturAbfrageResponse>
</soap:Body>
</soap:Envelope>
```

**Quellcode 3:** SOAP-Nachricht im HTTP-Response 200 OK

Wie der HTTP-Header anzeigt (HTTP/1.1 200 OK), wurde die Anforderung erfolgreich bearbeitet. Die Antwort des Dienstes findet man innerhalb des Body-Elementes. Man erkennt, dass am 22.02.2002 um 13:00 Uhr eine Temperatur von 4,4 Grad Celsius gemessen wurde.

## 2.3. WSDL - Web Service Description Language

### 2.3.1. Beschreibung von Webservices mit WSDL

Einleitend in diesem Kapitel soll zunächst die Notwendigkeit von WSDL erläutert werden. Ein Webservice kann als ein Objekt mit mehreren Methoden im Sinne einer objektorientierten

Programmiersprache dargestellt werden. Jede Methode erbringt dabei eine bestimmte Funktion eines Webservice, und um auf diese Funktionen zuzugreifen, muss die ihr entsprechende Methode aufgerufen werden. Für diesen entfernten Methodenaufruf ist die Beschreibung der Schnittstelle zum Webservice nötig, welche in WSDL realisiert wird.

Doch welche Informationen enthält ein solches WSDL-Dokument? Vereinfacht gesagt enthält ein WSDL-Dokument mehrere Elemente, in denen die Fragen Was?, Wie? und Wo? beantwortet werden [KL04].

- Was?
  Um welchen Webservice handelt es sich, d. h. welche Funktionen können mit ihm erbracht werden. Dies beschreibt das Element *portType*.

- Wie?
  Wie in Kapitel 2.2 beschrieben, kommen für den Zugriff auf Webservices mehrere Protokolle (HTTP, SMTP, FTP...) in Frage. An dieser Stelle wird geklärt, wie der Zugriff auf eine Funktion des Webservice mit Hilfe eines bestimmten Protokolls erfolgen kann. Diese Informationen werden im Element *binding* angegeben. Ein WSDL-Dokument kann mehrere *binding*-Elemente enthalten

- Wo?
  Um eine Funktion eines Webservice nutzen zu können, muss man ihre Lokation als Adresse kennen. Die URL wird durch das Element *service* angegeben. Werden von einem Webservice mehrere Funktionen angeboten, enthält das WSDL-Dokument entsprechend mehrere *service*-Elemente.

### 2.3.2. Aufbau eines WSDL-1.1-Dokumentes

Abbildung 6 zeigt den Aufbau eines WSDL-Dokumentes. Es ist in eine abstrakte und eine konkrete Ebene aufgeteilt. Während in der abstrakten Ebene die Frage "Was?"geklärt wird, kümmert sich die konkrete Ebene um die Fragen "Wie?"und "Wo?".

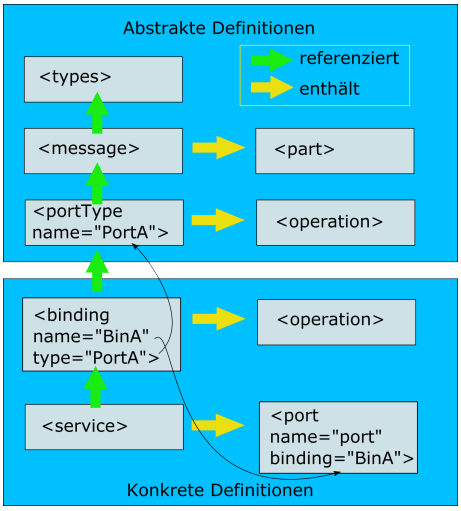


**Abbildung 6:** WSDL Spezifikation

Dargestellt ist ein WSDL-1.1-Dokument [CCMW01]. Mitlerweile hat das W3C bereits Version 2.0 [BL06] veröffentlicht, da Version 1.1 aber immer noch der Quasi-Standard ist, wird dieser genauer erläutert.

Um Abbildung 6 zu erklären, werden im nachfolgenden Kapitel die einzelnen Elemente vorgestellt.

#### 2.3.2.1. <types>

Das *types*-Element liefert Datentyp-Definitionen zum Spezifizieren der Datentypen innerhalb einer Nachricht. Zur Beschreibung der Datentypen wird auf XSD zurückgegriffen. Die Verwendung des *types*-Elementes ist optional und richtet sich nach der Komplexität der verwendeten Datentypen. Einfache Datentypen (Zahlenwert) müssen nicht extra definiert werden, sie ergeben sich implizit durch den angegebenen XSD-Namensraum, wohingegen komplexe Datentypen (Adresse, die sich aus Namen und Anschrift zusammensetzt) einer näheren Spezifikation bedürfen.

Quellcode 4 zeigt eine solche Spezifikation.

```
<types>
     <schema xmlns="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://wetter.fmi.uni-passau.de/
                 Temperaturdienst.xsd">
        <complexType name="TemperaturdienstAntwort">
         <sequence>
          <element name="temperatur" type="double"/>
          <element name="einheit" type="string"/>
          <element name="messdatum" type="dateTime"/>
         </sequence>
        </complexType>
     </schema>
</types>
```

**Quellcode 4:** Spezifikation eines komplexen Datentyps

Für den Temperatur-Web-Service wird bei einer Anfrage ein komplexer Datentyp verwendet. Dieser muss also deklariert werden, damit Sender und Empfänger ein identisches Verständnis dieses Datentyps haben. In diesem Beispiel setzt sich der komplexe Datentyp **TemperaturdienstAntwort** aus 3 einfachen Datentypen (double für die Temperatur, string für die Einheit und dateTime für das Messdatum) zusammen.

#### 2.3.2.2. <message>

Eine Nachricht ist eine Basiskommunikationseinheit zwischen einem Webservice und einer Anwender-Applikation. Sie beinhaltet die gesamten Daten innerhalb einer Nachricht zwischen beiden Teilnehmern. Das *message*-Element listet alle Parameter auf, die innerhalb dieser Nachricht verwendet werden. Es besitzt einen eindeutigen Namen innerhalb des Dokuments.

Logische *part*-Elemente sind die Bausteine eines Nachrichten-Elementes, die innerhalb eines solchen einen eindeutigen Namen haben müssen. Die *part*-Elemente, dessen Reihenfolge festgelegt ist und unbedingt eingehalten werden muss, bestimmen die übertragenden Parameter und legen den Datentyp fest.

```
<message name="temperaturAbfrageSoapIn">
        <part name="datum" type="xsd:dateTime"/>
</message>
<message name="temperaturAbfrageSoapOut">
        <part name="Result" type="ns1:TemperaturdienstAntwort"/>
</message>
```

**Quellcode 5:** Das <message>-Element

In der in Quellcode 5 dargestellten Nachricht namens *temeraturAbfrageSoapIn* wird in dem part-Element der Parameter *datum* mit dem zugehörigen Datenttyp dateTime festgelegt. In der Nachricht *temperaturAbfrageSoapOut* kommt der in Abschnitt 2.3.2.1 definierte Datentyp zum Einsatz.

#### 2.3.2.3. <porttype>

Ein *porttype*-Element gibt an, welche Funktionen über einen Port erbracht werden. Es besteht aus einer Menge von Operationen, die wiederum die zuvor definierten Nachrichtenelemente gruppieren. Angaben zum Transport-Protokoll werden nicht gemacht. Auch dieses Element besitzt einen eindeutigen Namen.

Das Kindelement ist *operation*. Innerhalb dieses Elementes werden die referenzierten Nachrichten genauer spezifiziert. Es wird angegeben, ob es sich um <input>-, <output>- oder <fault>-Nachrichten handelt. Dadurch werden implizit die verwendeten Transport-Primitiven festgelegt. WSDL unterstützt davon vier Stück, die festlegen, wie Webservices und Anwender-Applikationen miteinander kommunizieren [Apf04].

**One-way:** Der Service empfängt eine Nachricht vom Anwender und sendet keine Antwort zurück.
`<wsdl:input name="foo" message="bar"/>`
Anwendungsszenario: Hochladen einer Datei.

**Request-Response:** Der Service empfängt eine Nachricht und sendet eine Antwort zum Anwender.
`<wsdl:input name="foo" message="bar"/>`
`<wsdl:output name="foo" message="bar"/>`
`<wsdl:fault name="foo" message="bar"/>`
Anwendungsszenario: Anfrage an den Temperaturdienst und aktuelle Temperatur als Antwort.

**Solicit-Response:** Der Service sendet eine Nachricht und bekommt eine Antwort vom Anwender.
`<wsdl:output name="foo" message="bar"/>`
`<wsdl:input name="foo" message="bar"/>`
`<wsdl:fault name="foo" message="bar"/>`
Anwendungsszenario: Benachrichtigung, dass persönliche Daten überprüft werden sollen, als Antwort Bestätigung der Einsicht und evtl. Änderungen.

**Notification:** Der Service sendet eine Nachricht zum Anwender
`<wsdl:output name="foo" message="bar"/>`
Anwendungsszenario: Service sendet eine Nachricht, dass sich das Nutzungsangebot eines Dienstes erweitert hat.

```
<portType name="TemperaturdienstSoap">
  <operation name="temperaturAbfrage" parameterOrder="datum">
     <input name="temperaturAbfrageSoapIn" message="
         tns:temperaturAbfrageSoapIn"/>
     <output name="temperaturAbfrageSoapOut" message="
         tns:temperaturAbfrageSoapOut"/>
  </operation>
</portType>
```

**Quellcode 6:** Das <portType>-Element

Das Beispiel in Quellcode 6 illustriert eine Operation vom Typ Request-Response, da zuerst ein input- und dann ein output-Element definiert wird. Auf ein fault-Element wird verzichtet. Das Attribut *parameterOrder* legt die Eingabeparameter von temperaturAbfrage chronologisch fest. Da es aber nur einen Eingabeparameter gibt, ist die Reihenfolge in diesem Beispiel natürlich unerheblich.

#### 2.3.2.4. <binding>

An dieser Stelle wird dem abstrakten Porttyp ein konkretes Transportprotokoll zugeordnet. Prinzipiell kann jedes Transportprotokoll verwendet werden, welches XML-Daten transportieren kann (SOAP, HTTP, SMTP ...). Das *binding*-Element besitzt 2 Attribute. Das name-Attribut identifiziert es mit einem eindeutigen Namen, das type-Attribut referenziert den korrespondierenden Porttyp des WSDL-Dokumentes. Jedes *binding* bezieht sich also immer auf einen *portType* und konkretisiert ihn. Damit ist auch klar, dass das *binding*-Element das erste auf der konkreten Ebene ist.

Es ist möglich, mehrere Anbindungen für einen Porttyp zu spezifizieren, um so flexibel bei der Anwendung der unterschiedlichen Transportprotokolle zu sein.

```
<binding name="TemperaturdienstSoap" type="tns:TemperaturdienstSoap">
 <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/
     http"/>

   <operation name="temperaturAbfrage">
     <soap:operation soapAction="temperaturAbfrage" style="rpc"/>
        <input name="temperaturAbfrageSoapIn">
        ...
        </input>
        <output name="temperaturAbfrageSoapOut">
        ...
        </output>
   </operation>
```

```
</binding>
```

**Quellcode 7:** Das <binding>-Element

Quellcode 7 zeigt den Aufbau des *binding*-Elements. Das type-Attribut referenziert auf den PortTyp *TemperaturdienstSoap*. Da auf Applikationsebene mit dem SOAP-Protokoll kommuniziert werden soll, wird das Element *<soap:binding>* erwartet. Die Verwendung des HTTP-Protokolls auf Transportebene wird über das Attribut *transport* festgelegt.

#### 2.3.2.5. <service>

Hierbei handelt es sich um eine Kollektion von Ports und somit eine Zusammenstellung der angebotenen Webservices innerhalb eines WSDL-Dokumentes. Das *port*-Element beschreibt, wo ein bestimmter PortType über das binding zu erreichen ist. Es gibt also eine konkrete Netzwerkadresse an.

Quellcode 8 zeigt den Aufbau des Service-Elementes.

```
<service name="Temperaturdienst">
  <port name="TemperaturdienstSoap" binding="tns:TemperaturdienstSoap">
        <soap:address location="http://wetter.fmi.uni-passau.de:8004/
            service/Temperaturdienst"/>
  </port>
</service>
```

**Quellcode 8:** Das <service>-Element

Der Service *Temperaturdienst* enthält einen Port namens *TemperaturdienstSoap*. Ihm wird eine Anbindung zugewiesen. Innerhalb des Port-Elementes wird dem Port eine konkrete Netzwerkadresse zugeordnet, über die der Webservice zu erreichen ist.

# 3. DPWS

Als Webservice-Plattform wurden für die Projektgruppe verschiedene (Apache Axis, Websphere und DPWS in einer sehr kompakten Implementierung des Lehrstuhls IV der Universität Dortmund) in Betracht gezogen. Aufgrund der Kompaktheit und der Lauffähigkeit auf eingebetteten Systemen (JavaME-Kompatibilität) wurde der DPWS-Stack gewählt. Der Stack ist im Rahmen des europäischen Forschungsprojektes SIRENA (Service Infrastructure for Real-Time Embedded Networked Applications) [Sir] in einer Kooperation der Universität Dortmund mit dem Unternehmen *Materna Information and Communications* entstanden. Während der Arbeit der Projektgruppe wurde dieser DPWS-Stack noch weiter entwickelt.

*Devices Profile for Web Services* (DPWS) [Mic05] benutzt Teile verschiedener Webservice-Spezifikationen und erlaubt Geräten einen Standard-Mechanismus zu nutzen, um mit anderen DPWS-Geräten und Computern zu kommunizieren. DPWS konzentriert sich auf IP-fähige Geräte. Um ein Basisniveau von Interoperabilität zwischen Geräten und Webservices zu ermöglichen, wurden dem *Devices Profile* Webservice-Spezifikationen zugrunde gelegt, die folgende Bereiche betreffen:

- Versand von Nachrichten zu und von einem Webservice (messaging),
- dynamische Entdeckung von Webservices (discovery),
- Beschreibung von Webservices (description),
- Abonnieren und Empfang von Ereignissen von Webservices (eventing).

## 3.1. Überblick und Architektur

Webservices sind Software-Anwendungen, die auf XML und SOAP basieren. Ein Webservice unterstützt eine direkte Interaktion mit anderen Software-Agenten unter Verwendung von XML-basierten Nachrichten durch den Austausch über internetbasierte Protokolle. Obwohl Webservices unabhängig voneinander bleiben, können sie zu einer zusammenarbeitenden Gruppe, die eine bestimmte Aufgabe erfüllt, verbunden werden. Das *Devices Profile for Web Services* definiert ein Profil in einer Menge von WS-Spezifikationen, welches dem Ziel dient, die möglichen Verbindung und Nutzung von verschiedenen Geräten in Netzwerken zu vereinfachen. Es definiert einen UPnP-Nachfolgestandard.

Die Abbildung 7 gibt einen groben Überblick über das DPWS-Kommunikationsmodell.

### Endpoint

Das ist ein Endpunkt im Netzwerk, der Nachrichten über das Netzwerk empfängt.

### Port Type

Ein Porttyp ist ein Teil der WSDL, der beschreibt, welche Actions ein Gerät oder Service implementieren müssen. PortType entspricht der Schnittstellenbeschreibung.

### Scope

Scopes können benutzt werden, um Geräte in logische Bereiche zu gruppieren.

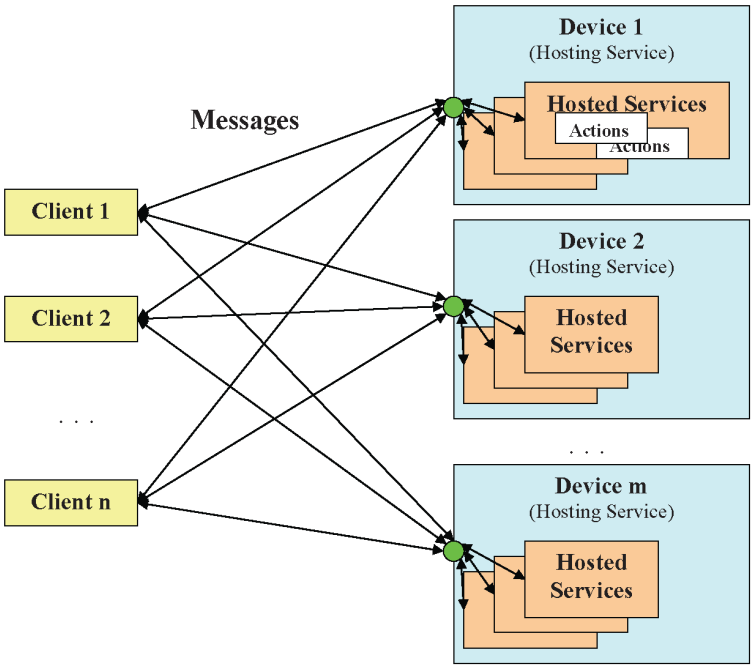


**Abbildung 7:** DPWS-Architektur

### Endpoint reference

Eine Endpunkt-Referenz ist eine Referenz auf eine spezifische Instanz eines Services. Ihre Adresse wird aus der HTTP-Adresse, Service-Portnummer und einem Endpunkt-Pfad zusammengesetzt.

### Device/Hosting Service

Ein Device (Gerät) hat einen oder mehrere Services. Das Gerät stellt Services mit allen üblichen Funktionen wie Messaging oder Discovery zur Verfügung. Es wird durch seine Porttypen und die auf ihm laufenden Dienste klassifiziert.

### Service/Hosted Service

Ein Service kommuniziert mit anderen DPWS-Services und -Clients im Netzwerk. Einzelne Funktionen eines Services können zum Action-Aufruf benutzt werden. Services empfangen/-senden Nachrichten und werden durch ihre Porttypen klassifiziert.

### Client

Ein Client ist ein Endpunkt, der nach Services sucht und als „Nutzer“ oder „Kunde“ interagiert.

### Action

Actions sind in Services enthalten und werden für die Interaktion mit Services von anderen DPWS-Instanzen benutzt. Sie können Input-, Output- und Fault-Parameter enthalten.

Clients interagieren mit Geräten und ihren Services mit Hilfe von SOAP-Nachrichten. Es gibt folgende Nachrichtentypen:

- **Hello** - diese Nachricht wird von einem Gerät gesendet, wenn es ins Netzwerk kommt.
- **Bye** - diese Nachricht wird von einem Gerät gesendet, wenn es das Netzwerk verlässt.
- **Probe** - diese Nachricht wird von einem Client/Service gesendet, wenn ein Device gesucht wird.
- **ProbeMatch** - diese Nachricht ist eine Antwort eines Devices auf die Probe-Nachricht.
- **Resolve** - diese Nachricht wird von einem Client/Service gesendet, wenn eine Device-UUID in eine Transportadresse umgewandelt werden soll.
- **ResolveMatch** - diese Nachricht ist eine Antwort eines Devices auf die Resolve-Nachricht.

## 3.2. Der DPWS-Protokoll-Stack

Der DPWS-Protokoll-Stack, der in Abbildung 8 dargestellt ist, integriert alle oben genannten Basisstandards. Mit DPWS basiert der ganze Datentransfer auf SOAP und WS-Addressing. DPWS fügt auch Webservice-Protokolle für Discovery und Eventing hinzu.

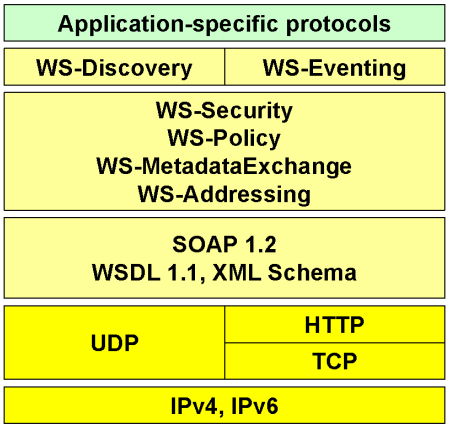


**Abbildung 8:** DPWS-Stack

Die grundlegenden Webservice-Standards [L.F04] sind folgende:

- **WSDL** für die abstrakte Beschreibung der Service-Schnittstellen,
- **XML-Schema** für die Definition der Datenformate, welche für die Konstruktion von Nachrichten verwendet werden, die den Services gesendet und von den Services empfangen werden
- **SOAP**, das Protokoll, das den Aufbau der verwendeten Nachrichten definiert
- **WS-Adressing** ist eine Netzwerk-Adressierung, die auf IPv4 oder IPv6 basiert, und nah mit SOAP verbunden ist

- **WS-Policy**, ein erweiterbares Framework, mit dem sich Webservices mit Vorgaben, sogenannten *Policy Assertions*, ausstatten lassen
- **WS-MetadataExchange**, Metadata in WSDL-Dokumenten, XML-Schemata und WS-Policies werden verwendet, um einen Webservice anzusprechen
- **WS-Security** enthält wichtige Mechanismen für die Sicherheit, wie Integrität, Vertraulichkeit und Authentisierung, um *Quality of Protection* zu gewährleisten
- **WS-Discovery** definiert ein Multicast-Protokoll zur Suche und Identifikation von Geräten
- **WS-Eventing** definiert ein Protokoll, in dem Nachrichten zum Abonnieren einer Eventquelle, zum Beenden einer Subscription und zum Senden von Events benutzt werden

## 3.3. Anwendung von DPWS für High-Level-Gerätekommunikation

### 3.3.1. WS-Addressing

WS-Addressing [F.J05, CKK+05] liefert Standardmechanismen für die Identifizierung und den Austausch von Webservice-Nachrichten zwischen verschiedenen Endpunkten in einem Netzwerk. Die Spezifikation definiert, wie Webservice-Adressen unabhängig von der verwendeten Transportschicht in SOAP-Nachrichten ausgedrückt werden. Neben der Identifikation von Services spezifiziert WS-Addressing auch, wie Nachrichten mithilfe von Message-IDs eindeutig identifiziert werden können. WS-Addressing definiert zwei Konstrukte, welche die Informationen übermitteln. Diese Konstrukte bringen diese Information in das Uniform-Format, das vom Transport oder von der Anwendung unabhängig verarbeitet werden kann. Zwei Konstrukte sind *Endpoint Reference* und *Message Information Headers*.

**Endpoint References** übermitteln die Information, die zur Identifikation/Referenz des Webservice-Endpunkts benötigt wird und können auch verwendet werden, um Adressen für Nachrichten, die an und von Webservices gesendet werden, zur Verfügung zu stellen. Um diese Aufgabe zu erfüllen, definiert diese Spezifikation die **Message Information Headers**, welche die Uniform-Adressierung unabhängig vom Transport erlaubt. Diese *Message Information Headers* enthalten eine Nachrichten-Charakteristik (Adressen für Quell- und Ziel-Endpunkte).

Die WS-Addressing definiert zusätzliche Eigenschaften von Standard-SOAP-Nachrichten, wie z.B. *To*, *Action*, *ReplayTo*, *FaultTo*, *MessageId*, *From* und *RelatesTo*.

### 3.3.2. WS-Discovery

WS-Discovery [F.J05, CKK+05] definiert ein Multicast-Discovery-Protokoll, um Geräte und Services finden zu können und beschreibt ein Nachrichtenaustausch-Modell zwischen einem Gerät und Client. Die Suche kann anhand des Typs und/oder Bereichs (*Scopes*) und des Namens durchgeführt werden. Wenn der Client die Ziel-Services anhand des Types und/oder Bereichs sucht, sendet er eine Probe-Nachricht an eine Multicast-Gruppe. Die Ziel-Services, welche die Nachricht empfangen haben, senden an den Client als Antwort eine *ProbeMatch*-Nachricht, wenn sie der Suchanfrage entsprechen. Statt *Probe* und *ProbeMatch* können *Resolve*- und *ResolveMatch*-Nachrichten versenden werden, die zur Namensauflösung gedacht sind.

Hello- (beim Betreten des Netzwerks) und Bye-Nachrichten (beim Verlassen des Netzwerks) werden an eine Multicast-Gruppe gesendet. Das reduziert die Anzahl von Nachrichten und damit den Netzverkehr.

### 3.3.3. WS-MetadataExchange

Ein Client braucht Information über Geräte und seine Services, um Nachrichten korrekt senden und empfangen zu können. Die verwendeten Metadaten enthalten Informationen über Geräte und Services. **WS-Policy** beschreibt Fähigkeiten, Anforderungen und allgemeine Charakteristiken von Webservices; **WSDL** beschreibt abstrakte Nachrichtenoperationen, bestimmte Netzwerkprotokolle und Endpunkt-Adressen, die von Webservices benutzt werden; **XML-Schema** beschreibt die Struktur und den Inhalt von XML-basierten Nachrichten, die von Webservices empfangen und gesendet werden.

Eine Gerätebeschreibung kann angefordert werden, indem ein Client ein *GetMetadata* dem Gerät sendet, sowie für das Gerät kann der Client auch die Beschreibung für einen oder mehrere Services anfordern. Solche Metadaten beinhalten z.B. *ThisModel*-Metadata (beschreibt das Geräte-Modell), *ThisDevice*-Metadata (beschreibt das Gerät), *Relationship*-Metadata (beschreibt die Beziehungen zwischen den *Hosted Services* und dem Gerät).

### 3.3.4. WS-Eventing

WS-Eventing [F.J05, CKK+05] ist ein Standard, der ein Publish-/Subscribe-System ermöglicht. Das WS-Eventing-Protokoll definiert Nachrichten zum Abonnieren einer Eventquelle, zum Beenden einer Subscription und zum Senden von Events.

WS-Eventing definiert das Protokoll für einen Webservice (*Subscriber*), der sich für Benachrichtigungen bei einem anderen Webservice (*Event Source*) registrieren möchte (*Subscription*), die Nachrichten über Ereignisse zu empfangen (*Notifications* oder *Event Messages*). **Event Sink** ist derjenige Webservice, der solche Notifications bekommt. Der **Subscriber** kann mit der Interaktion von Webservices das **Subscription** steuern (*Subscription Manager*).

Die Subscription kann vom **Event Source** erstellt und mit der Gültigkeitsdauer an den Subscriber gesendet werden. Dann hat der Subscriber die Wahl entweder die Subscription zu erneuern, solange es noch nicht abgelaufen ist, oder es annullieren zu lassen. Diese Aufgabe (erneuern, löschen, den Status verwalten) übernimmt der **Subscription Manager**.

### 3.3.5. WS-Security

Die WS-Security-Spezifikation [F.J05, CKK+05] ist die Basis für eine große Auswahl von Sicherheitslösungen.

Im Netzwerkmodell kommunizieren Clients und Geräte miteinander, nach der Authentisierung durch verschlüsselte Kanäle. Das IP-basierte Netzwerk kann ein oder mehrere administrative Domänen (wie *workgroup subnet*) umfassen, eine Domäne hat viele Subnets oder beinhaltet viele administrative Domänen (wie das globale Internet). Das Sicherheitsniveau ist durch Sicherheitsrichtlinien von administrativen Domänen bestimmt, die in verschiedenen Umgebungen unterschiedlich sind.

Die WS-Security-Spezifikation bietet drei Mechanismen:

- **Integrität**: schützt Nachrichten gegen Verfälschung mit Hilfe einer digitalen Signatur (*XML Signature*)
- **Vertraulichkeit**: schützt empfindliche Informationen gegen nicht autorisierten Zugriff mit Hilfe einer digitalen Verschlüsselung (*XML Encryption*)

- **Authentisierung**: stellt die Identität des Absenders mit Hilfe von Security-Tokens fest. Diese Security-Tokens können unsigned (*username/password*) oder signed (*X.509 Certificates, Kerberos tickets*) sein.

## 3.4. Implementierung des DPWS-Stacks

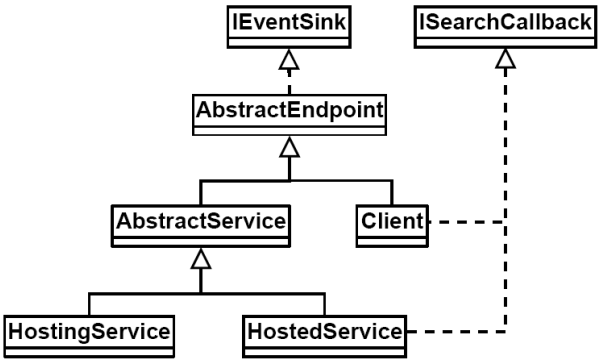


**Abbildung 9:** DPWS-Klassendiagramm

Der DPWS-Stack enthält Klassen gemäß der DPWS-Spezifikation, welche die benötigten Instanzen implementieren. Die Abbildung 9 gibt einen Klassendiagramm-Überblick. Die wichtigsten Klassen sind:

- **AbstractEndpoint** aktiviert die *Network endpoint*-Eigenschaften für Geräte, Services und Clients, die mithilfe dieses Stacks erstellt wurden, und implementiert die **IEventSink**-Schnittstelle, um auf Events zugreifen zu können.
- **AbstractService** enthält die Basisfunktionalität zur Verwaltung von Porttypen und Actions.
- **HostingService** implementiert die Gerätefunktionalität, kann Services hosten und Attribute, wie die Herstellerinformation, etc. einstellen. Außerdem kann er den Server für eine TCP-/UDP-Kommunikation starten.
- **HostedService** stellt eine Service-Funktionalität zur Verfügung.
- **Client** stellt die Client-Funktionalität zur Verfügung.

Beide Klassen *HostedService* und *Client* implementieren die ***ISearchCallback***-Schnittstelle, um Antworten auf Suchanfragen zu bekommen.

- **RemoteService** ist eine lokale Darstellung eines entfernten Services, um einen schnellen und einfachen Zugriff innerhalb des DPWS-Stacks zu ermöglichen.

### Packages des DPWS-Stacks

Der DPWS-Stack enthält folgende Packages:

- **edu.udo.cs.sirena.communication**
  Dieses Package enthält Klassen zur Durchführung der „I/O“-DPWS-Kommunikation (*Dispatcher, ServerEngine*).

- **edu.udo.cs.sirena.communication.http**
  Dieses Package enthält Klassen für die HTTP-Kommunikation (*HTTPServer, HTTPClient*).

- **edu.udo.cs.sirena.communication.soap**
  Dieses Package enthält Klassen für die SOAP-Kommunikation (*SOAPServer, SOAPMessageHandler*).

- **edu.udo.cs.sirena.communication.udp**
  Dieses Package enthält Klassen für die UDP-Kommunikation (*UDPServer, DatagramOutputStream*).

- **edu.udo.cs.sirena.constants**
  Dieses Package enthält Klassen mit den meisten Konstanten, die vom DPWS-Stack benötigt werden (*DPWSConstants, SOAPConstants*).

- **edu.udo.cs.sirena.service**
  Dieses Package enthält Klassen zum Erstellen, Behandeln und Benutzen von Services, Geräten und Clients (*AbstractEndpoint, AbstractService, HostedService, HostingService, Client, Action, Parameter*).

- **edu.udo.cs.sirena.service.attachment**
  Dieses Package enthält Klassen zum Behandeln von Anhängen (*Attachment, AttachmentData*).

- **edu.udo.cs.sirena.service.discovery**
  Dieses Package enthält Klassen für das DPWS-Discovery (*SearchManager, Resolver*).

- **edu.udo.cs.sirena.service.eventing**
  Dieses Package enthält Klassen für das DPWS-Eventing (*EventManager*).

- **edu.udo.cs.sirena.service.remote**
  Dieses Package enthält Klassen, die den Zugriff auf entfernte Geräte und Services gewährleisten (*RemoteService, RemoteHostingService*).

Die einzelnen Klassen von diesen Packages können aus der Java-Dokumentation entnommen werden.

## 3.5. DPWS-Explorer

Der DPWS-Explorer ist eine Anwendung zum Anzeigen und Nutzen von DPWS-Geräten und ihrem Services in einer Netzwerkumgebung. Jede gesendete oder empfangene Nachricht kann in *Message Log* angezeigt werden. Da der DPWS-Explorer bei Services Actions aufrufen kann, ist er ein geeignetes Werkezeug, um einfache Tests durchzuführen.

Um den DPWS-Explorer zu starten, muss die main-Mathode der Klasse **DPWSExplorer** ausgeführt werden.

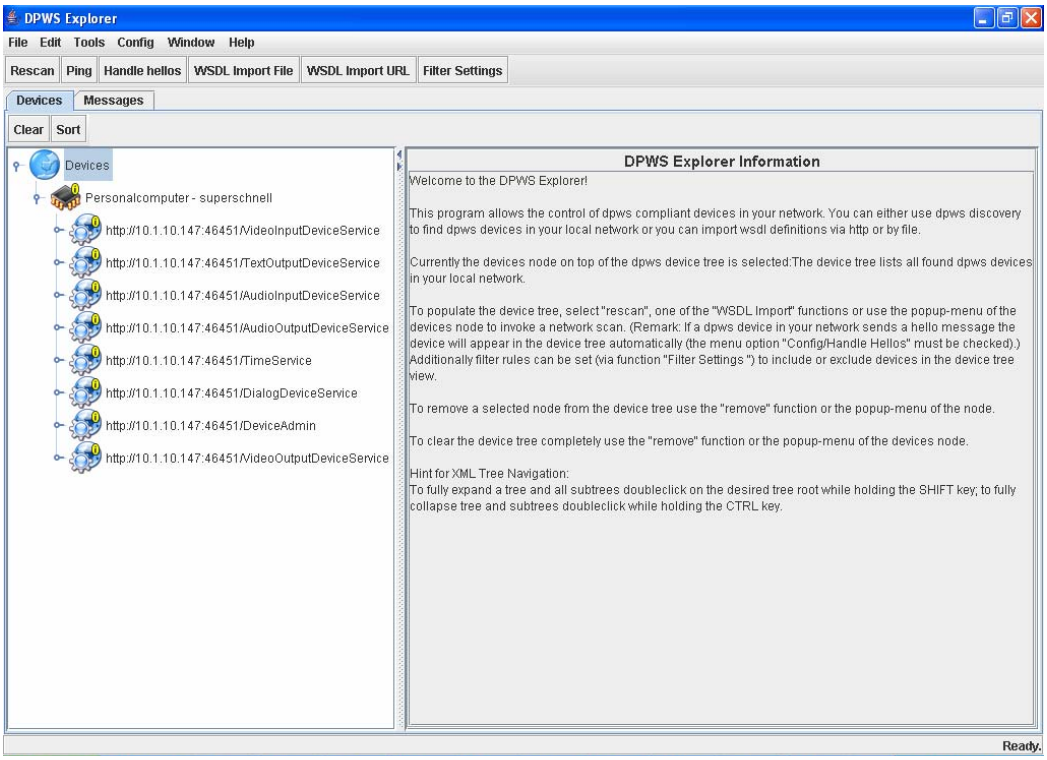


**Abbildung 10:** DPWS-Explorer

Alle verfügbaren Geräte sind in der linken Seite des DPWS-Explorers unter **Devices** aufgelistet. Jeder Device-Knoten hat Services. Services enthalten Actions, die man per Klick auf den Service sehen kann. Details über die markierten Knoten sind in der rechten Seite des DPWS-Explorers angezeigt. Die kompletten Informationen zu Geräten und ihren Services kann man als XML-Baum unter **Metadata as XML Tree**-Tab oder als Text unter **Metadata as Text**-Tab finden. Um zu sehen, welche Geräte laufen, kann man das Netzwerk mit der Taste **Rescan** scannen. Wenn **Handle hellos** ausgewählt ist, werden neue Geräte automatisch zum *Device Tree* hinzugefügt. Es ist möglich, Geräte auch manuell mit einer WSDL-Datei oder einer URL zu einer WSDL-Datei, welche die Service-Beschreibung enthält, hinzuzufügen. Geräte und Services können aus *Devices* automatisch oder manuell entfernt werden. **Remove** entfernt die ausgewählten Baumknoten. **Remove all** entfernt alle *Device*-Einträge.

# 4. Unsere Entwicklungsumgebung

## 4.1. Programmiersprachen

In der Projektgruppe wurden verschiedene Programmiersprachen verwendet.

Java ist eine plattformunabhängige Programmiersprache, welche von Sun Microsystems Mitte der 90er Jahre entwickelt wurde. Die Plattformunabhängigkeit wurde durch die Verwendung des so genannten *Bytescode* erreicht. Java-Programme werden, im Gegensatz zu vielen anderen Programmiersprachen, nicht direkt in nativen Code für eine bestimmtes System, sondern in den *Bytecode* übersetzt. Dieser *Bytecode* kann dann in einer *Java-Laufzeitumgebung* ausgeführt werden. Die wichtigste Komponente ist dabei die *Java Virtual Machine (JVM)*, welche den Bytecode interpretiert und das Programm ausführt. Die *Java Virtual Machine* ist für viele Plattformen verfügbar (z.B. Windows, Linux, Mac OSX, Solaris, etc.) und ermöglicht es, einmal geschriebene Java-Anwendungen auf verschiedensten Systemen auszuführen.

Neben der klassischen Java-Plattform war für uns besonders die *Java Micro Edition (J2ME)* von Interesse. Die Java Micro Edition definiert die Umsetzung der Programmiersprache Java für *Eingebettete Systeme*, wie beispielsweise PDAs und Handys. Ein für uns relevantes Kernstück der J2ME-Plattform ist die *Connected Limited Device Configuration (CLDC)*. Sie definiert eine minimale Laufzeitumgebung für die Ausführung einfacher Java-Programme. Heutzutage unterstützen beinahe alle marktüblichen Mobiltelefone mindestens CLDC 1.0. Programme laufen dann in einer minimalen Virtuellen Maschine, welche als *K Virtual Machine (KVM)* bezeichnet wird. Einen groben Überblick über die Java-ME-Komponenten und ihre Beziehung zur gesamten Java-Plattform ist in Abbildung 11 gezeigt.

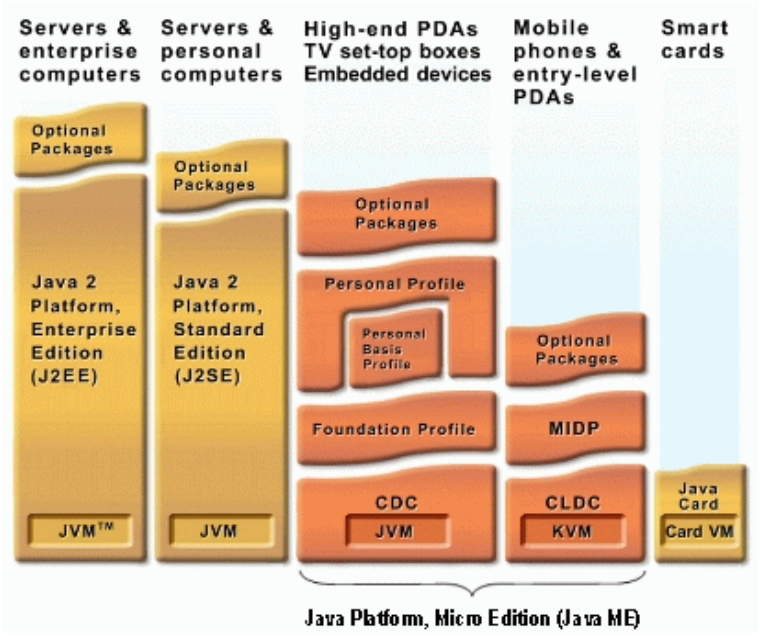


**Abbildung 11:** Java ME Komponenten (aus [JME])

Eine besonders wichtige Eigenschaft der Java-Plattform ist, dass sie trotz ihrer plattformunabhängigen Natur die Möglichkeit bietet, auf native betriebssystemspezifische Funktionen des jeweiligen Systems zuzugreifen. Für diesen Zweck bietet die Java-Plattform die *Java Native*

*Interfaces (JNI)* an. Diese Schnittstelle ermöglicht es, dass ein Java-Programm auf Funktionen zugreifen kann, die in einer nativen Programmbibliothek, wie z. B. einer *Windows DLL* oder *Linux Shared Library*, realisiert sind. Für die J2ME-Plattform bieten die *K Native Interfaces (KNI)* eine ähnliche Funktionalität wie JNI für die Java-Standard-Plattform. Die KNI ermöglichen es, native Funktionen direkt in die KVM zu integrieren (Siehe Kapitel 9).

Die nativen Funktionen wurden von uns in der Programmiersprache C realisiert. Die Sprache C entstand in den frühen 70er Jahren und bildete damals die Grundlage für das Betriebssystem Unix. Auch heute ist C eine äußerst beliebte Programmiersprache. Sie wird in erster Linie für grundliegende Programme sowie für den Systemkern vieler Betriebssysteme verwendet.

## 4.2. Eclipse

Bei der Entwicklung großer Softwareprojekte ist die Verwendung einer *Integrierten Entwicklungsumgebung (IDE)* unabdingbar, um bei der Arbeit optimal unterstützt zu werden. Im Open-Source-Bereich hat sich die Entwicklungsumgebung *Eclipse* (siehe [ECL]) einen besonders guten Ruf erarbeitet. Einen Screenshot von Eclipse zeigt Abbildung 12.

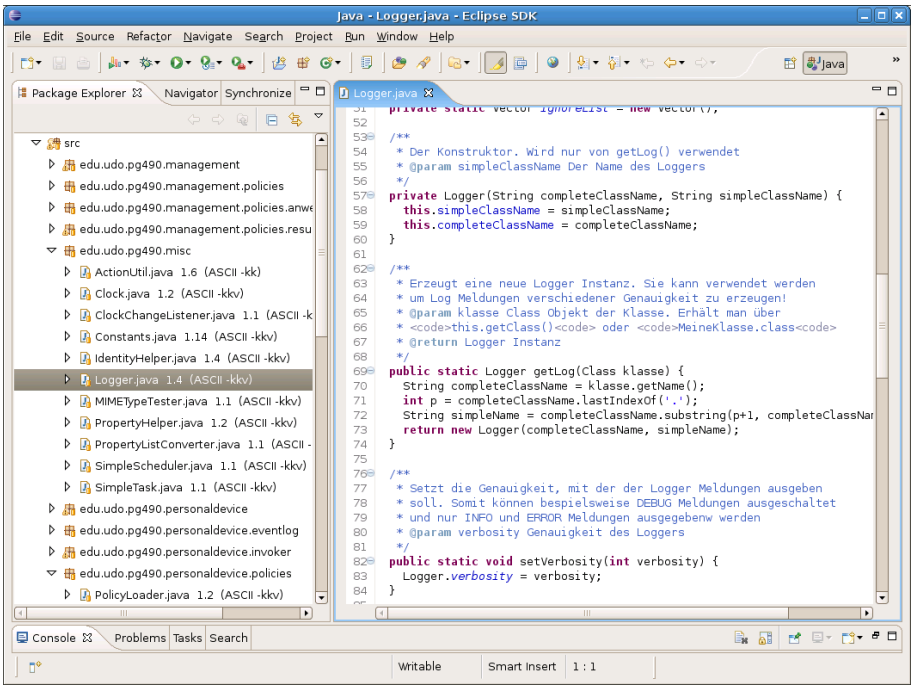


**Abbildung 12:** Eclipse IDE

Eigentlich handelt es sich bei Eclipse um ein universelles Open-Source-Framework für die Softwareentwicklung. Bekannt ist Eclipse jedoch in erster Linie als Entwicklungsumgebung für die Programmiersprache Java.

Aus unserer Sicht bot Eclipse eine Reihe von Vorteilen, die letztendlich dazu geführt haben, dass wir uns für die Verwedung von Eclipse entschieden haben:

- Eclipse ist ein Open-Source-Projekt und kostenfrei für verschiedene Betriebssysteme verfügbar
- Vollständige Entwicklungsumgebung für die Entwicklung von Java-Programmen

- Volle Unterstützung des Versionsverwaltungssystems CVS
- *Visual Editor* zu einfachen Gestaltung von GUIs
- Erweiterbarkeit durch Plugins (z.B. EclipseME[1])

## 4.3. Foxboard

Da wir nicht nur Geräte simulieren wollten, haben wir zusätzlich eine Minicomputer von der Firma *ACME Systems* eingesetzt, das so genannte **Fox-Board**. Es handelt sich hierbei um ein eingebettetes System, welches als Betriebssystem Linux benutzt, was für die Bluetooth-Funktionalität von großem Vorteil ist. In der vorliegenden Fassung besitzt es folgende Austattungsmerkmale:

| | |
|---|---|
| Cpu | Axis ETRAX 100LX 32 bit, RISC, 100MHz (100MIPS) |
| Memory | 16MB of RAM, 4MB of FLASH |
| Power | Single power supply, 5 Volt 280mA (1 watt) |
| Ports | 1 Ethernet (10/100 Mb/s), 2 USB 1.1, 1 serial console port |
| Kernel | Full Linux ver 2.4.31 |

Da wir uns vorgenommen haben, das Fox-Board als Bluetooth-Gateway zu nutzen, musste die original KVM angepasst und um zusätzliche Funktionen erweitert werden (siehe Kapitel C.2). Dazu gibt es eine Entwicklungsumgebung der Herstellerfirma [2], die alle vom Hersteller zur Verfügung stehenden Funktionen mitbringt. Das heißt, diese Entwicklungsumgebung umfasst alle Teile, die in der original KVM enthalten sind. Weitere optionale Elemente können während der Installation angegeben werden. Im Wesentlichen bildet diese Entwicklungsumgebung eine Verzeichnisstruktur, in der alle einzelnen Gerätetreiber im C-Quellcode enthalten sind. Das heißt, alle Geräte, die das Foxboard mitbringt wie USB-Ports, Ethernet etc., sind hier enthalten und werden individuell je nach Anwenderwunsch und benötigten Foxboard-Funktionen während der Installation zusammengestellt. Eine genaue Installationsanleitung für unsere Entwicklungsumgebung und die Anleitung, wie die KVM erstellt wird, steht im Anhang (siehe Kapitel C.1).

[1] Plugin für die Entwicklung auf der J2ME-Plattform
[2] www.acmesystems.it

# 5. Der Weg zum Digital Home

Das Projekt *Digital Home* stand zu Beginn der Projektgruppe noch nicht fest, sondern ist das Ergebnis eines längeren Lern- und Entscheidungsprozesses.

Zu Beginn der Projektgruppe wurde eine Seminarphase veranstaltet, um alle Teilnehmer auf einen einheitlichen Wissensstand in Sachen Theorie zu bringen. Anschließend wurde eine Praktikumsphase abgehalten, die zum Ziel hatte auch in Fragen der praktischen Anwendung von Webservices alle Teilnehmer auf ein Niveau zu bringen.

Zum Abschluss der Einarbeitungsphase wurden verschiedene Projektideen besprochen, in deren Rahmen das automatisierte Management von Web-Service-Systemen realisiert werden könnte. Daraufhin entschieden sich schließlich die meisten Projektgruppen-Teilnehmer dafür, das Projekt *Digital Home* zu realisieren.

## 5.1. Seminar

Das Seminar dieser Projektgruppe umfasst vier verschiedene Themenschwerpunkte: *Services*, *Fehlertoleranz und Self-X*, *Management und Policies* und *Web Service Management*.

Im Bereich *Services* finden sich die vier Vorträge *Service-Oriented Architecture*, *Web-Services: Übersicht und Standards*, *Web-Services: Plattformen (Websphere und Apache)* und *DPWS und UPnP*. Der zweite Themenschwerpunkt umfasst *Recovery-Oriented Computing* und *Autonomic Computing (IBM). Technisches Management: DTMF-Ansatz und CIM*, *Management-Prozesse und Aktionen nach ITIL*, *Policy-Based Management* und *Goal-Oriented Policy-Refinement* finden sich im dritten Schwerpunkt und schließlich *Web Services Distributed Management (WSDM)* und *IBM: Policy Management for Autonomic Computing (PMAC)* sind die Themen des letzten Themenschwerpunkts.

Im Folgenden werden die Inhalte der Seminarthemen grob umrissen.

### 5.1.1. Service-Oriented Architecture

SOA ist ein Konzept für eine Systemarchitektur, in dem Funktionen in Form von wiederverwendbaren, technisch voneinander unabhängigen und fachlich lose gekoppelten Services implementiert werden.

Services können unabhängig von zugrunde liegenden Implementierungen über Schnittstellen aufgerufen werden, deren Spezifikationen öffentlich sein können. Serviceinteraktion findet über eine dafür vorgesehene Kommunikationsinfrastruktur statt. [Kuh]

### 5.1.2. Web-Services: Übersicht und Standards

Webservices sind eine Technologie, die es ermöglicht, verteilte Anwendungen über das Web zu verbinden und die Daten als XML-Dokument zwischen ihnen zu übermitteln. Unter Webservices versteht man hierbei ein Bündel von Technologien zur Beschreibung von Schnittstellen, Eigenschaften und Implementierungen, Beschreibung von Dateiaustauschformaten usw.

Nähere Ausführungen zum Thema Webservices finden sich in Kapitel 2.

### 5.1.3. Web-Services: Plattformen (Websphere und Apache)

Webservice-Plattformen bezeichnen eine abgestimmte Menge mit Ewicklungsumgebung, Laufzeitumgebung und weiteren Bestandteilen wie Verzeichnissen und Webservice-Management-

Umgebungen sowie einzelne Webservices auf Basis eines Webservice-Frameworks.

*Apache Axis* (Apache eXtensible Interaction System) ist eine SOAP-Engine zur Konstruktion von darauf basierenden Webservices und Client-Anwendungen und wird durch die *Apache Software Foundation* entwickelt. Diese Organisation entwickelt auch den Apache-HTTP-Webserver und weitere Open-Source-Lösungen für den Internet-Bereich. Axis ist bereits die dritte Generation dieser Software, die ursprünglich als *Apache SOAP* startete.

Websphere umfasst eine Gruppe von Softwarelösungen rund um den *WebSphere Application Server*, wobei es sich um eine Laufzeitumgebung für J2EE-Anwendungen (Java 2 Enterprise Edition) handelt. Websphere existiert in drei Versionen: Die *Standard Edition* kombiniert die serverseitigen Geschäftsanwendungen mit Java-Technologien. Die *Advanced Edition* baut auf den *Standard Application Server* auf und fügt EJB-Support und Einbindung von Web-Anwendungen in andere Anwendungen hinzu. Die *Enterprise Edition* schließlich baut auf den *Advanced Application Server* auf und bietet zusätzlich die Einbindung von E-Business-Anwendungen in Unternehmensumgebungen auf der Basis von *TXSeries*.

### 5.1.4. DPWS und UPnP

UPnP ist eine *Erweiterung* des bekannten PnP (Plug and Play). Dieser Begriff wurde zum ersten Mal von Microsoft bei Windows 95 benutzt und ermöglicht eine automatische Konfiguration von Geräten, auch im Betrieb. PnP kümmert sich um eine automatische Erkennung und Konfiguration von Hardwarekomponenten ohne Interaktion des Benutzers. UPnP [UPn] ist eine im Jahr 1999 von Microsoft initiierte und vom UPnP-Forum entwickelte Architektur, die es Computern in einem Netzwerk ermöglicht, die Fähigkeiten von anderen Netzwerkkomponenten zu erkennen und zu nutzen.

DPWS wird für die nächste Version (V2) der UPnP Geräte-Architektur vorgeschlagen, und kennzeichnet die Konvergenz zwischen UPnP und Webservice-Technologien. Die aktuelle Generation der Windows-Plattform (*Vista*) hat DPWS integriert. DPWS benutzt Teile verschiedener Webservice-Spezifikationen und erlaubt Geräten einen Standard-Mechanismus zu nutzen, um mit anderen DPWS-Geräten und Computern zu kommunizieren.

Nähere Ausführungen zum Thema DPWS finden sich in Kapitel 3.

### 5.1.5. Recovery-Oriented Computing

*Recovery-Oriented Computing* geht davon aus, dass Fehler im Alltag von Computersystem unvermeidbar sind und die *Mean Time to Failure* (MTTF) nicht beliebig ausgedehnt werden kann. Deshalb ist es empfehlenswert, sich auf eine Reduktion der *Mean Time to Repair* (MTTR) statt auf eine noch größere Ausdehnung der MTTF zu konzentrieren, um die Verfügbarkeit zu erhöhen.

Zur Erreichung dieses Ziels gibt es ein Bündel an Techniken: Eine Testumgebung, die das Einspeisen von Fehlern in reale Systeme ermöglicht, ein System, das anhand von statistischer Analyse defekte Komponenten erkennt, Microreboots, um nur fehlschlagende Komponenten neu zu starten und nicht das ganze System, eine systemweite Undo-Funktion, um Fehler vor dem Auftreten beseitigen zu können und das Mittel der Redundanz.

### 5.1.6. Autonomic Computing (IBM)

Dieses Seminarthema beschäftigt sich mit Autonomic-Computing, einem Ansatz, den IBM seit ca. 2001 verfolgt. Autonomic-Computing-Systeme sollen mit dem maximalen Einsatz von

Self-Management entscheidende Vorteile gegenüber den bekannten Computersystemen besitzen. Self-Management lässt sich in Self-Configuration, Self-Healing, Self-Optimization und Self-Protection aufteilen und deckt damit alle Aufgaben der System-Verwaltung ab. Um zu verdeutlichen, wie Komponenten und deren Interaktionen in einem Autonomic-Computing-System aussehen können, wird die Autonomic-Computing-Referenz-Architektur betrachtet. Die Kernfunktionalität des automatisierten Managements befindet sich dabei in den Autonomic-Managern, die eine Kontroll- und Steuerungsschleife enthalten. Abschließend wird auf die Beziehung zwischen Autonomic-Computing und der Service-Oriented-Architecture eingegangen. Anhand von Beispielen wird dabei der Nutzen der SOA-Standards in einer Autonomic-Computing-Umgebung gezeigt.

### 5.1.7. Technisches Management: DTMF-Ansatz und CIM

Das *Common Information Model* (CIM) ist ein Standard, der von der *Distributed Management Task Force* geschaffen wurde, um das Management von Systemen und Geräten über ein Netzwerk zu vereinfachen. Basierend auf der Objektorientierung, soll mit Hilfe eines einheitlichen Schemas eine Managementunterstützung geschaffen werden. Hierbei hilft nicht nur der einheitliche und strukturierte Aufbau, sondern das Konzept der Technologieunabhängigkeit sichert eine Standardisierung, die plattformunabhängig ist.

### 5.1.8. Management-Prozesse und Aktionen nach ITIL

ITIL, die *Information Technology Infrastructure Library*, ist ein *Best Practice Framework* für die Definition und den Betrieb von IT-Prozessen, die in IT-Organisationen erfolgreich eingesetzt wird. Träger dieser öffentlich zugänglichen Bibliothek ist das OGC (Office of Government Commerce), vormals als CCTA (Central Computer and Telecommunications Agency) bekannt. Das OGC ist eine britische Regierungsbehörde, unter deren Regie ITIL gegen Ende der 80er Jahre entwickelt wurde. ITIL ist somit ein herstellerunabhängiges, prozessorientiertes Regelwerk. ITIL beschreibt die Prozesse, die einen effizienten und effektiven Betrieb der gesamten IT-Infrastruktur ermöglichen, um die zwischen der IT-Organisation und ihren Kunden vereinbarten Service-Levels einhalten zu können. Es enthält Erfahrungswerte von 30 Jahren IT-Betrieb.

ITIL ist heute der weltweite De-facto-Standard im Bereich Service-Management und beinhaltet eine umfassende und öffentlich verfügbare fachliche Dokumentation zur Planung, Erbringung und Unterstützung von IT-Serviceleistungen. ITIL bietet die Grundlage zur Verbesserung von Einsatz und Wirkung einer operationell eingesetzten IT-Infrastruktur.

### 5.1.9. Policy-Based Management

Beim *Policy-Based Management* geht es um die automatische Verwaltung von Ressourcen auf der Grundlage von sogenannten *Policies* (in etwa *Richtlinien* in deutscher Übersetzung). Der Einsatz eines solchen Konzeptes bietet sich beispielsweise in der Verwaltung von Computersystemen und Netzwerken an, um bei deren Administration Zeit und Kosten sparen zu können. So können ursprünglich manuell durchgeführte Aufgaben maschinell und automatisch verarbeitet werden, wodurch schnellere Reaktionen auf Änderungen und eine geringere Fehleranfälligkeit, trotz einer wachsenden Komplexität des Systems, erreicht werden können.

### 5.1.10. Goal-Oriented Policy-Refinement

Da das Interesse an policy-basierten Verfahren für Systemmanagement steigt, wird es immer wichtiger, Methoden für die Analyse und die Verfeinerung von Policy-Spezifikationen zu entwickeln. Obwohl sich die Forschung damit schon relativ lange beschäftigt, gibt es noch keine Lösung, bei der die zu implementierenden Policies aus High-Level-Zielen abgeleitet werden. Ein wichtiger Teil der Lösung ist es, die Bestimmung der verfügbaren Operationen zu ermöglichen. In diesem Seminarthema wird ein Verfahren vorgestellt, bei dem eine formale Darstellung basierend auf dem *Event Calculus* zusammen mit einer speziellen Beweisführungstechnik benutzt wird, um die Sequenz der Operationen für ein zu erreichendes Ziel abzuleiten. Abschließend wird an einem Beispiel gezeigt, wie das Verfahren zusammen mit UML-Modellierung angewandt wird.

### 5.1.11. Web Services Distributed Management (WSDM)

Eines der wichtigsten und schwerwiegendsten Probleme der Informationstechnologie ist das Management von IT-Ressourcen. Zwar existieren weitverbreitete Management-Tools, jedoch sind diese meist auf spezielle Ressourcen und Systeme zugeschnitten und bieten keinen ganzheitlichen Ansatz.

Die *Web Services Distributed Management Spezifikation* (WSDM) definiert die Methoden und Strukturen eines Systems, das es ermöglicht, sowohl klassische Netzwerk-Ressourcen, wie z. B. Drucker, Server und Router, als auch Webservices zu verwalten.

Die WSDM-Spezifikation ist dabei in zwei Teilbereiche gegliedert. MUWS (Management Using Web Services) behandelt das Management von Informationstechnologie-Ressourcen unter der Verwendung von Webservices. Die Art der Ressourcen, die mit Hilfe dieser Spezifikation managebar sind, ist sehr vielfältig und nicht auf einzelne spezifische Ressourcen beschränkt. Eine universale Behandlung verschiedenster Ressourcen ermöglicht dadurch ein einfaches und einheitliches Management.

Die andere Hälfte der WSDM-Spezifikation behandelt MOWS (Management Of Web Services). MOWS setzt dabei auf die Ideen von MUWS auf und erweitert diese Ideen für das Management von Webservices.

### 5.1.12. IBM: Policy Management for Autonomic Computing (PMAC)

PMAC ist eine von IBM entwickelte Infrastruktur, die eine selbstkonfigurierende, selbstheilende, selbstoptimierende und selbstschützende Umgebung ermöglichen soll. Dabei werden Applikationen durch einen autonomen Manager geleitet, der seine Entscheidungen durch zentral gespeicherte Policies trifft. Die Policies sind also von den Applikationen getrennt und können verändert werden, ohne die Anwendung ändern zu müssen. Gleichzeitig kann eine Policy mit demselben autonomen Manager für mehrere, verschiedene Applikationen benutzt werden. Konkret geht es darum, Policies zu definieren, einzusetzen (deploy), zu evaluieren und schließlich anzuwenden und *Policy Control* in die Anwendungen zu implementieren. [Bel04], [IBM05a]

## 5.2. Praktikum

In der praktisch orientierten Vorbereitungsphase dieser Projektgruppe wurden von den Teilnehmern in selbstständiger Arbeit Aufgaben zur Bearbeitung entwickelt. Die Themen sind *Apache Axis* und *Websphere*, *Eclipse mit CVS*, *PMAC*, *Workflow Orchestration* und *MoBaSeC*.

### 5.2.1. Apache Axis

Das Tutorial zu *Apache Axis* bietet zunächst eine ausführliche Installationsanleitung für *Jakarta Tomcat*, einen Servlet-Container, für *Axis* selbst und für *Apache Ant*, ein Build-Werkzeug, dessen Benutzung das Kompilieren und Deployen von Webservices einfacher gestaltet.

In der ersten Aufgabe muss der Rahmen eines vorgefertigten Services, mit dem der Titel eines Films anhand einer ID erfragt werden kann, gefüllt werden, um diesen Service dann zu deployen und wieder zu undeployen. Anschließend wird in Java ein Client geschrieben, mit dem auf den erstellten Webservice zugegriffen werden kann.

Aufgabe zwei gibt einen tieferen Einblick in die Verwendung von komplexeren Datentypen. Dazu wird eine Klasse in Form einer *Java-Bean* programmiert und deployt. Getestet wird dieser Service mit einem modifizierten Client aus der ersten Aufgabe.

Die dritte und letzte Aufgabe geht davon aus, dass schon eine per WSDL beschriebene Schnittstelle zur Verfügung steht. Um nun diese Schnittstelle möglichst effizient zu nutzen, soll das zu *Axis* gehörende Werkzeug *WSDL2Java* verwendet werden, das aus einer WSDL-Datei fast allen für *Axis* nötigen Code generiert und so nur noch der eigentliche Aufruf programmiert werden muss. Rein praktisch soll dies in dieser Aufgabe mit der Webservice-Schnittstelle von Google getestet werden.

### 5.2.2. Websphere Application Server

Der *Websphere Application Server* von IBM ist ein kommerzielles Konkurrenzprodukt zu *Apache*. Die Aufgaben dazu geben zunächst die Download-URL an und den Hinweis, dass der Server zu starten ist.

Dann soll eine WAR-Datei zusammengestellt werden. Dazu muss eine vorgefertigte XML-Datei mit Inhalt gefüllt werden und entsprechend diverser Formvorschriften gezippt werden. Anschließend soll das Archiv menügesteuert deployt werden, um sich anschließend eine Test-HTML-Seite anzeigen zu lassen.

Da *Websphere* zum Installieren die WAR-Datei in eine EAR-Datei automatisiert verpackt hat, soll dieser Vorgang per Hand nochmal nachvollzogen werden. Dazu muss ein weiteres Grundgerüst in Form einer XML-Datei gefüllt werden und gezippt werden und diesmal mit einem Scripting-Tool in der Konsole deployt werden.

In der nächsten Aufgabe soll der erste Webservice dieses Tutorials geschrieben werden, ein Zeitdienst, mit dem die aktuelle Zeit des Servers abgefragt werden kann. Um jedoch nicht erneut von Hand die nötige WAR-Datei erstellen zu müssen, wird das Deploytool von *Sun* kurz beschrieben. Zum Schluss muss noch ein vorgefertiger Client kompiliert und installiert werden, mit dem der Webservice getestet werden soll.

Die letzte Aufgabe bietet wiederum einen vorgefertigen Client, der es ermöglicht, über ein Webformular eine Zahl einzugeben, die wiederum auf einer HTML-Seite ausgegeben wird. Die Aufgabe ist es, analog zur vorhergehende Aufgabe, einen Webservice zu schreiben, der die ausgegebene Zahl verdoppelt.

### 5.2.3. Eclipse mit CVS

Als Arbeitsmaterialien für dieses Tutorial sind zunächst *Eclipse* selbst und das Plugin *EclipseUML* herunterzuladen und zu installieren. Anschließend werden einige Optionseinstellungen vorgeschlagen (Zeichenkodierung UTF-8, richtiger Compliance-Level des Compilers) und die Aufteilung der *Eclipse*-Oberfläche mit Editoren, Views und Perspektiven vorgestellt.

In der ersten Aufgabe sollen ein neues Projekt und drei Klassen erstellt werden, wobei eine Klasse von der anderen erbt. Die Klassen werden mit einfachem Beispielcode gefüllt und getestet.

Als Nächstes wird anhand dieser Klassen der Debug-Modus von *Eclipse* demonstriert.

Zuletzt soll das UML-Plugin verwendet werden, um sich die zuvor erstellten Klassen als UML-Diagramm anzuschauen.

### 5.2.4. IBM – Policy Management for Autonomic Computing

Im ersten Teil des Tutorials soll das PMAC-Paket installiert werden, das eine Entwicklungsumgebung mit einem lokalen *Policy Editor Storage* (PES) und einen eingebetteten autonomen Manager (AM) enthält.

Nach einer kurzen Konfigurationseinführung soll man zunächst lernen, schon vorgefertigte WSDL- und XML-Beispieldateien zum PES hinzuzufügen und zu deployen. Dies geschieht per Kommandozeile.

Das Java-Programm *cldp* kann benutzt werden, um einen *Decision Point* zu implementieren und durch Aufruf von *cldp* soll nun eine *solicited Decision* an den autonomen Manager ausgelöst werden.

Als Nächstes wird ein *Decision Point* erstellt. Dies geschieht mittels der Decider-Klasse, die eine Funktion hat, um eine Entscheidungsanfrage zu senden. Dabei wird ein Decision-Input-Objekt übergeben und man erhält als Rückgabe ein Decision-Objekt.

Abschließend werden noch *unsolicited Decisions* vorgestellt, die z. B. durch Events ausgelöst werden können.

Der zweite Teil macht mit der ACPolicy-API vertraut, die es ermöglicht eine *Autonomic Policy*-Datei zu erstellen und zu ändern.

Zunächst muss eine Java-Klasse erstellt werden, in die dann das ACPolicy-Package importiert wird und ein ACPolicy-Objekt erstellt wird. Durch Kompilieren kann eine XML-Policy-Datei in ein ACPolicy-Objekt transformiert werden, welches deployt werden kann.

Im Folgenden wird beschrieben, wie man auf Policy-Informationen bei einem ACPolicy-Objekt zugreifen kann. Danach soll man eine Policy selbst schreiben.

### 5.2.5. Workflow Orchestration

Bevor zur Installation von *Netbeans* aufgefordert wird, wird *Workflow Orchestration* als ein Vorgang beschrieben, Prozesse in eine Reihenfolge zu bringen, in der sie ablaufen bzw. aufeinander zugreifen.

Nach der Installation soll man ein Tutorial auf der Website von *Sun* lesen. Im Beispiel dazu wird eine durch eine SOAP-Nachricht initiierte Nachricht einfach kopiert und an den Webservice zurückgeschickt.

Dann wird die Aufgabe gegeben, ein neues Projekt zu erstellen und den mitinstallierten Application-Server zu starten. Dem folgt eine Erläuterung der Entwicklungsumgebung und einfacher Basis-Operationen, die man damit ausführen kann. Abschließend wird das Deployen erklärt.

Die erste eigentliche Übungsaufgabe fordert, dass man das Beispielprojekt so abändert, dass eine Zahl quadriert wird.

In der zweiten Aufgabe soll man einen asynchronen Ablauf modellieren, bei dem jeder Webservice eine Antwort erwartet.

### 5.2.6. MoBaSeC

Das *Mobasec*-Tutorial enthält eine umfangreiche Anleitung zur Bedienung dieser Software, die erläutert, wie man Knoten erstellt und verbindet, Patterns erstellt und darauf basierend Graph-Transformationen durchführt.

In der ersten Aufgabe wird eine Grafik mit drei Clients, einem Proxy-Server und einem Server vorgegeben. Zu dieser Grafik soll ein Metamodell erstellt werden und mit *Mobasec* auf Korrektheit geprüft werden.

In Aufgabe zwei soll zunächst der Ansichten-Filter von *Mobasec* getestet und anschließend ein Metamodell erstellt werden, das Mehrfachvererbung nutzt. Zuletzt soll noch eine Graph-Transformation unter Verwendung von Pre- und Post-Patterns durchgeführt werden.

## 5.3. Konkurrierende Projektideen

Nach der theoretischen und praktischen Einarbeitung rund um das Thema des automatisierten Managements von Web-Service-Systemen galt es, ein Thema zu finden, in dem das Erlernte umgesetzt werden kann.

Die grundsätzliche Frage, ob Teilprobleme eines bestehenden Projekts, in diesem Fall CANDELA [Can], bearbeitet werden sollten oder ein eigenes Projekt mit eigenen Zielen verwirklicht werden sollte, wurde schnell zugunsten eines eigenen Projekts entschieden.

Das Ergebnis einer anschließenden Ideensammelphase waren mehrere Projektvorschläge (*Webservices für Krankenhäuser*, *Webservices für den Tourismus*, *Enterprise Service Bus*, *Ubiquitous Computing*), wobei drei Projekte einer tiefer gehenden Betrachtung als würdig empfunden wurden: *Lokaler Lokalisations- und Informationsdienst*, *Zentrale Internetforenverwaltung* und *Digital Home*, welches schließlich am meisten Teilnehmer überzeugte und in Kapitel 6 vorgestellt wird.

### 5.3.1. Lokaler Lokalisations- und Informationsdienst

Der Grundgedanke beim lokalen Lokalisations- und Informationsdienst ist, dass bestehende Lokalisationsdienste nicht detailreich genug sind. So ist es z. B. nicht möglich, einen Getränkeautomaten zu suchen und zu finden. Die These ist, dass dies für einen einzelnen Anbieter zu aufwendig ist, weshalb in diesem Projekt eine Umgebung geschaffen werden sollte, mit der jeder bequem einen eigenen Webservice mit Beschreibungspunkten für ein eng lokal begrenztes Gebiet zur Verfügung stellen könnte, die vernetzt eine sehr detaillierte Lokalisation ermöglichen sollten.

Im Detail sah der Vorschlag vor, dass in einer Suchmaske entweder mit Postleitzahl oder mit Längengrad und Breitengrad und einem Suchradius gesucht werden kann. Im Suchergebnis sollten Treffer nach Webservices sortiert angezeigt und durch Klick darauf hinterlegte Informationen angezeigt werden.

Die hinterlegten Informationen hätten z. B. im Falle einer Suche nach einem Getränkeautomaten aus einer Karte der Position, einem Luftbild, einem Foto des Gebäudes, in dem sich der Automat befindet, und einem Foto des Getränkeautomaten selbst bestehen können.

Technisch wäre der Suchvorgang realisiert worden, indem sich die einzelnen Webservices bei einer zentralen Registry anmelden und Geokoordinaten für gespeicherte Orte übergeben. Bei einer Suchanfrage an den zentralen Kommunkationsserver wären die Webserver ermittelt worden, die anhand des Suchradius infrage kommen, und wären dem Suchenden inklusive der

infrage kommenden Geokoordinaten übermittelt worden. Die Detailinformationen wären dann durch direkte Kommunikation mit den Webservices abgefragt worden.

Als Aufgaben für das Management dieses Webservice-Systems sah der Projektvorschlag zunächst die Pflege und Wartung des zentralen Registry-Servers vor, außerdem ein Notfallmanagement, wenn ein externer Geodaten-Server, der PLZ in Geodaten übersetzt hätte, ausgefallen wäre. Eine policy-basierte Auswahl der Informationsdarstellung (für Handy, Desktop-Rechner etc.), ein Sicherheitsmanagement, das das Verbergen privater Lokalisationspunkte ermöglicht und ein Kollaborationsmanagement, womit sich einzelne Webservices verschiedene Detail-Informationen hätten teilen können.

### 5.3.2. Zentrale Internetforenverwaltung

Die zentrale Internetforenverwaltung stellt sich des Problems der Suche eines Forums, wenn man Hilfe benötigt. Es ist aufwendig ein passendes und kompetentes Forum zu finden und in der Regel ist eine extra Anmeldung bei jedem neuen Forum nötig und man muss in jedem Forum separat suchen.

Dieser Projektvorschlag sah deshalb vor, dass sich Internetforen bei der zentralen Internetforenverwaltung registrieren, sodass dort mit einer einzigen Anmeldung viele verschiedene Foren genutzt werden können. Wenn z. B. eine Suche nach *Golf* gestartet werden soll, könnte der Nutzer anhand hinterlegter Metadaten die zu durchsuchenden Foren auf den Golfsport eingrenzen. Die Suche selbst wäre durch Aufruf der Suchfunktionen der jeweiligen Foren realisiert worden. Die empfangenden Suchergebnisse wären dann beim Zentraldienst anhand von Policies (z. B. für die Qualität der Foren) gelistet worden.

Die Managementaufgaben dieses Projektvorschlags liegen in zwei Teilbereichen: Technik und Service. Im ersten Teilbereich hätten demnach die Administration der Datenbank, die Authentifizierung der Foren und die Überprüfung der Verfügbarkeit der Forenserver gemanagt werden müssen. Im Servicebereich wäre es um die Qualitätskontrolle der Foren, die Metasuche und die Klassifizierung der Foren gegangen.

# 6. Projekt Digital Home

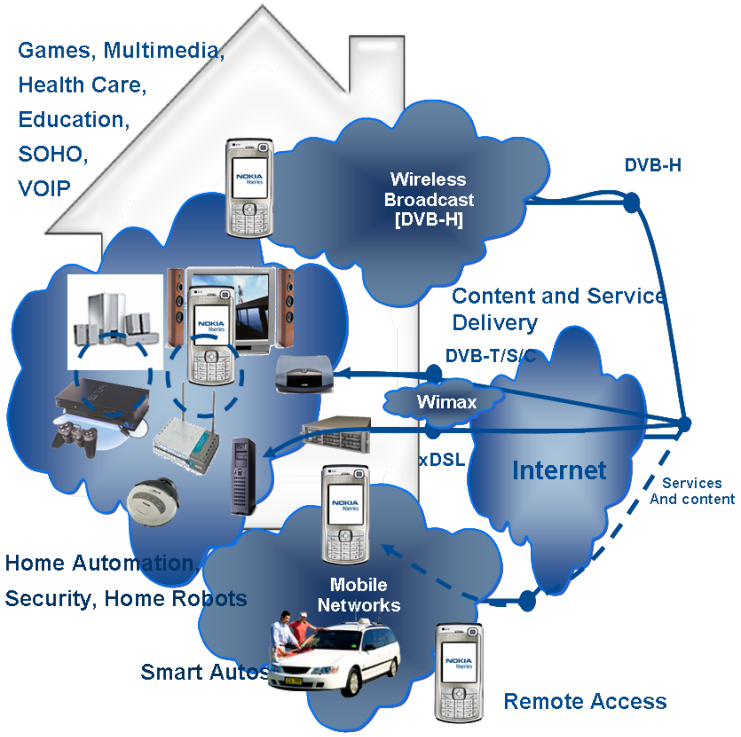


**Abbildung 13:** Die Vision von Nokia mit derselben Idee, die im Projekt Digital Home realisiert wurde. [Nokia Connecting People/www.nokia.de]

Digital Home ist eine Vereinfachung des alltäglichen Lebens. Es dient dazu, dass der Nutzer seine multimedialen Geräte nicht mehr manuell bedienen muss, sondern diese sich gegenseitig aktivieren und verwalten. Es nimmt dem Nutzer die Arbeit, wie zum Beispiel das Ein- oder Ausschalten eines Gerätes. Dies geschieht über das *Webservice System*, wodurch die Geräte automatisiert werden. Abbildung 13 zeigt die Vision von Nokia. Hier dient das Nokia Handy als persönliches Gerät, worüber die multimedialen Geräte verwaltet werden.

## 6.1. Die Idee

Die Idee entstand, nachdem die Seminarphase beendet war. Da es ein Projekt über *Automatisiertes Management von Web-Service-Systemen* sein sollte, wurden in der Praktikumsphase einige Ausarbeitungen in Betracht gezogen und der Entschluss fiel auf das Projekt *Digital Home*.

Die Idee im *Projekt Digital Home* besteht darin, dass es intelligente Gegenstände gibt, die unterschiedliche Anwendungen haben und diese autonom verwaltet werden. Daher werden die Geräte im Digital Home simuliert. Die nicht simulierten Geräte sind das *Foxboard* und *Bluetooth*.

Der Zugriff auf die Geräte soll über Policies geregelt und die Geräte sollen so gemanagt werden, dass sie automatisch ein anderes Gerät aktivieren können.

Als spezielle Anwendung für das Digital Home haben wir ein *Brandschutzszenario* erstellt,

in dem Feuermelder in den Räumen durch eine Brandmeldezentrale verwaltet werden. Die Idee in diesem Fall ist, sobald ein Feuer ausbricht, sollen die aktiven Geräte die Personen im Haus benachrichtigen, wo ein Feuer ausgebrochen ist.

## 6.2. Die Struktur

Durch eine gute Gruppenarbeit und Tools wurde das *Projekt Digital Home* erfolgreich abgeschlossen. Im Projekt Digital Home wurde mit der Entwicklungsumgebung Eclipse gearbeitet und unter anderem mit dem DPWS-Stack, Foxboard und Bluetooth.

Der DPWS-Stack wird für eingebettete Systeme benutzt und er basiert auf *Java ME*. Da in dem Server des Lehrstuhls IV der Universität Dortmund schon ein DPWS-Stack vorhanden ist, wurde dieser Server für das Digital Home genutzt. Der *DPWS-Stack* ist bei dem Unternehmen *Materna Information and Communications* noch in Entwicklung (siehe Kapitel 3).

Das Foxboard, ein Gerät, das nicht im Digital Home simuliert wurde, ist ein Minicomputer von der Firma *ACME Systems*. Es handelt sich hierbei um ein eingebettetes System, welches unter dem Betriebssystem Linux läuft und für die Bluetooth-Funktionalität ist dessen Verwendung vorteilhaft (Kapitel 5.2).

Der Projekt Digital Home wurde in vier Gruppen aufgeteilt, um das Projekt systematisch angehen zu können. Dabei wurden die folgenden Gruppen gebildet: *Architektur*, *Management*, *Die Geräte (Persönliches Gerät/Fernseher/PC)* und *Spezielles Anwendungszenario*. Jede Gruppe hat sich in ihr Thema eingearbeitet und ihre Aufgabe so entwickelt, dass die Gruppen überschneidend arbeiten konnten. Im Folgenden werden kurze Zusammenfassungen der Gruppen gegeben.

### 6.2.1. Architektur

Die Architektur-Gruppe hat sich mit der Installation von Bluetooth beschäftigt, damit die Personenerkennung in einem Raum realisiert werden kann. Für jeden Raum ist ein Bluetooth-Access-Point (Foxboard) vorgesehen. Es sind somit für das Digital Home zwei Foxboards ausreichend.

Die *Foxboards* (FoxboardSchlafzimmer und FoxboardWohnzimmer) bieten Services an, die miteinander interagieren, um die Personenerkennung im Digital Home zu realisieren (siehe Kapitel 9.2). Die beiden *RoomControlServices* (Services des Foxboards) können eine gegenseitige Servicesuche realisieren, die so lange abläuft, bis der andere Service gefunden wird. Danach wird die eigentliche Funktionalität des Services angestoßen. Nun startet in jedem Service die Funktion zur Personenerkennung und die Foxboards suchen in der Umgebung die vorhandenen Blutooth-Geräte.

Die Geräte im Digital Home sind über einen *Switch* miteinander verbunden und bekommen statisch eine IP-Adresse aus dem Subnetz 192.168.0.x (siehe Genaueres unter Kapitel 9.3).

### 6.2.2. Management

Die Management-Gruppe hat durch die Management-Komponenten das automatisierte Management im Digital Home realisiert. *Policies* wurden für die Devices im Digital Home definiert, damit die Geräte automatisch ihre Funktionen bekommen. Da es im Digital Home unterschiedliche Devices gibt, wie das Personal Devices und andere Geräte, unterscheidet sich auch die Management-Konfigurationen. Die unterschiedlichen Komponenten sind: *Access-Management*, *State-Management* und *Personal Management* (siehe Kapitel 10).

**Access-Management:** Das Access-Management kann in jedem Device (außer dem Personal Device) aktiviert werden. Dort kann es die Verwaltung und Umsetzung von Zugriffsrechten realisieren. Wenn ein Action-Aufruf in DPWS geschieht, wird das Access-Management befragt und es wird für den Benutzer überprüft, ob er diese Aktion durchführen darf. Wenn die Action nicht erlaubt ist, wird eine Fehlermeldung zurückgegeben (Kapitel 10.1.1).

**State-Management:** Das State-Management kann ebenfalls in jedem Device (außer dem Personal Device) aktiviert werden. Es kann dort den Zustand des Devices (beschrieben durch die Service-Properties) beobachten und abhängig von diesem handeln. Das State-Management wird bei jeder Propertyänderung befragt, ob die Bedingungen in den Properties erfüllt sind, und nach der Auswertung wird eine Änderung lokaler Properties durchgeführt (Kapitel 10.1.2).

**Personal Management:** Das Personal Management stellt die Funktionalität für das Personal Device bereit und kann in folgenden Fällen aufgerufen werden: Wenn ein Timer abläuft, ein Event eintrifft, wenn der Nutzer Präferenzgeräte aufruft, wenn Actionaufrufe fehlschlagen und wenn Propertyänderungen auftreten (Kapitel 10.1.3).

Alle Management-Komponenten sind mit dem *PolicyPool* verbunden. Da in diesem *PolicyPool* die Policies gespeichert werden, treten die Policies je nach ihrer Priorität und ihrem Zeitpunkt auf.

### 6.2.3. Die Geräte

Im Digital Home gibt es die folgenden Geräte: 2 Fernseher, PC, Handy/PDA, 2 Lichtschalter, 2 Foxboards (s. o. Architektur), Brandschutzzentrale (s.u. Spezielle Anwendung), 2 Feuermelder.

Jeder Bewohner im Digital Home besitzt ein Persönliches Gerät, also ein Handy oder einen PDA. Das *Personal Device* stellt die zentrale Steuerkomponente dar. Was macht das persönliche Gerät eigentlich? Es bietet eine einheitliche Schnittstelle für Meldungen an den Benutzer und überwacht den aktuellen Aufenthaltsort des Benutzers durch Interaktion mit dem Foxboard. Es verwaltet die Interaktionsfolgen mit anderen DPWS-Geräten und führt transparent einzelne Aufrufe von entfernten Services aus (siehe Kapitel 8.2.1). Zu den Services des persönlichen Geräts gehören die Kategorien *DialogDevice* und *TextOutputDevice* (siehe Kapitel 8.2.2). In der Swing-GUI des Personal Devices hat der Benutzer Zugriff auf die folgenden Elemente: Die Uhrzeit, den Aufenthaltsort, die bekannten Geräte (die momentan erreichbar sind), das Display (für die Meldungen an den Benutzer über TextOutpuService und DialogService), die Log-Meldungen (alle erzeugten Meldungen in einer Liste), das Aktionen Menü (*RemoteTasks* ausführen) und das Policies Menü (verwalten/exportieren/importieren) (Kapitel 8.2.3).

Die Fernseher in Wohnzimmer und Schlafzimmer sind in folgenden Kategorien: *VideoInputDevice*, *VideoOutpuDevice*, *AudioInputDevice*, *AudioOutputDevice*, *RecorderDevice*, *TextOutputDevice* (siehe Kapitel 8.5). Die Grundfunktion des Fernsehers ist das Betrachten eigener Videos und des simulierten TV-Programms. Durch eine grafische Benutzeroberfläche können Videos und Audiodateien gestartet werden. Des Weiteren kann man die Helligkeit und den Kontrast mit Schiebereglern ändern. Diese Funktionen werden ohne die Verwendung von Webservices direkt am Gerät vollzogen. Sobald eine Kommunikation mit anderen Geräten stattfindet, wie zum Beispiel mit dem PC, kann man durch Webservices auf dem PC mit VideoOutputDevices ein Video empfangen oder durch VideoInputDevices freigeben. Genauso läuft es mit AudioOutput/InputDevices auch ab. Der Fernseher kann auch in einem Fenster Texte anzeigen, wie z. B. im Fall eines Brandes als Warnung. Unter der Kategorie RecorderDevice verfügt der Fernseher über einen Festplattenrekorder zur zeitgesteuerten simulierten Aufnahme des simulierten TV-Programms. Durch einen Timer wird die Zeit bestimmt, zu der ein simulier-

tes TV-Programm aufgenommen wird. Dabei wird das Video als Datei in einem Verzeichnis gespeichert.

Der PC hat die Kategorien: *VideoInputDevice*, *VideoOutpuDevice*, *AudioInputDevice*, *AudioOutputDevice*, *RecorderDevice*, *TextOutputDevice*, *DialogDevice*. Der PC hat die gleichen Funktionen wie der Fernseher, zusätzlich kann er Textnachrichten verschicken sowie Dialoge über ein Menü initialisieren (siehe Kapitel 8.4).

Der Lichschalter gehört zur Kategorie *LightDevice* und hat die einfache Funktion, das Licht in den Räumen ein- und auszuschalten (siehe Kapitel 8.3).

### 6.2.4. Die spezielle Anwendungsszenario

Die Brandschutzzentrale in der Kategorie *FireCenterDevice* verwaltet die Feuermelder in den einzelnen Räumen. Sie abonniert das Event der Feuermelder und bekommt eine ID von ihnen zurück. Im Falle eines Brandes bekommt die Zentrale eine Nachricht, von welchem Feuermelder dieses Signal kommt und durch Policies wird dieser Alarm an Geräte der Kategorie *TextOutputService* weitergeleitet, um die Bewohner zu warnen.

Die Feuermelder (Kategorie *FireDetectorDevice*) messen die Temperatur in den Räumen und in einem Brandfall feuern sie ein Event an die Zentrale. Nach dem Brandfall werden die Feuermelder in den Startzustand zurückgesetzt (siehe Kapitel 8.6).

# 7. Gerätekategorien

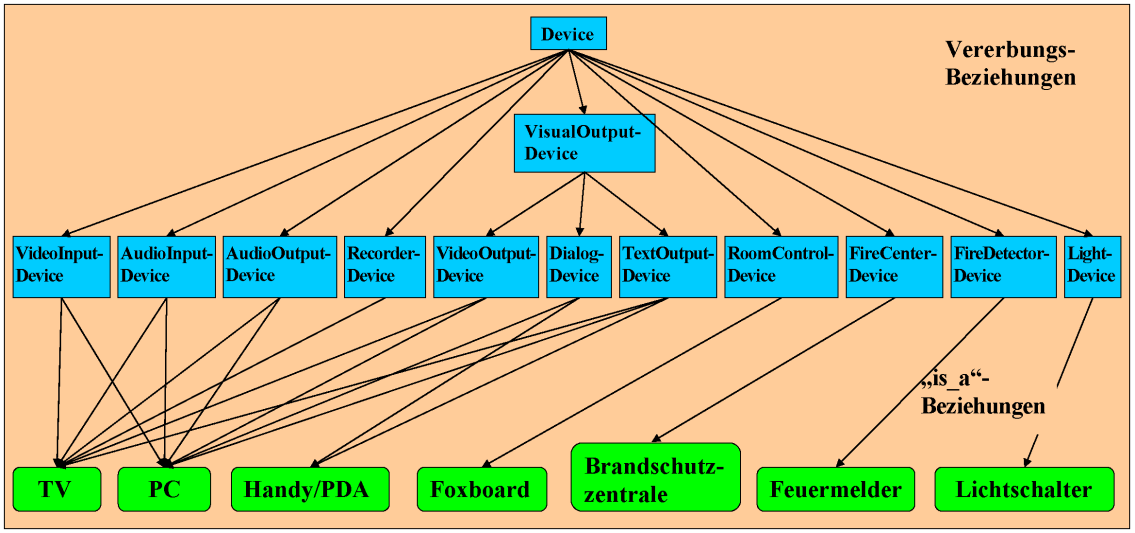


**Abbildung 14:** Spezifikation der Gerätekategorien

Geräte werden in Kategorien eingeteilt, um eine einheitliche Struktur für gleichartige Geräte zu haben. Befindet sich ein Gerät in einer Kategorie, so ist automatisch klar, dass eine bestimmte Menge von Diensten auf jeden Fall unterstützt werden. Geräte können sich dabei in mehreren Kategorien befinden.

Vererbung bedeutet bei Kategorien, dass die erbende Kategorie auch alle Properties und Dienste der vererbenden Kategorie besitzt. Wird in einer Kategorie eine Property oder ein Dienst spezifiziert, obwohl diese eine Property oder einen Dienst gleichen namens geerbt hat, so wird das Geerbte überschrieben. Namenskonflikte können ebenfalls bei Mehrfachvererbung oder bei der Einteilung eines Gerätes in mehrere Kategorien auftreten. Daher muss bei der Spezifikation darauf geachtet werden, dass nur dann Gleichnamiges von mehreren Kategorien an eine Kategorie vererbt wird bzw. in ein Gerät eingeht, wenn diese Kategorien diese Property bzw. diesen Dienst von der gleichen Oberkategorie geerbt haben.

Diese Spezifikation umfasst alle Kategorien von Geräten. Sie beinhaltet den Kategorienamen, eine Beschreibung der Kategorie, Vererbungsbeziehungen (direkte Oberkategorien), Properties und Actions.

## 7.1. Device

Device ist die Oberkategorie aller Geräte. Jede andere Kategorie stammt (direkt oder indirekt) von ihr ab. Daher besitzt ihre abstrakte Klasse schon den Großteil der Eigenschaften, die die eigentlichen Device-Services später benötigen. So stellt sie beispielsweise die grundliegende Funktionalität für das Property-Framework bereit.

Das Property-Framework fügt einem Webservice Eigenschaften hinzu, die als *Properties* bezeichnet werden. Eine Property kann mit einer Variable verglichen werden und hat einen Namen, einen Datentyp und einen aktuellen Wert. Die von uns definierten Datentypen für Properties sind in Tabelle 3 zu sehen. Mit Hilfe des Property-Frameworks kann jede Kategorie Properties definieren. Die Properties können vom Service selbst und von entfernten Services

gelesen und verändert werden. Zu diesem Zweck stellt die abstrake Klasse der *Device*-Kategorie die Actions `getProperty` und `setProperty` zur Verfügung, mit denen einzelne Properties gelesen und geschrieben werden können. Gleichzeitig besteht die Möglichkeit, gleich mehrere Properties zu lesen oder zu schreiben, indem die Actions `getProperties` und `setProperties` benutzt werden. Möchte jemand über Änderungen an Properties eines entfernten Services informiert werden, so kann er bei diesem Service das Event `onPropertyChange` abonnieren.

| Typ | Java Typ | Beschreibung |
|---|---|---|
| `String` | `java.lang.String` | Eine Zeichenkette |
| `Integer` | `java.lang.Integer` | Eine Ganzzahl |
| `Boolean` | `java.lang.Boolean` | Eine boolesche Variable |
| `StringList` | `java.util.Vector` | Eine Liste von Zeichenketten |

**Tabelle 3:** Property Datentypen

Neben dem Property-Framework selbst definiert die Kategorie *Device* auch zwei grundliegende Properties. Die Property `LocationID` beschreibt den Raum, in dem sich ein Gerät befindet, und die Property `Status` definiert den operationalen Status des Service. Als Status werden die vier Konstanten `IDLE`, `RUNNING`, `STOPPED` und `ERROR` verwendet.

Die Anbindung an das Management ist an zwei Stellen im Device vorgenommen. Alle erbenden Kategorien besitzen diese dadurch auch. Jedesmal, wenn sich eine Property ändert, wird das State-Management benachrichtigt. Werden auf dem Device Actions aufgerufen, so wird zuerst das Access-Management befragt. Details hierzu finden sich in Kapitel (10).

## 7.2. VisualOutputDevice

Die *VisualOutputDevice*-Kategorie umfasst alle visuellen Ausgabegeräte, worunter die Kategorien *DialogDevice*, *TextOutputDevice* und *VideoOutputDevice* fallen. Diese Kategorie setzt ein paar Properties: Helligkeit und Kontrast, jeweils ein aktueller Wert und ein maximaler- und minimaler Wert, und eine Property für die Breite und Höhe des Ausgabebildschirms wird hier vorgegeben. In dieser Kategorie werden keine zusätzlichen Actions bereitgestellt.

Die Spezifikation der Kategorie *VisualOutputDevice* wird im Anhang (siehe A.2) dokumentiert.

### 7.2.1. DialogDevice

Die *DialogDevice*-Kategorie umfasst alle Geräte, die eine dialogartige Kommunikation mit dem Benutzer ermöglichen. Bei *DialogDevice*-Geräten ist es möglich einen Dialog auf einem anderen Gerät anzeigen zu lassen und eine Antwort zu erhalten. Vorgegeben sind die Dialogtypen „Ja, Nein, Abbrechen“, „Ja, Nein“ und „OK“. Zusätzlich kann bei Aufruf der Dialog-Action ein Text für den Dialog übergeben werden und ein Timeout, nach dessen Ablauf keine Antwort des empfangenden Geräts mehr möglich sein soll. Wenn der Timeout-Wert null ist, soll dies als unendlich interpretiert werden. Die Art und Weise, wie ein Dialog angezeigt wird, ist der Implementation des jeweiligen Geräts überlassen.

Die Spezifikation der Kategorie *DialogDevice* wird im Anhang (siehe A.2.1) dokumentiert.

### 7.2.2. VideoOutputDevice

Die *VideoOutputDevice*-Kategorie umfasst alle Geräte, die die Ausgabe von Videos unterstützen. Es gibt zwei Actions: `playVideostream` und `stopVideostream`. Letztere kommt völlig ohne Parameter aus und stoppt lediglich die Wiedergabe eines Videostreams.

Die Action `playVideostream` hingegen verfügt über einen Input-Parameter, der mit der URL zu einer Videoquelle bestückt sein sollte. Bei Aufruf dieser Action wird zunächst versucht, eine eventuell schon bestehende Videowiedergabe zu stoppen. Wenn dies erfolgreich ist oder nicht nötig, wird das Gerät, das diese Kategorie implementiert hat, aufgefordert, das Video, das sich hinter dem Parameter verbirgt, in irgendeiner Form wiederzugeben. Ob das Video dabei gestreamt wird oder zuerst kopiert und dann wiedergegeben ist abhängig vom Typ der Videoquelle und der Art, wie die Videowiedergabe im konkreten Gerät abläuft. Als zusätzliche Properties stehen die aktuelle, maximale und minimale Lautstärke der Videowiedergabe zur Verfügung.

Die Spezifikation der Kategorie *VideoOutputDevice* wird im Anhang (siehe A.2.2) dokumentiert.

### 7.2.3. TextOutputDevice

Die *TextOutputDevice*-Kategorie umfasst alle Ausgabegeräte, die Text anzeigen können. Durch Aufruf der Action `showText` wird ein Text angezeigt. Per Parameter können dazu der anzuzeigende Text und ein Timeout, nach dem der Text nicht mehr angezeigt werden soll, übergeben werden. Ein Timeout von null soll dabei als unendlich verstanden werden. Zusätzlich stehen noch Properties bereit, mit denen sich die aktuelle, maximale und minimale Textgröße, die Schriftart, Hintergrundfarbe und die Textfarbe bestimmen lassen.

Die Spezifikation der Kategorie *TextOutputDevice* wird im Anhang (siehe A.2.3) dokumentiert.

## 7.3. AudioInputDevice

Befindet sich ein Gerät in der *AudioInputDevice*-Kategorie, so stellt es eine Audioquelle bereit. Drei Actions werden von dieser Kategorie angeboten: Das Senden einer Audiodatei, das bloße Bereitstellen einer Audioquelle und das Anzeigen von verfügbaren Audioquellen.

Die Action `createAudiostreamURL` erwartet eine Zahl als Input-Parameter. Diese Zahl bezeichnet die alphabetische Position einer Audioquelle des implementierenden Geräts. Zu dieser Audioquelle wird eine URL mittels des kategorieeigenen HTTP-Servers bereitgestellt und im Output-Parameter an den Aufrufer übergeben. Dieser kann dann selbsttätig auf die Audioquelle zugreifen. Es ist dabei immer nur maximal eine URL aktiv. Wenn `createAudiostreamURL` ein zweites Mal aufgerufen wird, wird die erste URL gelöscht.

Bei Aufruf der Action `sendAudiostream` ist der Ablauf zunächst identisch mit `createAudiostreamURL`. Es wird allerdings neben dem Input-Parameter, der eine Audioquelle bezeichnet, noch die UUID eines Geräts der Kategorie *AudioOutputDevice* verlangt. Dies kann auch die eigene UUID des Audio-Input-Geräts sein. Diese Action ruft dann von alleine nach dem Bereitstellen der URL die Action `playAudiostream` des *AudioOutputDevices* auf und gibt keinen Output-Parameter an den Aufrufer zurück.

Als Letztes steht noch die Action `listAudiostreams` bereit, nach dessen Aufruf ein Output-Parameter zurückgegeben wird, der eine Liste von aufrufbaren Audioquellen des Geräts beinhaltet.

Die Spezifikation der Kategorie *AudioInputDevice* wird im Anhang (siehe A.3) dokumentiert.

## 7.4. AudioOutputDevice

Die *AudioOutputDevice*-Kategorie umfasst alle Geräte, die die Wiedergabe von Audiodateien oder -streams unterstützen. Diese Kategorie verhält sich äquivalent zur Kategorie *VideoOutputDevice* (siehe Kapitel 7.2.2). Die zwei Actions heißen hier nur entsprechend `playAudiostream` und `stopAudiostream`. Ebenfalls stehen in dieser Kategorie die zusätzlichen Properties für aktuelle, maximale und minimale Lautstärke der Audiowiedergabe zur Verfügung.

Die Spezifikation der Kategorie *AudioOutputDevice* wird im Anhang (siehe A.4) dokumentiert.

## 7.5. VideoInputDevice

Befindet sich ein Gerät in der *VideoInputDevice*-Kategorie, so stellt es eine Videoquelle bereit. Die drei Actions dieser Kategorie verhalten sich äquivalent zur Kategorie *AudioInputDevice* (siehe Kapitel 7.3). Die Actions sind lediglich entsprechend der Funktion einer *VideoInput*-Kategorie mit `createVideostreamURL`, `sendVideostream` und `listVideostreams` benannt.

Die Spezifikation der Kategorie *VideoInputDevice* wird im Anhang (siehe A.5) dokumentiert.

## 7.6. RecorderDevice

Geräte der *RecorderDevice*-Kategorie stellen Funktionen für Timer-Programmierungen zur zeitgesteuerten Aufnahme bereit. In dieser Device-Kategorie kann mit der Aktion `setTimer` eine Timer-Programmierung gesetzt werden, mit `deleteTimer` eine Timer-Programmierung gelöscht werden und mit `getTimers` wird eine Tabelle mit Timer-Programmierungen zurückgegeben. Funktionen zum tatsächlichen Aufzeichnen von Streams, Programmen oder Dateien müssen in der konkreten Implementierung eines Geräts realisiert werden, sie werden nicht von dieser Geräteklasse bereitgestellt.

`setTimer` verfügt über fünf Input-Parameter:

- `startTime`: Über diesen Parameter wird die Startzeit einer Timer-Programmierung gesetzt. Es wird ein String im Format *hh:mm* erwartet.
- `endTime`: Über diesen Parameter wird die Stoppzeit einer Timer-Programmierung gesetzt. Wenn die Stoppzeit vor der Startzeit liegt, soll dies in der Umsetzung in einem Gerät als Stoppzeit am folgenden Tag interpretiert werden. Es wird ebenfalls ein String im Format *hh:mm* erwartet.
- `date`: Mit diesem Parameter wird das Startdatum einer Timer-Programmierung gesetzt. Es wird ein String im Format *tt.mm.jjjj* erwartet.
- `program`: Über diesen Parameter wird das aufzuzeichnende Objekt einer Timer-Programmierung gesetzt. Es wird kein bestimmtes Format oder Verfügbarkeit eines damit bezeichneten Streams oder Datei erwartet, da dies hochgradig von der konkreten Implementierung in einem Gerät abhängig ist.
- `name`: Mit diesem Parameter wird der Bezeichner einer Timer-Programmierung gesetzt. Es wird kein bestimmtes Format erwartet, allerdings wird überprüft, ob der Name schon vergeben ist.

Die Aktion `getTimers` gibt die vorgenannten Parameter, die Bestandteil einer Timer-Programmierung sind, im Output-Parameter in Form einer Tabelle zurück.

`deleteTimer` benötigt im Unterschied zu `setTimer` nur den Input-Parameter `name`, anhand dessen eine zu löschende Timer-Programmierung identifiziert wird. Wenn mehrere Timer-Programmierungen den gleichen Bezeichner haben, wobei dies eigentlich nicht möglich sein sollte, werden alle Timer-Programmierungen dieses Namens gelöscht.

Die Spezifikation der Kategorie *RecorderDevice* wird im Anhang (siehe A.6) dokumentiert.

## 7.7. LightDevice

*LightDevice* ist die Kategorie, die einen einfachen Lichtschalter repräsentiert. In dieser Kategorie gibt es keine neuen Actions, eine Aktion ist nur mittels der Properties möglich. Als neue Properties kommen hier hinzu die aktuelle, maximale und minimale Helligkeit der Umgebung des Lichtschalters und der Status des Schalters. Über das Ändern letzterer Property kann ein Lichtschalter ein- uns ausgeschaltet werden.

Die Spezifikation der Kategorie *LightDevice* wird im Anhang (siehe A.7) dokumentiert.

## 7.8. FireCenterDevice

In dieser Kategorie wird durch die Brandschutzzentrale *FireCenterDevice* beschrieben. Es gibt eine Property `Fire` und zwei Actions `searchSensors`, `AlarmOff`.

In der Property `Fire` wird die aktuelle Liste von Orten, wo das Feuer ausgebrochen ist, abgespeichert. Mit der Action `searchSensors` werden alle Feuermelder gesucht und bei den gefundenen Feuermeldern werden die Actions `sendAlarmAction`, `AlarmOff`(siehe unten FireDetectorDevice) abonniert.

Durch die Action `AlarmOff` wird die bei Feuermelder abonnierte Action `AlarmOff` aufgerufen. Die Brandschutzzentrale bekommt als Antwort den aktuellen Ort, wo das Feuer gelöscht wurde. Die Property `Fire` wird dann entsprechend aktualisiert.

## 7.9. FireDetectorDevice

In dieser Kategorie werden alle Fueurmelder umfasst. Es gibt vier Actions: `startState`, `fireUp`, `sendAlarmAction` und `AlarmOff`.

Die Action `startState` bringt Feuermelder in den initialen Zustand.

Mit der Action `fireUp` kann man einen Feuerausbruch simulieren.

Die Action `sendAlarmAction` ist eine event-basierte Action. Die Action wird von dem *FireCenterDevice* abonniert, so wie die Action `AlarmOff`. Wenn ein Feuerausbruch erkannt wurde, wird durch die Action `sendAlarmAction` eine Alarmmeldung an die *FireCenter* geschickt. Dabei wird der Ausgabeparameter `sendAlarm` mit dem aktuellen Ort mitgeschickt. Bei Brandschtuzzentrale wird die Property `Fire` entsprechend aktualisiert.

Wenn die Temperatur unter 50 Grad Celsius liegt, wird der Alarm durch die Action `AlarmOff` automatisch abgeschaltet, in dem eine Nachricht vom Feuermelder an die Feuerzentrale mit dem aktuellen Ort geschickt wird.

## 7.10. RoomControlDevice

Der Kategorie der *RoomControlDevices* gehören nur die Foxboards an. Sie haben die Aufgabe, den Raum zu überwachen und das Eintreten und Verlassen von Personen zu registrieren. Eine

*RoomControlDevice* ist dabei immer für einen einzelnen Raum zuständig. Das Verfahren zur Personenerkennung wird in Kapitel 9 näher beschrieben.

Damit auch andere Geräte über die Raumwechsel der Personen informiert werden, bietet ein *RoomControlDevice* zwei Events an. Das Event *deviceEnters* wird immer dann versendet, wenn eine Person einen Raum betritt und das Event *deviceLeaves*, wenn eine Person den Raum verlässt. Die Geräte, die diese Informationen benötigen, können die Events abonnieren und werden dann immer über die aktuelle Position der Bewohner des Digial Home informiert.

Neben den Events bietet das *RooomControlDevice* noch eine einzelne Property an. Die Property *DevicesPresent* ist vom Typ *StringList* und enthält eine Liste von UUIDs der Personal Devices aller im Raum anwesenden Personen.

Die genaue Spezifikation der Kategorie *RoomControlDevice* wird im Anhang (siehe A.8) dokumentiert.

# 8. Geräte

Um die Entwicklung der Geräte des *Digital Home* zu vereinheitlichen, wurde eine gemeinsame Architektur geschaffen, auf der alle Geräte aufbauen. Zentraler Bestandteil jedes Gerätes ist die Klasse `DeviceHostingService`. Sie erweitert die Klasse `HostingService` des DPWS Stacks um die Basisfunktionalität der Geräte des Digial Home.

Jedes Gerät bietet ein oder mehrere Services an. Die Basisklasse eines Services ist der `AbstractDeviceService`, welcher von `HostedService` erbt.

Die Geräte haben die Möglichkeit auf einige Komponenten zuzugreifen, die die Basisklassen `DeviceHostingService` und `AbstractDeviceService` zur Verfügung stellen. Diese Komponenten werden im Folgenden näher beschrieben.

## 8.1. Gemeinsame Komponenten

### 8.1.1. Der Logger

Um die Entwicklung und auch die Fehlersuche im Prototypen des *Digital Home* zu vereinfachen, wurde eine zentrale Komponente benötigt, mit welcher die verschiedenen Programmteile Meldungen unterschiedlicher Priorität erzeugen konnten. Es bestand also der Bedarf nach einer einfachen Logging Komponente. Die Klasse `Logger` wurde in Anlehnung an bekannte Logging Frameworks für Java SE, wie beispielsweise Log4J[2], entwickelt und bietet folgende Kernfunktionen:

- Einfache Instanziierung über eine statische Factory Methode
- Erzeugung von Meldungen verschiedener Priorität (DEBUG, INFO, ERROR)
- Möglichkeit zur Deaktivierung von Meldungen bestimmer Priorität oder aus bestimmten Packages oder Klassen

Obwohl die Möglichkeiten des Logging sehr eingeschränkt sind, ermöglicht uns dieses einfache zentrale Logging eine effizientere Fehlersuche. Abbildung 9 zeigt die grobe Struktur dieser Klasse.

```
public class Logger {

  public static final int VERBOSIY_DEBUG =    0;
  public static final int VERBOSIY_INFO =     1;
  public static final int VERBOSIY_ERROR =    2;
  public static final int VERBOSIY_NONE =     3;

  public static Logger getLog(Class klasse);

  public static void setVerbosity(int verbosity);

  public static void addToBlacklist(String name);

  public void debug(String msg);
```

[2] http://logging.apache.org/log4j/docs/

```
  public void info(String msg);

  public void error(String msg);

}
```

**Quellcode 9:** Die `Logger` Klasse

Die Factory Methode `getLog()` erwartet eine Instanz der Klasse `Class` als Argument. Klassen, die eine Logger Instanz verwenden wollen, sollen hier eine Referenz auf ihr eigenes `Class` Objekt übergeben. Somit ist es dem Logger möglich, die Klasse eindeutig zu identifizieren. Nach dem eine Klasse eine `Logger` Instanz erhalten hat, können mit den Methoden `debug()`, `info()` und `error()` Meldungen abgesetzt werden.

### 8.1.2. Der ServiceLookupManager

Für das Projekt *Digital Home* war es von wesentlicher Bedeutung für alle beteiligten Geräte, schnell einen Überblick über die anderen vorhandenen Geräte zu erlangen. Der DPWS Stack bietet für diesen Zweck eine asynchrone Suche an. Man kann mit Hilfe der Klasse `SearchManager` einen WS-Discovery Probe versenden und wird dann mit Hilfe das Callback Interface `ISearchCallback` über die Antworten der einzelnen Services informiert.

Für uns schien diese Lösung jedoch nicht die benötigte Flexibilität zu bieten. So hätte man bei einer Suche stets eine gewisse Zeit (mindestens 5 Sekunden) abwarten und während dieser Zeit alle Suchantworten sammeln müssen, um die Gesamtzahl der Suchergebnisse zu erhalten.

Im *Digital Home* übernimmt der *ServiceLookupManager* die Aufgabe nach Geräten oder einzelnen Services zu suchen. Der *ServiceLookupManager* hat dabei die Aufgabe, das *Digital Home* zu überwachen und somit immer über alle existierenden Geräte und ihre Services Bescheid zu wissen. Wird ein Gerät oder eines, welches einen bestimmten Service anbietet, gesucht, so kann der *ServiceLookupManager* befragt werden. Der große Vorteil dieser Lösung ist, dass die Suche synchron ist und die Ergebnisse somit sofort zur Verfügung stehen.

Um möglichst einfach mit verschiedenen Verfahren experimentieren zu können, wurde ein allgemeines Interfaces definiert, welches von verschiedenen Klassen implementiert werden kann. Eine Realisierung des *ServiceLookupManager*s muss das Interface implementieren, welches in Abbildung 10 zu sehen ist.

```
public interface ServiceLookupManager {

  public RemoteService[] findRemoteServices(QualifiedNameVector
      serviceTypes, QualifiedNameVector deviceTypes);

  public RemoteService[] findRemoteServices(QualifiedNameVector
      serviceTypes, QualifiedNameVector deviceTypes, Vector scopes,
      String scopeMatchingRule);

  public RemoteService[] findRemoteServicesByLocation(
      QualifiedNameVector serviceTypes, QualifiedNameVector deviceTypes
      , String location);
```

```
  public RemoteService[] findRemoteServicesByServicePortType(String
      porttype);

  public RemoteService[] findRemoteServicesByDeviceUUID(String uuid);

  public RemoteService[] findRemoteServicesByDeviceUUID(String uuid,
      QualifiedNameVector serviceTypes);

  public String[] getKnownDeviceUUIDs();

  public HostingServiceMetadata getHostingServiceMetadataByDeviceUuid(
      String uuid);

  public void addManagedDeviceListener(ManagedDeviceListener listener);

}
```

**Quellcode 10:** Das Interface `ServiceLookupManager`

Das Interface bietet mit den Methoden `findRemoteService()` die Möglichkeit zur synchronen Suche nach Services. Dabei besteht die Möglichkeit verschiedenste Kriterien für die Suche zu verwenden. Die möglichen Suchkriterien sind:

- Die UUID des Gerätes, auf dem der Service läuft
- Der *PortType* des Services
- Der Typ des Gerätes (Der so genannte *DevicePortType*)
- Der Raum, in dem sich das Gerät befindet
- Angabe eines *Scope* und der entsprechenden *Matching Rule* (siehe [CCK+06])

Neben der Möglichkeit zur synchronen Suche nach Services können die Geräte den *ServiceLookupManager* noch auf eine andere Weise benutzen. Eine Klasse, welche das Interface `ManagedDeviceListener` implementiert, kann sich beim *ServiceLookupManager* als Listener registrieren. Das Interface `ManagedDeviceListener` ist in Abbildung 11 zu sehen.

```
public interface ManagedDeviceListener {

  public void onManagedDeviceFound(String uuid);

  public void onManagedDeviceUpdate(String uuid);

  public void onManagedDeviceRemoved(String uuid);
}
```

**Quellcode 11:** Das Interface `ManagedDeviceListener`

Ist eine Klasse als *ManagedDeviceListener* registriert, so wird sie über die Callback Methoden stets darüber informiert, wenn der *ServiceLookupManager* ein neues Gerät entdeckt, sich Änderungen an einem bekannten Gerät ergeben haben und wenn ein Gerät verschwindet.

Im Zuge der Arbeit am *Digital Home* hat sich eine Implementierung des *ServiceLookupManagers* als besonders effizient und zuverlässig erwiesen. Die implementierende Klasse mit dem Namen `ServiceLookupManagerProbeRadar` arbeitet auf der Basis von regelmäßig ausgesendeten Multicast WS-Discovery *Probe* Nachrichten. Die Klasse sendet regelmäßig einen *Probe*, bei dem sich jedes Gerät angesprochen fühlt und somit auf den *Probe* antwortet. Erhält die Klasse eine WS-Discovery *ProbeMatch* Nachricht von einem bis dato unbekannten Gerät, wird der eigentliche Discovery Vorgang eingeleitet. Mit Hilfe der physikalischen Netzwerkadresse aus dem *ProbeMatch* werden die Metadaten des Gerätes und die WSDL Dokumente seiner Services angefordert. Mit diesen Informationen ist das Gerät dem *ServiceLookupManager* vollständig bekannt.

Die regelmäßigen WS-Discovery *Probe* Nachrichten haben neben dem Auffinden neuer Geräte aber noch eine andere wichtige Aufgabe. Mit Hilfe dieser Probes kann der *ServiceLookupManager* feststellen, ob ein Gerät noch erreichbar ist. Antwortet ein Gerät ordnungsgemäß auf den regelmäßigen *Probe*, so ist davon auszugehen, dass das Gerät noch existiert und auch ansprechbar ist. Bleibt die Antwort eines bereits bekannten Geräts aber nach einigen Probes jedesmal aus, so kann der *ServiceLookupManager* von der Annahme ausgehen, dass das Gerät nicht mehr vorhanden ist und entfernt es aus seiner Liste von bekannten Geräten.

Ein Sonderfall tritt ein, wenn ein Gerät abstürzt und unmittelbar danach neu gestartet wird. Das Gerät behält dabei seine eindeutige UUID, jedoch kann es passieren, dass das Gerät eine neue physikalische Netzwerkadresse erhält. Deshalb überprüft der *ServiceLookupManager* bei jedem *ProbeMatch* eines bereits bekannten Gerätes, ob sich die physikalische Netzwerkadresse verändert hat. Ist die Adresse die selbe, so wird das *ProbeMatch* als Zeichen dafür interpretiert, dass das Gerät noch vorhanden ist. Hat sich die Adresse hingegen verändert, wird das Gerät aus der Liste der bekannten Geräte entfernt. Ab diesem Zeitpunkt wird das empfangene *ProbeMatch* so behandelt, als wäre das Gerät bis dato nicht bekannt gewesen. Somit werden alle Metadaten und WSDL Dokumente erneut abgefragt.

## 8.2. PersonalDevice

Das PersonalDevice stellt die zentrale Steuerkomponente im *Digital Home* dar. Jeder Bewohner des *Digital Home* besitzt eine solche Steuerkomponente. Als Geräte sind sowohl PDAs, als auch Mobiltelefone denkbar. Die Realisierung in *CLDC* stellt dabei sicher, dass die Steuersoftware auf kleinen mobilen Geräten lauffähig ist.

Die Kernimplementierung des PersonalDevice ist *CLDC 1.1* kompatibel. Sie enthält alle Komponenten, welche für den grundliegenden Betrieb des Personal Devices notwendig sind. Aufgrund der Einschränkungen der *CLDC Spezifikation* enthält die Kernimplementierung kein Benutzerinterface. Durch einen Aufbau auf der Kernimplementierung ist es jedoch möglich, verschiedene Benutzerschnittstellen zu integrieren.

Im Laufe der Projektgruppe sind dabei zwei verschiedene Benutzerinterfaces entstanden. Bei der Entwicklung der Kernimplementierung wurde auf eine JavaSE Swing GUI zurückgegriffen, welche eine möglichst einfache Bedienung des Personal Devices zur Entwicklungszeit ermöglichte (siehe Kapitel 8.2.3).

Für eine realistische Simulation des Personal Device auf kleinen Geräte wurde jedoch noch eine weitere Benutzerschnittstelle für die Kernimplementierung geschaffen. Dieses Benutzerin-

terface wurde mit Hilfe der *MIDP 2.0 Spezifikation* entwickelt. MIDP ist ein CLDC Profil für mobile Endgeräte wie Mobiltelefone und PDAs. Da heute jedes marktübliche Mobiltelefon mindestens CLDC 1.1 und MIDP 2.0 implementiert, ist das MIDP Benutzerinterface zusammen mit der Kernimplementierung auf heutigen Mobiltelefonen und PDAs lauffähig (siehe Kapitel 8.2.4).

Im Folgenden wird zunächst auf die Details der Kernimplementierung eingegangen und in den darauffolgenden Kapiteln die beiden Benutzerschnittstellen beschrieben.

### 8.2.1. Kernimplementierung

Genau wie bei allen anderen Geräten im *Digital Home* ist die Hauptklasse des PersonalDevice ebenfalls ein *DeviceHostingService* und somit ein Gerät im Sinne des DPWS Stacks. Diese Klasse instanziiert und verwaltet alle für den Betrieb des PersonalDevice notwendigen Programmkomponenten. Abbildung 4 zeigt eine Aufstellung aller wichtigen Komponenten.

| **Komponente** | **Aufgabe** |
|---|---|
| EventLog | Einheitliche Schnittstelle für Meldungen an den Benutzer |
| LocationMonitor | Überwacht den aktuellen Aufenthaltsort des Benutzers durch Interaktion mit den Foxboards |
| RemoteTaskManager | Klasse zur Verwaltung von Interaktionsfolgen mit anderen DPWS Geräten |
| ServiceInvoker | Führt transparent einzelne Aufrufe von entfernten Services aus |

**Tabelle 4:** Komponenten des PersonalDevice

Im Folgenden werden alle in dieser Tabelle genannten Komponenten näher betrachtet und ihre Funktionsweise erklärt.

#### 8.2.1.1. EventLog

Die Klasse `EventLog` bietet jeder Komponente des PersonalDevice die Möglichkeit, auf einfache Art Meldungen zu erzeugen. Sie bildet gleichzeitig eine Schnittstelle zwischen der Kernimplementierung und einer Benutzerschnittstelle. Der grobe Aufbau der Klasse ist in Quellcode 12 zu sehen.

```
public class EventLog {

  public void addDebugEvent(String msg);

  public void addInfoEvent(String msg);

  public void addWarnEvent(String msg);

  public void addErrorEvent(String msg);

  public void addEventLogChangeListener(EventLogChangeListener listener
      );
```

```
  public Vector getAllEventLogEntries();

  public void deleteAll();

}
```

**Quellcode 12:** Die Klasse `EventLog`

Damit eine Programmkomponente an eine Instanz der Klasse `EventLog` kommt, muss sie die statische Methode `getInstance()` der Factory Klasse `EventLogFactory` aufrufen. Zur Erzeugung von Meldungen bietet die Klasse `EventLog` die Methoden `addDebugEvent()`, `addInfoEvent()`, `addWarnEvent()` und `addErrorEvent()` an, welche Nachrichten der verschiedenen Prioritäten erzeugen.

Wie auch die Klasse `ServiceLookupManager` realisiert auch `EventLog` das Observer Entwurfsmuster. Will eine Klasse über Log-Meldungen informiert werden, muss sie das Interface `EventLogChangeListener` implementieren und sich mit der Methode `addEventLogChangeListener()` registrieren. Das Interface definiert zwei Methoden. Die Methode `newEventAdded()` wird immer dann aufgerufen, wenn eine neue Log Meldung erzeugt wird und die Methode `eventLogCleaned()`, falls das EventLog geleert wird.

#### 8.2.1.2. LocationMonitor

Wie in Kapitel 6 beschrieben wurde, besitzt jeder Bewohner des Digial Home ein Personal Device, welches seine Steuerkomponente für die anderen Geräte darstellt. Neben dieser Funktion hat das Personal Device jedoch noch die Aufgabe, den Benutzer lokalisierbar zu machen. Befindet sich ein Personal Device in einem bestimmten Raum, so wird davon ausgegangen, dass sich der Besitzer ebenfalls in diesem Raum aufhält.

Der *LocationMonitor* hat die Aufgabe, immer über den aktuellen Aufenthaltsort des Personal Device informiert zu sein und stets alle abhängigen Komponenten über Änderungen zu informieren. Die eigentliche Lokalisation über die Bluetooth-Verbindung führt das Personal Device allerdings nicht selbst aus. Statt dessen interagiert das Personal Device mit den *RoomControlServices*, die durch die Foxboards bereitgestellt werden. Jeder Raum des *Digital Home* wird durch mindestens ein Foxboard abgedeckt. Die Foxboards überwachen die einzelnen Räume und können somit feststellen, welche Person sich in welchem Raum aufhält. Sobald ein Personal Device den Raum betritt oder verlässt, wird dies über Events allen interessierten Geräten mitgeteilt.

Der *LocationMonitor* überprüft in regelmäßigen Abständen, welche *RoomControlServices* dem *ServiceLookupManager* bekannt sind. Findet er dabei einen *RoomControlService*, der ihm bis dahin nicht bekannt war, abonniert er die Events `DeviceEnters` und `DeviceLeaves` bei diesem Foxboard. Da die Subscription eines Events immer zeitlich begrenzt ist, ist es dabei notwendig auslaufende Subscriptions regelmäßig zu erneuern.

Betritt nun der Besitzer des Personal Devices einen Raum, so wird dies durch das Foxboard erkannt. Falls der *RoomControlService* dieses Foxboards dem Personal Device bekannt ist und das Personal Device die nötigen Events abonniert hat, so erhält es ein Event, welches über das Eintreten in den Raum informiert. Ein Beispiel für eine entsprechende SOAP Nachricht ist in Quellcode 13 aufgezeigt.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
```

```
<s12:Envelope xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
  <s12:Header>
    <wsa:MessageID>uuid:133c7d20-b05d-11db-8036-350ad2035f11</
        wsa:MessageID>
    <wsa:Action>http://digitalhome.pg490.udo.edu/
        portTypeRoomControlDevice/deviceEnters</wsa:Action>
    <wsa:To>http://127.0.0.1:47523/LocationMonitor/OndeviceEnters-1</
        wsa:To>
  </s12:Header>
  <s12:Body>
    <n0:device xmlns:n0="http://digitalhome.pg490.udo.edu">
        uuid:66666666-6666-6666-6666-666666666666</n0:device>
    <n1:location xmlns:n1="http://digitalhome.pg490.udo.edu">Wohnzimmer
        </n1:location>
  </s12:Body>
</s12:Envelope>
```

**Quellcode 13:** Ein `DeviceEnters` Event

Durch dieses Event ist der *LocationMonitor* darüber informiert worden, dass ein Personal Device mit der angegebenen UUID in einen Raum eingetreten ist. Falls die UUID aus dem Event mit der UUID des eigenen Gerätes übereinstimmt, hat sich der aktuelle Aufenthaltsort des Personal Devices verändert. Da der neue Aufenthaltsort für weitere Komponenten innerhalb des Personal Devices von Interesse sein kann, müssen diese vom *LocationMonitor* informiert werden. Dazu können die Klassen das Interface `LocationListener` implementieren, welches in Abbildung 14 gezeigt ist.

```
public interface LocationListener {

  public void onNewLocation(String oldLocation, String newLocation);

}
```

**Quellcode 14:** Das `LocationListener` Interface

Implementiert eine Klasse dieses Interface und registriert sich als Listener, so wird die Methode `onNewLocation()` bei jeder Änderung am aktuellen Aufenthaltsort aufgerufen. Die Argumente `oldLocation` bzw. `newLocation` enthalten dabei stets die Bezeichnungen der jeweiligen Räume.

#### 8.2.1.3. RemoteTaskManager

Der *RemoteTaskManager* hat die Aufgabe eine Menge von *RemoteTasks* zu überwachen. Ein RemoteTask ist eine Interaktionsfolge mit einer gewissen Anzahl von Web Services des *Digital Home*. Eine Interaktionsfolge wäre zum Beispiel, ein Gerät zu suchen, dass einen bestimmten Videostream anbietet und diesen Stream auf einem anderen Gerät abzuspielen.

Ein RemoteTask ist dem *Command* Entwurfsmuster nachempfunden, welches besagt, dass Aktionen in einer Klasse gekapselt werden sollen. Jede Aufgabe wird dabei als eigener RemoteTask realisiert. Solche RemoteTask Objekte können nach Belieben instanziiert und dem

*RemoteTaskManager* übergeben werden, der diese Tasks ausführt, über eine gewisse Zeit hinweg überwacht und dann beendet.

```
public abstract class AbstractRemoteTask {

  public AbstractRemoteTask(long taskTimeout);

  public abstract void startTask();

  public abstract String getDescription();

  public abstract void onServiceFound(String probeId, RemoteService
      remoteService);

  public abstract boolean equals(Object obj);

  protected void startSearchService(String[] portTypes);

  protected void finished();

}
```

**Quellcode 15:** Die Klasse `AbstractRemoteTask`

Konkrete RemoteTasks erben von der abstrakten Klasse `AbstractRemoteTask`, welche in Quellcode 15 zu sehen ist. Diese Klasse schreibt vier Methoden vor, die konkrete RemoteTasks implementieren müssen. Die Methode `startTask()` wird einmal zu Begin der Ausführung des RemoteTasks ausgeführt. Die Methode `getDescription()` muss eine einfache kurze textuelle Beschreibung der Aufgabe eines Tasks liefern. Diese Methode wird für Log Meldungen verwendet. Die Methode `equals()` muss den RemoteTask mit einem weiteren Task vergleichen. Dies wird benötigt, um feststellen zu können, ob bereits ein RemoteTask mit der gleichen Aufgabe ausgeführt wird. Die Methode `onServiceFound()` wird immer dann aufgerufen, wenn der Task über eine erfolgreiche Suche nach einem Service informiert wird. Dies gilt allerdings nur, wenn der Task direkt über den DPWS Stack eine Suche ausführt. Wird der *ServiceLookupManager* verwendet, wird das Ergebnis der Suche direkt geliefert und die Callback Methode nicht verwendet.

Hat ein RemoteTask seine Aufgabe erfüllt, ruft er die Methode `finished()` auf, welche von `AbstractRemoteTask` bereitgestellt wird. Diese markiert den RemoteTask als beendet, so dass der RemoteTaskManager ihn automatisch aus der Task Liste entfernen kann. Damit Tasks nicht für immer in der Queue verbleiben können, ermöglicht es die Klasse `AbstractRemoteTask` jedem Task einen Timeout zu zuordnen. RemoteTasks, welche den Timeout überschreiten werden automatisch abgebrochen. Das kann beispielsweise in Situationen passieren, in denen die asynchrone Suche des DPWS Stacks verwendet wird, jedoch nie ein passender Service gefunden wird und somit auch niemals die Callback Methode `onServiceFound()` aufgerufen wird. Das PersonalDevice ruft in regelmäßigen Intervallen die Methode `removeFinishedAndTimedOutTasks()` des RemoteTaskManagers auf, welche erfolgreich beendete RemoteTasks und solche, die den Timeout überschritten haben, aus der Queue entfernt.

Möchte eine Komponente des PersonalDevice nun einen RemoteTask starten, so muss diese Komponente zunächst eine von `AbstractRemoteTask` erbende Klasse instanziieren. Dieses Objekt wird entweder durch Argumente im Konstruktor oder durch Set-Methoden genauer konfiguriert. Dann holt sich die Komponente eine Referenz auf den *RemoteTaskManager* und fügt den Task mit Hilfe der Methode `addNewTask()` der Task Liste hinzu. Der Task wird somit in die interne Queue des RemoteTaskManagers aufgenommen und ausgeführt.

#### 8.2.1.4. ServiceInvoker

Ein häufiger Anwendungsfall im *Digital Home* ist es, einen entfernten Service aufzurufen. In einigen Situationen schien es jedoch Sinn zu ergeben, den Aufruf eines Services zu verallgemeinern und von dem konkreten Service zu abstrahieren.

Um dieses Ziel zu erreichen wurde das Konzept des *ServiceInvoker*s vorgeschlagen. Der *ServiceInvoker* kapselt den eigentlichen Aufruf des Services. Mit seiner Hilfe ist es möglich, einen Webservice Aufruf durchzuführen, ohne jedoch den konkreten Zielservice mit angeben zu müssen. Stattdessen spezifiziert man Eigenschaften, die der Zielservice erfüllen muss. Aber das Konzept des *ServiceInvoker*s hat noch einen weiteren wichtigen Vorteil. Der *ServiceInvoker* kann die Fehlerbehandung eines Webservice Aufrufs kapseln und kann somit transparent nach alternativen Services suchen.

Der *ServiceInvoker* ist genau wie der *ServiceLookupManager* als Interface realisiert, um mit verschiedenen Implementierungen experimentieren zu können (siehe Kapitel 8.1.2). Abbildung 16 zeigt das von uns definierte Interface.

```
public interface ServiceInvoker {

  public Action invokeRemoteServiceFromList(String actionName, String
      porttype, ActionPreparer preparer, Vector remoteServiceList);

  public Action invokePreferedRemoteService(String actionName, String
      porttype, ActionPreparer preparer);

  public Action invokePreferedRemoteServiceInRoom(String actionName,
      String porttype, ActionPreparer preparer, String location);

  public void invokeSingleRemoteService(RemoteService remoteService,
      Action action) throws SOAPException;

}
```

**Quellcode 16:** Das `ServiceInvoker` Interface

Das Interface bietet vier Möglichkeiten einen entfernten Service aufzurufen:

- `invokeRemoteServiceFromList()`
  Dieser Methode wird eine Liste von `RemoteService` Objekten übergeben. Die Implementierung des Interfaces versucht nacheinander die Services aufzurufen, bis ein Aufruf auch tatsächlich gelingt. Das `Action` Objekt des gelungenen Aufrufs wird dabei als Rückgabewert geliefert.

- `invokePreferedRemoteService()`
  In diesem Fall werden die für den Aufruf in Frage kommenden Services bestimmt, in dem das *Personal Management* bezüglich der Präferenzen des Benutzers befragt wird. Die Aufrufe werden nacheinander entsprechend der Priorität der Präferenzen bei den entsprechenden Geräten durchgeführt, bis das erste Mal ein Erfolg zu verzeichnen ist.

- `invokePreferedRemoteServiceInRoom()`
  Diese Methode arbeitet fast genau wie `invokePreferedRemoteService()`. Die Auswahl bevorzugter Geräte wird jedoch auf Geräte eingeschränkt, die sich im angegebenen Raum befinden.

- `invokeSingleRemoteService()`
  In diesem Fall wird der `RemoteService` aufgerufen, der über das Argument `RemoteService` spezifiziert wird.

### 8.2.2. Services

Wie jedes andere Gerät im *Digital Home* bietet auch das Personal Device einige Services an. Das Personal Device gehört dabei in die Gerätekategorien *TextOutputDevice* (siehe Kapitel 7.2.3) und *DialogDevice* (siehe Kapitel 7.2.1). Das grundsätzliche Problem besteht darin, dass die Realisierung der Services in die Kernimplementierung gehören, die Kernimplementierung jedoch, aufgrund der geforderten Kompatibilität zur *CLDC Spezifikation*, keinerlei Funktionen für eine Benutzerschnittstelle bereitstellt.

Wie auch bei der Implementierung anderer Kernkomponenten wurde deshab ein Interface definiert, welches als Schnittstelle zwischen der Kernimplementierung und einem Benutzerintface diente (siehe Quellcode 17).

```
public interface UserInterface {

  public void textOutputSetText(String text);

  public void textOutputClearAfterDelay(int seconds);

  public void dialogShow(String msg, int type, int timeout,
      DialogAnswerCallback callback);

}
```

**Quellcode 17:** Das `UserInterface` Interface

Dabei hat die Methode `textOutputSetText()` die Aufgabe, einen Text auf der Benutzerschnittstelle auszugeben. Falls der Text nur eine gewisse Zeit angezeigt werden soll, so wird dieser Timeout über die Methode `textOutputClearAfterDelay()` gesetzt. Die Methode `dialogShow()` ist für einfache Dialoge zuständig. Dabei kann für den Dialog ein Text, der genaue Dialogtyp und ein Timeout angegeben werden. Des Weiteren erwartet die Methode `dialogShow()` eine Referenz vom Typ `DialogAnswerCallback`. Über dieses Interface wird mitgeteilt, wie der Benutzer auf den Dialog reagiert hat.

Die Implementierungen der beiden Services greifen auf die Klasse `UserInterfaceProxy` zu, welche das Interface `UserInterface` implementiert. Der `UserInterfaceProxy` hat dabei die

Aufgabe, die Aufrufe der Service Implementierungen entgegenzunehmen und an die jeweilge Implementierung der Benutzerschnittstelle weiter zu leiten. Will eine Klasse der Benutzerschnittstelle dem *TextOutputService* als Ausgabe dienen, so muss sie das Interface `UserInterface` implementieren und sich bei der Klasse `UserInterfaceProxy` registrieren.

### 8.2.3. Swing GUI

Neben der Kernimplementierung wurde eine einfache Benutzerschnittstelle für das Personal Device benötigt. Bei der Benutzerschnittstelle entschieden wir uns für eine *JavaSE* Anwendung mit einem Java Swing Benutzerinterface. Eine solche Schnittstelle versprach bequeme und schnelle Bedienung des Personal Devices, was besonders bei der Entwicklung von Nutzen war.

Abbildung 15 zeigt das Hauptfenster der Swing Oberfläche des Personal Devices.

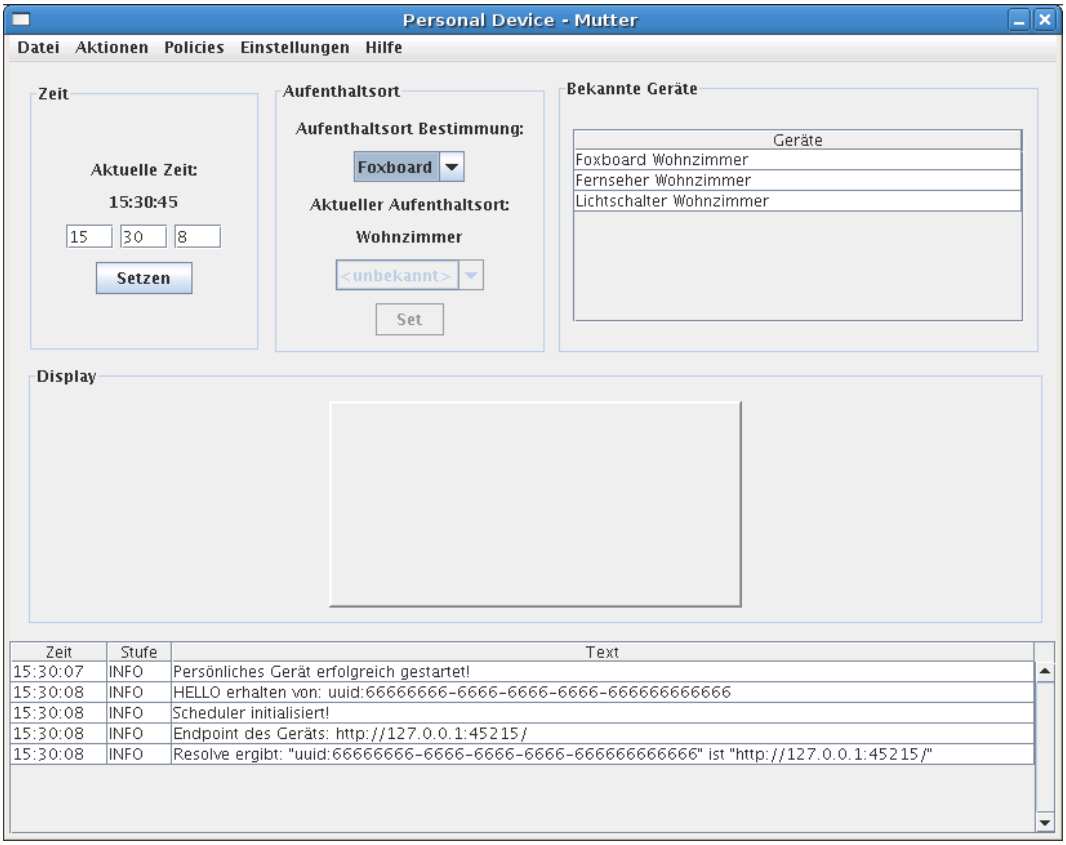


**Abbildung 15:** Personal Device - Swing GUI

Um es dem Benutzer zu ermöglichen, möglichst schnell einen vollständigen Überblick über den Gesamtstatus des Personal Device zu erlangen, wurden die wichtigsten Bedienelemente direkt im Hauptfenster untergebracht:

- **Die Uhrzeit**
  Ein Element des Hauptfensters ist eine einfache Uhr. Sie zeigt die aktuelle Uhrzeit aus Sicht des Personal Device. Für Simulationszwecke ist es möglich die Uhrzeit zu verändern. Diese wird vor allem benötigt, um Policies testen zu können, die abhängig von der aktuellen Uhrzeit sind.

- **Aufenthaltsort**
  Ein Teil des Hauptfensters zeigt ständig den aktuellen Aufenthaltsort des Personal Device an. Gleichzeitig kann das Verfahren zur Bestimmung des Aufenthaltsortes verändert werden. Das Verfahren *Foxboard* interagiert mit Hilfe des LocationMonitors mit den *RoomControlServices* und nutzt die ausgesendeten Events um die eigene Position zu bestimmen (siehe Kapitel 8.2.1.2). Zu Testzwecken besteht auch die Möglichkeit, die Bestimmung des Aufenthaltsorts auf *Manuell* zu setzen. In diesem Fall ist es möglich den Aufenthaltsort selbst festzulegen.

- **Bekannte Geräte**
  Die Tabelle *Bekannte Geräte* zeigt alle Geräte, die für den *ServiceLookupManager* momentan erreichbar sind. Die Tabelle wird dabei über den Listener Mechanismus des *ServiceLookupManager*s stets aktuell gehalten (siehe Kapitel 8.1.2).

- **Display**
  Der *Display*-Bereich des Personal Device simuliert einen einfachen kleinen Bildschirm, über den der *TextOutputService* und der *DialogService* des Personal Device Meldungen an den Benutzer ausgeben können. Das Display implementiert dabei das Interface `UserInterface` und registriert sich bei der Klasse `UserInterfaceProxy` (siehe Kapitel 8.2.2)

- **Log Meldungen**
  Im unteren Bereich des Hauptfensters befindet sich die Liste der vom Personal Device erzeugten Meldungen. Diese Tabelle zeigt also stets alle Nachrichten, die an das *EventLog* übergeben wurden. Die Tabelle implementiert das Interface `EventLogChangeListener` und ist beim EventLog registriert (siehe Kapitel 8.2.1.1).

- **Menü *Aktionen***
  Über das Menü *Aktionen* ist es möglich einige vordefinierte RemoteTasks auszuführen. Die zur Ausführung benötigten Informationen werden dabei vor dem Starten des Tasks abgefragt. Die für das Personal Device realisierten Tasks sind:

  - Anzeigen von Text auf bevorzugten Geräten in bestimmten Räumen
  - Starten von Videos auf bevorzugten *VideoOutput* Geräten im Raum
  - Licht im aktuellen Raum ein- oder ausschalten
  - Deaktivierung des Feueralarms an der Brandschutzzentrale

- **Menü *Policies***
  Das Menü *Policies* bietet die Möglichkeit mit den auf dem Personal Device aktiven Policies zu arbeiten. Unterstützt wird das Anzeigen, das Bearbeiten und das Löschen einzelner bzw. mehrerer Policies einer bestimmten Kategorie. Dabei werden die Policies stets in ihrer XML Repräsentation dargestellt. Zusätzlich besteht die Möglichkeit, Policies in eine Datei zu exportieren und aus einer Datei wieder zu importieren.

### 8.2.4. MIDP Implementierung

Neben der Swing Benutzeroberfläche, welche in erster Linie zur Entwicklung und Demonstration dient, wurde auch ein Benutzerinterface für *MIDP 2.0* implementiert. Da die Kernimplementierung und die *MIDP* Benutzerschnittstelle zusammen lediglich *CLDC 1.1* und *MIDP*

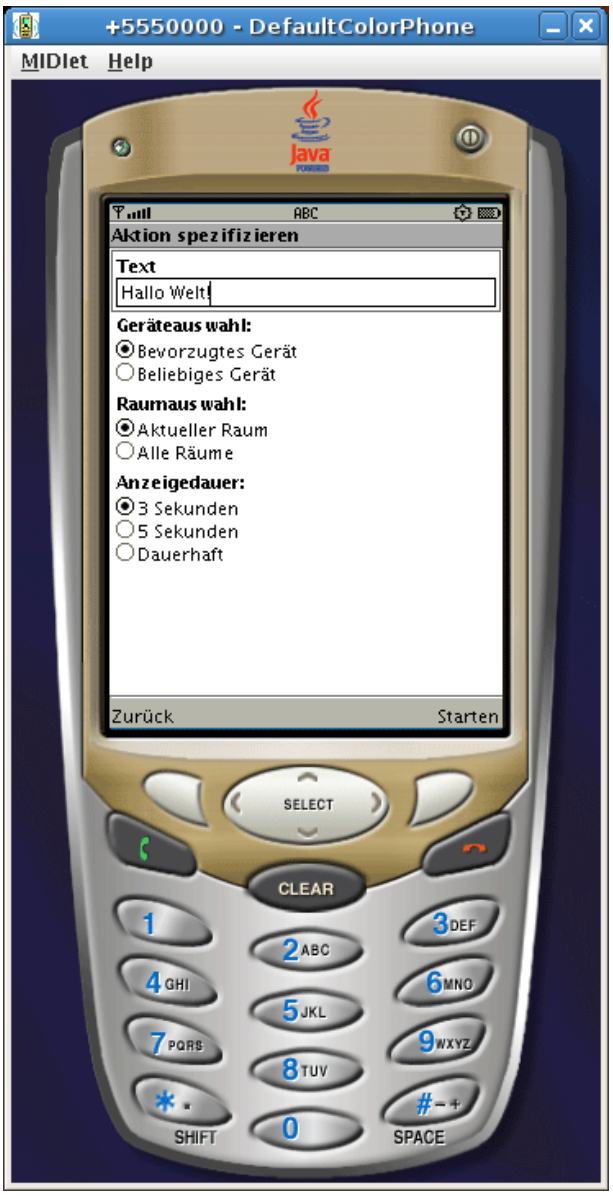


**Abbildung 16:** Personal Device - MIDP Version

*2.0* benötigen, ist die Implementierung des Personal Device auf marktüblichen Mobiltelefonen und PDAs lauffähig.

Bei der Entwicklung wurde das *Sun Wireless Toolkit 2.2 (WTK)* verwendet, welches einen leistungsfähigen Emulator für *MIDP* Anwendungen bereitstellt. Abbildung 16 zeigt das Personal Device im *WTK Emulator*.

Da *MIDP* im Vergleich zu *Swing* in den Möglichkeiten zum Design von Benutzerschnittstellen extrem eingeschränkt ist, wurden bei der Entwicklung der *MIDP* Schnittstelle nur die wichtigsten Features implementiert. Es folgt ein kurzer Überblick über die Möglichkeiten der *MIDP* Implementierung:

- **Menü *Aktionen ausführen*** Die *MIDP* Schnittstelle ermöglicht es, alle RemoteTasks

zu starten, die auch die *Swing* Oberfläche kennt. Dabei werden die nötigen Informationen über einfache Formulare abgefragt.

- **Menü *Bekannte Geräte***
  Das Menü erlaubt, eine Auflistung aller bekannten Geräte zu erhalten, die dem *ServiceLookupManager* bekannt sind. Aus Performance Gründen wurde jedoch nicht der *ManagedDeviceListener*-Mechanismus zur Aktualisierung verwendet. Statt dessen wird die Liste über die Auswahl des Buttons *Update* aktualisiert.

- **Menü *Aktive Policies***
  Diese Funktion erlaubt es eine Liste aller im Gerät aktiven Policies zu erhalten. Im Gegensatz zur *Swing* Implementierung ist ein Bearbeiten der Policies jedoch nicht realisiert worden.

- **Menü *Gerätestatus***
  Über dieses Menü ist dem Benutzer ermöglicht worden, grundlegende Eigenschaften des Personal Devices abzufragen. Zu diesen Eigenschaften gehören die UUID des Geräts, der aktuelle Aufenthaltsort und die momentane Uhrzeit aus Sicht des Personal Device.

## 8.3. Lichtschalter

Der Lichtschalter ist ein einfaches Gerät, das in jedem Zimmer verfügbar ist. Dieser hat ausschließlich die Aufgabe, das Licht ein- und auszuschalten. Die Klasse `LightDeviceService` verwaltet den Status des Lichtschalters. Dabei ändert die Methode `newLightSwitchStatus()` den Status des Lichtschalters und schaltet ihn ein und aus.

## 8.4. Personalcomputer

Der PC implementiert die Kategorien *VideoOutputDevice* (Kapitel 7.2.2), *VideoInputDevice* (7.5), *AudioOutputDevice* (7.4) und *AudioInputDevice* (7.3). Damit ist ein Empfangen und Senden von Video- und Audiodateien möglich. Außerdem kann der PC einen Dialog (7.2.1) initialisieren und Textnachrichten (7.2.3) verschicken, empfangen und anzeigen.

### 8.4.1. Video- und Audiokommunikation

Über eine menügestützte Oberfläche kann eine Video- bzw. eine Audio-Datei ausgewählt werden und an ein beliebiges anderes Gerät der Kategorie *VideoOutputDevice* gesendet werden. Ebenso ist es möglich, im Dateisystem eines anderen *VideoInputDevices* nach Videos Ausschau zu halten und diese zu empfangen.

#### VideoInputDevice

Um die oben genannten Aktionen ausführen zu können, wird in der Implementierung der Kategorie *VideoInputDevice* diese um die Action `verzeichnisAuswaehlenAction` ergänzt. Damit ist es möglich, in einem Input-Parameter einen Verzeichnispfad zu übergeben und es werden in den zwei Output-Parametern dieser Methode die Unterverzeichnisse und Videodateien des Verzeichnispfads zurückgegeben.

Die Spezifikation der Implementierung der Kategorie *VideoInputDevice* für den Personalcomputer wird im Anhang (siehe B.2.1) dokumentiert.

#### VideoOutputDevice

Bei der Implementierung dieser Kategorie geht es nur darum, die bestehende Action `playVideostream` auszubauen. Das Abspielen von Videos wird beim PC durch Aufruf eines beliebigen Mediaplayers realisiert. Dieser kann in einem Dateiauswahldialog beim ersten Aufrufen eines Videos ausgewählt werden.

Die Spezifikation der Implementierung der Kategorie *VideoOutputDevice* für den Personalcomputer wird im Anhang (siehe B.2.3) dokumentiert.

#### AudioInputDevice

Äquivalent zur Implementierung der Kategorie *VideoInputDevice* wird auch hier die neue Action `verzeichnisAuswaehlenAction` hinzugefügt. Der einzige Unterschied besteht lediglich darin, dass im Output-Parameter keine Videodateien, sondern nur Audiodateien zurückgegeben werden.

Die Spezifikation der Implementierung der Kategorie *AudioInputDevice* für den Personalcomputer wird im Anhang (siehe B.2.2) dokumentiert.

#### AudioOutputDevice

Auch diese Implementierung ist äquivalent zum Video-Gegenpart. Audiodateien werden abgespielt, indem eine externe Software, die dazu befähigt ist, beim ersten Abspielen ausgewählt werden muss.

Die Spezifikation der Implementierung der Kategorie *AudioOutputDevice* für den Personalcomputer wird im Anhang (siehe B.2.4) dokumentiert.

### 8.4.2. Text empfangen und verschicken

Im GUI-Menü *Text senden* kann an ein Gerät der Kategorie *TextOutputDevice* eine beliebige Textnachricht geschickt werden und es kann ein Timeout vorgegeben werden, nach dessen Ablauf der Text nicht mehr angezeigt werden soll. Dabei entspricht ein Timeout von null einer unendlich Anzeigedauer. Über die Art und Weise der Realisierung des Timeouts entscheidet jedoch der Zielservice.

Die Spezifikation sieht vor, dass bei einem *TextOutputDevice* über Properties die Textgröße, Textfarbe, Hintergrundfarbe und Schriftart geändert werden können. Diese Einstellungen können jedoch in der GUI des Personalcomputers nicht vorgenommen werden.

Der PC kann auch selbst Text empfangen und nicht nur verschicken. Dies geschieht in einem separaten Fenster, das immer geöffnet ist.

Die Timeout-Einstellung, die bei Aufruf des Text-Services per Parameter übergeben wird, wird korrekt realisiert. Jedoch werden die Property-Einstellungen für Textgröße, Textfarbe, Hintergrundfarbe und Schriftart nicht berücksichtigt.

Die Spezifikation der Implementierung der Kategorie *TextOutputDevice* für den Personalcomputer wird im Anhang (siehe B.2.5) dokumentiert.

### 8.4.3. Dialoge starten und beantworten

Dialoge können über ein einfaches Menüfesnter initialisiert werden. Es ist, so wie in der Spezifikation vorgesehen, eine Auswahl des Zieldevices möglich, des Dialogtyps („Ja, Nein, Abbrechen“, „Ja, Nein“, „OK“), der Angabe einer Frage und eines Timeouts.

Nach dem Starten eines Dialogs wird die Antwort, *Yes*, *No* oder *OK*, im selben Fenster angezeigt. Wenn das Fenster vor dem Erhalt der Antwort geschlossen wurde, ist keine Anzeige der Antwort möglich.

Wenn der PC selbst das Ziel einer Dialog-Anfrage ist, dann öffnet sich ein separates Dialogfenster, das die Frage und abhängig vom Dialogtyp ein bis drei Schaltflächen anzeigt. Wenn eine dieser Schaltflächen, mit Ausnahme von *Cancel*, das den Dialog ohne Aktion beendet, betätigt wird, wird eine Antwort gesendet.

Die Timeout-Vorgaben des Dialog-Initialisierers werden korrekt beachtet, allerdings wird die noch verbleibende Antwortzeit nicht eingeblendet.

Die Spezifikation der Implementierung der Kategorie *DialogDevice* für den Personalcomputer wird im Anhang (siehe B.2.6) dokumentiert.

## 8.5. Fernseher

Der Fernseher ist ein Gerät der Kategorien *VideoOutputDevice* (Kapitel 7.2.2), *VideoInputDevice* (7.5), *AudioOutputDevice* (7.4) und *AudioInputDevice* (7.3). Er kann damit Videodateien und Audiodateien verschicken und empfangen. Zusätzlich verfügt der Fernseher über Funktionen, um eigene Videos und ein simuliertes TV-Programm zu betrachten. Außerdem ist der Fernseher noch ein Gerät der Kategorie *RecorderDevice* (7.6) und der Kategorie *TextOutputDevice* (7.2.3). Damit kann er zur zeitgesteuerten Aufnahme des simulierten Fernsehprogramms auf Festplatte verwendet werden und zur Anzeige eines Textes in einem Fenster.

### 8.5.1. Grundfunktionen des Fernsehers

Zu den Grundfunktionen des Fernsehers gehört das Betrachten fernseheigener Videos und des simulierten TV-Programms. Diese Funktionen kommen ohne die Verwendung von Webservices aus, da dazu keine Kommunikation mit anderen Geräten nötig ist.

Der Festspeicher des Fernsehers, auf dem Video- und Audiodateien gespeichert werden, wird im temporären Verzeichnis im Unterordner `Fernseher` des ausführenden Computers simuliert. Videos und das simulierte TV-Programm unterscheiden sich lediglich dadurch, dass nach Videos im Unterordner `Video` nach Videodateien (das sind Dateien mit einer Dateiendung, die einem für Video registriertem MIME-Typ entsprechen) gesucht wird, während das TV-Programm mangels eines echten Fernsehanschlusses mit Videodateien simuliert wird, die im Verzeichnis `TV` liegen und mit der Zeichenkette `channel` beginnen.

Innerhalb einer grafischen Benutzeroberfläche können die Videos und das simulierte TV-Programm gestartet werden. Die Wiedergabe erfolgt jedoch nicht innerhalb der Fernseher-Umgebung. Dazu ist eine externe Medienabspielsoftware nötig, deren Startdatei bei erstmaligem Aufruf eines Videos in einem Dateiauswahldialog gewählt werden kann.

Von einer Verwendung des *Java Media Frameworks*, das eine eingebundene Abspielmöglichkeit geboten hätte, wurde abgesehen, da damit die Anzahl unterstützter Videoformate stark reduziert worden wäre.

Eine weitere Grundfunktion des Fernsehers, die prinzipiell ohne Webservices auskommt, ist die Einstellung der Helligkeit und des Kontrasts der Benutzeroberfläche. Im Optionsmenü können diese Werte mittels zweier Schieberegler auf Werte von 0 bis 100 geändert werden. Zusätzlich ist auch ein Zugriff von außen her möglich durch Ändern der Properties `Brightness` und `Contrast` der Services für *VideoOutput* und *TextOutput*. Die Properties dieser zwei Services werden dabei immer automatisch synchron gehalten.

Praktisch realisiert werden die Helligkeits- und Kontraständerungen durch Ändern der Farbwerte aller Grafiken und Java-Menüfarben, soweit ein Zugriff darauf auf einfache Art und Weise möglich ist. So behalten die Rahmen der eingebetten Fenster, die Hintergründe der Tabellen und die Schieberegler ihre Helligkeits- und Kontrastwerte bei.

### 8.5.2. Kommunikation mit anderen Geräten

Neben dem Anzeigen der eigenen Videos des Fernsehers ist es möglich, diese per Webservice für andere *VideoOutputDevices* freizugeben oder Videos von anderen *VideoInputDevices* zu empfangen und abzuspielen. Das Gleiche gilt für auf der Festplatte des Fernsehers gelagerte Audiodateien. Außerdem ist der Fernseher auch ein Gerät der Kategorie *TextOutputDevice* und kann deshalb Textnachrichten empfangen.

Um mit anderen Geräten zu kommunizieren, mussten die abstrakten Input- und Output-Devices der Gerätehierarchie an die konkreten Erfordernisse des Fernsehers angepasst werden.

#### VideoInputDevice

Die Kategorie *VideoInputDevice* wird in der Implementierung des Fernsehers um die Action `verzeichnisAuswaehlenAction` ergänzt. Diese Action wurde im Wesentlichen aus praktischen Erwägungen zwecks Kompatibilität mit dem Personalcomputer hinzugefügt und gibt unabhängig vom Input-Parameter im Output-Parameter immer nur die Liste an Videos, die sich im Unterordner `Video` befinden, zurück.

Die Spezifikation der Implementierung der Kategorie *VideoInputDevice* für den Fernseher wird im Anhang (siehe B.1.1) dokumentiert.

#### VideoOutputDevice

Für den *VideoOutput* ist keine neue Action nötig. Es werden lediglich die Mechanismen zum Ausführen der Action `playVideostream` hinzugefügt. Dies beinhaltet die freie Auswahl eines Mediaplayers per Dialogfenster, sofern nicht zuvor schon mal einer gewählt wurde, und das Aufrufen desselben mit der per Parameter übergebenen URL zu einem Video.

Eine Besonderheit ist dabei beim Interpretieren des Input-Parameters, der laut Beschreibung in Kapitel 7.2.2 eine URL enthalten soll, zu beachten: Wenn der Input-Parameter ausschließlich numerische Zeichen enthält, interpretiert der Service dies als einen Identifier für das simulierte TV-Programm im Unterverzeichnis `TV` und ruft die bezeichnete Videodatei auf.

Die Spezifikation der Implementierung der Kategorie *VideoOutputDevice* für den Fernseher wird im Anhang (siehe B.1.3) dokumentiert.

#### AudioInputDevice

Diese Kategorie wird ebenso wie die *VideoInputDevice*-Kategorie um die Action `verzeichnisAuswaehlenAction` ergänzt. Unterschiedlich ist lediglich, dass im Output-Parameter die Liste an Audiodateien zurückgegeben wird, die sich im Unterordner `Audio` befinden.

Die Spezifikation der Implementierung der Kategorie *AudioInputDevice* für den Fernseher wird im Anhang (siehe B.1.2) dokumentiert.

#### AudioOutputDevice

Analog zur Kategorie *VideoOutputDevice* werden auch hier keine neuen Actions hinzugefügt und nur die Methode `playAudiostream` mit Leben gefüllt.

Die Spezifikation der Implementierung der Kategorie *AudioOutputDevice* für den Fernseher wird im Anhang (siehe B.1.4) dokumentiert.

#### TextOutputDevice

Textnachrichten dieses Services werden in einem separaten Textfenster angezeigt, das zentral über die GUI gelegt wird. Wenn im Service ein Timeout vorgegeben ist, wird dieser sekundengenau im Textfenster angezeigt und nach Ablauf des Timeouts wird das Fenster geschlossen, sofern dies nicht zuvor über den Schließen-Button erfolgt ist.

Abweichend von der Spezifikation der Kategorie *TextOutputDevice* ist es nicht möglich über Properties die Textgröße, Schriftart, Textfarbe oder Hintergrundfarbe zu ändern. Diese Werte werden immer einheitlich im Design des Fernsehers gehalten.

Die Spezifikation der Implementierung der Kategorie *TextOutputDevice* für den Fernseher wird im Anhang (siehe B.1.5) dokumentiert.

### 8.5.3. Aufnehmen mit dem integrierten Festplattenrekorder

Als Typ der Kategorie *RecorderDevice* verfügt der Fernseher über einen Festplattenrekorder zur zeitgesteuerten simulierten Aufnahme des simulierten TV-Programms.

Unter dem Menüpunkt *Timer-Einstellungen* kann der Festplattenrekorder programmiert werden. In einer Tabelle muss dazu ein x-beliebiger Name vergeben werden, ein Datum, Startzeit und Stoppzeit. Wenn die Stoppzeit vor der Startzeit liegt, wird das, wie von der Spezifikation gefordert, als Stoppzeit am nächsten Tag interpretiert. Wenn Werte in einem anderen Format als *tt.mm.jjjj* bzw. *hh:mm* eingegeben werden, wird dies per Textfenster bemängelt.

Das Programm, das aufgezeichnet werden soll, kann in einer Liste ausgewählt werden. Der Inhalt der Liste besteht dabei aus den Videodateien des Unterverzeichnisses `TV`, die mit `channel` beginnen.

Wenn eine Timer-Programmierung hinzugefügt wurde, kann sie auch wieder bequem durch Auswahl in der Übersichtstabelle gelöscht werden.

Alle zwanzig Sekunden überprüft der Fernseher, ob die Zeit einer Timer-Programmierung erreicht ist, und startet in diesem Fall die Aufnahme. Die Aufnahme wird dabei dadurch simuliert, dass die Videodatei des programmierten TV-Senders in den Unterordner `Video` kopiert wird und mit dem Namen der Timer-Programmierung benannt wird. Zusätzlich wird bei Aufnahmebeginn für zehn Sekunden eine Textmeldung angezeigt, dass und was aufgenommen wird.

Wenn bei einer der Überprüfungen der Timer-Einträge, die der Fernseher alle zwanzig Sekunden vornimmt, festgestellt wird, dass eine Timer-Programmierung nicht mehr gültig ist, da sie zu alt ist, wird sie gelöscht.

Die Spezifikation der Implementierung der Kategorie *RecorderDevice* für den Fernseher wird im Anhang (siehe B.1.6) dokumentiert.

## 8.6. Brandschutz

Der Brandschutz ist eine spezielle Anwendung im *Digital Home*. Sie wurde in die Kategorien *Feuermelder* und *Brandschutzzentrale* unterteilt.

Die Funktion der Brandschutzzentrale (*FireCenterDevice*) ist, die Feuermelder in den Räumen Wohnzimmer und Schlafzimmer zu erkennen und dem entsprechend zu abonnieren. Er soll auch zur Kontrolle dienen, damit die Feuermelder richtig funktionieren und keine Störungen haben.

Die Brandschutzzentrale bekommt von den aktiven Feuermeldern Ihre IDs und so weiss man welche Feuermelder in welchem Raum sind.

Sobald ein Feuer ausbricht, bekommt die Zentrale ein Signal von dem Feuermelder im Zimmer. Daraufhin kommen die Policies ins Spiel und die Brandschutzzentrale schickt Nachrichten in Textform zu den Geräten, welche *TextOutputDevice* implementiert haben um die Personen zu warnen.

Die Funktion des Feuermelders (*FireDetectorDevice*) besteht darin, die Zimmertemperatur zu messen. Sobald der Wert über 50 Grad steigt, sendet der Feuermelder an die Brandschutzzentrale ein Signal, dabei kann der Raum identifiziert werden, indem das Feuer ausgebrochen ist.

Die Feuermelder werden nach einem Brandfall in den Startzustand zurückgesetzt.

# 9. Bluetooth

Hinter dem Namen Bluetooth verbirgt sich eine Funktechnologie, die entwickelt wurde, um verschiedene elektronische Geräte wie z. B. Laptops, PDAs, Drucker, Mobiltelefone, Kopfhörer u.v.m miteinander auf unkomplizierte Art und Weise zu verbinden [IEE02]. Diese Funkverbindung nutzt das 2,4 GHz ISM Band (ISM = Industrial, Scientific, Medical). Die eindeutige Identifizierung der verschiedenen Bluetooth Geräte erfolgt hierbei durch eine Art MAC Adresse, wie sie z. B. aus dem Ethernet her bekannt ist, im folgenden BT-MAC genannt.

Mit Bluetooth lassen sich Reichweiten von maximal 100 Metern (in der 20 dBm Spezifikation) erreichen. In der ursprünglichen Version, welche auch in dieser PG zum Einsatz kommt, werden jedoch nur 10 Meter erreicht; diese Version kommt aufgrund des geringeren Energiebedarfs in Mobiltelefonen zum Einsatz. Die durch Bluetooth übertragenen Daten werden mit einem, von einem Zufallsgenerator erzeugten, bis zu 128 Bit langem Schlüssel codiert.

800 mal weniger als bei einem Handy - nämlich nur 1 mW Sendeleistung - entsteht beim Betrieb dieser Technologie, bei der Datenübertragungsraten von 1 MBit/s brutto bzw. 723 KBit/s netto erreicht werden.

## 9.1. Der Linux Bluetooth Stack

Unter Linux wurden anfangs verschiedene Ansätze verfolgt die Technologie zugänglich zu machen. Letzten Endes jedoch hat sich das BlueZ Projekt durchsetzen können. Seit Kernel 2.6 ist der BlueZ Protokoll Stack standardmäßig im Kernel integriert. Der Protokollstapel (Abbildung 17) gliedert sich im wesentlichen in zwei Schichten, die unteren drei Bluetooth Radio, Baseband und LMP und darüber die Schichten ab L2CAP. Die Schnittstelle dazwischen bildet das Host Controller Interface (HCI). Die unteren Schichten bilden die Basis zur Kommunikation mit und über ein Bluetooth-Interface, dazu gehören der Zugriff auf die Bluetooth Hardware als auch die Aushandlung der Betriebsmodi, wozu z.B. gehört, festzulegen welches Gerät jeweils die Master und Slave Rolle übernimmt. Im folgenden werden wir nur noch von der HCI-Ebene sprechen, womit gemeint ist, daß nur eine Verbindung auf den unteren drei Protokollschichten aufgebaut wird. Die HCI-Ebene ist als Vorbereitung zu einem TCP-Verbindungsaufbau für uns wichtig, da es erst auf diese Weise möglich ist RSSI Werte und MAC Adressen auszulesen. Die Logical Link Control and Adaption Protokoll (L2CAP) Schicht, hat die Aufgabe mehrere logische Verbindungen über ein Interface zu ermöglichen, um so zu gewährleisten, daß ein ganzes Bluetooth Netzwerk aufgebaut werden kann, über nur einen Master. Ansonsten könnte man immer nur eine direkte Verbindung zwischen zwei Geräten aufbauen. Um eine TCP Verbindung zu erstellen bietet BlueZ zwei Profile an: Dial Up Networking (DUN) und Personal Area Network (PAN).

PAN ist aus unserer Sicht die elegantere Lösung, da TCP/IP direkt auf dem L2CAP Protokoll aufbaut und nicht wie DUN RFCOMM mit PPP (dem Point-to-Point Protocoll) als Träger für TCP/IP verwendet.

Als problematisch erwies sich die Tatsache, dass die Java Bluetooth APIs nicht die Protokoll Stacks wie in Abbildung 17 unterstüzen, sondern auf der L2CAP Ebene aufhören und dadurch der Aufbau einer TCP Verbindung unmöglich war. Infolgedessen mussten die Java Bluetooth APIs um die zusätzlichen Funktionen des BlueZ Protokollstack erweitert werden.

Der BlueZ Protokollstack gliedert sich in zwei große Teile. Der erste Teil, im Quellcode durch das Bibliotheksverzeichnis */lib* repräsentiert, umfasst die Grundfunktionalität, die bei jeglicher Kommunikation über das Bluetooth-Interface benötigt wird. Da generell jeder der Bluetooth-

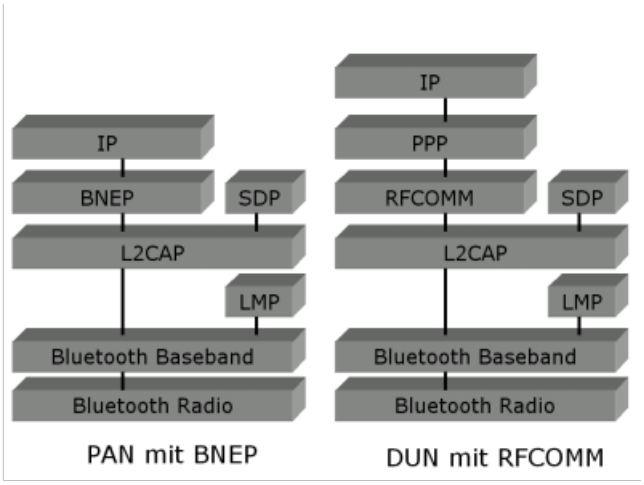


**Abbildung 17:** PAN und DUN im Vergleich

Dienste auf der HCI-Ebene aufbaut, ist die HCI-Funktionalität als Basis anzusehen. Hierüber werden all jene Funktionen gesteuert, die für den Aufbau einer Verbindung elementar sind. Zum Beispiel ist die Suchfunktion grundlegend für die unterste Bluetooth Protokollschicht, da hier nicht nur die verfügbaren Dienste auf der Gegenseite abgefragt werden, sondern auch die Uhrzeiten, um später einen Offset zu berechnen. Jeder einzelne Parameter, der für einen erfolgreichen Verbindungsaufbau nötig ist, kann über entsprechende Methoden aufgerufen werden. So ist die Funktionalität, welche in der Datei */lib/src/hci.c* implementiert ist, sozusagen der Kern des Stacks.

Der andere Teil, repräsentiert durch das Verzeichnis */utils*, liefert sowohl die Funktionalität der höheren Schichten im Bluetooth Protokoll Stack, als auch diverse Dienste für den Umgang mit Bluetooth. Der Aufbau einer TCP/IP Verbindung mittels Bluetooth ist, wie schon erwähnt, mit Hilfe des PAN-Diensts, welcher auf das Bluetooth-Network-Encapsulation-Protokoll, kurz BNEP, aufbaut, realisierbar. Beide Teile (PAN und BNEP) sind Bestandteil des utils-Stackteils.

Der PAN-Dienst baut auf der HCI-Ebene auf, so dass funktional gesehen vor dem Aufbau einer BNEP-Verbindung eine HCI-Verbindung aufgebaut wird, um die nötigen Geräteparameter auszulesen.

Um nun die Java Bluetooth API um die zusätzlichen Funktionen zum Aufbau einer TCP/IP Verbindung zu ergänzen, müssen in Java Interfaces definert werden, welche native C-Funktionen aufrufen. Wie im Kapitel 4.1 geschildert, gibt es für diesen Zweck in der Java Standard-Edition das so genannte Java-Native-Interface. Anschaulich gesehen wird hier eine Möglichkeit geschaffen dem Javaprogramm mitzuteilen wie eine entsprechende C-Funktion heißt und welche Parameter diese hat.

Als Beispiel soll hier die Methode `pandCreateConnection` von beiden Seiten beleuchtet werden, einmal aus Java Sicht und auf der anderen Seite das C-Pendant.

Auf der Java Seite definiert man durch den Zusatz *native* ein Interface, welches aus Java heraus aufgerufen wird und dabei die native C-Funktion ausführt. Die übergebenen Parameter sind aus den JNI Datentypen auszuwählen, da hier nicht alles in Java Mögliche übergeben werden kann.

```
public native int pandCreateConnection(String jbdaddr, String netdev)
```

```
    throws BlueZException;
```

Das Beispiel zeigt, dass es sogar möglich ist, Exceptions aus C-Funktionen heraus zu werfen. Das ist eine sehr wichtige und komfortable Funktionalität, um die ansonsten schwer zu beherschenden nativen Methoden zu kontrollieren, da ansonsten nur der *Standart-Error-Kanal* übrig bleibt um Fehler an den Nutzer weiter zu geben. Der Nachteil dabei wäre, dass die Meldungen nur auf der Konsole ausgegeben werden und nie im Java Teil ankommen.

Die entsprechende native C-Seite sieht dann wie folgt aus:

```
JNIEXPORT jint JNICALL
    Java_edu_udo_pg490_knifox_BlueZ_pandCreateConnection
  (JNIEnv *env, jobject obj, jstring jbdaddr, jstring netdev){
```

Dabei gibt der Rückgabewert `jint` an, dass es sich um einen Java-Datentyp handelt und das `JNICALL` referenziert direkt die Javamethode. Im vorliegenden Fall entspricht dies dem Java-Package `edu.udo.pg490.knifox.BlueZ` mit der Methode `pandCreateConnection`. Nur die beiden rechten Parameter entsprechen den von Java übergebenen, die beiden anderen sind JNI spezifisch und bilden die Objektreferenz, weil C nichts von Javaobjekten weiß, bzw. nicht weiß wie diese referenziert werden.

Die selbe native Methode von einem JavaME Programm aus aufzurufen gestaltet sich wesentlich schwieriger, da nicht mehr einfach Parameter übergeben werden können. Genauer gesagt bleibt die Java Seite unangetastet, nur auf der C-Seite müssen die Parameter auf eine sehr umständliche Weise "abgeholt" werden. Vereinfacht gesagt liegen die übergebenen Parameter immer an einer bestimmten Stelle auf dem Stack in Relation zur aufrufenden Methode und an genau dieser Stelle muss man sich die Parameter in C wieder suchen. Das gleiche gilt dann in ähnlicher Form für die Rückgabewerte. Die gleiche C-Methode sieht dann mit Implementierung des KNI wie folgt aus:

```
KNIEXPORT KNI_RETURNTYPE_INT
    Java_edu_udo_pg490_knifox_JavaBlueZ_pandCreateConnection()
  {
```

Im Vergleich zur JNI C-Methode fällt auf, dass die wesentlichen Teile, die Übergabeparameter, komplett fehlen. Diese muss man sich dann wie folgt selber wieder holen:

```
/* Declare handle */
    KNI_StartHandles(2);
    KNI_DeclareHandle(stringHandle);

    /* Read parameter #1 to stringHandle */
    KNI_GetParameterAsObject(1, stringHandle);

    /* Get the length of the string */
    size = KNI_GetStringLength(stringHandle);

    /* Copy the Java string to our own buffer (as Unicode) */
    KNI_GetStringRegion(stringHandle, 0, size, buffer);
```

Es werden zunächst zwei *handles* definiert, eines für jeden komplexen Datentyp, der übergeben wird, in diesem Beispiel zwei Strings. Simple Datentypen können wesentlich einfacher ausgelesen werden, für ein Integer reicht dafür ein einfaches `KNI_GetParameterAsInt(0)`.

Der folgende Befehl `KNI_GetParameterAsObject(1, stringHandle)` sagt aus, dass der erste Parameter vom Stack geholt wird und da Strings in Java Objekte sind, muss dieser als Objekt geholt werden. Dadurch wird sozusagen die Referenz auf den richtigen Speicherplatz gesetzt und durch die Methode `KNI_GetStringLength(stringHandle)` die Länge des Strings ausgelesen. Mit diesen beiden Informationen, also wo ein Datum steht und wie lang es ist, kann der eigentliche Ausleseprozess gestartet werden. Das realisiert dann die letzte oben gezeigte Methode `KNI_GetStringRegion(stringHandle, 0, size, buffer)`, wobei `stringHandle` die Speicherreferenz ist und `buffer` die in C lokale Variable. Wie der Name der Methode impliziert, wird ein Bereich aus dem übergebenen String ausgelesen. In dem Beispiel soll der komplette String übergeben werden, so dass als untere Schranke die 0 und als obere Schranke die Länge des Strings, wie schon erwähnt in `size` hinterlegt, festgelegt wird.

Strukturell haben wir eine Menge von Java-Klassen innerhalb des Package `de.udo.pg490.knifox` (siehe Anhang D, Seite 134) geschaffen, in der sämtliche nativen Interfaces stehen, und eine umfangreiche Java Bluetooth API repäsentieren. Die C-Funktionen gibt es dann einmal als JNI Version für die JavaSE und als KNI Version für die JavaME. Letztere Version läuft eingefügt in die KVM auf dem Fox-Board, die andere Version realisiert die simulierten Fox-Boards.

## 9.2. Personenerkennung

Das Ziel der Arbeit mit Bluetooth war eine Personenerkennung im Digital Home zu realisieren. Dabei genügt eine raumbasierte Erkennung, da es für die Aufgaben des Digital Home unerheblich ist, wo genau im Raum sich eine Person befindet, sondern nur wichtig ist, dass sie im Raum ist. Daraus ergibt sich, dass für jeden Raum ein Bluetooth Access Point (Foxboard) zur Lokalisierung ausreicht. Das Digital Home der PG besitzt 2 Räume, so dass sich die Personenerkennung auf eben 2 Foxboards beschränkt.

Jedes Foxboard beinhaltet einen Service, FoxboardSchlafzimmer respektive FoxboardWohnzimmer, die miteinander interagieren und die Personenerkennung im Digital Home realisieren. Diese Interaktion wird im Petri-Netz, Abbildung 18, bildhaft gemacht.

Die beiden RoomControlServices können nur miteinander existieren. Dieses Verhalten wird über eine gegenseitige Servicesuche realisiert, die so lange abläuft, bis der jeweils andere Service gefunden wird. Erst danach wird die eigentliche Funktionalität des Service angestoßen.

Wenn sich die beiden RoomControlServices gegenseitig bekannt gemacht haben, startet in jedem Service der erste Schritt der Personenerkennung. Beide Foxboards suchen durch Aufruf der Methode `inquiryKnownDevices()` in der Umgebung nach vorhandenen Bluetooth Geräten. Die Besonderheit dieser Methode liegt darin, dass der Ergebnisvektor nicht sämtliche Bluetooth Geräte auflistet, sondern direkt die Geräte, welche nicht Bewohner des Digital Home sind, verwirft. Technisch wird ein normales Bluetooth Inquiry mittels der Methode `hciInquiry(int DeviceID)` (siehe Anhang D.1, Seite 136) ausgeführt, und aus dem Ergebnisvektor werden alle Geräte, die nicht in der Constants Klasse als Bewohner des Digital Home gekennzeichnet sind, entfernt. Dadurch ist gesichert, dass keine illegale Person die Funktionen des Digital Home benutzt.

Der nächste Schritt besteht darin, zu jedem der Geräte, die durch ein `inquiryKnownDevice()` zurückgeliefert wurden, eine HCI Verbindung herzustellen. Diese HCI Verbindung ist notwen-

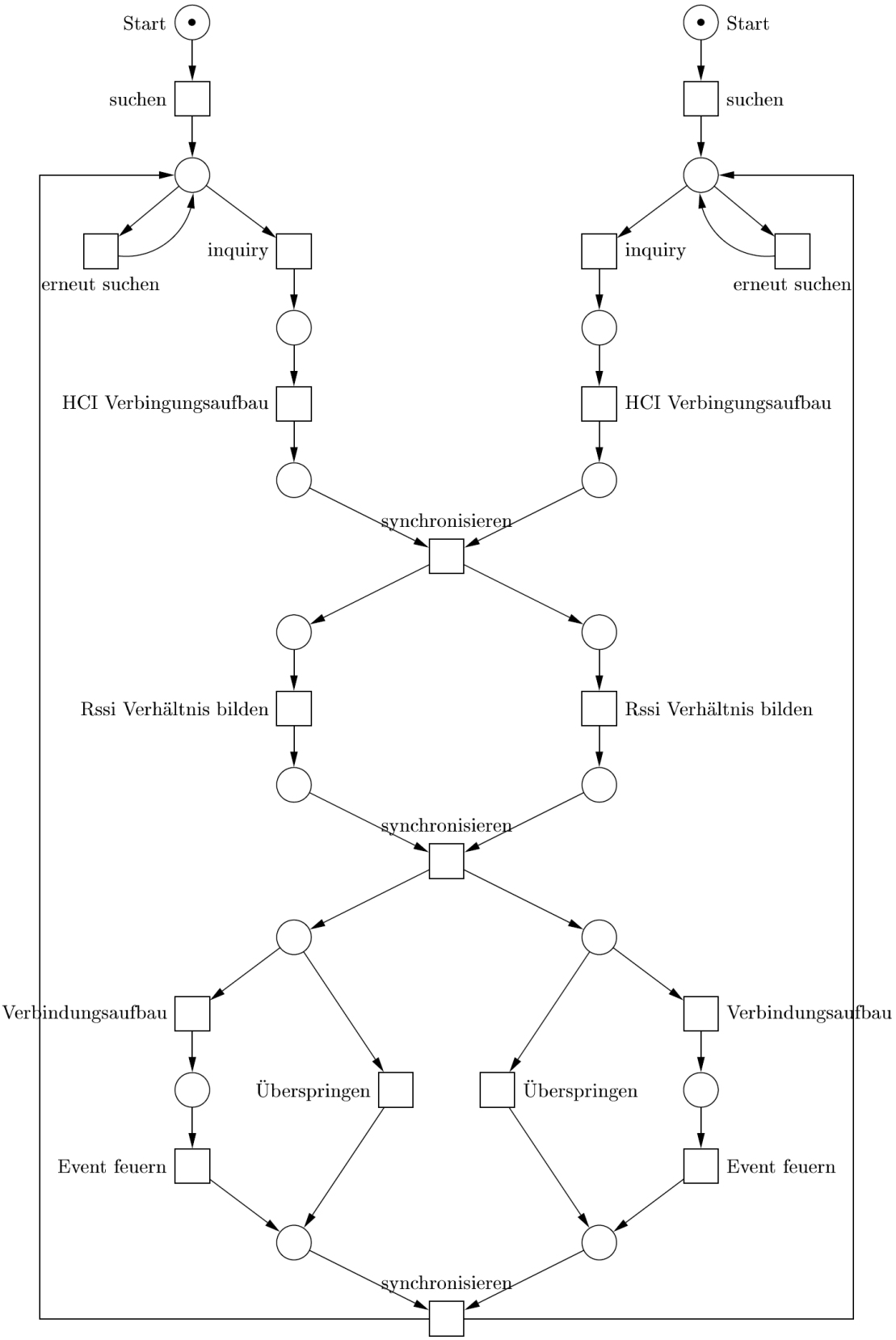


**Abbildung 18:** Der Roomcontrol Service

dig, um an späterer Stelle den Rssi Wert, also die Signalqualität, auszulesen. Innerhalb der Methode `hciConnectKnownDevices()` wird die Verbindung durch den Aufruf der Methode `hciCreateConnection()` (siehe Anhang D.1, Seite 135) aufgebaut. Der Rückgabewert, ein sogenanntes Verbindungshandle, wird gespeichert, um so an späterer Stelle die HCI Verbindung wieder trennen zu können.

Darauf folgt ein Abgleich der Liste mit Personen im Raum, mit tatsächlich aufgebauten bnep Verbindungen, welche durch einen Aufruf der Methode `pandListConnections()` (siehe Anhang D.1, Seite 138) ermittelt werden. Diese Abfrage ist nötig, um Benutzer aus dem DigitalHome abzumelden, wenn sie das Haus verlassen. Besteht keine Verbindung mehr zu einer Person, die sich laut Personenliste im Raum befinden müsste, so wird sie aus dieser Liste entfernt. Dies geschieht durch Aufruf der Methode `personLeavesRoom()`. In dieser Methode wird automatisch ein Event zur Statusänderung der Personenliste gefeuert. Dieser Abgleich könnte an jeder beliebigen Stelle des Service stehen, es ist nur wichtig, dass er einmal pro Durchlauf stattfindet.

Als nächstes findet die erste Synchronisation der Services statt. Damit die beiden Foxboards zur gleichen Zeit den Rssi Wert zu einem bestimmten Gerät auslesen, warten die Services an dieser Stelle aufeinander. Dies geschieht durch den Aufruf der Methode `synchronizeServices()`. Im Detail wird die Synchronisation über das gegenseitige Setzen und Auslesen einer Property SytemCounter realisiert. Jeder Service erhöht diese vor jedem Synchronisationspunkt und liest den SystemCounter des entfernten Foxboards aus. Ist der eigene SystemCounter größer als der entfernte, muss gewartet werden, ansonsten wird fortgefahren.

Wie schon erwähnt folgt darauf das Auslesen des Rssi Wertes für das erste Gerät aus der Liste der bekannten Bluetooth Geräte in der Umgebung mit Hilfe der Methode `hciGetRssi()` (siehe Anhang D.1, Seite 136) . Dabei wird zuerst der eigene Rssi Wert ausgelesen, und danach der Rssi Wert des entfernten Foxboards. Ist das entfernte Foxboard nicht in Reichweite des Gerätes, so wird der Wert -888 zurückgeliefert, um sicherzustellen, dass der lokale Rssi Wert erheblich besser ist. Als nächstes wird das Verhältnis zwischen lokalem und entferntem Rssi Wert, also die Differenz, berechnet und die Property RssiRatio gesetzt. Darauf folgt der nächste Synchronisationspunkt, um sicherzustellen, dass beide Services das Auslesen der Rssi Werte beendet haben.

Beide Services holen sich nun mittels `remoteRssiRatio()` das jeweils entfernte Rssi Verhältnis, um es mit dem Lokalen zu vergleichen und eine Entscheidung zu treffen, welches Foxboard eine bnep Verbindung zum Personal Device aufbauen darf. Ein Rssi Verhältnis berechnet sich dabei aus dem eigenen Rssi Wert subtrahiert von dem entfernten. Ein Service, also Wohnzimmer oder Schlafzimmer, muss der präferierte Raum sein, der bei Gleichheit der Rssi Verhältnisse die Verbindung aufbauen darf. Damit wird verhindert, dass bei Gleichheit beide Services eine Verbindung aufbauen, und somit die Person in 2 Räumen lokalisiert ist. Der Service, der keine Verbindung aufbauen darf, springt direkt zum nächsten Synchronisationspunkt und wartet bis der andere Service den Verbindungsaufbau abgeschlossen hat.

Der Service, der eine Verbindung aufbauen darf, muss nun zuerst Sorge tragen, dass das entfernte Foxboard seine eventuell bestehende bnep Verbindung zum entsprechenden Personal Device trennt. Dieses kann der Service durch den Aufruf von `remotePandDisconnect()` erreichen. Innerhalb dieser Methode wird eine Action aufgerufen, die auf dem entfernten Foxboard die Methode `pandDisconnect()` (siehe Anhang D.1, Seite 138) ausführt. Nur wenn dieser Aufruf einen Erfolg zurückliefert, darf der Service mit dem Verbindungsaufbau fortfahren. Der nächste Schritt ist nun, dass der Service sich aus der Liste mit HCIConnection-Handles das passende zu der Bluetooth Adresse des Personal Device raussucht, und die HCI Verbindung

trennt.

Natürlich kann es vorkommen, dass ein Service schon im letzten Durchlauf ein besseres Rssi Verhältnis als sein Gegenüber hatte, und somit schon eine bestehende bnep Verbindung zum Personal Device aufrecht erhält. Dadurch ist es unnötig zu versuchen eine neue Verbindung aufzubauen. Diese Information wird durch einen Aufruf von `pandListConnections()` (siehe Anhang D.1, Seite 138) eingeholt und bei Übereinstimmung zum nächsten Synchronisationspunkt gesprungen. Ist jedoch keine Verbindung vorhanden, so wird diese mittels Aufruf von `pandCreateConnection(bdaddr, interface)` (siehe Anhang D.1, Seite 137) aufgebaut. Die explizite Angabe des Interface Namens ist notwendig, da ansonsten für jede Verbindung versucht werden würde das Interface bnep0 einzurichten. Ist dieses einmal vorhanden, wird der nächste Verbindungsaufbau verweigert. Bei einem Versuch eines Verbindungsaufbaus sollte eigentlich laut Spezifikation des BlueZ Protokoll Stack eine 0 oder 1 für Erfolg oder Misserfolg zurückgegeben werden. Leider wird aber anstelle dessen ein scheinbar randomisierter Integer Wert überliefert, so dass erneut über ein Abgleich der Rückgabe von `pandListConnections()` mit der Bluetooth Adresse des Personal Device der Erfolgsstatus ermittelt werden muss.

Nach Beendigung des Verbindungsaufbaus wird ein entsprechendes Event durch Aufruf von `personEntersRoom()` gefeuert und zum nächsten Synchronisationspunkt gesprungen. Danach wird der ganze Vorgang ausgehend vom Auslesen des Rssi Wertes bis zum Verbindungsaufbau für das nächste Gerät in der Liste der bekannten Bluetooth Geräte in der Umgebung wiederholt, bis die Liste abgearbeitet ist. Anschliessend startet der Service von neuem.

Damit nicht bei jedem Durchlauf des Services die relativ zeitintensive Suche nach dem entfernten Foxboard von neuem gestartet wird, passiert dieser Teil nur alle zwanzig mal. Dadurch ist aber immer noch gesichert, dass bei kurzzeitigem Ausfall eines Foxboards der Betrieb nach einiger Zeit wieder aufgenommen werden kann.

Weiterhin wird bei jedem Servicedurchlauf jede Person, die sich im Raum befindet, kurzzeitig entfernt und wieder hinzugefügt, um dadurch nochmalig ein Event zu feuern. Durch die Trägheit der Initialisierung des Personal Devices bei einem erfolgtem Verbindungsaufbau kann ansonsten das Event verpasst werden, und der Benutzer ist fortan als Geist im Digital Home unterwegs.

## 9.3. Netzwerk / Routing

Die Geräte, die über das Digital-Home Netzwerk verbunden sind, lassen sich in drei Kategorien einteilen.

- **Fest vernetzte Geräte**
  Geräte, die dauerhaft per LAN mit dem Netzwerk verbunden sind. - Sie benötigen nur ein Netzwerk Interface und besitzen eine fest zugewiesene IP Adresse.

- **Bluetooth Access Points**
  Die Aufgabe der Bluetooth Access Points übernehmen zwei (ggfs. simulierte) Foxboards. Sie sind ebenfalls per LAN mit dem gesamten Netzwerk verbunden. Zusätzlich kann jedes Foxboard noch bis zu 7 Bluetooth PAN Verbindungen zu den mobilen Personal Devices aufbauen.

- **Bluetooth Geräte**
  Jedes Personal Device ist zu jedem Zeitpunkt mit höchstens einem Foxboard über ein Personal Area Network (PAN) verbunden. Es kann also zwischenzeitig auch gar nicht

mit den Digital-Home Netzwerk verbunden sein.

Aus dieser Aufteilung ergaben sich einige Probleme, die gelöst werden mussten:

- Services, die auf den einzelnen Geräten laufen, sollen von jedem anderen Gerät auffindbar sein. Dazu muss einerseits der gesamte Netzwerk Verkehr der *fest vernetzten Geräte* von den Foxboards an die über Bluetooth verbundenen Personal Devices weitergeleitet werden (und umgekehrt), andererseits müssen aber auch Services, die auf dem Foxboard laufen, den gesamten Traffic empfangen.
- Alle Datenpakete der Foxboards (RoomControl Services) müssen in das LAN und über jede Bluetooth Verbindung geschickt werden.
- Die Personal Devices können im laufenden Betrieb ihre Verbindung zum Netzwerk verlieren und nach einiger Zeit über eine neue, andere Verbindung wieder mit dem Netzwerk verbunden werden. Die auf den Personal Devices laufenden Services dürfen davon nichts mitbekommen.

### 9.3.1. Konfiguration der Geräte

Abbildung 19 zeigt eine Übersicht der im Digital Home vorkommenden Geräte mit ihren Netzwerk Interfaces und IP Adressen. Damit die unter Java laufenden Services die richtige IP-Adresse als Absender nutzen, muss in jedem Gerät in der Datei */etc/hosts* dem Hostnamen die Haupt-IP Adresse des Gerätes zugewiesen werden.

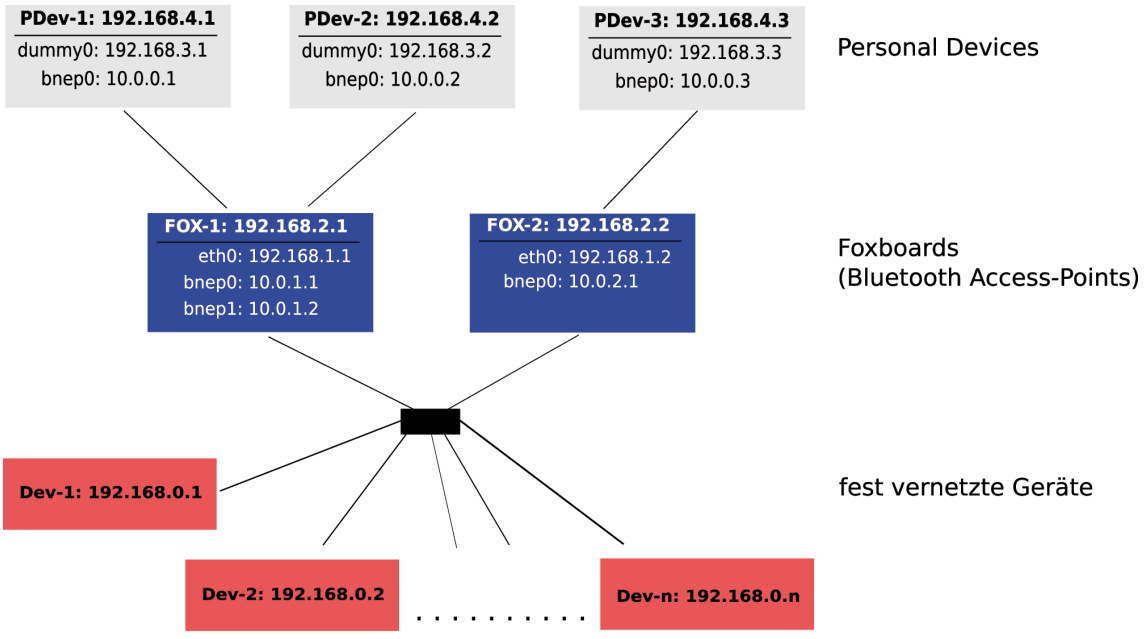


**Abbildung 19:** Geräte mit ihren Interfaces und Adressierungen

### Fest vernetzte Geräte

Diese Geräte benötigen keine besondere Konfiguration. Die Geräte sind über einen Switch miteinander verbunden und jedes Gerät bekommt statisch eine IP Adresse aus dem Subnetz 192.168.0.x zugwiesen. In der Routing Tabelle muss die default Route auf das LAN Interface (eth0) eingerichtet werden, damit alle IP und Multicast Nachrichten in dieses Netz geschickt werden.

### Foxboards

Hier ist die Konfiguration komplizierter. Jedes Foxboard besitzt mehrere Netzwerk Interfaces, die mittels einer sogenannten Bridge zusammengefasst werden. Diese Bridge kann dann als virtuelles Interface angesprochen werden und bekommt eine IP Adresse aus dem Subnetz 192.168.2.x zugewiesen. Die default Route wird ebenfalls auf dieses Interface gesetzt. Alle weiteren Interfaces (eth0 für das LAN Netzwerk und pro PAN Verbindung ein bnep Interface) müssen bei ihrer Initialisierung dieser Bridge hinzugefügt werden. Das dev-up Skript der bnep Interfaces befindet sich im Ordner */etc/bluetooth/pan/*.

Skript 18 zeigt beispielhaft alle nötigen Schritte zur Initialisierung von *Fox1*.

```
#!/bin/sh
ifconfig eth0 192.168.1.1 allmulti promisc
        # richtet das eth0 Interface ein
brctl addbr Fox1
        # richtet die Bridge ein
brctl setfd Fox1 1
        #setzt die Forwarding delay time
brctl addif Fox1 eth0
        # fuegt das eth0 Interface der Bridge hinzu
ifconfig Fox1 192.168.2.1 allmulti promisc
        # richtet das Bridge Interface ein
route del -net 192.168.2.0/24
        # entfernt unnoetige standard Route
route del -net 192.168.1.0/24
        # entfernt unnoetige standard Route
route add default Fox1
        # Setzt die Default Route auf das Bridge Interface
```

**Quellcode 18:** Init Skript für Foxboard 1

Das folgende Skript 19 muss im *Foxboard 1* in der Datei */etc/bluetooth/pan/dev-up* gespeichert werden und wird dann automatisch nach dem Aufbau einer Bluetooth PAN Verbindung ausgeführt. Da es für alle bnep Interfaces nur ein dev-up Skript gibt, müssen alle Möglichkeiten bei jedem Aufruf abgearbeitet werden.

```
#!/bin/sh
ifconfig bnep0 10.0.1.1 allmulti promisc
ifconfig bnep1 10.0.1.2 allmulti promisc
```

```
ifconfig bnep2 10.0.1.3 allmulti promisc
        # bnep Interface einrichten
brctl addif Fox1 bnep0
brctl addif Fox1 bnep1
brctl addif Fox1 bnep2
        # bnep der Bridge hinzufuegen
route del -net 10.0.1.0/24
route del -net 10.0.1.0/24
route del -net 10.0.1.0/24
        # standard Routen von bnep entfernen
```

**Quellcode 19:** Dev-up Skript für Foxboard 1

### Personal Devices

Genau wie bei den Foxboards wird bei diesen Devices eine Bridge eingerichtet, die eine IP Adresse aus dem Subnetz 192.168.4.x erhält und als default Route eingerichtet wird. Dieser Bridge wird jeweils ein Dummy Interface *dummy0* hinzugefügt, das immer verfügbar ist, unabhängig davon, ob das Personal Device mit dem Netzwerk verbunden ist oder nicht. Somit steht einem gestarteten Service zu jedem Zeitpunkt ein Netzwerk zur Verfügung, mit dem er kommunizieren kann. Baut das Personal Device nun eine PAN Verbindung zu einem Foxboard auf, wird ein neues Interface (bnep0) eingerichtet, das eine IP Adresse aus dem Subnetz 10.0.0.x bekommt und der Bridge hinzugefügt werden muss. So wird der gesamte Netzwerkverkehr, den der Service an die Bridge schickt, sofort über die PAN Verbindung in das Netzwerk weitergeleitet.

Skript 20 zeigt beispielhaft alle nötigen Schritte zur Initialisierung von *Personal Device 1*.

```
#!/bin/sh
ifconfig dummy0 192.168.3.1 allmulti promisc
        # richtet das Dummy Interface ein
brctl addbr PDev1
        # richtet die Bridge ein
brctl setfd PDev1 1
        # setzt die Forwarding delay time
brctl addif PDev1 dummy0
        # fuegt das Dummy Interface der Bridge hinzu
sleep 1
ifconfig PDev1 192.168.4.1 allmulti promisc
        # richtet das Bridge Interface ein
route add default PDev1
        # Setzt die Default Route auf das Bridge Interface
route del -net 192.168.3.0/24 dummy0
        # entfernt unnoetige standard Route
route del -net 192.168.4.0/24 PDev1
        # entfernt unnoetige standard Route
pand -r NAP -M -s
```

```
sdpd
        # Auf PAN Verbindung warten und diese annehmen
```

**Quellcode 20:** Init Skript für Personal Device 1

Die im folgenden Skript 21 aufgeführten Befehle, die zum Initialisieren der bnep Interfaces benötigt werden, müssen in das PAN dev-up Skript geschrieben werden, das unter */etc/bluetooth/pan/* zu finden ist. Sie werden dann automatisch nach dem Aufbau einer Bluetooth PAN Verbindung ausgeführt.

```
#!/bin/sh
ifconfig bnep0 10.0.0.1 allmulti promisc
        # bnep0 Interface einrichten
brctl addif PDev1 bnep0
        # bnep0 der Bridge hinzufuegen
route del -net 10.0.0.0/72
        # standard Route von bnep0 entfernen
```

**Quellcode 21:** Dev-up Skript für Personal Device 1

# 10. Management

Im Zuge der Seminarphase haben wir uns unter anderem mit *Policy Management for Autonomic Computing (PMAC)* von IBM befasst und es stand zur Auswahl, dieses Framework für unsere Managementfunktionen zu benutzen oder das Management komplett selbst zu entwickeln. Die Entscheidung fiel dabei auf die Eigenentwicklung. Übernommen wurde dabei allerdings die Idee, Policies als Grundlage der Managemententscheidungen und einen PolicyPool zu benutzen, der alle Policies speichert und verwaltet und auch die Möglichkeit bietet, Policies zu deaktivieren, um sie von der Policy-Auswertung auszuschließen. Jedoch gibt es sonst gravierende Unterschiede bei unserer Implementierung. So benutzt das Management der PG statt nur eines Policytyps eine Reihe von spezialisierten Policies, die mehr Anwendungsmöglichkeiten bieten oder den Gebrauch zumindest vereinfachen. Auf der anderen Seite nutzt das selbst implementierte Management für die Auswertung nur lokale Properties und Informationen, die mit Actions oder Events übertragen werden und ist somit nicht so mächtig, wie die PMAC-Auswertung, die global auf alle Werte zugreifen kann. Im Gegensatz zu einem weiteren Ansatz von IBM – *Autonomic-Computing* genannt – gibt es im Management des *Digital Home* keine Kontroll- und Regelschleife, wie sie der *Autonomic-Manager* benutzt (siehe [IBM05b]). Diese erschien für Systeme mit geringer Leistung (wie z. B. das Personal Device) nicht geeignet.

Der Rahmen eines policybasierten Managements (siehe [Slo94]) soll es allgemein ermöglichen, Aufgaben und Vorgänge einfacher und automatischer zu gestalten. Wenn beispielsweise die konfigurierbaren oder sich wiederholenden Systemabläufe einmal fehlerfrei in Policies spezifiziert sind, lässt sich der Verwaltungsaufwand in einem System reduzieren, da nun vorher möglicherweise manuell durchzuführende Aufgaben selbständig vom Computersystem bzw. vom Management anhand von Policies überprüft und ausgeführt werden können. Das Verhalten der Geräte des *Digital Home* kann also durch die Spezifikation von Policies oder deren Änderung gesteuert bzw. beeinflusst werden.

Das automatisierte Management im Digital Home wird über Management-Komponenten realisiert, die Entscheidungen wie oben schon erwähnt mit Hilfe von Policies treffen. Aus Sicht der Managementfunktionalität werden Management-Komponenten für das Personal-Device (Personal Management) und für übrige Devices (Access-Management und State-Management) unterschieden. Im Digital Home können daher verschiedene Management-Konfigurationen der Devices vorkommen. Dabei gibt es die Unterteilung in folgenden Gruppen:

- Devices ohne Management (z. B. ein Lichtschalter). In dieser Kategorie fallen hauptsächlich kleine Steuergeräte die kein Management benötigen oder aufgrund ihrer beschränkten Ressourcen dieses gar nicht ermöglichen. Falls für diese trotzdem Management notwendig ist, so kann dieses von einem Personal Device (Personal Management) übernommen werden.

- Devices, die nur Access-Management besitzen. Diese ermöglichen dadurch Zugriffbeschränkungen verschiedenster Art.

- Devices, die nur State-Management besitzen (z. B. ein Foxboard). In dieser Klasse gehören Geräte, die bei Zustandsänderungen (Property-Änderungen) Managementfunktionen ermöglichen sollen.

- Devices, die beide Management-Komponenten benötigen – State- und Access-Management (z. B. ein Computer).

- Personal Devices. Diese besitzen stets das Personal Management, welches verschiedene, nutzerspezifische Management-Funktionen anbietet.

Das Management gliedert sich an die eigentliche Anwendung an und agiert nur, wenn bestimmte Anwendungsereignisse oder -zustandsänderungen auftreten (mit Zustand ist hier eine Wertebelegung der Properties gemeint). Es verhält sich also – mit Ausnahme eines zeitgesteuerten Mechanismus zur Auswertung von TimedPolicies – passiv. Seine Reaktionen berechnet das Management aus den Policies, wobei nur der Anwendungszustand und das jeweilige Ereignis Grundlage der Berechnung sind. Informationen über Management-Aktionen aus der Vergangenheit fließen dabei nicht mit ein, und daher kann man das Management als *gedächtnislos* bezeichnen.

## 10.1. Management-Komponenten

Während das Personal-Management jedem Personal Device zugeschaltet ist, sind State- und Access-Management optional. Ihre statischen Schnittstellenmethoden werden auch im ausgeschalteten Zustand immer aufgerufen. Allerdings bleiben diese Aufrufe dann ohne Effekt. Zugeschaltet werden die Management-Komponenten mittels boolescher Flags im Konstruktor eines Devices. Mit der Methode `isActive()` lässt sich zur Laufzeit überprüfen, ob eine Management-Komponente aktiviert wurde.

### 10.1.1. Access-Management

Das Access-Management kann zu jedem Device (außer dem Personal Device) zugeschaltet werden und kann dort die Verwaltung und Umsetzung von Zugriffsrechten realisieren. Wenn ein Device über DPWS-Mechanismen angesprochen wird (Action-Aufruf), wird das Access-Management (`edu.udo.pg490.management.AccessManagement`) befragt. Es prüft, ob der Benutzer die Erlaubnis hat, die Action aufzurufen. Sollte dieses der Fall sein, so wird die Action durchgeführt. Ansonsten (wenn z.B. eine Policy dieses verbietet) wird eine Fehlermeldung zurückgegeben. Der Nutzer wird dabei über die InvokingEntityID aus der Action identifiziert.

### 10.1.2. State-Management

Das State-Management (`edu.udo.pg490.management.StateManagement`) kann zu jedem Device (außer dem Personal Device) zugeschaltet werden und kann dort den Zustand des Devices (Menge aller ServiceProperties) beobachten und abhängig von diesen agieren. Bei jeder Property-Änderung wird das State-Management befragt, das zur Auswertung nur lokale Properties benötigt und prüft, ob Bedingungen über den Properties erfüllt sind. Nach der Policy-Auswertung wird die Änderung lokaler Properties durchgeführt.

### 10.1.3. Personal Management

Das Personal Management (`edu.udo.pg490.management.PersonalManagement`) kapselt die Funktionalität für das Personal Device und kann in verschiedenen Fällen aufgerufen werden:

- Wenn der Timer abläuft, wird eine Action durchgeführt.
- Wenn ein Event eintrifft, wird ebenso eine Action durchgeführt.

- Wenn der Nutzer Services auf Präferenzgeräten aufrufen will, werden die Präferenzgeräte der Reihenfolge nach durchlaufen und der Service auf dem bevorzugten Gerät aufgerufen.

- Wenn lokale Policies geändert werden müssen, steht dem Nutzer die Möglichkeit zur Verfügung sie zu erstellen, zu verändern oder zu löschen.

- Wenn Actionaufrufe fehlschlagen (z. B. eine Fehlermeldung wird als Ergebnis aus dem Access-Management zurückgegeben), wird der Nutzer benachrichtigt, indem er eine textuelle Fehlermeldung auf dem Display vom Personal Device erhält.

- Wenn Property-Änderungen auftreten (z. B. die Location wird geändert), wird die Änderung lokaler Properties durchgeführt.

Das Personal Management liefert als Ergebnis Action-Aufrufe, lokale Property-Änderungen, eine Liste bevorzugter Geräte, eine automatische Konfiguration oder Policy-Änderungen.

#### 10.1.3.1. PolicyTimer

Der PolicyTimer ist ein Teil des PersonalManagements, der es ermöglicht, zeitgetriggerte Policies (TimedPolicies) zu entwerfen (So kann die Zeit spezifiziert werden, wann die Policies ausgewertet werden sollen).

Im PolicyTimer wird eine sortierte Liste verwaltet, die die nächsten Überprüfungen für TimedPolicies enthält. Ein Thread stößt die Auswertung der nächsten Policy in der Liste automatisch an.

Wenn der PolicyTimer gestartet wird, holt sich dieser die im lokalen PolicyPool enthaltenen TimedPolicies, untersucht diese und erstellt die Liste, in der Referenzen auf die Policies der Reihenfolge ihrer nächsten Auswertung nach einsortiert werden. Sofern überhaupt TimedPolicies vorliegen, wird der Thread gestartet, der die sortierte Liste verwaltet und die Wartezeit berechnet, nach der die erste beziehungsweise nächste Policy überprüft werden soll. Es wird abgewartet bis diese Zeit verstrichen ist und daraufhin das PersonalMangement informiert (über die Methode `onTimerExpiration()`), welches dann die PolicyAuswertung anstößt. Danach wird die nächste Prüfzeit berechnet und wieder in die Liste einsortiert.

In der TimedPolicy wird jeweils die Zeit (durch den Datentyp Date) angegeben, zu der die Policy das erste Mal ausgewertet werden soll. Die Interpretation der Startzeit durch den PolicyTimer erfolgt dabei nicht als absolutes Datum, sondern relativ zur aktuellen Zeit. Je nachdem wie es gewünscht ist, kann angegeben werden welche Bestandteile der Startzeit (Tag, Monat) dazu relevant sein sollen. Die weiteren Auswertungen erfolgen periodisch nach der in der TimedPolicy angegebenen Periodenlänge. Das geschieht so lange bis die in der Policy eventuell angegebene Endzeit erreicht wurde. Danach wird ein neuer Startzeitpunkt berechnet.

Mittels der Methode `addPolicy()` wird es ermöglicht, zur Laufzeit weitere Policies durch den PolicyTimer schedulen zu lassen und die Liste der nächsten Prüfzeiten zu akualisieren. Durch die Methode `removePolicy()` wiederum kann der PolicyTimer aufgefordert werden, das Scheduling für die als Parameter angegebene Policy aufzugeben.

Falls während der Laufzeit die Uhr umgestellt wird, wird automatisch die Wartezeit im Thread korrigiert. Wird die Uhr auf einen Zeitpunkt in der Vergangenheit gestellt, startet sich der PolicyTimer zur eventuellen Fehlerbehandlung automatisch neu.

## 10.2. Policy-Auswertung

Die Klasse PolicyEvaluation ist für die Auswertung der Bedingungen (Conditions) von Policies zuständig und wird von den drei Management-Komponenten State-Management, Access-Management und Personal-Management benutzt. Diese rufen Methoden von PolicyEvaluation auf und erhalten daraufhin die Ergebnisse, die in den Policies festgelegt sind und deren Conditions erfüllt sind. PolicyEvaluation wertet dabei jeweils die Conditions von Policies eines bestimmten Typs aus und liefert nur die Ergebnisse gültiger Policies der jeweils höchsten Priorität zurück (hohe Priorität heißt hier: je höher der Wert des Attributs „priority“ im Policyobjekt, desto höher die Priorität). Die weitere Behandlung erfolgt dann jeweils im Access-Management, State-Management oder Personal-Management. Daraus können Actionaufrufe über eine UUID, Namespace und Porttype sowie Events oder Property-Änderungen resultieren.

Als atomare Bedingungen in den Conditions einer Policy dienen zumeist Vergleiche über den Properties der Geräte, Parameter von Actions etc. Es kann beispielsweise geprüft werden, ob diese einem bestimmten Wert entsprechen oder in einer bestimmten Relation zu einem anderen Wert stehen. Solch ein Wert kann direkt in der Condition angegeben sein, kann aber auch der Wert einer anderen Property des gleichen Datentyps sein. Dazu werden in der Policy angegebene Bedingungen mit Schlüssel-Wert-Paaren des aktuellen Gerätezustands oder z.B. den Parametern von Action-Aufrufen abgeglichen. Für die Darstellung der Schlüssel für die Properties der Geräte wurde folgendes Format gewählt: namespace:porttype.propertyname. Außerdem ist es möglich, bei Propertyänderungen den alten Wert mit dem neu zu setzenden zu vergleichen. Dazu wird dem Schlüssel des neu zu setzenden Propertywertes von den Managementkomponenten ein * angehängt, um ihn vom Schlüssel des alten Wertes unterscheiden zu können.

Die atomaren Bedingungen können mit Hilfe logischer Operationen zu komplexeren Bedingungen miteinander verknüpft werden. PolicyEvaluation geht bei der Auswertung der Bedingungen einer Policy davon aus, dass die in den Bedingungen genutzten UND- und ODER-Verknüpfungen eine konjunktive Normalform bilden. Diese Einschränkung wurde ursprünglich gewählt, damit bei komplexen Policy-Bedingungen die Übersicht über die Bedingungen für den Policy-Ersteller und für die, die eine Policy nachvollziehen wollen, gewahrt bleibt (auch zum Beispiel im Hinblick auf eine grafische Benutzerschnittstelle). PolicyEvaluation prüft also Bedingung für Bedingung, ob sie erfüllt ist oder nicht. Diese Teilergebnisse werden nach den Regeln der Logik innerhalb der gesamten komplexeren Bedingung ausgewertet, solange nicht eindeutig feststeht, ob die Bedingung noch erfüllt werden kann oder bereits erfüllt ist.

### 10.2.1. Condition

Jede Policy hat ein Condition-Objekt als Attribut. Es gibt verschiedene Typen von Condition-Objekten. Zur Verfügung stehen dazu Objekte vom Typ KNFCondition, Clause und Literal. In einer KNFCondition wird eine Liste von Clause- oder Literal-Objekten verwaltet, die untereinander logisch UND-verknüpft sein sollen. Auf Grund der oben erwähnten Einschränkung muss einer Policy als Condition-Objekt eine KNFCondition zugewiesen werden. Clause-Objekte enthalten eine Liste von Literal-Objekten, die hier per logischem ODER miteinander verknüpft sein sollen. In Literal-Objekten lassen sich die Werte definieren, die als eigentliche Bedingung auszuwerten sind. So können Werte des Gerätezustands, Actions etc. auf einen bestimmten Wert geprüft, Größenvergleiche durchgeführt oder gestestet werden, ob sich ein Zahlenwert um eine zulässige Abweichung vom angegebenen Wert unterscheidet. Die in Literal-Objekten

angegebenen Bedingungen lassen sich außerdem negieren. Die Conditions lassen sich als Teil einer Policy vom PolicyBuilder erstellen. (Sie können aber natürlich auch separat instanziiert und einer Policy zugewiesen werden.)

In den Literalen können folgende unterschiedliche Typen von Bedingungen definiert werden:

- EqualCondition

  Hier werden Werte miteinander verglichen und geprüft, ob ein Wert "gleich" einem anderen ist.

- GreaterCondition

  Hier werden Werte miteinander verglichen und geprüft, ob ein Wert "größer" als ein anderer ist.

- GreaterEqualCondition

  Hier werden Werte miteinander verglichen und geprüft, ob ein Wert "größer oder gleich" einem anderen ist.

- LessCondition

  Hier werden Werte miteinander verglichen und geprüft, ob ein Wert "kleiner" als ein anderer ist.

- LessEqualCondition

  Hier werden Werte miteinander verglichen und geprüft, ob ein Wert "kleiner oder gleich" einem anderen ist.

- TimeCondition

  Spezifiziert, wann die Condition gültig sein soll: man kann jeweils eine Start- und eine End-Uhrzeit angeben. Die aktuelle Zeit wird dann in der PolicyEvaluation mit diesem Zeitintervall verglichen. Wenn die End-Uhrzeit vor der Start-Uhrzeit liegt, wird angenommen, dass die End-Uhrzeit für den folgenden Tag gilt.

Dabei können die Werte der EqualCondition Elemente folgender Datentypen sein, die auch bei den Properties der Devices von der Klasse ServiceProperty unterstützt werden: Boolean, Integer, String oder Elemente einer StringList. Bei Werten vom Typ Integer können auch die anderen Typen von Bedingungen (abgesehen von der TimeCondition) genutzt werden. Außerdem kann man angeben, falls benötigt, welchen Abstand maximal oder minimal ein Wert von einem anderen haben soll.

## 10.3. PolicyPool

Der Policy Pool verwaltet und speichert die vom Benutzer definierten Policies und PolicyTemplates. Er ist ein Singleton und mit allen Management-Komponenten verbunden.

Beim PolicyPool wird zwischen lediglich gespeicherten (passiven) und aktiven Policies unterschieden. Nur die aktiven Policies einer bestimmten Policyart werden bei einem Aufruf von `getPolicies()` zurückgegeben, welche normalerweise von der Klasse `PolicyEvaluation` aufgerufen wird. Somit werden die gespeicherten, aber nicht aktiven Policies bei der Evaluierung ignoriert. Die Policies lassen sich über die Schnittstelle des PolicyPools aktivieren und

deaktivieren (`activatePolicy()`,`deactivatePolicy()`) und natürlich speichern und löschen. Wird eine Policy gespeichert, wird sie automatisch aktiviert. Zum Speichern benutzt man die Methode `addPolicy()`und zum Löschen `removePolicy()`.

### 10.3.1. Policies

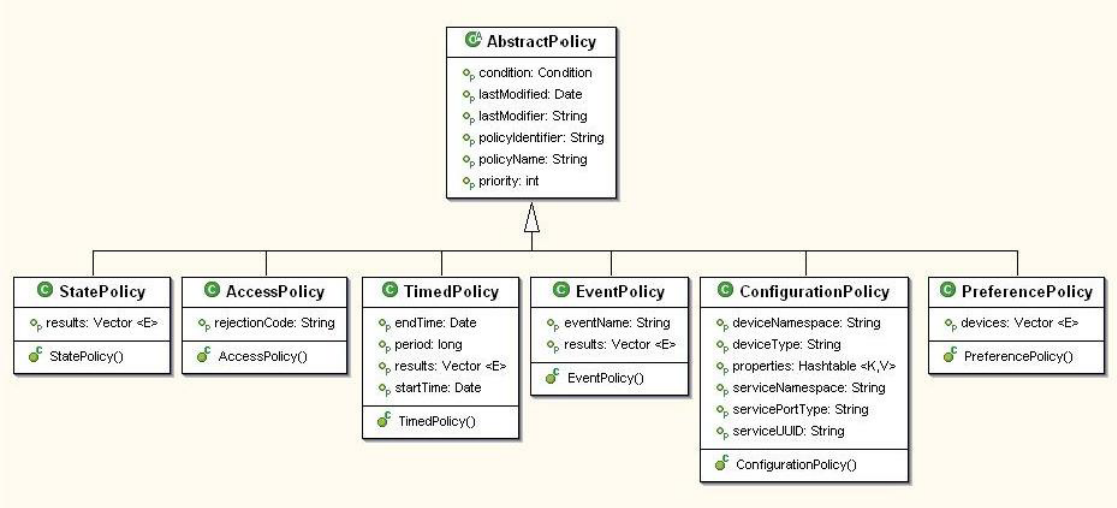


**Abbildung 20:** Struktur der Policies

Die Policies, die *WENN-DANN* Regeln entsprechen, enthalten die Logik des Management-Systems. Sie sollen zuerst vom Programmierer abgelegt werden und später vom Benutzer geändert werden können. Das Klassendiagramm in Abbildung 20 veranschaulicht die Policy-Klassen-Hierarchie.

Alle Policy-Typen erben von der abstrakten Klasse AbstractPolicy, die folgende Einträge / folgenden Aufbau hat:

- condition

  Der WENN-Teil der Policy. Wenn die *Condition* erfüllt ist, wird die *Response* zurückgegeben, wenn die Policy die höchste Priorität aller erfüllten Policies hat. Auf die Condition wird noch weiter unten eingegangen.

- lastModified [optional]

  Datumsangabe, wann die Policy zum letzten Mal geändert wurde

- lastModifier [optional]

  Hier kann der Name des Nutzers angegeben werden, der die Policy erstellt / zum letzten Mal geändert hat. Für die Policy-Auswertung irrelevant.

- policyIdentifier

  Dient dazu, eine Policy nach UUID eindeutig zu identifizieren. Für die Policy-Auswertung irrelevant.

- policyName

  Dient dazu, einer Policy einen sie beschreibenden Namen zu geben, dieser Name ist nicht zwangsläufig eindeutig. Für die Policy-Auswertung ist dieses Element irrelevant.

- priority

  Durch die Priorität wird angegeben, welche erfüllte Policy schließlich ausgewählt wird, nämlich diejenige mit der höchsten Priorität. Bei mehreren Policies mit erfüllter Condition und gleicher Priorität werden alle entsprechenden Results ausgeführt.

Es gibt ein Interface `ResultPolicy`, welche von denjenigen Policies implementiert wird, die auch PolicyResults enthalten. Diese Policies sind: EventPolicy, StatePolicy und TimedPolicy. Das Interface definiert eine get- und eine set-Methode für den PolicyResult-Vector.

Für normale Devices gibt es zwei Arten von Policies:

- StatePolicy

  Policies, die vom State-Management benötigt werden und die bei einer Änderung von Propertywerten evaluiert werden. Eine StatePolicy enthält einen Vector von Result-Objekten. Ein Result kann ein zu feuerndes Event oder auszuführende Actions beschreiben.

- AccessPolicy

  Enthält Policies, die vom Access-Management benötigt werden und beschreiben, wem der Zugriff nicht gestattet wird und mit welcher Begründung (RejectionCode), welcher für gewöhnlich *PERMISSION_DENIED* ist.

Es wird zwischen folgenden Policies vom Personal Device unterschieden:

- TimedPolicy

  Zeitgesteuerte Policies, die durch Angabe von Zeiten eine Policy-Überprüfung starten können. Man kann eine Periode definieren, nach der, ausgehend von der Startzeit, die Überprüfung erneut gestartet wird sowie eine Endzeit, nach der durch die Periode wiederholtes Überprüfen gestoppt wird. Diese Policy enthält ebenso wie eine StatePolicy einen Vector mit Results.

- EventPolicy

  Dieser Policy-Typ wird ausgewertet, wenn ein Event eintrifft. Dafür wird neben der Condition, die ausgewertet werden soll auch der Name des Events mit angegeben. Als Ergebnis hat man auch hier einen Result-Vector.

- ConfigurationPolicy

  Konfigurationseinstellungen für Geräte und ihre Services. Diese Policies werden überprüft, wenn man mit einem PersonalDevice manuell eine Aktion durchführt. Wird eine Aktion jedoch selber durch eine Policy ausgelöst, muss man auf eine automatische Konfiguration verzichten, da hierbei diese Policies nicht ausgewertet werden. Die eigentliche einzustellende Konfiguration wird durch eine Hashtable von Properties angegeben. Man kann festelegen, ob die Konfiguration der Policy für einen Service allgemein (servicePorttype), eine bestimmte Geräteklasse (deviceType) oder für einen einzelnen Service(serviceUUID) eingestellt werden soll. Indem man z.B. gleichzeitig für servicePorttypes und etwa serviceUUIDs Konfigurationen erstellt und die Prioritäten passend wählt,

können Konfigurationen für spezielle Endpunkte gesetzt werden ohne dass diese von Konfigurationen für allgemeinere Endpunkte überschrieben werden. Zu den servicePorttypes bzw. deviceTypes muss der Namespace mit angegeben werden.

- PreferencePolicy

  Diese Policies werden für die Priorisierung von Geräten benutzt. Die Informationen, die man benötigt, um ein Gerät anzusprechen, sind in dem sogenannten PreferenceObject gespeichert. Man kann sagen, dass man ein bestimmtes Gerät (deviceUUID) oder lieber eine bestimmte Geräteklasse (deviceType) ansprechen möchte. Zusätzlich kann man angeben, ob diese Auswahl allgemein gelten soll oder nur, wenn man sich im selben Raum wie das Device aufhält. Dazu muss man im Konstruktor der Klasse PreferenceObject neben der Information zum Gerät das Flag useLocalDevice auf true setzen. Ansonsten bietet die Klasse nur Methoden zum Abfragen der Informationen. Ein deviceType wird dabei durch seinen QualifiedName beschrieben, während eine UUID einfach durch einen String angegeben wird. Die Reihenfolge der Geräte und damit die Präferenz ergibt sich aus der Reihenfolge der PreferenceObjects in dem Vector in der Policy.

Ein Beispiel für die Erstellung einer StatePolicy mit Hilfe des PolicyBuilders sieht man in Quellcode 22. Diese Policy dient der Benachrichtigung im Feuerfall und wird so auch im Anwendungsszenario benutzt, siehe hierfür auch Kapitel 11.2.

```
builder.createNewPolicy(ManagementConstants.POLICY_TYPE_STATE,
    "OnFireShowAlarmSchlafzimmer");
        builder.setPriority(Integer.MAX_VALUE-1);

        builder.createNewLiteral(ManagementConstants.
            EQUAL_CONDITION);
        builder.compareStringWithList("Wohnzimmer", Constants.
            PORTTYPE_FIRE_CENTER_DEVICE, "Fire", true);
        builder.setNegated();
        builder.addToANDLevel();

        builder.createNewLiteral(ManagementConstants.
            EQUAL_CONDITION);
        builder.compareStringWithList("Schlafzimmer", Constants
            .PORTTYPE_FIRE_CENTER_DEVICE, "Fire", true);
        builder.addToANDLevel();

        Hashtable parameters = new Hashtable();
        parameters.put(AbstractTextOutputDeviceService.
            SHOWTEXT_TEXT_INPUT, "Feueralarm:␣Es␣brennt␣im␣
            Schlafzimmer!");
        parameters.put(AbstractTextOutputDeviceService.
            SHOWTEXT_TIMEOUT_INPUT, "0");
        builder.addResult(null,new QualifiedName(Constants.
            PORTTYPE_TEXT_OUTPUT_DEVICE,Constants.NAMESPACE),
            null,null,AbstractTextOutputDeviceService.
            SHOWTEXT_ACTION,parameters);
```

```
try {
        builder.finishPolicy();
} catch (InvalidPolicyException e) {}
```

**Quellcode 22:** Erstellung einer StatePolicy mit Hilfe des PolicyBuilders

## 10.4. PolicyBuilder

Der PolicyBuilder wurde entwickelt, um als API für eine grafische Benutzeroberfläche zu dienen. Er wurde um etliche überladene und zusätzliche Methoden ergänzt, um zum Einen die API auch ohne GUI nutzen zu können, da die GUI niedrigere Priorität hat und wahrscheinlich nicht genug Zeit für ihre Fertigstellung vorhanden ist. Dadurch ist die Anzahl der Methoden sehr hoch und ein bißchen unübersichtlich geworden. Die Erstellung auf API-Ebene ist noch recht unkomfortabel. Zu beachten ist auch, dass durch die Zwischenschaltung dieser Schnittstelle die Erstellung von Policies zwar vereinfacht wird und einige Fehlerquellen ausgeschaltet werden, jedoch gleichzeitig zumindest bei den AccesPolicies die Möglichkeiten sehr beschränkt werden. Hier können nur einfache Policies erstellt werden, die den Zugriff für eine Person erlauben. Komplexere Policies, wie sie z.B. im Anwendungsszenario benutzt werden, können so nicht erstellt werden, dies muss noch ohne API gemacht werden. Sollen hier komplexere Policies, die über eine reine Erlaubnis des Zugriffes einer Person hinaus gehen, erstellt werden, wie z.B. für das Anwendungsszenario, so muss dies direkt gemacht werden oder über Umwege. Je nachdem welche Policy erstellt werden soll, sind unterschiedliche Methodenaufrufe vonnöten. Es werden nun für die jeweilige Policy die notwendigen Methodenaufrufe aufgezählt, ohne dabei auf die Parameter der Aufrufe genauer einzugehen, diese werden im Anhang genau beschrieben(Anhang E.5).

- `AccessPolicy`
  - `createNewPolicy()`
  - `setAccessContent()`
  - `finishPolicy()`
- `ConfigurationPolicy`
  - `createNewPolicy()`
  - `setConfigurationContent()`:
    Es wird eine Hashtable mit Properties benötigt, welche die Konfiguration darstellt
  - `finishPolicy()`
- `EventPolicy`
  - `createNewPolicy()`
  - `setEventName()`
  - `setPriority()`
  - Condition erstellen:
    Wird weiter unten noch genauer erläutert

  - `addResult()`:
    Mindestens einmal aufrufen, kann aber mehrmals aufgerufen werden, um mehrere Results in einer Policy zu haben
  - `finishPolicy()`

- `PreferencePolicy`
  - `createNewPolicy()`
  - `setPriority()`
  - `setPreferenceContent()`:
    Es wird ein Vector mit Preference-Objekten benötigt. Die Reihenfolge der bevorzugten Geräte ergibt sich aus der Reihenfolge der Objekte in dem Vector
  - `finishPolicy()`

- `StatePolicy`
  - `createNewPolicy()`
  - `setPriority()`
  - Condition erstellen:
    Wird weiter unten noch genauer erläutert
  - `addResult()`:
    Mindestens einmal aufrufen, kann aber mehrmals aufgerufen werden, um mehrere Results in einer Policy zu haben
  - `finishPolicy()`

- `TimedPolicy`
  - `createNewPolicy()`
  - `setTimeContent()`
  - Condition erstellen:
    Wird weiter unten noch genauer erläutert
  - `addResult()`:
    Mindestens einmal aufrufen, kann aber mehrmals aufgerufen werden, um mehrere Results in einer Policy zu haben
  - `finishPolicy()`

Um einer Policy eine Condition hinzuzufügen kann die Methode `setCondition()` aufgerufen werden, der eine schon fertige Condition übergeben wird. Ansonsten können Literale mit der Methode `createNewLiteral()` erstellt werden, um daraus die Condition zu erstellen. Der Inhalt eines Literals wird dann mit der Methode `setFirstPropertyToCompare()` und eine der Methoden `setValueToCompareWith(), setSecondPropertyToCompare()` oder `setListToCompareWith()` definiert. Beim setValueToCompareWith-Aufruf wird die Property, welche mit der setFirstPropertyToCompareWith-Methode gesetzt wird, mit einem primitiven Datentyp verglichen. Bei setSecondPropertyToCompare-Aufruf wird diese Property mit einem anderem Property verglichen und beim setListToCompareWith-Aufruf mit einem Wert aus einer Liste. Zu beachten ist bei setFirstPropertyToCompare, dass hier optional ein

Flag `onlyOnPropertyChange` gesetzt werden kann. Ist dieses Flag auf *true* gesetzt, wird ein Property nur betrachtet, wenn es einen neuen Wert erhalten soll. Wird das Property nicht geändert, wird es nicht betrachtet. Soll ein Property betrachtet werden, wenn es nicht geändert werden soll, kann das Flag weggelassen oder auf *false* gesetzt werden. Dadurch kann auch ein neuer Wert eines Properties mit dem ursprünglichen Wert verglichen werden. Es gibt noch die Möglichkeit, das Literal per `setNegated()`-Aufruf zu negieren und wenn ein Property mit einem anderem Property verglichen wird, den Wert des zweiten Properties zu vervielfachen(`setMultiplierForSecondProperty()`). Alternativ zu der Vervielfachung kann man ein Intervall definieren, in dem trotz unterchiedlicher Werte der Properties das Literal doch noch als wahr anzusehen ist(`setDifferenceForSecondProperty()`). Die zwei letzten Möglichkeiten schließen sich aber gegenseitig aus und können nicht in Kombination benutzt werden. Um ein Literal abzuschließen, muss es der Condition hinzugefügt werden. Die Condition liegt in Konjunktiver Normalform vor. D.h. man hat eine konjunktive(UND) Verknüpfung von disjunktiv(ODER) verknüpften Literalen. Ruft man die Methode `addToANDLevel()` auf, so wird das Literal konjunktiv mit eventuell anderen Literalen verknüpft. Wird die Funktion `addToORLevel()` aufgerufen, wird das Literal zu einer Liste von ODER-Verknüpfungen hinzugefügt, welche dann wiederrum UND verknüpft wird. Um eine neue Liste mit ODER-Verknüpfungen zu starten, muss die Methode `addToNewOrLevel()` aufgerufen werden, danach kann wieder normal `addToORLevel()`aufgerufen werden, um Literale zu der neu erzeugten Ebene hinzu zu fügen.

## 10.5. PolicySerializer

Der PolicySerializer wandelt Policy-Objekte in XML um und umgekehrt. Dabei wird jedes Attribut eines Policy-Objekts in ein XMLElement umgewandelt und in die Gesamtstruktur eines XML-Objekts eingefügt. Auf dem umgekehrten Wege wird der XML-Inhalt geparst und die Attribute des Policy-Objekts werden gesetzt. So wird durch die XML-Struktur der Inhalt eines Policy-Objektes repräsentiert. Der PolicySerializer wurde implementiert, weil die MicroEdition von Java keine automatische Serialisierung von Java-Objekten unterstützt. Daher sollte es mittels des PolicySerializers eigentlich ermöglicht werden, Policies persistent abspeichern zu können oder sie per DPWS-Nachrichten an andere Devices zu verteilen. Durch eine Funktionalität im Personal Device ist es möglich, Teile geladener Policies direkt zu bearbeiten und damit Policies zur Laufzeit zu verändern oder neue Policies zu erstellen auf der Basis bereits vorhandener. Damit ergibt sich neben dem Schreiben der Policies im Quelltext des Projekts eine zweite Möglichkeit, Policies zu erstellen. Es stehen verschiedene Methoden zur Verfügung, je nachdem, in welche Richtung die Umwandlung erfolgen soll oder ob z.B. Strings im XML-Format genutzt werden oder die vom DPWS-Stack genutzten XML-Objekte. Außerdem können Policies im XML-Format direkt in den lokalen PolicyPool transferiert oder umgekehrt Policies aus dem PolicyPool in XML-Objekte umgewandelt werden.

Im Listing 23 ist ein Beispiel einer Policy zu sehen, die im XML-Format abgespeichert wurde. Dort kann man die Struktur einer Policy erkennen.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<p0:Policy xmlns:p0="http://policies.management.digitalhome.pg490.udo.
    edu">
```

```
<p0:PolicyType>StatePolicy</p0:PolicyType>
<p0:PolicyName>OnRoomChangeLightsOn</p0:PolicyName>
<p0:LastModifier />
<p0:LastModifiedInMillis>1169922270348</p0:LastModifiedInMillis>
<p0:Priority>8</p0:Priority>
<p0:Condition>
  <p0:AndCondition>
    <p0:Literal>
      <p0:LiteralType>0</p0:LiteralType>
      <p0:ValueType>3</p0:ValueType>
      <p0:FirstString>http://digitalhome.pg490.udo.
          edu:portTypePersonalDeviceControllerService.LocationID*</
          p0:FirstString>
      <p0:SecondString>http://digitalhome.pg490.udo.
          edu:portTypeDialogDevice.LocationID</p0:SecondString>
      <p0:ComparingValue>0</p0:ComparingValue>
      <p0:IsPositive>false</p0:IsPositive>
    </p0:Literal>
    <p0:Literal>
      <p0:LiteralType>5</p0:LiteralType>
      <p0:StartHour>18</p0:StartHour>
      <p0:StartMinute>30</p0:StartMinute>
      <p0:StartSecond>0</p0:StartSecond>
      <p0:StartMillisecond>0</p0:StartMillisecond>
      <p0:EndHour>7</p0:EndHour>
      <p0:EndMinute>0</p0:EndMinute>
      <p0:EndSecond>0</p0:EndSecond>
      <p0:EndMillisecond>0</p0:EndMillisecond>
      <p0:IsPositive>true</p0:IsPositive>
    </p0:Literal>
  </p0:AndCondition>
</p0:Condition>
<p0:PolicyTypeSpecific>
  <p0:Results>
    <p0:Result Type="PolicyResultActionInvocationMultiple">
      <p0:InvokeLocal>true</p0:InvokeLocal>
      <p0:DeviceTypeLocalPart>Lichtschalter</p0:DeviceTypeLocalPart>
      <p0:DeviceTypeNamespace>http://digitalhome.pg490.udo.edu</
          p0:DeviceTypeNamespace>
      <p0:ServicePortTypeLocalPart>portTypeLightDevice</
          p0:ServicePortTypeLocalPart>
      <p0:ServicePortTypeNamespace>http://digitalhome.pg490.udo.edu</
          p0:ServicePortTypeNamespace>
      <p0:Scopes />
      <p0:ScopeMatchingRule />
      <p0:ActionName>setProperty</p0:ActionName>
      <p0:InputParameters>
        <p0:Parameter Name="key">
          <p0:Value>LightSwitchStatus</p0:Value>
```

```
            </p0:Parameter>
            <p0:Parameter Name="value">
              <p0:Value>on</p0:Value>
            </p0:Parameter>
          </p0:InputParameters>
        </p0:Result>
      </p0:Results>
    </p0:PolicyTypeSpecific>
</p0:Policy>
```

**Quellcode 23:** Aufbau einer Beispielpolicy in XML

Der Aufbau orientiert sich stark am Policy-Objekt und seinen Attributen. Es sind Daten wie Policy-Typ, Name, Priorität etc. ersichtlich. In den Unterelementen des XML-Elements Condition wird die Bedingung der Policy genauer spezifiziert. Eine negierte EqualCondition und eine TimeCondition sind hier per logischem UND verknüpft (innerhalb des Elements AND-Condition). Die negierte EqualCondition veranlasst den Vergleich des neuen und alten Werts der Property „LocationID“ eines Service (die „LocationID“ gibt den Standort an) und ist demnach bei einem Raumwechsel erfüllt. In den Unterelementen des Elements „PolicyTypeSpecific“ sind die für den aktuellen Policy-Typ relevanten Daten festgehalten. Bei dem Beispiel handelt es sich also um eine StatePolicy mit dem Namen „OnRoomChangeLightsOn“, die beim Raumwechsel dafür sorgt, dass im Raum, der betreten werden soll, automatisch das Licht eingeschaltet wird (durch Aufrufen der Action „setProperty“, die hier die Property “LightSwitchStatus“ auf „on“ setzt), falls es zwischen 18:30 Uhr und 7 Uhr ist.

## 10.6. PolicyResult

Einige Policy-Typen (StatePolicy, TimedPolicy und EventPolicy) liefern nach deren Auswertung das Ergebnis (ein oder mehrere), das in der Klasse **ResultPerformer** (edu.udo.pg490.management.policies.result.ResultPerformer) ausgeführt wird. Die wichtigste Methode dieser Klasse ist **performResults()** und sie wird nach der Policy-Auswertung aufgerufen.

Es gibt noch vier zusätzliche Hilfsklassen, die sich auch im gleichen Package befinden und in dieser Methode benutzt werden:

- **PolicyResultPropertyChange** ermöglicht es, als Ergebnis der Auswertung von Policies eine Property zu ändern. Der Konstruktor dieser Klasse erwartet als Parameter einen Propertynamen, Propertywert, Servicenamen und einen Serviceporttyp.
- **PolicyResultEvent** ermöglicht es, als Ergebnis der Auswertung von Policies ein Event zu feuern. Der Konstruktor dieser Klasse erwartet als Parameter einen Ereignisnamen, Servicenamen, Serviceporttyp und eine Hashtabelle mit Ausgabeparametern.
- **PolicyResultActionInvocationSingle** ermöglicht es, als Ergebnis der Auswertung von Policies eine Action bei einem Service eines eindeutigen Devices aufzurufen. Der Konstruktor dieser Klasse erwartet als Parameter einen Actionnamen, Servicenamen, Serviceporttyp, eine UUID von einem Gerät und eine Hashtabelle mit Eingabeparametern.
- **PolicyResultActionInvocationMultiple** ermöglicht es, als Ergebnis der Auswertung von Policies eine Action bei mehreren Devices, die durch deviceType oder scopes angegeben werden, aufzurufen. Man kann hier ein Flag setzen, wenn nötig ist, dass die

Ausführung nur im aktuellen Raum durchgeführt werden muss. Hier können zwei Konstruktoren benutzt werden, die verschiedene Parameter erwarten:

```
public PolicyResultActionInvocationMultiple(QualifiedName deviceType, Qua-
lifiedName servicePortType, String actionName, Hashtable inputParameters, -
Vector scopes, String scopeMatchingRule)
```

```
public PolicyResultActionInvocationMultiple(QualifiedName deviceType, Qua-
lifiedName servicePortType, String actionName, Hashtable inputParameters, -
boolean invokeLocal)
```

Die genannten Policies implementieren auch ein Interface **ResultPolicy**, das sich im gleichen Package befindet und lediglich eine get- und eine set-Methode für den PolicyResult-Vector definiert.

# 11. Testszenario

Das Testszenario läuft mit drei Personen (Mutter, Vater und Sohn) in zwei Zimmern (Wohn- und Schlafzimmer) ab. Dabei wurde darauf geachtet, dass die Funktionalität des Managements getestet und demonstriert werden kann.

## 11.1. Ausgangssituation und Gegebenheiten

Jede Person im DigitalHome hat eine Reihenfolge von bevorzugten Geräten. So bevorzugt der Vater den Fernseher im Wohnzimmer, wenn der nicht verfügbar ist, kommt der PC an der zweiten Stelle und zuletzt kommt der Fernseher im Schlafzimmer. Die Mutter will immer den Fernseher im aktuellen Raum für die Wiedergabe benutzen. Der Sohn hat wie der Vater die gleiche feste Reihenfolge. Der Fernseher wird auch selber nach folgenden Zugriffsrechten verwaltet: Die Mutter besitzt die höchste Zugriffspriorität danach kommt der Vater in der zweiten Position und der Sohn hat die niedrigste Stufe.

Die Wiedergabeeinstellungen werden auch für jeden Benutzer gespeichert und bei jeder Wiedergabe automatisch geladen:

- Vater: Helligkeit: 60, Kontrast: 55
- Mutter: Helligkeit: 50, Kontrast: 60
- Sohn: Helligkeit: 55, Kontrast: 50

Die vorprogrammierten Sendungen bzw. Filme sind in der folgenden Tabelle dargestellt:

| ***Person*** | ***Sendung*** | ***Dauer*** |
|---|---|---|
| Mutter | Film M | 2 Stunden |
| | Sendung M1 | 18:00-21:00 |
| | Sendung M2 | 22:00-24:00 |
| Vater | Film V | 1 Stunde |
| | Sendung V1 | 20:00-22:00 |
| Sohn | Film S1 | 1 Stunde |
| | Film S2 | 2 Stunden |
| | Sendung S1 | 20:30-23:00 |

**Tabelle 5:** TV-Programme

Damit Konflikte bei der Ausführung des Testszenarios gelöst und auch alle Management-Komponenten abgedeckt werden können, wurden folgende Regeln festgelegt:

- Das Licht Kann zwischen 18:30 Uhr und 7:00 Uhr eingeschaltet werden
- Filme werden manuell gestartet
- Der Sohn darf nicht nach 22:00 Uhr im Wohnzimmer fernsehen
- Eine TV-Sendung wird aufgenommen wenn das Wiedergabegerät nicht verfügbar ist

## 11.2. Policies

Für das Testszenario wurden mehrere Policies geschrieben, um die Funktionalität des Managements testen und demonstrieren zu können. Die Policies dienten dabei verschiedenen Zwecken:

### Steuerung der Beleuchtung

Zum automatisierten Ein- und Ausschalten des Lichtes werden drei Policies benutzt. Zwei davon befinden sich in jedem Personal-Device. Die erste Policy schaltet beim Raumwechsel eines Personal-Devices zwischen 18.30 und 7.00 Uhr das Licht im neuen Raum ein. Die zweite schaltet das Licht um 18.30 Uhr in dem Raum ein, in dem sich das Personal-Device der ausgewerteten Policy befindet. Eine weitere Policy wurde in den beiden PolicyPools der Foxboards eingefügt. Wenn alle Personen den Raum verlassen haben, sorgt diese dafür, dass das Licht ausgeschaltet wird.

### Konfiguration und Geräteauswahl

Für die Konfiguration werden in den Policies einfach ServiceProperties gespeichert, welche den Namen der zu konfigurierenden Einstellung samt Wert enthalten. Für die Auswahl von bevorzugten Geräten für Vater und Sohn werden deren *UUID*s in den entsprechenden Policies gespeichert. Für die Mutter wird im Preference-Object für die Policy der *DevicePorttype* Fernsehen angegeben, zusammen mit dem Flag *useLocalDevice*, so dass gemäß Anwendungsszenario nur der Fernseher, der im gleichen Raum wie die Mutter ist, angesprochen wird.

### Zugriffskontrolle

Die Policies für die Zugriffskontrolle liegen auf den jeweiligen Geräten, auf denen der Zugriff beschränkt werden soll. Die erste Policy überprüft, ob die ID des Aufrufers gleich der des Sohnes ist und ob es nach 22 oder vor 15 Uhr ist und verbietet den Zugriff, wenn diese Bedingungen erfüllt sind. Die zweite Policy ist etwas komplexer und regelt die Prioritäten der Nutzer für den VideoOutput-Service, d.h. ein neuer Benutzer mit höherer Priorität kann einen schon vorhandenen Nutzer mit niedrigerer Priorität vom Gerät verdrängen, nicht jedoch umgekehrt. Gleichzeitig werden andere Services jedoch nicht beschränkt, so dass etwa der TextOutut-Service oder der Recorder-Service von jedem benutzt werden kann. Zum Beispiel wird der TextOutput-Service bei einem Feueralarm aufgerufen. Man könnte hier natürlich den Zugriff per Policy so konfigurieren, dass die Familie und die Feuerzentrale Zugriff auf genau diesen Service haben. Darauf wurde an dieser Stelle jedoch verzichtet.

Hier die Policy im Detail:
*Wenn*
$\neg(Aufrufer = Mutter)$
$\wedge\ (\neg(Aufrufer = Vater) \vee (aktuellerBenutzer = Mutter))$
$\wedge\ (\neg(Aufrufer = Sohn) \vee (aktuellerBenutzer = Mutter) \vee (aktuellerBenutzer = Vater))$
$\wedge\ (ServicePorttype = VideoOutput)$
*Dann:* $ACCESS\_DENIED$

### Steuerung der Multimediageräte

Um zu einer bestimmten Zeit eine Sendung wiederzugeben, oder sie aufzunehmen, werden TimedPolicies benutzt. Sie befinden sich in PersonalDevices.
Die Policy „V1_WZ_Aufnahme_20:00“ programmiert um 20 Uhr den Festplattenrekorder im Wohnzimmer so, dass der Vater seine Sendung aufnehmen kann. Dafür werden als Eingabe folgende Inputparameter erwartet: „startTime“, „endTime“, „date“, „program“ und „name“.
Zwei weitere Policies „S1_SZ_Wiedergabe_20:30“ und „M2_SZ_Wiedergabe_22:00“ sind von der Konstruktion her ähnlich und unterschieden sich nur in den Orten, wo sie gespeichert werden. Beide Policies schalten zu einer bestimmten Zeit (um 20:30 Uhr für den Sohn und um 22 Uhr für die Mutter) den Fernseher mit einer Lieblingssendung im Schlafzimmer ein. Es werden nur die Nummer der Sendung (Mediadateien sind in der alphabetischen Reihenfolge sortiert und die Nummerierung fängt bei „0“ an) und eine UUID des Fernsehers im Schlafzimmer angegeben. Wenn Events vom DPWS-System eintreffen, können EventPolicies ausgewertet werden. Die Beispielpolicy, deren Auswertung durch das Eventing getriggert wird, befindet sich im PersonalDevice der Mutter. Es handelt sich um die Policy „MutterInRaumAb1800“. Diese Policy bewirkt, falls die Mutter im Zeitrahmen zwischen 18 und 21 Uhr das Wohnzimmer betritt, ihre tägliche Lieblingssendung M1 auf dem Fernseher im Wohnzimmer angezeigt wird. Es wird dabei das vom RoomControlDeviceService empfangene Event genutzt, dass die Mutter das Wohnzimmer betreten hat, um die Auswertung der Policy anzustoßen.

### Alarmierung im Falle eines Brandes

Für die Benachrichtigung bei einem Feueralarm wurden vier Policies geschrieben, die je nachdem, wo es brennt bzw. ob das Feuer gelöscht wurde, auf allen erreichbaren TextOutPut-Geräten eine Benachrichtigung der Bewohner veranlassen. Bei den Policies handelt es sich um StatePolicies, die im FireCenter abgelegt werden. So wird jeweils bei Änderung der Property „Fire“ ihr Inhalt geprüft, der angibt, wo oder ob ein Feuer im Digital-Home vorhanden ist.

## 11.3. Ablauf

Der genaue Ablauf des Testszenarios findet sich in der Tabelle in Abbildung 22. Um den Gesamtablauf besser nachvollziehen zu können, zeigt Abbildung 21 ein Sequenzdiagramm des Ablaufs eines Teilstücks des Szenarios.

In Schritt 1 des Diagramms betritt der Vater den Raum. Dies erkennt das Foxboard und sendet ein Event, welches den Namen des Raums und die UUID des Personal Device des Vaters beinhaltet. Das Personal Device des Vaters empfängt dieses Event und weiß somit nun, dass es sich in einem neuen Raum befindet. Nun möchte der Vater einen Film sehen und startet die entsprechende Aktion auf seinem Personal Device (Schritt 2). Im Zuge dieser Aktion muss nun zunächst der gewünschte Videostream gefunden werden. Dazu befragt das Personal Device alle ihm bekannten *VideoInputDevices* und fordert eine Liste der jeweils verfügbaren Streams an (Schritte 3 und 4).

Wurde der gesuchte Stream gefunden, muss dieser nun auf dem bevorzugten Gerät des Vaters ausgegeben werden. Dazu wird das *PersonalManagement* damit beauftragt, den entsprechenden Service-Aufruf auf dem bevorzugten Gerät auszuführen (Schritt 5). Das *PersonalManagement* wertet die Policies aus und ruft dann den *VideoOutputService* des Zielgeräts auf (Schritt 6). Als Argument erhält das Zielgerät dabei eine Referenz auf das in den Schritten 3 und 4 bestimmte Quellgerät des Videostreams.

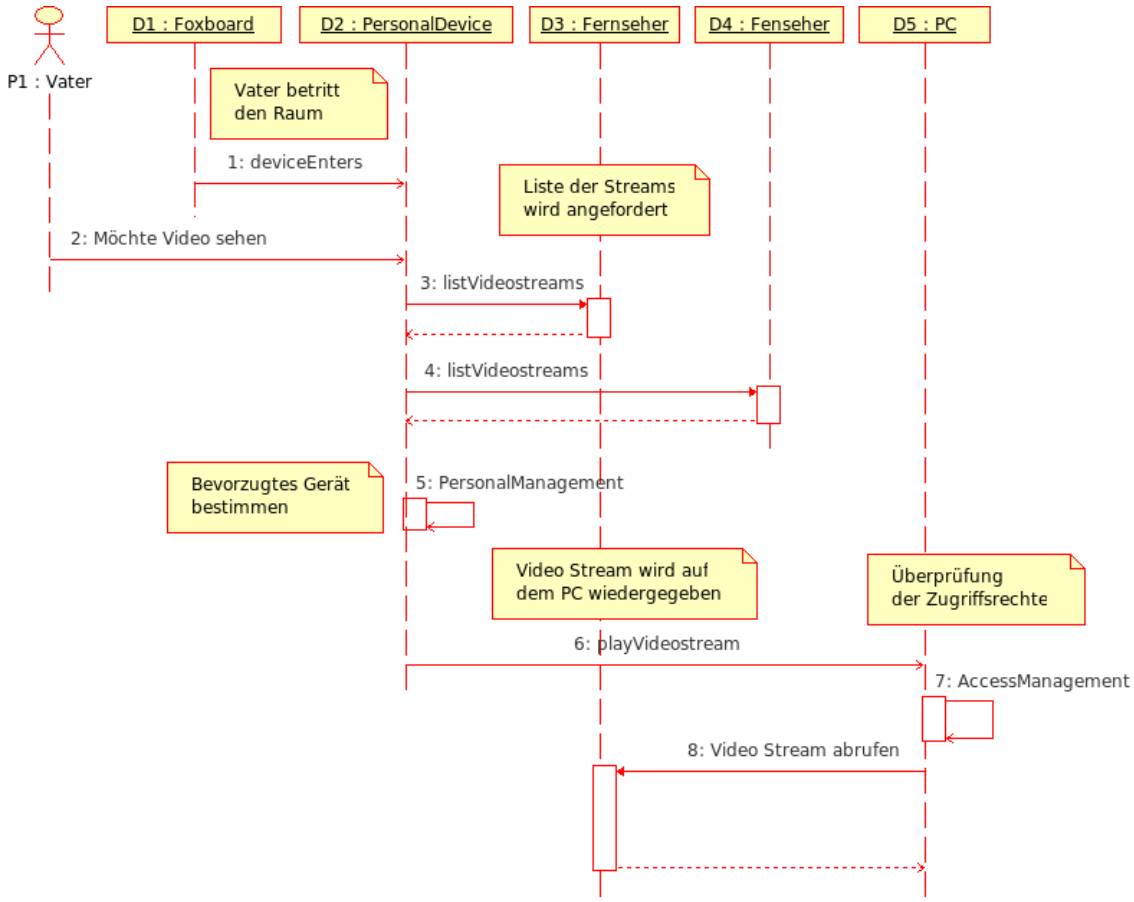


**Abbildung 21:** Abspielen eines Videos

Sobald der Webservice Aufruf (Schritt 6) beim PC eingetroffen ist, wird das *AccessManagement* befragt, um die Erlaubnis für den Aufruf zu prüfen (Schritt 7). Wird der Aufruf als autorisiert eingestuft, kann der PC mit der Wiedergabe des Streams beginnen. Dazu greift er dann, wie in Schritt 8 gezeigt, auf den Videostream zu, der durch den Fernseher (Gerät D3) bereitgestellt wird.

## 11.4. Schwierigkeiten

Für die Durchführung des Testszenarios wurde die Lokalisierung mittels Foxboard realisiert. Dabei ist man auf das Problem gestoßen, dass es schwierig war, den genauen Aufenthaltsort zu bestimmen. Die Stärke des Bluetoothsignals hat das Problem verursacht, so war es möglich für das persönliche Gerät sich auch an das Foxboard in dem benachbarten Zimmer anzumelden. Um das Problem zu lösen wurde auf das simulierte Foxboard umgestiegen, um die Lokalisierung zu realisieren. So konnte der Raumwechsel simuliert werden und die für den Testszenario geschriebenen Policies getestet werden.

| Zeit | Wohnzimmer | | Schlafzimmer | |
|---|---|---|---|---|
| | *Ereignis* | *Aktion* | *Ereignis* | *Aktion* |
| **18:00** | - Mutter kommt rein | - Sendung M1 auf TV WZ wiedergeben | - Sohn kommt rein | - Film S auf TV SZ wiedergeben |
| **18:30 (Es wird dunkel)** | - Mutter ist im Zimmer | - Licht einschalten | - Sohn ist im Zimmer | - Licht einschalten |
| **19:00** | - Vater kommt rein | - TV ist besetzt<br>-Film V aufs PC abspielen | | |
| **20:00** | - TV ist besetzt von M | - Sendung V1 wird aufgenommen | - Ende von Film S | |
| **20:05** | | | - Sohn verlässt das Zimmer | - Licht ausschalten |
| **20:15** | | | - Feuermeldung | - Brandschutzzentrale sendet Alarme auf allen verfügbaren Geräten |
| | - Feueralarme | - Alarm wird angezeigt | | |
| | - M und V verlassen Zimmer | - Licht ausschalten | - M und V gehen ins Zimmer (Feuer löschen) | - Licht einschalten |
| **20:20** | | | - Feuer gelöscht | |
| | | | - M und V gehen raus | - Licht ausschalten |
| | - M und V sind im Zimmer | - Licht einschalten | | |
| **20:25** | | | - Sohn kommt rein | - Licht einschalten |
| **20:30** | | | | - Sendung S1 auf TV SZ wiedergeben |
| **21:00** | - Ende von M1 | | | |
| | - Vater und Mutter gehen raus | - Licht ausschalten | | |
| **22:00** | | | - Mutter kommt rein | - Sendung S1 Stoppen<br>- Sendung M2 auf TV SZ wiedergeben |
| **22:05** | - Sohn kommt rein | - Licht einschalten<br>- TV wird nicht eingeschaltet | | |
| **24:00** | | | - Ende von M2 | |

**Abbildung 22:** Ablauf vom Testszenario

# 12. Fazit

Wir standen vor der Aufgabe, ein automatisiertes Managementsystem für eine verteilte Anwendung auf der Basis von Webservices zu entwickeln. Die Vision des *Digital Home* bot uns die Möglichkeit den Ansatz des policybasierten Managements von Webservices in der Praxis zu erproben. Das *Digital Home* als hoch dynamische Umgebung mit interagierenden Services ist ein Anwendungsfeld, das durch seine Komplexität verschiedener Arten des Managements bedarf.

Im Laufe der Arbeit konnten alle gesteckten Ziele erreicht werden, jedoch wurden an manchen Stellen auch zeitliche Grenzen erreicht. Die Arbeit mit Bluetooth und dem Foxboard zum Beispiel konnte nicht in absolut zufriedenstellender Weise abgeschlossen werden. Zwar wurde eine komplette Java Bluetooth API entwickelt, jedoch traten Probleme auf, die nicht gelöst werden konnten. So ist zum Beispiel der RSSI Wert nicht standardisiert, und von der verwendeten Rechnerkonfiguration abhängig, und auch der Linux Bluetooth Stack, vor allem der für das Projekt Digital Home wichtige *utils-Teil*, ist nicht frei von Fehlern. Im Rahmen einer solchen Projektarbeit muss aber an gewissen Stellen eine Grenze gezogen werden, und vorhandene Technologie genutzt werden können. Wenn diese aber fehlerhaft ist, liegt das außerhalb des Bereiches einer Projektgruppe. Ebenso problematisch im Rahmen einer Projektgruppe sind Probleme, die erst kurz vor Ende offenbar werden. Konkret im Falle der PG490 war dieses beispielsweise bei der Verwendung des Foxboards der Fall. Die Entwicklung der KVM mit Bluetooth Funktionalität und des Services für die Raumerkennung wurde erst kurz vor Ende der PG abgeschlossen. Bei dem Test von allen Komponenten auf dem Foxboard stellte sich heraus, dass insgesamt resourcenschonender programmiert werden muss, da der Minicomputer anscheinend etwas überfordert war mit unserer Implementierung. Die darauf folgende Verbesserung konnte dann leider nicht mehr ausreichend genug getestet werden, so dass letztendlich auf simulierte Foxboards zurückgegriffen werden musste. Insgesamt ist hier also eine Problematik entstanden, die von uns am Anfang des Projektes nicht zu überblicken war. Die Probleme sind jedoch, wie unser Kurztest zeigte, lösbar, bedürfen jedoch ein wenig mehr Zeit und standen auch nicht im Hauptfokus unseres Projektes. Hier bestehen sicherlich noch Möglichkeiten für weitere Gruppen, wobei die Themenstellung dann auch sicherlich eine gänzlich andere wäre und die Technik mehr im Blickpunkt stehen sollte.

Die Realisierung des Managements wurde so weit entwickelt, dass von einer soliden Basis für eine große Zahl an Managementaufgaben gesprochen werden kann. Die Managementunterstützung für die in den Vorüberlegungen aufgestellten Anwendungsfälle wurde vollständig und funktionstüchtig fertiggestellt. Zur vereinfachten Erstellung der Policies wurde der PolicyBuilder entwickelt. Hier handelt es sich allerdings um ein Werkzeug, das immer noch auf Programmierebene Policies erstellt. Eine einfache zu bedienende, grafische Oberfläche, die auf dem PolicyBuilder aufsetzt, wäre nötig, um auch Nutzern des *Digital Home*, die keine Programmierkenntnisse besitzen, eine Policyerstellung zu ermöglichen. Diese grafische Oberfläche müsste dann aus Gründen der Benutzerfreundlichkeit auf einem PC laufen, so dass es notwendig wäre einen Mechanismus zu implementieren, der Policies in entfernte PolicyPools überträgt bzw. von dort ausliest. Dieser Mechanismus wurde aus zeitlichen Gründen nicht implementiert. Es wurde jedoch eine wichtige Vorraussetzung für ihn geschaffen. Da es unter JavaME nicht einfach möglich ist (wie etwa in JavaSE) Objekte zu serialisieren (und dann auf entfernte Devices zu übertragen), wird eine andere Serialisierungsfunktion benötigt. Diese kann der PolicySerialzier übernehmen, der Policies in eine XML-Form bringt.

Insgesamt stellt es sicherlich eine interessante Aufgabenstellung dar, das ganze Projekt *Di-*

*gital Home* von der simulierten Variante zu übertragen auf echte Geräte. Einen ersten Ansatz haben wir schon mit der Nutzung des Foxboards getan, da hiermit gezeigt wurde, dass mit dem nötigen Aufwand eingebettete Systeme auch in komplexen Einsatzgebieten zurecht kommen. Auch das zweite von uns eingesetzte Gerät, das Motorola Handy A780 bietet weitere Ansatzpunkte, um ein solches Projekt von der Simulation in die Realität zu bringen. Dabei ist besonders interessant, dass es sich hierbei um ein handelsübliches Gerät handelt, welches nur durch Programmierung und ohne technischen Eingriff so umgeändert werden kann, dass es die Funktion des Personal Devices übernehmen kann. Abschließend kann gesagt werden, das mit unserer Motivation und ein wenig mehr Zeit, ein noch komfortableres und stabileres System entstanden wäre, welches außerhalb des universitären Umfeldes bestehen könnte.

# A. Spezifikation der abstrakten Webservices der Gerätekategorien

In diesem Kapitel werden die abstrakten Klassen der Webservices der Gerätehierarchie und ihre zum Aufruf benötigten Parameter dokumentiert.

## A.1. Device

***Oberkategorie(n):***

-

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *LocationID* | Nummern des Raumes bzw. des nächstgelegenen Access Points |
| *Status* | IDLE, RUNNING, STOPPED oder ERROR |

**Tabelle 6:** Device-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| getProperty | String key | String value |
| setProperty | String key<br>String value | - |
| getProperties | - | propertyOutputList |
| setProperties | propertyInputList | - |

**Tabelle 7:** Device-Actions

## A.2. Class AbstractVisualOutputDeviceService

Declaration

```
public abstract class AbstractVisualOutputDeviceService
extends edu.udo.pg490.services.AbstractDeviceService
```

### A.2.1. Class AbstractDialogDeviceService

Declaration

```
public abstract class AbstractDialogDeviceService
extends edu.udo.pg490.services.AbstractVisualOutputDeviceService
```

| Properties | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *Brightness* | aktuelle Helligkeit der Anzeige |
| *MinBrightness* | Minimale Helligkeit der Anzeige |
| *MaxBrightness* | Maximale Helligkeit der Anzeige |
| *Contrast* | Aktueller Kontrast der Anzeige |
| *MinContrast* | Minimaler Kontrast der Anzeige |
| *MaxContrast* | Maximaler Kontrast der Anzeige |
| *ScreenWidth* | Breite der Anzeige |
| *ScreenHeight* | Höhe der Anzeige |

**Tabelle 8:** VisualOutputDevice-Properties

| Actions | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| - | - | - |

**Tabelle 9:** VisualOutputDevice-Actions

| Properties | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| - | - |

**Tabelle 10:** DialogDevice-Properties

| Actions | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| showDialog | String message,<br>int timeout,<br>String type | - |

**Tabelle 11:** DialogDevice-Actions

| Events | |
|---|---|
| ***Event*** | ***Parameter*** |
| dialogAnswer | String answer |

**Tabelle 12:** DialogDevice-Events

Methods

- `protected void` **antwortFeuern**`( java.lang.String` **answer**`)`

- `protected abstract void` **zeigeDialog**`( java.lang.String` **message**`, java.lang.String` **type**`, java.lang.String` **timeout**`)`

### A.2.2. Class AbstractVideoOutputDeviceService

Declaration

public abstract class AbstractVideoOutputDeviceService
**extends** edu.udo.pg490.services.AbstractVisualOutputDeviceService

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *Volume* | Aktuelle Lautstärke |
| *MaxVolume* | Maximale Lautstärke |
| *MinVolume* | Minimale Lautstärke |

**Tabelle 13:** VideoOutputDevice-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| playVideostream | String smilData | - |
| stopVideostream | - | - |

**Tabelle 14:** VideoOutputDevice-Actions

Methods

- `public abstract void` **smilDatenVerarbeiten**`( java.lang.String` **smilData**`)`
  - **Usage**
    - Diese Methode muss in der Klasse der konkreten Implementierung überschrieben werden, um die Verarbeitung der im Parameter Übergebenen URL an das jeweilige Gerät anzupassen, sodass ein Video angezeigt werden kann.
  - **Parameters**
    - `smilData` - Eine URL, die auf den Ort eines Videos des Video-Input-Devices verweist

- `protected abstract boolean` **stoppeVideowiedergabe**`( )`
  - **Usage**
    - Diese Methode muss in der Klasse der konkreten Implementierung überschrieben werden. Sie stoppt die Videowiedergabe auf dem Device.
  - **Returns** - Gibt `true` zurück, wenn die Videowiedergabe gestoppt wurde.

### A.2.3. Class AbstractTextOutputDeviceService

#### Declaration

public abstract class AbstractTextOutputDeviceService
**extends** edu.udo.pg490.services.AbstractVisualOutputDeviceService

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *CurrentTextSize* | Aktuelle Schriftgröße |
| *MaxTextSize* | Maximale Schriftgröße |
| *MinTextSize* | Minimale Schriftgröße |
| *Font* | Aktuell verwendete Schriftart |
| *BackgroundColor* | Hintergrundfarbe des Textes |
| *FontColor* | Schriftfarbe |

**Tabelle 15:** TextOutputDevice-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| showText | String text,<br>int timeout | - |

**Tabelle 16:** TextOutputDevice-Actions

#### Methods

- `public abstract void` **setBground**`( java.lang.String` **arg0**`)`
- `public abstract void` **setFground**`( java.lang.String` **arg0**`)`
- `public abstract void` **setTxt**`( java.lang.String` **text**`)`

- `public abstract void` **setTxtSize**`( int` **TextSize**`)`

---

- `public abstract void` **timeout**`( int` **num_sec**`)`

## A.3. Class AbstractAudioInputDeviceService

### Declaration

public abstract class AbstractAudioInputDeviceService
**extends** edu.udo.pg490.services.AbstractDeviceService

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| - | - |

**Tabelle 17:** AudioInputDevice-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| sendAudiostream | String identifier,<br>String AudioOutput-UUID | String smilData |
| createAudiostreamURL | String identifier | String audioURL |
| listAudiostreams | - | audioliste |

**Tabelle 18:** AudioInputDevice-Actions

### Methods

- `protected abstract InputStream` **getInputStream**`( java.lang.String` **streamID**`)`

  - **Usage**
    - Diese Methode gibt für einen Identifikationswert ein Objekt der Klasse Input-Stream zurück.
  - **Parameters**
    - `streamID` - Ein String, der in Abhängigkeit von der konkreten Implementierung einen Audiostream eindeutig identifiziert
  - **Returns** - Gibt ein Objekt der Klasse InputStream zurück.
  - **Exceptions**
    - `java.io.IOException` - Wird geworfen, wenn der Eingabeparameter keinen existierenden Stream bezeichnet.

- `protected abstract String` **getStreamList**`( )`
  - **Usage**
    - Diese Methode gibt eine Liste von Audiostreams zurück.
  - **Returns** - Gibt ein String-Array zurück, das eine Auflistung von Audiostreams enthält.

## A.4. Class AbstractAudioOutputDeviceService

### Declaration

public abstract class AbstractAudioOutputDeviceService
**extends** edu.udo.pg490.services.AbstractDeviceService

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *Volume* | Aktuelle Lautstärke |
| *MaxVolume* | Maximale Lautstärke |
| *MinVolume* | Minimale Lautstärke |

**Tabelle 19:** AudioOutputDevice-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| playAudiostream | String smilData | - |
| stopAudiostream | - | - |

**Tabelle 20:** AudioOutputDevice-Actions

### Methods

- `protected abstract void` **smilDatenVerarbeiten**`( java.lang.String` **smilData**`)`
  - **Usage**
    - Diese Methode muss in der Klasse der konkreten Implementierung Überschrieben werden, um die Verarbeitung der im Parameter übergebenen URL an das jeweilige Gerät anzupassen, sodass eine Audiodatei abgespielt werden kann.
  - **Parameters**
    - `smilData` - Eine URL, die auf den Ort einer Audiodatei des Audio-Input-Devices verweist

- `protected abstract void` **stoppeAudiowiedergabe**`( )`
  - **Usage**
    - Diese Methode muss in der Klasse der konkreten Implementierung überschrieben werden. Sie stoppt die Audiowiedergabe auf dem Device.

## A.5. Class AbstractVideoInputDeviceService

### Declaration

public abstract class AbstractVideoInputDeviceService
**extends** edu.udo.pg490.services.AbstractDeviceService

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| - | - |

**Tabelle 21:** VideoInputDevice-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| sendVideostream | String identifier,<br>String VideoOutput-UUID | String smilData |
| createVideostreamURL | String identifier | String videoURL |
| listVideostreams | - | videoliste |

**Tabelle 22:** VideoInputDevice-Actions

### Methods

- `protected abstract InputStream` **getInputStream**`( java.lang.String` **streamID**`)`
  - **Usage**
    - Diese Methode gibt für einen Identifikationswert ein Objekt der Klasse InputStream zurück.
  - **Parameters**
    - `streamID` - Ein String, der in Abhängigkeit von der konkreten Implementierung einen Videostream eindeutig identifiziert
  - **Returns** - Gibt ein Objekt der Klasse InputStream zurück.
  - **Exceptions**

  * `java.io.IOException` - Wird geworfen, wenn der Eingabeparameter keinen existierenden Stream bezeichnet.

---

- `protected abstract String` **getStreamList( )**
  - **Usage**
    * Diese Methode gibt eine Liste von Videostreams zurück.
  - **Returns** - Gibt ein String-Array zurück, das eine Auflistung von Videostreams enthält.

## A.6. Class AbstractRecorderDeviceService

### Declaration

public abstract class AbstractRecorderDeviceService
**extends** edu.udo.pg490.services.AbstractDeviceService

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| - | - |

**Tabelle 23:** RecorderDevice-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| setTimer | String startTime<br>String endTime<br>String date<br>String program<br>String name | - |
| getTimers | - | timerList |
| deleteTimer | String name | - |

**Tabelle 24:** RecorderDevice-Actions

### Methods

- `protected abstract Calendar` **getCalendar( )**
  - **Usage**

* Diese Methode gibt eine Instanz der Klasse Calendar zurück. Ein Calendar-Objekt muss deshalb über diese Methode geholt werden, weil sonst in konkreten Implementierungen dieser Klasse auf die Klasse `java.util.Calendar` des JavaSE-Programmpakets zurückgegriffen würde, die sich in wesentlichen Punkten von der Klasse Calendar des JavaME-Pakets unterscheidet, wodurch ein AbstractMethodError geworfen würde.

– **Returns** - Gibt jeweils eine neue Instanz der Klasse Calendar zurück.

## A.7. Class AbstractLightDeviceService

### Declaration

public abstract class AbstractLightDeviceService
**extends** edu.udo.pg490.services.AbstractDeviceService

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *Brightness* | aktuelle Helligkeit |
| *MinBrightness* | Minimale Helligkeit, die gemessen werden kann |
| *MaxBrightness* | Maximale Helligkeit, die gemessen werden kann |
| *LightSwitchStatus* | Status des Schalters |

**Tabelle 25:** LightDevice-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| - | - | - |

**Tabelle 26:** LightDevice-Actions

### Methods

- `public abstract void` **newBrightness**`( int` **value**`)`

---

- `public abstract void` **newLightSwitchStatus**`( boolean` **status**`)`

---

- `public abstract void` **newLocationID**`( java.lang.String` **ID**`)`
  - **Usage**
    * Die Methode wird von konkreten Implementierungen dieser Klasse realisiert. Sie wird aufgerufen, wenn der Schalter an oder ausgeschaltet wird.

– **Parameters**
  * `status` - true bedeutet eingeschaltet, false ausgeschaltet

## A.8. Class AbstractRoomControlDeviceService

### Declaration

```
public abstract class AbstractRoomControlDeviceService
extends edu.udo.pg490.services.AbstractDeviceService
```

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *DevicesPresent* | UUIDs der Personal Devices im Raum |

**Tabelle 27:** RoomControlDevice-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| - | - | int - |

**Tabelle 28:** RoomControlDevice-Actions

| **Events** | |
|---|---|
| ***Event*** | ***Parameter*** |
| deviceEnters | String device<br>String location |
| deviceLeaves | String device<br>String location |

**Tabelle 29:** RoomControlDevice-Events

## A.9. Class AbstractFireCenterDeviceService

### Declaration

```
public abstract class AbstractFireCenterDeviceService
extends edu.udo.pg490.services.AbstractDeviceService
```

## A.10. Class AbstractFireDetectorDeviceService

| Properties | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *Fire* | zeigt das Ort des Feuerausbruches |

**Tabelle 30:** FireCenterDevice-Properties

| Actions | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| searchSensors | - | - |
| AlarmOff | - | - |

**Tabelle 31:** FireCenterDevice-Actions

Declaration

public abstract class AbstractFireDetectorDeviceService
**extends** edu.udo.pg490.services.AbstractDeviceService

| Events | |
|---|---|
| ***Event*** | ***Parameter*** |
| - | - |

**Tabelle 32:** FireCenterDevice-Events

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| startState | - | int temperature |
| fireUp | - | - |

**Tabelle 33:** FireDetectorDevice-Actions

| **Events** | |
|---|---|
| ***Event*** | ***Parameter*** |
| sendAlarmAction | String sendAlarm |
| AlarmOff | String String sendAlarmOffParam |

**Tabelle 34:** FireDetectorDevice-Events

# B. Spezifikation der Geräte-Implementierungen

In diesem Kapitel werden die Implementierungen der einzelnen Geräte dokumentiert.

## B.1. Fernseher

### B.1.1. Class TVVideoInputDeviceImpl

#### Declaration

public class TVVideoInputDeviceImpl
**extends** edu.udo.pg490.services.AbstractVideoInputDeviceService

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| verzeichnisAuswaehlenAction | verzeichnisAuswaehlenInput | verzeichnisOutput, dateiOutput |

**Tabelle 35:** Zusätzliche VideoInputDevice-Action des Fernsehers

#### Methods

- `protected InputStream` **getInputStream**`( java.lang.String` **streamID**`)`
  - **Usage**
    - Diese Methode gibt für einen Dateipfad ein Objekt der Klasse InputStream zurück.
  - **Parameters**
    - `streamID` - Ein Dateipfad in Form eines Strings
  - **Returns** - Gibt ein Objekt der Klasse InputStream zurück.
  - **Exceptions**
    - `java.io.IOException` - Wird geworfen, wenn der Eingabeparameter keine existierende Datei bezeichnet.

- `protected String` **getStreamList**`( )`
  - **Usage**
    - Diese Methode gibt eine Liste von Videostreams zurück.
  - **Returns** - Gibt ein String-Array zurück, das eine Auflistung von Videostreams enthält.

### B.1.2. Class TVAudioInputDeviceImpl

Declaration

public class TVAudioInputDeviceImpl
**extends** edu.udo.pg490.services.AbstractAudioInputDeviceService

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| verzeichnisAuswaehlenAction | verzeichnisAuswaehlenInput | verzeichnisOutput, dateiOutput |

**Tabelle 36:** Zusätzliche AudioInputDevice-Action des Fernsehers

Methods

- `protected InputStream` **getInputStream**`( java.lang.String` **streamID**`)`
  - **Usage**
    - Diese Methode gibt für einen Dateipfad ein Objekt der Klasse InputStream zurück.
  - **Parameters**
    - `streamID` - Ein Dateipfad in Form eines Strings
  - **Returns** - Gibt ein Objekt der Klasse InputStream zurück.
  - **Exceptions**
    - `java.io.IOException` - Wird geworfen, wenn der Eingabeparameter keine existierende Datei bezeichnet.

---

- `protected String` **getStreamList**`( )`
  - **Usage**
    - Diese Methode gibt eine Liste von Audiostreams zurück.
  - **Returns** - Gibt ein String-Array zurück, das eine Auflistung von Audiostreams enthält.

### B.1.3. Class TVVideoOutputDeviceImpl

Declaration

public class TVVideoOutputDeviceImpl
**extends** edu.udo.pg490.services.AbstractVideoOutputDeviceService

Methods

- `public void` **smilDatenVerarbeiten**`( java.lang.String` **smilData**`)`
  - **Usage**
    - Diese Methode startet einen Mediaplayer mit den Daten des Parameters dieser Methode. Dieser Parameter soll eine URL zu einem Video beinhalten. Wenn der Parameter ausschließlich eine Zahl enthält, wird dies als Identifier für das Unterverzeichnis "Video"des Fernsehers implementiert.

      Wenn nie zuvor ein Mediaplayer gestartet wurde, wird ein Dateiauswahldialog angezeigt, in dem die Startdatei eines Mediaplayers gewählt werden kann. Der Dateifilter für die Auswahl des Mediaplayers im Dateiauswahldialog ist auf das Betriebssystem Windows zugeschnitten und ist unter Linux überflüssig.

---

- `protected boolean` **stoppeVideowiedergabe**`( )`
  - **Usage**
    - Diese Methode beendet den Mediaplayer.

### B.1.4. Class TVAudioOutputDeviceImpl

Declaration

public class TVAudioOutputDeviceImpl
**extends** edu.udo.pg490.services.AbstractAudioOutputDeviceService

Methods

- `protected void` **smilDatenVerarbeiten**`( java.lang.String` **smilData**`)`
  - **Usage**
    - Diese Methode startet einen Mediaplayer mit den Daten des Parameters dieser Methode. Dieser Parameter soll eine URL zu einer Audiodatei beinhalten.

      Wenn nie zuvor ein Mediaplayer gestartet wurde, wird ein Dateiauswahldialog angezeigt, in dem die Startdatei eines Mediaplayers gewählt werden kann. Der Dateifilter für die Auswahl des Mediaplayers im Dateiauswahldialog ist auf das Betriebssystem Windows zugeschnitten und ist unter Linux überflüssig.

---

- `protected void` **stoppeAudiowiedergabe**`( )`
  - **Usage**
    - Diese Methode beendet den Mediaplayer.

### B.1.5. Class TVTextOutputDeviceImpl

#### Declaration

public class TVTextOutputDeviceImpl
**extends** edu.udo.pg490.services.AbstractTextOutputDeviceService

#### Methods

- `public void` **setTxt**`( java.lang.String` **text**`)`
  - **Usage**
    - Diese Methode öffnet ein Textfenster im Fernseher durch Aufruf der Methode TVDevice#textAnzeigen(String) .

---

- `public void` **timeout**`( int` **num_sec**`)`
  - **Usage**
    - Diese Methode startet einen Thread, der jede Sekunde die noch verbleibende Zeit eines Timeouts an die Methode TVDevice#timeoutAnzeigen(int) weiterreicht. Nach Ablauf des Timeouts wird das TVDevice angewiesen das Textfenster zu schließen durch Aufruf der Methode TVDevice#textanzeigeLoeschen() .

### B.1.6. Class TVRecorderDeviceImpl

#### Declaration

public class TVRecorderDeviceImpl
**extends** edu.udo.pg490.services.AbstractRecorderDeviceService

Methods

- `protected Calendar` **getCalendar**`( )`
  - **Usage**
    - Gibt eine Instanz der Klasse GregorianCalendar zurück.
  - **Returns** - Gibt jeweils eine neue Instanz der Klasse GregorianCalendar zurück.

## B.2. Personalcomputer

### B.2.1. Class PCVideoInputDeviceImpl

Declaration

public class PCVideoInputDeviceImpl
**extends** edu.udo.pg490.services.AbstractVideoInputDeviceService

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| verzeichnisAuswaehlenAction | verzeichnisAuswaehlenInput | verzeichnisOutput, dateiOutput |

**Tabelle 37:** Zusätzliche VideoInputDevice-Action des Personalcomputers

Methods

- `protected InputStream` **getInputStream**`( java.lang.String` **streamID**`)`
  - **Usage**
    - Diese Methode gibt für einen Dateipfad ein Objekt der Klasse InputStream zurück.
  - **Parameters**
    - `streamID` - Ein Dateipfad in Form eines Strings
  - **Returns** - Gibt ein Objekt der Klasse InputStream zurück.
  - **Exceptions**
    - `java.io.IOException` - Wird geworfen, wenn der Eingabeparameter keine existierende Datei bezeichnet.

- `protected String` **getStreamList**`( )`
  - **Usage**

  * Diese Methode gibt eine Liste von Videostreams zurück.
- **Returns** - Gibt ein String-Array zurück, das eine Auflistung von Videostreams enthält.

### B.2.2. Class PCAudioInputDeviceImpl

#### Declaration

public class PCAudioInputDeviceImpl
**extends** edu.udo.pg490.services.AbstractAudioInputDeviceService

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| verzeichnisAuswaehlenAction | verzeichnisAuswaehlenInput | verzeichnisOutput, dateiOutput |

**Tabelle 38:** Zusätzliche AudioInputDevice-Action des Personalcomputer

#### Methods

- `protected InputStream` **getInputStream**`( java.lang.String` **streamID**`)`
  - **Usage**
    * Diese Methode gibt für einen Dateipfad ein Objekt der Klasse InputStream zurück.
  - **Parameters**
    * `streamID` - Ein Dateipfad in Form eines Strings
  - **Returns** - Gibt ein Objekt der Klasse InputStream zurück.
  - **Exceptions**
    * `java.io.IOException` - Wird geworfen, wenn der Eingabeparameter keine existierende Datei bezeichnet.

---

- `protected String` **getStreamList**`( )`
  - **Usage**
    * Diese Methode gibt eine Liste von Audiostreams zurück.
  - **Returns** - Gibt ein String-Array zurück, das eine Auflistung von Audiostreams enthält.

### B.2.3. Class PCVideoOutputDeviceImpl

Declaration

public class PCVideoOutputDeviceImpl
**extends** edu.udo.pg490.services.AbstractVideoOutputDeviceService

Methods

- `public void` **smilDatenVerarbeiten**`( java.lang.String` **smilData**`)`
  - **Usage**
    - Diese Methode startet einen Mediaplayer mit den Daten des Parameters dieser Methode. Dieser Parameter soll eine URL zu einem Video beinhalten.

      Wenn nie zuvor ein Mediaplayer gestartet wurde, wird ein Dateiauswahldialog angezeigt, in dem die Startdatei eines Mediaplayers gewählt werden kann. Der Dateifilter für die Auswahl des Mediaplayers im Dateiauswahldialog ist auf das Betriebssystem Windows zugeschnitten und ist unter Linux überflüssig.

- `protected boolean` **stoppeVideowiedergabe**`( )`
  - **Usage**
    - Diese Methode beendet den Mediaplayer.

### B.2.4. Class PCAudioOutputDeviceImpl

Declaration

public class PCAudioOutputDeviceImpl
**extends** edu.udo.pg490.services.AbstractAudioOutputDeviceService

Methods

- `protected void` **smilDatenVerarbeiten**`( java.lang.String` **smilData**`)`
  - **Usage**
    - Diese Methode startet einen Mediaplayer mit den Daten des Parameters dieser Methode. Dieser Parameter soll eine URL zu einer Audiodatei beinhalten.

      Wenn nie zuvor ein Mediaplayer gestartet wurde, wird ein Dateiauswahldialog

angezeigt, in dem die Startdatei eines Mediaplayers gewählt werden kann. Der Dateifilter für die Auswahl des Mediaplayers im Dateiauswahldialog ist auf das Betriebssystem Windows zugeschnitten und ist unter Linux überflüssig.

---

- `protected void` **stoppeAudiowiedergabe**`( )`
  - **Usage**
    - Diese Methode beendet den Mediaplayer.

### B.2.5. Class PCTextOutputDeviceImpl

#### Declaration

public class PCTextOutputDeviceImpl
**extends** edu.udo.pg490.services.AbstractTextOutputDeviceService

#### Methods

- `public void` **setTxt**`( java.lang.String` **text**`)`

---

- `public void` **setTxtSize**`( int` **TextSize**`)`

---

- `public void` **timeout**`( int` **num_sec**`)`

### B.2.6. Class PCDialogDeviceImpl

#### Declaration

public class PCDialogDeviceImpl
**extends** edu.udo.pg490.services.AbstractDialogDeviceService
**implements** java.awt.event.ActionListener

#### Methods

- `protected void` **showDialog**`( java.lang.String` **message**`, int` **type**`, int` **timeout**`)`
  - **Parameters**

* message -
* type -
* timeout -

---

- protected void **zeigeDialog**( java.lang.String **message**, java.lang.String **type**, java.lang.String **timeout**)

## B.3. RoomControlDevice

### B.3.1. Class FoxboardRoomControl

#### Declaration

public class FoxboardRoomControl
**extends** edu.udo.pg490.services.AbstractRoomControlDeviceService

| **Properties** | |
|---|---|
| ***Properties*** | ***Beschreibung*** |
| *DevicesRoomPresent* | BT-Adressen aller Devices in der Umgebung |
| *SystemCounter* | Der System Counter zur Synchronisation |
| *RssiRatio* | Das Rssi Verhältnis zwischen lokalem und entfertem Foxboard |

**Tabelle 39:** FoxboardRoomControl-Properties

| **Actions** | | |
|---|---|---|
| ***Action*** | ***Input-Parameter*** | ***Output-Parameter*** |
| getLinkQuality | String bdaddr | int rssi |
| remoteDisconnect | String bdaddr | boolean success |
| remoteSystemCounter | - | int systemCounterParameter |
| remoteRssiRatio | - | int rssiRatioParameter |

**Tabelle 40:** FoxboardRoomControl-Actions

#### Methods

- public void **PersonEntersRoom**( java.lang.String **name**)

---

- public void **PersonLeavesRoom**( java.lang.String **name**)

---

- public void **getRssi**( java.lang.String **strbdaddr**)

---

- public void **pandRemoteDisconnect**( java.lang.String **strbdaddr**)

## B.4. Persönliches Gerät

### B.4.1. Interface UserInterface

Das Interface beschreibt eine Klasse, die das User Interface für den Benutzer realisiert. Andere Implementierungen könnten z.B. die Dialoge auf einem Handy Display anzeigen.

#### Declaration

```
public interface UserInterface
```

#### Methods

- `public void` **dialogShow**`( java.lang.String` **msg**`, int` **type**`, int` **timeout**`, edu.udo.pg490.personaldevice.services.DialogAnswerCallback` **callback**`)`
  - **Usage**
    - Anzeige eines Dialoges
  - **Parameters**
    - `msg` - Frage bzw. Nachricht des Dialogs
    - `type` - Typ des Dialogs (siehe Konstanten)
    - `timeout` - Timeout des Dialogs (0 für keinen)
    - `callback` - Callback Interface für die Antwort des Benutzers

---

- `public void` **textOutputClearAfterDelay**`( int` **seconds**`)`
  - **Usage**
    - Text nach Delay wieder löschen
  - **Parameters**
    - `seconds` - Timeout des Textes

---

- `public void` **textOutputSetText**`( java.lang.String` **text**`)`
  - **Usage**
    - Einen Text anzeigen
  - **Parameters**
    - `text` - der Text

### B.4.2. Class AbstractRemoteTask

Diese abstrakte Klasse bietet die Basisklasse für eigene RemoteTasks. RemoteTasks sind Interaktionsfolgen mit entfernten Services. Die Klasse bietet die grundsätzliche Funktionalität, um solche Tasks zu entwickeln. Die Tasks können dann dem RemoteTaskManager zur Ausführung übergeben werden.

Declaration

public abstract class AbstractRemoteTask
**extends** java.lang.Object

Constructors

- `public` **AbstractRemoteTask**`( long` **taskTimeout**`)`
  - **Usage**
    - Der Konstruktor muss von der implementierenden Klassen aufgerufen werden
  - **Parameters**
    - `taskTimeout` - Timeout das Tasks

Methods

- `public abstract boolean` **equals**`( java.lang.Object` **obj**`)`
  - **Usage**
    - Diese Methode liefert `true`, wenn zwei Tasks identisch sind und somit die gleiche Aufgabe erfüllen. Somit soll verhindert werden, dass zwei gleiche Tasks ausgeführt werden.

---

- `protected void` **finished**`( )`
  - **Usage**
    - Wird aufgerufen, wenn ein Task fertig ist. Der Task wird dann vom Scheduler entfernt.

---

- `public abstract String` **getDescription**`( )`
  - **Usage**
    - Liefert eine einfache Beschreibung dieses Tasks für Log Nachrichten
  - **Returns** - Beschreibende String

---

- `protected String` **getInvokingServiceId**`( )`
  - **Usage**
    - Liefert die UUID des Personal Device
  - **Returns** - UUID des Personal Device

---

- `protected ServiceInvoker` **getServiceInvoker**`( )`
  - **Usage**
    - Liefert eine Referenz auf den ServiceInvoker
  - **Returns** - Referenz auf den ServiceInvoker

---

- `protected ServiceLookupManager` **getServiceLookupManager**`( )`
  - **Usage**
    - Liefert eine Referenz auf den ServiceLookupManager
  - **Returns** - Referenz auf den ServiceLookupManager

---

- `public long` **getValidUntil**`( )`
  - **Usage**
    - Bis zu diesem Zeitpunkt ist der Tasks noch gültig. Ist dieser Zeitpunkt überschritten, ist ein Timeout des Tasks aufgetreten.
  - **Returns** - Zeit als long

---

- `public boolean` **isFinished**`( )`
  - **Usage**
    - Fragt ab, ob ein Task bereits seine Aufgabe erfüllt hat.
  - **Returns** - `true`, wenn Task fertig

---

- `public abstract void` **onServiceFound**`( java.lang.String` **probeId**`, edu.udo.cs.sirena.service.remote.RemoteService` **remoteService**`)`
  - **Usage**
    - Wird aufgerufen, wenn ein Service bei einer Suche über den DWPS Stack gefunden wurde. Eine implementierende Klasse kann dann überprüfen ob es ein für sie interessanter Service ist und mit der Interaktion beginnen

---

- `protected boolean` **sameValue**`( java.lang.Object x, java.lang.Object y)`
  - **Usage**
    - Liefert `true`, wenn die Object x und y den gleichen Inhalt haben. Im Gegensatz zu equals() liefert die Methode auch dann true, wenn beide Werte `null` sind
  - **Parameters**
    - `x` - Erstes Object
    - `y` - Zweites Object

  - **Returns** - `true`, falls Objekte gleich

---

- `public void` **setRemoteTaskManager**`(`
  `edu.udo.pg490.personaldevice.tasks.RemoteTaskManager` **remoteTaskManager**`)`

  - **Usage**
    * Setzt den zu verwendenden RemoteTaskManager .
  - **Parameters**
    * `remoteTaskManager` - Instanz des RemoteTaskManager

---

- `public void` **startSearchService**`( java.lang.String []`**nameSpacesAndTypes**`)`

  - **Usage**
    * Diese Methode reicht einen searchService() Aufruf an den SearchManager des DPWS Stacks weiter.
  - **Parameters**
    * `nameSpacesAndTypes` - Eine Array von Namespaces und PortTypes

---

- `public abstract void` **startTask**`( )`

  - **Usage**
    * Wird zu Begin der Ausführung aufgerufen. Hier sollten die ersten Schritte der Service Interaktion realisiert werden.

### B.4.3. Class RemoteTaskManager

Diese Klasse verwaltet die Ausführung von RemoteTasks.

#### Declaration

public class RemoteTaskManager
**extends** java.lang.Object

#### Constructors

- `public` **RemoteTaskManager**`( edu.udo.pg490.personaldevice.PersonalDevice` **personalDevice**`)`

  - **Usage**
    * Der Konstruktor
  - **Parameters**
    * `personalDevice` - Eine Referenz auf das Personal Device

Methods

- `public void` **addNewTask(**
  `edu.udo.pg490.personaldevice.tasks.AbstractRemoteTask` **task)**
  - **Usage**
    - Fügt dem RemoteTaskManager einen neuen Tasks hinzu, der dann ausgeführt wird
  - **Parameters**
    - `task` - Der neuer Task

---

- `public PersonalDevice` **getPersonalDevice( )**
  - **Usage**
    - Liefert eine Referenz auf das PersonalDevice

---

- `public void` **handleServiceFound(** `java.lang.String` **probeId,**
  `edu.udo.cs.sirena.service.remote.RemoteService` **remoteService)**
  - **Usage**
    - Reicht die *ServiceFound* Meldungen an alle Tasks weiter

---

- `public void` **removeFinishedAndTimedOutTasks( )**
  - **Usage**
    - Diese Methode wird regelmäßig vom Scheduler aufgerufen. Sie entfernt Tasks, die den Timeout erreicht haben und Tasks, die als *beendet* markiert wurden.

---

- `public void` **removeTask(**
  `edu.udo.pg490.personaldevice.tasks.AbstractRemoteTask` **task)**
  - **Usage**
    - Entfernt einen Task aus der Queue
  - **Parameters**
    - `task` - zu löschender Task

---

- `public void` **startSearchService(** `java.lang.String []` **nameSpacesAndTypes)**
  - **Usage**
    - Leitet die searchService() Aufrufe an das persönliche Gerät weiter

### B.4.4. Interface ServiceInvoker

Der ServiceInvoker stellt ein allgemeines Interface bereit, mit dem das PersonalDevice entfernte RemoteServices aufrufen kann. Die Verwendung des Interfaces ermöglicht verschiedene Implementierungen wie z.B. mit oder ohne dem PersonalManagement .

#### Declaration

```
public interface ServiceInvoker
```

#### Methods

- `public Action` **invokePreferedRemoteService**`( java.lang.String` **actionName**`, java.lang.String` **porttype**`, edu.udo.pg490.personaldevice.invoker.ActionPreparer` **preparer**`)`
  - **Usage**
    - Diese Methode dient dazu einen durch den Benutzer bevorzugten Service aufzurufen. Die Implementierung kümmert sich dabei selbstständig darum, den entsprechenden bevorzugten Service zu ermitteln und eventell Alternativen zu suchen. Die Action wird duch den Action Namen und den PortType identifiziert. Das füllen der Action Input Parameter übernimmt die Instanz des ActionPreparer
  - **Parameters**
    - `actionName` - Name der Action
    - `porttype` - PortType der Action
    - `preparer` - ActionPreparer Klasse, die die Action mit Werten füllt
  - **Returns** - Aufgerufene Action oder `null` bei Fehlerschlag

---

- `public Action` **invokePreferedRemoteServiceInRoom**`( java.lang.String` **actionName**`, java.lang.String` **porttype**`, edu.udo.pg490.personaldevice.invoker.ActionPreparer` **preparer**`, java.lang.String` **location**`)`
  - **Usage**
    - Diese Methode dient dazu einen durch den Benutzer bevorzugten Service innerhalb eines bestimmten Raums aufzurufen. Die Implementierung kümmert sich dabei selbstständig darum, den entsprechenden bevorzugten Service zu ermitteln und eventell Alternativen zu suchen. Die Action wird duch den Action Namen und den PortType identifiziert. Das füllen der Action Input Parameter übernimmt die Instanz des ActionPreparer
  - **Parameters**
    - `actionName` - Name der Action

    * `porttype` - PortType der Action
    * `preparer` - ActionPreparer Klasse, die die Action mit Werten füllt
    * `location` - Raum, in dem gesucht werden soll
  - **Returns** - Aufgerufene Action oder `null` bei Fehlschlag

---

- `public Action` **invokeRemoteServiceFromList**`( java.lang.String` **actionName**`, java.lang.String` **porttype**`, edu.udo.pg490.personaldevice.invoker.ActionPreparer` **preparer**`, java.util.Vector` **remoteServiceList**`)`
  - **Usage**
    * Diese Methode dient dazu eine Action eines bestimmten PortTypes aufzurufen. Dabei werden alle Services der angegeben Liste der Reihe nach ausprobiert, bis der erste Aufruf erfolgreich war. Um die Action Objekte vor dem eigentlichen Aufruf vorzubereiten (um die Input Parameter mit Werte zu füllen), wird ein ActionPreparer Objekt übergeben.
  - **Parameters**
    * `actionName` - Name der Action
    * `porttype` - PortType der Action
    * `preparer` - ActionPreparer Klasse, die die Action mit Werten füllt
    * `remoteServiceList` - Liste von RemoteService Objekten
  - **Returns** - Aufgerufene Action oder `null` bei Fehlschlag

---

- `public void` **invokeSingleRemoteService**`(`
  `edu.udo.cs.sirena.service.remote.RemoteService` **remoteService**`,`
  `edu.udo.cs.sirena.service.Action` **action**`)`
  - **Usage**
    * Dies ist die einfachste Methode des ServiceInvoker Interfaces. Die Methode ruft die Action eines einzelnen RemoteService Objekts auf.
  - **Parameters**
    * `remoteService` - zu verwendender RemoteService
    * `action` - Action die ausgeführt werden soll
  - **Exceptions**
    * `edu.udo.cs.sirena.communication.soap.SOAPException` - Bei SOAPExceptions

### B.4.5. Interface LocationListener

Alle Services, welche dieses Interface implementieren, können sich registrieren, um über Raumwechsel informiert zu werden. So können Services beispielsweise ihre LocationID aktualisieren.

#### Declaration

```
public interface LocationListener
```

Methods

- `public void` **onNewLocation**`( java.lang.String` **oldLocation**`, java.lang.String` **newLocation**`)`
  - **Usage**
    - Wird aufgerufen, wenn ein Raumwechsel stattgefunden hat
  - **Parameters**
    - `oldLocation` - Alter Raum
    - `newLocation` - Neuer Raum

### B.4.6. Interface EventLogChangeListener

Das Interface kann von beliebige Klassen implementiert werden, wenn sie über Änderungen am Event-Log informiert werden wollen.

Declaration

```
public interface EventLogChangeListener
```

Methods

- `public void` **eventLogCleaned**`( )`
  - **Usage**
    - Wird aufgerufen, wenn das Log geleert wurde

---

- `public void` **newEventAdded**`(`
  `edu.udo.pg490.personaldevice.eventlog.EventLogEntry` **newEvent**`)`
  - **Usage**
    - Wird aufgerufen, wenn ein neues Event ins Log eingefügt wurde
  - **Parameters**
    - `newEvent` - das neue Event

### B.4.7. Class EventLog

Das EventLog ist eine Klasse, über die Log Meldungen generiert werden. Listener können sich bei der Klasse registrieren und werden über neue Log Einträge informiert.

Declaration

public class EventLog
**extends** java.lang.Object

Constructors

- `protected` **EventLog**`( )`
  - **Usage**
    - Erzeugt ein neues EventLog

Methods

- `public synchronized void` **addDebugEvent**`( java.lang.String` **msg**`)`
  - **Usage**
    - Erzeugt eine Nachricht mit der Priorität DEBUG
  - **Parameters**
    - `msg` - Log Nachricht

---

- `public synchronized void` **addErrorEvent**`( java.lang.String` **msg**`)`
  - **Usage**
    - Erzeugt eine Nachricht mit der Priorität ERROR
  - **Parameters**
    - `msg` - Log Nachricht

---

- `public synchronized void` **addEventLogChangeListener**`(`
  `edu.udo.pg490.personaldevice.eventlog.EventLogChangeListener` **listener**`)`
  - **Usage**
    - Neue Listener können sich über diese Methode registrieren
  - **Parameters**
    - `listener` - Referenz auf den Listener

---

- `public synchronized void` **addInfoEvent**`( java.lang.String` **msg**`)`
  - **Usage**
    - Erzeugt eine Nachricht mit der Priorität INFO

  - **Parameters**
    * `msg` - Log Nachricht

---

- `public synchronized void` **addWarnEvent**`( java.lang.String msg)`
  - **Usage**
    * Erzeugt eine Nachricht mit der Priorität WARN
  - **Parameters**
    * `msg` - Log Nachricht

---

- `public synchronized void` **deleteAll**`( )`
  - **Usage**
    * Löscht das komplette LOG

---

- `public synchronized Vector` **getAllEventLogEntries**`( )`
  - **Usage**
    * Liefert eine Liste aller Log Einträge
  - **Returns** - Liste aller Log Einträge

## B.5. Brandschutz

### B.5.1. Fire-Center

#### Methode serviceFound

```
protected void serviceFound(RemoteService remoteService)
```

Diese Methode sucht die Feuermelder in den Räumen und abonniert die Events *sendAlarmAction* und *AlarmOff* bei den Feuermeldern.

#### Methode receiveEvent

```
public void receiveEvent(Action notif, String subscrip)
```

Wenn die Events *sendAlarmAction* oder *AlarmOff* kommen, dann wird die Property *Fire* entsprechend aktualisiert.

### B.5.2. Fire-Detector

#### Methode getLocation

```
public public String getLocation()
```

***Returns:***
loc. - Ein String mit dem aktuellen Ort.(Wohnzimmer oder Schlafzimmer)

In FireCenter und in FireDetector gibt es nur wenige Funktionen, die zudem nicht wichtig genug sind, um hier aufgelistet zu werden. Die Benachrichtigung im Falle eines Brandes erfolgt im Management durch Policies.

# C. Installationsanleitung der Foxboard Entwicklungsumgebung

In diesem Kapitel wird eine hoffentlich ausführliche Anleitung gegeben, wie eine brauchbare Entwicklungsumgebung für das Foxboard installiert wird. Ausserdem wird dargestellt wie unsere KVM entstanden ist.

## C.1. Installation

Zunächst wird die Entwicklungsumgebung benötigt, die man sich unter www.acmesystems.it/?id=714 herunter laden kann. Man hat die Auswahl für die Betriebssystem Windows und Linux, wobei die Windows Version auf eine Linux Emulation zurückgreift und deshalb eher nicht zu empfehlen ist. Vorher sollte Überprüft werden ob folgende nötigen abhängigen Pakete schon installiert sind :
*GCC C compiler, CRIS cross-compiler, GNU make, GNU wget, Subversion, awk (or gawk), bc, byacc, lex or flex, perl, sed, tar, zlib, md5sum, pmake, curses oder ncurses, bison, which,* u.U. müssen diese vor der eigentlichen Installation noch installiert werden. Die ganze Paketinstallation wird vereinfacht durch ein Shell-Skript, das nur noch ausführbar gemacht werden muss. Nach dem Ausführen des Skriptes

```
./install_svn_sdk.sh
```

werden alle nötigen Dateien automatisch heruntergeladen und die Installation startet.
Als nächstes muss die Umgebung an unsere Bedürfnisse angepasst werden, dazu wird in das Verzeichnis

```
cd devboardR2_01
```

gewechselt und

```
make menuconfig
```

ausgeführt. Es öffnet sich das unter 23 gezeigte Fenster und man kann jetzt durch alle Menüs durchgehen und sich so seine Entwicklungsumgebung konfigurieren. Die Einstellungen werden gespeichert und mit dem Befehl

```
./configure
```

wird ein Makefile erzeugt. Als nächstes muss nur noch

```
make
```

eingegeben werden und die Installation startet. Der ganze Vorgang kann schon recht lange dauern. Danach ist eine für die eigenen Anforderungen speziell zugeschnittene Entwicklungsumgebung erstellt worden, auf Basis derer auch das Fox-Board geflasht werden kann.
Als letztes muss noch der Crosscompiler installiert werden, dazu muss die Datei `cris-dist-1.63-1.i386.tar.gz` heruntergeladen und entpackt werden. Die folgenden Umgebungsvariablen müssen dann noch richtig gesetzt werden:

PATH=PATH:<cris-dist-compiler>
JAVA_HOME=<java-sdk home>
JDK_HOME=<java-sdk home>
BOOTDIR=<java-sdk home>
JAVAC=<java-sdk home>/bin/javac
CLASSPATH=<java-sdk home>/jre/lib/rt.jar:

Es ist weiter zu beachten, dass die KVM nur mit einem Java 1.4 SDK Compiler zu com-

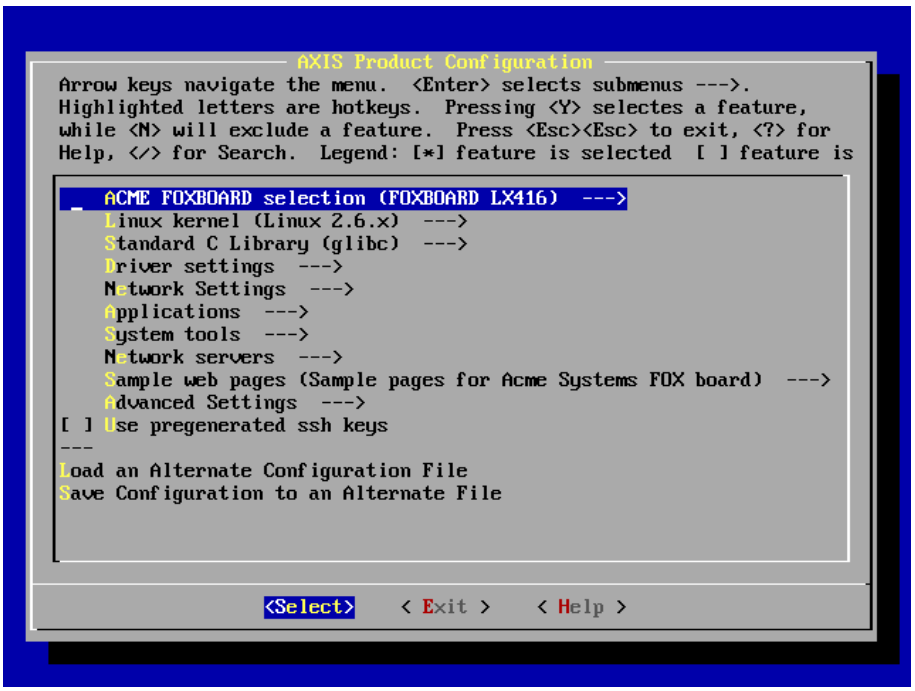


**Abbildung 23:** Installationsmenü

pilieren ist. Danach muss ins devboard root-Verzeichnis gewechselt werden und `./configure` ausgeführt werden
Danach ist mit `make cris-axis-linux-gnu` oder `make cris-axis-linux-gnuuclibc` der Crossompiler zu erstellen.

## C.2. Compilierung der eigenen KVM

Vor dem compillieren ist es wichtig auf jedenfall das Patch

`cldc_104dpws4jme.patch`

zu installieren, damit auch die Multicast Nachrichten empfangen werden können, was in der normalen KVM nicht geht. Das zum bauen der KVM interessante Verzeichnis lautet

`devboardR2_01/apps/j2me_cldc` .

Es folgt exemplarisch eine Auflistung der Verzeichnisse, die ergänzt werden müssen um eigene Funktionalitäten hinzuzufügen.

Eigene Funktionalitäten gehören hier herein, das heißt eigene Packages werden komplett in dieses Verzeichnis kopiert :

`devboardR2_01/apps/j2me_cldc/api/src/`

Die dazugehörigen Klassendateien werden in folgendes Verzeichnis kopiert.

`devboardR2_01/apps/j2me_cldc/api/classes`

Eventuell weitere benötigte Zusatzfunktionen welche aus der Java2SE kommen, kann man natürlich auf diese Weise auch über entsprechende Pakete hier einpflegen. Für unser Projekt sind z.B. noch erweiterte Exception-Methoden hinzugefügt worden und nicht zu vergessen der StringTokenizer. Das die Kilobyte Virtual Machine dann irgendwann eher zur Megabyte-VM mutiert, ist jedem selber überlassen. Theoretisch kann auf diesem Weg alles zu der KVM hin-

zugefügt werden, was sonst beim Starten der KVM über die Option `-classpath` hinzugefügt werden muss.
An folgender Stelle liegen die C-Quelldateien, über die das KNI auf den Bluetooth Stack zugreift, in unserem Projekt ist dies die Datei `bluez.c`.

`devboardR2_01/apps/j2me_cldc/kvm/VmUnix/src`

Entsprechend müssen die C-Header Dateien in folgendes Verzeichnis kopiert werden.

`devboardR2_01/apps/j2me_cldc/kvm/VmUnix/h`

Wenn dann alles soweit angefügt wurde, müssen noch die Makefiles angepasst werden.Im Verzeichnis

`devboardR2_01/apps/j2me_cldc`

liegt das Haupt-Makefile welches aufgerufen wird um die KVM zu erstellen, die anderen werden in Abhängigkeit aufgerufen. Das erste Makefile sieht folgendermaßen aus:

```
AXIS_USABLE_LIBS = UCLIBC GLIBC
include $(AXIS_TOP_DIR)/tools/build/Rules.axis

PROGS      = kvm

ifeq ($(DEBUG), true)
   KVMBIN = kvm_g
else
   KVMBIN = kvm
endif

clean:
        cd build/linux/; $(MAKE) clean
all:
        cd build/linux/; export FOX=true && $(MAKE) all
        file kvm/VmUnix/build/$(KVMBIN)
        cp kvm/VmUnix/build/kvm /srv/www/htdocs/egroup/
        @ echo ""
        @ echo "--------------------------------"
        @ echo ""
        @ echo "Yeah! KVM successfully compiled! :-)"
        @ echo "--------------------------------"
        @ echo ""
```

Wie man sieht gibt es hier eine Option *DEBUG=true* die zu setzen ist, um eine KVM zu erstellen, die am Ende kvm_g heißt und zusätzlich Debug Funktionen unterstützt. Hier werden ausserdem die beiden anderen Makefiles aufgerufen. Das erste liegt unter

`devboardR2_01/apps/j2me_cldc/build/linux`

Und das andere liegt im Verzeichnis

`devboardR2_01/apps/j2me_cldc/kvm/VmUnix/build`

Das letzte enthält die Erweiterungen um die Funktionalität, daß die Bluetooth Funktionen mit eingebunden werden und daß das native Interface benutzt wird. Dazu muss daß Standard-Makefile um folgende Zeilen erweitert werden.

```
ifeq ($(USE_KNI), false)
  OTHER_FLAGS += -DUSE_KNI=0
else
  OTHER_FLAGS += -DUSE_KNI=1
  SRCFILES += kni.c
endif

ifeq ($(USE_BLUE), true)
  SRCFILES += bluez.c
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:../../../../bluetooth/bluez/
      libs/src/.libs
  LIBS += -lbluetooth
  ADDOBJECTS = /home/share/fox/devboard-R2_01/apps/j2me_cldc/kvm/VmUnix
      /pgobj/dund/*.o /home/share/fox/devboard-R2_01/apps/j2me_cldc/kvm
      /VmUnix/pgobj/pand/sdp.o /home/share/fox/devboard-R2_01/apps/
      j2me_cldc/kvm/VmUnix/pgobj/hcid/*.o /home/share/fox/devboard-
      R2_01/apps/j2me_cldc/kvm/VmUnix/pgobj/hidd/*.o /home/share/fox/
      devboard-R2_01/apps/j2me_cldc/kvm/VmUnix/pgobj/rfcomm/*.o /home/
      share/fox/devboard-R2_01/apps/j2me_cldc/kvm/VmUnix/pgobj/sdpd/*.o
       /home/share/fox/devboard-R2_01/apps/j2me_cldc/kvm/VmUnix/pgobj/
      tools/*.o /home/share/fox/devboard-R2_01/apps/j2me_cldc/kvm/
      VmUnix/pgobj/libs/*.o
else

endif
```

Wenn die Makefiles wie oben beschrieben angepasst wurden, kann man mit *make USE_KNI=true USE_BLUE=true* die KVM erstellen, oder alternativ mit *make USE_KNI=true USE_BLUE=true DEBUG=true* eine Debug-KVM namens *kvm_g* .

## C.3. Konfiguration des Fox-Board

Vor dem Anschalten des Board wird ein Bluetooth Dongle in einen der beiden USB Ports gesteckt und ein USB-Speicherstick in den anderen. Damit dieser nun bei jedem Start automatisch konfiguriert wird, muss ein Init-Script geschrieben werden, welches folgendes leisten muss.

1. Aktivieren des Bluetooth Dongel
2. Initialisieren des Bluetooth Dongels als Network Access Point
3. Starten des SDP Dämons
4. Mounten des USB-Speichersticks

Das Script *Blue* wird in `/etc/init.d` erstellt und für den init Level 3 in `/etc/rc.3` mit dem Namen `S99Blue` verlinkt.

```
#!/bin/sh -e
# /etc/init.d/blue
case "$1" in
        start)
                hciconfig hci0 up
                pand -s -r NAP -M
                sdpd
                mount /dev/sda1 /mnt/flash/usb
                ;;
        stop)
                hciconfig hci0 down
                ;;
        *)
                error "Usage: $0 start|stop"
                ;;
esac

exit 0
```

Wenn nun ein anderes Bluetooth Gerät eine BNEP Verbindung als PANU mit dem Fox-Board eingeht, entsteht ein virtuelles Netzwerkinterface *bnep0*. Um eine TCP Verbindung aufbauen zu können muss diesem Interface eine IP zugewiesen werden. Damit dies automatisch erledigt wird, muss ein Konfigurationsscript *dev-up* in /etc/bluetooth/pan erstellt werden.

```
#!/bin/sh
ifconfig $1 10.0.0.1
```

Das Fox-Board ist an dieser Stelle fertig konfiguriert und für den Einsatz bereit.

# D. Spezifikationen der Raumerkennung

In diesem Kapitel werden die Bluetooth Schnittstellen und ihre zum Aufruf benötigten Parameter dokumentiert. Es handelt sich hierbei um native Interfaces, deren Funktionalität in C implementiert ist. Die C implementierung erfolgte sowohl in Form von JNIs für den i386 Prozessor, als auch in Form von KNIs für das Foxboard. Die Schnittstellen bleiben dabei gleich.

## D.1. Class BlueZ

This class provides the methods to access the underlying BlueZ functions. All of the native methods defined in this class are implemented, via the Java Native Interface (JNI) in C. See the bluez.c file and associated comments for full details.

If you wish to incorporate Bluetooth functionality (using the BlueZ stack) into your Java application, create a new instance of this BlueZ class. Calling the methods provided here will call the functions defined by the BlueZ libraries, primarily those defined in the files `hci.c` and `hci_lib.h`.

### Declaration

public class BlueZ
**extends** java.lang.Object

### Methods

- `public native void` **`hciCloseDevice`**`( int dd)`
  - **Usage**
    - Close the HCI device.
  - **Parameters**
    - `dd` - The HCI device descriptor (as returned from `hciOpenDevice`)

---

- `public int` **`hciCreateConnection`**`( int dd, edu.udo.pg490.knifox.BTAddress bdaddr, int ptype, int clkoffset, short rswitch, int timeOut)`
  - **Usage**
    - Create a HCI connection. Note that this requires higher than normal privileges, and so will often only work when executed as root. If you call this method with insufficient privileges, an exception will be thrown with the message text Ünable to create connection".
      See HCI_Create_Connection in the Bluetooth Specification for further details of the various arguments.
  - **Parameters**
    - `dd` - HCI device descriptor.

    * `bdaddr` - Bluetooth address as a BTAddress object.
    * `ptype` - Packet type, for example 0x0008 for DM1.
    * `clkoffset` - Clock offset (usually set to 0).
    * `rswitch` - Role switch (usually set to 0 or 1).
    * `timeOut` - Timeout, in milliseconds.
  - **Returns** - A handle for the connection.
  - **Exceptions**
    * `edu.udo.pg490.knifox.BlueZException` - Unable to create the connection.

---

- `public int` **hciCreateConnection**`( int dd, edu.udo.pg490.knifox.BTAddress bdaddr, int timeOut)`
  - **Usage**
    * This is the same as `hciCreateConnection(int dd, BTAddress bdaddr, int ptype, int clkoffset, short rswitch, int timeOut)`, except that the `ptype`, `clkoffset` and `rswitch` fields are preset to 'default' values. These values are 0x0008, 0 and 0, respectively.
  - **See Also**
    * `public int hciCreateConnection( int dd, edu.udo.pg490.knifox.BTAddress bdaddr, int ptype, int clkoffset, short rswitch, int timeOut)` ( in D.1, page 134)

---

- `public int` **hciDeviceID**`( edu.udo.pg490.knifox.BTAddress bdaddr)`
  - **Usage**
    * Gets the device ID for a specified local HCI device.
  - **Parameters**
    * `bdaddr` - Bluetooth address as a BTAddress object.
  - **Returns** - The device ID for the local device.
  - **Exceptions**
    * `edu.udo.pg490.knifox.BlueZException` - If unable to get the device ID.

---

- `public native int` **hciDeviceID**`( java.lang.String bdaddr)`

---

- `public native void` **hciDisconnect**`( int dd, int handle, short reason, int timeOut)`
  - **Usage**
    * Disconnect an established HCI connection.
      See HCI_Disconnect in the Bluetooth Specification for further details of the various arguments.
  - **Parameters**
    * `dd` - HCI device descriptor.

    * `handle` - HCI connection handle.
    * `reason` - Code detailing reason for disconnection (often 0x13).
    * `timeOut` - Timeout, in milliseconds.
  - **Exceptions**
    * `edu.udo.pg490.knifox.BlueZException` - Unable to disconnect.

---

- `public int` **hciGetRssi**`( edu.udo.pg490.knifox.BTAddress` **bdaddr**`)`
  - **Usage**
    * Gives you the RSSI value of a specifiied Bluetooth connection. For best connection quality the return value is 0
  - **Parameters**
    * `bdaddr` - Bluetooth address as a BTAddress object you want to have the RSSI value from in the form `"00:12:34:56:78:9A"`.
  - **Returns** - The RSSI value
  - **Exceptions**
    * `edu.udo.pg490.knifox.BlueZException` - Read RSSI failed

---

- `public native int` **hciGetRssi**`( java.lang.String` **jbdaddr**`)`

---

- `public native InquiryInfo` **hciInquiry**`( int` **hciDevID**`, int` **len**`, int` **max_num_rsp**`, long` **flags**`)`
  - **Usage**
    * Perform a HCI inquiry to discover remote Bluetooth devices.
      See HCI_Inquiry in the Bluetooth Specification for further details of the various arguments.
  - **Parameters**
    * `hciDevID` - The local HCI device ID (see the hciconfig tool provided by BlueZ for further information).
    * `len` - Maximum amount of time before inquiry is halted. Time = len x 1.28 secs.
    * `max_num_rsp` - Maximum number of responses allowed before inquiry is halted. For an unlimited number, set to 0.
    * `flags` - Additional flags. See BlueZ documentation and source code for details.
  - **Returns** - An InquiryInfo object containing the results of the inquiry.
  - **Exceptions**
    * `edu.udo.pg490.knifox.BlueZException` - If the inquiry failed.
  - **See Also**
    * `edu.udo.pg490.knifox.BlueZ.hciInquiry(int hciDevID, int len, int max_num_r long flags)` ( in D.1, page 136)

---

- `public InquiryInfo` **hciInquiry**`( int` **hciDevID**`)`

- **Usage**
  - This is the same as `hciInquiry(int hciDevID, int len, int max_num_rsp, long flags)`, except that the `len`, `max_num_rsp` and `flags` fields are preset to 'default' values. These values are 8, 10 and 0, respectively.
- **See Also**
  - `edu.udo.pg490.knifox.BlueZ.hciInquiry(int hciDevID)` (in D.1, page 136)

---

- `public native HCIConnectionInfoList` **hciListConnections**`( int dev_id)`
  - **Usage**
    - List all HCI connections
  - **Parameters**
    - `dev_id` - The device id from which the search is initiated
  - **Returns** - A `HCIConnectionInfoList` object representing the local HCI connections
  - **Exceptions**
    - `edu.udo.pg490.knifox.BTAddressFormatException` -

---

- `public native int` **hciOpenDevice**`( int hciDevID)`
  - **Usage**
    - Opens the HCI device.
  - **Parameters**
    - `hciDevID` - The local HCI device ID (see the hciconfig tool provided by BlueZ for further information).
  - **Returns** - A device descriptor (often named `dd`) for the HCI device.
  - **Exceptions**
    - `edu.udo.pg490.knifox.BlueZException` - Unable to open the HCI device.

---

- `public int` **pandCreateConnection**`( edu.udo.pg490.knifox.BTAddress` **bdaddr**`, java.lang.String` **netdev**`)`
  - **Usage**
    - Creates a PAN connection to a specified Bluetooth device
  - **Parameters**
    - `bdaddr` - Bluetooth address as a BTAddress object.
    - `netdev` - The name of the interface wich will be created
  - **Returns** - An integer 0 or 1 depending on whether it was successfull or not
  - **Exceptions**
    - `edu.udo.pg490.knifox.BlueZException` - Unable to create the connection

---

- `public native int` **pandCreateConnection**`( java.lang.String` **jbdaddr**`, java.lang.String` **netdev**`)`

---

- `public int` **pandDisconnect**`( edu.udo.pg490.knifox.BTAddress` **bdaddr**`)`
  - **Usage**
    - Kills a specified PAN connection
  - **Parameters**
    - `bdaddr` - Bluetooth address as a BTAddress object.
  - **Returns** - An integer 0 or 1 depending on whether it was successfull or not
  - **Exceptions**
    - `edu.udo.pg490.knifox.BlueZException` - Unable to kill the connection

---

- `public native int` **pandDisconnect**`( java.lang.String` **jbdaddr**`)`

---

- `public native int` **pandDisconnectAll**`( )`
  - **Usage**
    - Kills all PAN connections
  - **Returns** - An integer 0 or 1 depending on whether it was successfull or not
  - **Exceptions**
    - `edu.udo.pg490.knifox.BlueZException` - Unable to kill the connections

---

- `public native PANConnectionInfoList` **pandListConnections**`( )`
  - **Usage**
    - Lists all active PAN connections
  - **Returns** - A `PANConnectionInfoList` object representing the local PAN connections
  - **Exceptions**
    - `edu.udo.pg490.knifox.BlueZException` - Failed to get connection list

---

- `public native int` **pandSearchAndCreateConnection**`( java.lang.String` **jrole**`, int` **jsearch_duration**`)`
  - **Usage**
    - Searches for Bluetooth devices and tries to create a PAN connection to them.
  - **Parameters**
    - `jrole` - Specifies the role of the local device (PAN, NAP or GN)
    - `jsearch_duration` - Specifies the duration of the search
  - **Returns** - An integer, 0 for finding no Bluetooth Device, 1 for a connection failure and 2 for success

- **Exceptions**
  - `edu.udo.pg490.knifox.BlueZException` - Unable to create Connection

---

- `public int` **pandSearchAndCreateConnection**`( java.lang.String jrole)`
  - **Usage**
    - This is the same as `pandSearchAndCreateConnection(String jrole, int jsearch_duration)`, except that the `jsearch_duration` field is preset to 'default' value. These value is 10.
  - **See Also**
    - `public int pandSearchAndCreateConnection(String jrole, int jsearch_duration)` ( in D.1, page 138)

---

- `public native int` **pandWaitforConnection**`( )`
  - **Usage**
    - Listens for incoming PAN connections
  - **Returns** - An integer 0 or 1 depending on whether it was successfull or not
  - **Exceptions**
    - `edu.udo.pg490.knifox.BlueZException` -

## D.2. Class BlueZException

Exception thrown when problems occur within the BlueZ native code.

### Declaration

public class BlueZException
**extends** java.lang.Exception

### Constructors

- `public` **BlueZException**`( )`
  - **Usage**
    - Constructs a BlueZException with no detail message.

---

- `public` **BlueZException**`( java.lang.String s)`
  - **Usage**
    - Constructs a BlueZException including a detail message.

## D.3. Class BTAddress

Java representation of a Bluetooth device address.

A Bluetooth device address is comprised of six pairs of hex digits, for example `00:12:34:56:78:9A`. This class is based upon the `bdaddr_t` type, as defined in `bluetooth.h` of the BlueZ libraries.

### Declaration

public class BTAddress
**extends** java.lang.Object

### Constructors

- `public` **BTAddress**`( )`
  - **Usage**
    - Default constructor

---

- `public` **BTAddress**`( java.lang.String` **addr_str**`)`
  - **Usage**
    - Creates a `BTAddress` object from the address `addr_str`. The address string should be in the form "`00:12:34:56:78:9A`".
  - **Parameters**
    - `addr_str` - `String` representation of the Bluetooth device address.

### Methods

- `public boolean` **equals**`( edu.udo.pg490.knifox.BTAddress` **compare**`)`
  - **Usage**
    - Compares two BTAddress objects to see if they represent the same Bluetooth device address.

---

- `public void` **setValue**`( java.lang.String` **addr_str**`)`
  - **Usage**
    - Set the Bluetooth device address. Valid string format is "`00:12:34:56:78:AB`", that is 6, colon separated pairs of hex digits.
  - **Exceptions**

* `edu.udo.pg490.knifox.BTAddressFormatException` - If the String is not a parsable Bluetooth address.

---

- `public String` **toString**`( )`
  - **Usage**
    * Returns a String representation of the Bluetooth device address in the form "`00:12:34:56:78:9A`".

## D.4. Class BTAddressFormatException

Exception thrown when problems occur when parsing a String to a BTAddress object.

### Declaration

public class BTAddressFormatException
**extends** java.lang.Exception

### Constructors

- `public` **BTAddressFormatException**`( )`
  - **Usage**
    * Constructs a BTAddressException with no detail message.

---

- `public` **BTAddressFormatException**`( java.lang.String s)`
  - **Usage**
    * Constructs a BTAddressException including a detail message.

## D.5. Class HCIConnectionInfo

Represents the informations of a HCI connection

### Declaration

public class HCIConnectionInfo
**extends** java.lang.Object

Constructors

- `public` **HCIConnectionInfo**`( )`
  - **Usage**
    - Default Constructor

---

- `public` **HCIConnectionInfo**`( java.lang.String` **_direction**`, java.lang.String` **_link**`, edu.udo.pg490.knifox.BTAddress` **_bdaddr**`, int` **_handle**`, int` **_state**`, java.lang.String` **_role**`)`
  - **Usage**
    - Constructor which sets all the fields in the class. This is provided since when creating a new PANConnectionInfo object in the native code (namely C).
  - **Parameters**
    - `_direction` - < or > for an incoming or outgoing connection
    - `_link` - There are two types of physical link between a master and a slave: SCO (synchronous connection oriented) and ACL (asynchronous connectionless)
    - `_bdaddr` - Bluetooth device address
    - `_handle` - Connection handle
    - `_state` - Connection state
    - `_role` - The role of the connection (Master, Slave)

Methods

- `public boolean` **equals**`( edu.udo.pg490.knifox.HCIConnectionInfo` **con**`)`
  - **Usage**
    - Compares two HCIConnectionInfo objects to see if they represent the same Bluetooth device. This is done by calling the `equals` method on the `bdaddr` field, hence seeing if they have the same Bluetooth device address.

---

- `public String` **toString**`( )`
  - **Usage**
    - Returns a String representation of all the information represented by this object.

## D.6. Class HCIConnectionInfoList

Stores the result of a hciListConnections. Each result is a `HCIConnectionInfo` object

Declaration

public class HCIConnectionInfoList
**extends** java.lang.Object

Constructors

- `public` **HCIConnectionInfoList**`( )`
  - **Usage**
    - Default constructor.

---

- `public` **HCIConnectionInfoList**`( byte` **_num_responses**`)`
  - **Usage**
    - Constructor.
  - **Parameters**
    - `_num_responses` - The number of responses.

Methods

- `public void` **addConnection**`( edu.udo.pg490.knifox.HCIConnectionInfo` **con**`)`
  - **Usage**
    - Adds an HCIConnectionInfo object to the Vector of Connections.
  - **Parameters**
    - `con` - A HCIConnectionInfo object.

---

- `public Vector` **devices**`( )`
  - **Usage**
    - Returns the Vector representation of the HCIConnectionInfo objects.
  - **Returns** - Devices found.

## D.7. Class InquiryInfo

Stores the results of an HCI inquiry. Each result is an InquiryInfoDevice object.

Declaration

public class InquiryInfo
**extends** java.lang.Object

Constructors

- `public` **InquiryInfo**`( )`
  - **Usage**
    - Default constructor.

---

- `public` **InquiryInfo**`( byte` **_num_responses**`)`
  - **Usage**
    - Constructor.
  - **Parameters**
    - `_num_responses` - The number of responses.

Methods

- `public void` **addDevice**`( edu.udo.pg490.knifox.InquiryInfoDevice` **dev**`)`
  - **Usage**
    - Adds an InquiryInfoDevice object to the Vector of devices.
  - **Parameters**
    - `dev` - An InquiryInfoDevice object.

---

- `public Vector` **devices**`( )`
  - **Usage**
    - Returns the Vector representation of the InquiryInfoDevice objects.
  - **Returns** - Devices found.

## D.8. Class InquiryInfoDevice

Represents the information gather about a remote device during an HCI Inquiry.

Many devices may be found for an inquiry. The `InquiryInfoDevice` class is the information relating to a single device. All the discovered devices are collectively kept in the `InquiryInfo` class.

Declaration

public class InquiryInfoDevice
**extends** java.lang.Object

Constructors

- `public` **InquiryInfoDevice**`( )`
  - **Usage**
    - Default constructor.

---

- `public` **InquiryInfoDevice**`( edu.udo.pg490.knifox.BTAddress` **_bdaddr**`, short` **_pscan_rep_mode**`, short` **_pscan_period_mode**`, short` **_pscan_mode**`, short` **_dev_class0**`, short` **_dev_class1**`, short` **_dev_class2**`, int` **_clock_offset**`)`
  - **Usage**
    - Constructor which sets all the fields in the class. This is provided since when creating a new InquiryInfoDevice object in the native code (namely C), it is much more efficient since we do not have to ask the JVM for pointers to each individual field.

Methods

- `public boolean` **equals**`( edu.udo.pg490.knifox.InquiryInfoDevice` **dev**`)`
  - **Usage**
    - Compares two InquiryInfoDevice objects to see if they represent the same Bluetooth device. This is done by calling the `equals` method on the `bdaddr` field, hence seeing if they have the same Bluetooth device address.

---

- `public String` **toString**`( )`
  - **Usage**
    - Returns a String representation of all the data held in this object.

## D.9. Class PANConnectionInfo

Represents the informations of a PAN connection

Declaration

public class PANConnectionInfo
**extends** java.lang.Object

Constructors

- `public` **PANConnectionInfo**`( )`
  - **Usage**
    - Default Constructor

---

- `public` **PANConnectionInfo**`( java.lang.String` **_interfacename,** `edu.udo.pg490.knifox.BTAddress` **_bdaddr,** `java.lang.String` **_role**`)`
  - **Usage**
    - Constructor which sets all the fields in the class. This is provided since when creating a new PANConnectionInfo object in the native code (namely C).
  - **Parameters**
    - `_interfacename` - Name of the interface which will be created (e.g. bnep0)
    - `_bdaddr` - The bluetooth device address
    - `_role` - The role of the connection (Master, Slave)

Methods

- `public boolean` **equals**`( edu.udo.pg490.knifox.PANConnectionInfo` **dev**`)`
  - **Usage**
    - Compares two PANConnectionInfo objects to see if they represent the same Bluetooth device. This is done by calling the `equals` method on the `bdaddr` field, hence seeing if they have the same Bluetooth device address.

---

- `public String` **toString**`( )`
  - **Usage**
    - Returns a String representation of all the information represented by this object.

## D.10. Class PANConnectionInfoList

Stores the result of a pandListConnections. Each result is a `PANConnectionInfo` object

Declaration

public class PANConnectionInfoList
**extends** java.lang.Object

Constructors

- `public` **PANConnectionInfoList**`( )`
  - **Usage**
    - Default constructor.

---

- `public` **PANConnectionInfoList**`( byte` **_num_responses**`)`
  - **Usage**
    - Constructor.
  - **Parameters**
    - `_num_responses` - The number of responses.

Methods

- `public void` **addConnection**`( edu.udo.pg490.knifox.PANConnectionInfo` **dev**`)`
  - **Usage**
    - Adds an PANConnectionInfo object to the Vector of Connections.
  - **Parameters**
    - `dev` - An PANConnectionInfo object.

---

- `public Vector` **devices**`( )`
  - **Usage**
    - Returns the Vector representation of the PANConnectionInfo objects.
  - **Returns** - Devices found.

# E. Spezifikation der Management-Implementierung

## E.1. Class ResultPerformer

### Declaration

public class ResultPerformer
**extends** java.lang.Object
**implements** edu.udo.cs.sirena.service.discovery.ISearchCallback

### Constructors

- `public` **ResultPerformer**`( )`

### Methods

- `public String` **getEndpointLocation**`( )`

---

- `public String` **getEndpointPath**`( )`

---

- `public void` **onCachedServiceFound**`( java.lang.String` **searchID**`, edu.udo.cs.sirena.s` **rm**`)`

---

- `public void` **onGetResponseFound**`( java.lang.String` **arg0**`, java.lang.String` **arg1**`, java.lang.String` **arg2**`, edu.udo.cs.sirena.util.QualifiedNameVector` **arg3**`)`

---

- `public void` **onLocalServiceFound**`( java.lang.String` **arg0**`, java.util.Vector` **arg1**`)`

---

- `public void` **onProbeMatchFound**`( java.lang.String` **arg0**`, java.lang.String` **arg1**`, java.lang.String` **arg2**`, edu.udo.cs.sirena.util.QualifiedNameVector` **arg3**`, java.util.Vector` **arg4**`, int` **arg5**`)`

---

- `public void` **onServiceFound**`( java.lang.String` **searchID**`, edu.udo.cs.sirena.service` **rs**`)`

---

- `public void` **performResults**`( java.util.Vector` **results**`, java.util.Hashtable` **additionalTable**)
  - **Usage**
    - performResults wird aufgerufen, wenn das Ergebnis der Policyauswertung ein oder mehrere Resultobjekte sind (nur bei Policies der Fall, die das ResultPolicy-Interface implementieren). Die Results werden im Vector results übergeben und in dieser Reihenfolge ausgeführt. Der Vector sollte nur Objekte enthalten, die das PolicyResult-Interface implementieren.
  - **Parameters**
    - `results` -
    - `additionalTable` -

## E.2. Class PolicySerializer

Diese Klasse wandelt Policy-Objekte in XML-Objekte um und umgekehrt.

### Declaration

public class PolicySerializer
**extends** java.lang.Object

### Methods

- `public static AbstractPolicy` **fromXML**`( edu.udo.cs.sirena.xml.XMLElement` **xml**)
  - **Usage**
    - Wandelt ein XML-Objekt in eine Policy um.
  - **Parameters**
    - `xml` - ein XMLElement
  - **Returns** - ein Policy-Objekt

---

- `public static AbstractPolicy` **fromXMLString**`( java.lang.String` **xml**)
  - **Usage**
    - Wandelt einen String im XML-Format in ein Policy-Objekt um.
  - **Parameters**
    - `xml` - String im XML-Format
  - **Returns** - Policy-Objekt

---

- `public static XMLElement` **getPoliciesFromPool**`( )`
  - **Usage**
    - Diese Methode holt alle Policies aus dem PolicyPool und liefert diese in einem XMLElement zurück.
  - **Returns** - XMLElement, das alle Policies aus dem lokalen PolicyPool enthält.

---

- `public static Vector` **policiesFromXML**`( java.lang.String` **policyXMLString**`)`
  - **Usage**
    - Wandelt die im gegebenen String im XML-Format enthaltenen Policies in Policy-Objekte um und liefert sie in einem Vector zurück.
  - **Parameters**
    - `policies` - String im XML-Format
  - **Returns** - Vector, der Policy-Objekte enthält.

---

- `public static Vector` **policiesFromXML**`( edu.udo.cs.sirena.xml.XMLElement` **policies**`)`
  - **Usage**
    - Wandelt die im gegebenen XML-Objekt enthaltenen Policies in Policy-Objekte um und liefert sie in einem Vector zurück.
  - **Parameters**
    - `policies` - XMLElement, das die Policies enthält
  - **Returns** - Vector, der Policy-Objekte enthält.

---

- `public static XMLElement` **policiesToXML**`( java.util.Vector` **policies**`)`
  - **Usage**
    - Wandelt die im gegebenen Vector enthaltenen Policies in ein XMLElement um.
  - **Parameters**
    - `policies` - Vector, der Policy-Objekte enthält.
  - **Returns** - XMLElement, das die in XML-Objekte konvertierten Policies enthält.

---

- `public static void` **putPoliciesIntoPool**`( java.lang.String` **policies**`)`
  - **Usage**
    - Wandelt die im übergebenen String im XML-Format enthaltenen Policies in Policy-Objekte um und transferiert diese in den lokalen PolicyPool.
  - **Parameters**
    - `policies` - String im XML-Format, der Policies enthält

  - **Exceptions**
    * `java.lang.IllegalArgumentException` -

---

- `public static void` **putPoliciesIntoPool**`( edu.udo.cs.sirena.xml.XMLElement` **policies**)
  - **Usage**
    * Wandelt die im übergebenen XML-Objekt als XMLElement enthaltenen Policies in Policy-Objekte um und transferiert diese in den lokalen PolicyPool.
  - **Parameters**
    * `policies` -
  - **Exceptions**
    * `java.lang.IllegalArgumentException` -

---

- `public static XMLElement` **toXML**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**)
  - **Usage**
    * Wandelt ein Policy-Objekt in ein XML-Objekt um.
  - **Parameters**
    * `Policy-Objekt` -
  - **Returns** - XML-Objekt

---

- `public static String` **toXMLString**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**)
  - **Usage**
    * Wandelt ein Policy-Objekt in einen XML-formatierten String um.
  - **Parameters**
    * `Policy-Objekt` -
  - **Returns** - XML-String

## E.3. Class AccessManagement

Diese Klasse beinhaltet die Managementfunktionalität für Devices. Sie wird bei jedem Action-Aufruf befragt und prüft, ob der Benutzer die Erlaubnis hat, die Action aufzurufen und ob der Action-Aufruf überhaupt möglich ist.

### Declaration

public class AccessManagement
**extends** java.lang.Object

Methods

- `public static void` **activate**`( )`
  - **Usage**
    - Diese Methode aktiviert das AccessManagement.

---

- `public static boolean` **isActive**`( )`
  - **Usage**
    - Diese Methode prüft, ob das AccessManagement aktiviert ist.
  - **Returns** - true, wenn das Management schon aktiviert wurde

---

- `public static void` **onActionInvocation**`( edu.udo.pg490.services.AbstractDeviceServi` **service,** `edu.udo.cs.sirena.service.Action` **action**`)`
  - **Usage**
    - onActionInvokation wird jedes mal aufgerufen, wenn in einem Device eine Action aufgerufen wird. Falls das AccessManagement vorher aktiviert wurde, entscheidet das Management dann, ob dieses Action durchgeführt werden kann/darf. Falls das AccessManagement nicht aktiviert wurde, wird die Action einfach durchgeführt.
  - **Parameters**
    - `service` - Der Service auf die Action aufgerufen wurde.
    - `action` - Die aufgerufene Action.
  - **Exceptions**
    - `edu.udo.cs.sirena.communication.soap.SOAPException` - Falls der Actionaufruf nicht durchgeführt werden kann/darf wird eine Exception geworfen, die beim Actionaufrufer als SOAP-Fault in einer SOAP-Exception landet.

## E.4. Class PersonalManagement

Diese Klasse beinhaltet die Managementfunktionalität für das PersonalDevice und entscheidet nach jedem Aufruf, was als nächstes gemacht werden soll.

Declaration

public class PersonalManagement
**extends** java.lang.Object

Methods

- `public void` **addPolicy**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**)
  - **Usage**
    - Diese Methode wird aufgerufen, wenn eine neue/modifizierte Policy zum PolicyPool hinzugefügt werden muss.
  - **Parameters**
    - `policy` - Die Policy, die im PolicyPool gespeichert wird

---

- `public void` **dialogOnPD**`( java.lang.String` **message**`, int` **type**`, int` **timeout**)
  - **Usage**
    - Diese Methode wird benutzt, wenn die Fehlermeldung auf dem Display von Personal Device angezeigt werden soll.
  - **Parameters**
    - `message` - Nachricht des Dialogs
    - `type` - Typ des Dialogs
    - `timeout` - Timeout des Dialogs (0 für keinen)
  - **See Also**
    - `edu.udo.pg490.personaldevice.services.PersonalDeviceDialogServiceImpl`

---

- `public static PersonalManagement` **getInstance**`( )`
  - **Usage**
    - Mit dieser Methode bekommt man eine Instanz des PersonalManagements. Damit erhählt man jedesmal die gleiche Referenz auf das Objekt.
  - **Returns** - Eine Instanz des PersonalManagements

---

- `public Vector` **getPoliciesByType**`( int` **policyType**)
  - **Usage**
    - Diese Methode wird aufgerufen, wenn eine Policy geändert werden muss. Dazu werden alle Policies vom bestimmten Typ aus dem PolicyPool geholt.
  - **Parameters**
    - `policyType` - Policytyp als int (aus der Konstantenklasse)
  - **Returns** - Alle aktiven Policies vom angebenen Typ als Vector
  - **See Also**
    - `edu.udo.pg490.management.ManagementConstants`

---

- `public Vector` **getPreferredDevice**`( java.lang.String` **serviceNamespace**`, java.lang.S`
  **servicePortType**)
  - **Usage**
    - Diese Methode wird aufgerufen, wenn der Benutzer den Service aufrufen will und nach dem Präferenzgerät fragt.
  - **Parameters**
    - `serviceNamespace` - Namespace vom Service
    - `servicePortType` - Porttype vom Service
  - **Returns** - foundServices Eine Liste (Vector) mit remote Services von Präferenzgeräten

---

- `public static boolean` **isActive**`( )`
  - **Usage**
    - Mit dieser Methode kann abgefragt werden, ob das PersonalManagement aktiv ist!
  - **Returns** - `true` wenn das PersonalMangement aktiv ist. Ansonsten `false`.

---

- `public void` **onActionFailure**`( edu.udo.cs.sirena.communication.soap.SOAPException`
  **exception**)
  - **Usage**
    - Diese Methode wird aufgerufen, wenn Actionaufrufe fehlgeschlagen sind und eine SOAPException empfangen wurde. Die Methode onActionFailure entscheidet, welche Fehlerbehandlung durchgeführt werden muss. Der Nutzer kriegt eine Fehlermeldung mit dem Grund des Fehlers als Dialog angezeigt.
  - **Parameters**
    - `exception` - Die empfangene SOAPException

---

- `public void` **onDeviceServiceInvocation**`( edu.udo.cs.sirena.service.remote.RemoteSer`
  **service**`, edu.udo.cs.sirena.service.Action` **action**)
  - **Usage**
    - Diese Methode wird aufgerufen, wenn der Nutzer eine Action auf dem Service aufrufen will. Zu dieser Action werden nach der Policy-Auswertung entsprechende Konfigurationen geladen.
  - **Parameters**
    - `service` - Service, auf dem Action aufgerufen werden muss
    - `action` - Die aufgerufene Action
  - **Exceptions**
    - `edu.udo.cs.sirena.communication.soap.SOAPException` - Falls der Actionaufruf nicht durchgeführt werden kann/darf wird eine Exception geworfen, die beim Actionaufrufer als SOAP-Fault in einer SOAP-Exception landet.

---

- `public void` **onEventIncoming**`( edu.udo.cs.sirena.service.Action` **event**`)`
  - **Usage**
    - Diese Methode wird aufgerufen, wenn ein Ereignis eintrifft.
  - **Parameters**
    - `event` - Ereignis, das eingetroffen ist

---

- `public void` **onServicePropertyChange**`( edu.udo.pg490.services.AbstractDeviceService` **service**`, edu.udo.pg490.services.ServiceProperty` **property**`)`
  - **Usage**
    - Diese Methode wird aufgerufen, wenn Propertyänderungen auftreten. Sie werden nach der Policy-Auswertung durchgeführt.
  - **Parameters**
    - `service` - Service, auf dem Properties geändert wurden
    - `property` - Die geänderte Property

---

- `public void` **onTimerExpiration**`( edu.udo.pg490.management.policies.TimedPolicy` **policy**`)`
  - **Usage**
    - Diese Methode wird aufgerufen, wenn der Timer abläuft.
  - **Parameters**
    - `policy` - Zeitgesteuerte Policies vom Typ TimedPolicy, die für die Policy-Auswertung benötigt werden.

---

- `public void` **removePolicy**`( java.lang.String` **policyIdentifier**`)`
  - **Usage**
    - Diese Methode wird aufgerufen, wenn eine Policy gelöscht werden muss. Sie wird aus dem PolicyPool und aus dem PolicyTimer entfernt.
  - **Parameters**
    - `policyIdentifier` - UUID der Policy

## E.5. Class PolicyBuilder

---

Mit Hilfe dieser Klasse können die verschiedenen Policies zentral erstellt und im PolicyPool gespeichert werden

Declaration

public class PolicyBuilder
**extends** java.lang.Object

Constructors

- `public` **PolicyBuilder**`( )`

Methods

- `public void` **addParameterToListForResult**`( java.lang.String` **propertyKey**`, java.lang.String` **value**`)`
  - **Usage**
    - anstelle der Methode addResult eine Hashtable mit Properties als InputParamtern zu übergeben, kann man auch mit Hilfe dieser Methode die Properties erstellen. Bei Aufruf der Methode addResult mit dem Wert null für die Hashtable werden dann die hier erzeugten Properties als InputParameter genommen. Gilt für die Methoden addResult, die eine auszuführende Action definieren.
  - **Parameters**
    - `propertyKey` -
    - `value` -

---

- `public void` **addResult**`( edu.udo.cs.sirena.util.QualifiedName` **deviceType**`, edu.udo.c` **servicePortType**`, java.util.Vector` **scopes**`, java.lang.String` **scopeMatchingRule**`, java.lang.String` **actionName**`, java.util.Hashtable` **inputParameters**`)`
  - **Usage**
    - fügt der Policy ein Result hinzu, welches die spezifizierte Aktion auf allen dem Muster entsprechenden Geräten ausführt
  - **Parameters**
    - `deviceType` - der Gerätetyp(QualifiedName) der aufzurufenden Dienste
    - `servicePortType` - der ServicePorttype(QualifiedName) der aufzurufenden Aktionen
    - `scopes` - Scope zum Einschränken der Ergebnisse
    - `scopeMatchingRule` - Die Methode, mit dem die Scopes der Geräte verglichen werden soll
    - `actionName` - Name der Action, die auf den Geräten ausgeführt werden soll
    - `inputParameters` - Parameter der Aktion

---

- `public void` **addResult(** `java.lang.String` **eventName,** `java.lang.String` **servicePortType,** `java.util.Hashtable` **outputParameters)**
  - **Usage**
    - fügt der Policy ein Result hinzu, welches ein Event feuert
  - **Parameters**
    - `eventName` - Name des zu werfenden Events
    - `servicePortType` -
    - `outputParameters` - Parameter des Events

---

- `public void` **addResult(** `java.lang.String` **propertyName,** `java.lang.String` **propertyValue,** `java.lang.String` **servicePortType)**
  - **Usage**
    - fügt der Policy ein Result hinzu, das auf dem Gerät, auf dem die Policy ausgewertet wird eine Property ändert.
  - **Parameters**
    - `propertyName` - Name der zu Ändernden Property
    - `propertyValue` - Wert der zu Ändernden Property
    - `servicePortType` - Porttype, zu dem die Property gehört

---

- `public void` **addResult(** `java.lang.String` **actionName,** `java.lang.String` **deviceUUID,** `java.lang.String` **servicePortType,** `java.util.Hashtable` **inputParameters)**
  - **Usage**
    - fügt der Policy ein Result hinzu, welches die spezifizierte Action auf einem bestimmten Gerät ausführt
  - **Parameters**
    - `actionName` - Name der Action, die auf den Geräten ausgeführt werden soll
    - `deviceUUID` - UUID des Gerätes, auf dem die Action ausgeführt werden soll
    - `servicePortType` - der ServicePorttype der aufzurufenden Aktionen
    - `inputParameters` - Parameter der Aktion

---

- `public void` **addResult(** `java.lang.String` **deviceType,** `java.lang.String` **servicePortType,** `java.lang.String` **actionName,** `java.util.Hashtable` **inputParameters,** `boolean` **useLocal)**
  - **Usage**
    - fügt der Policy ein Result hinzu, welches die spezifizierte Aktion auf allen dem Muster entsprechenden Geräten ausführt
  - **Parameters**
    - `deviceType` - der Gerätetyp der aufzurufenden Dienste
    - `servicePortType` - der ServicePorttype der aufzurufenden Aktionen
    - `actionName` - Name der Action, die auf den Geräten ausgeführt werden soll

    * `inputParameters` - Parameter der Aktion
    * `useLocal` - gibt an, ob eine action nur auf Geräten ausgeführt wird, welche sich in dem gleichen Raum wie der Nutzer befindet

---

- `public void` **addResult**`( java.lang.String` **deviceType**`, java.lang.String` **servicePortType**`, java.util.Vector` **scopes**`, java.lang.String` **scopeMatchingRule**`, java.lang.String` **actionName**`, java.util.Hashtable` **inputParameters**`)`
  - **Usage**
    * fügt der Policy ein Result hinzu, welches die spezifizierte Aktion auf allen dem Muster entsprechenden Geräten ausführt
  - **Parameters**
    * `deviceType` - der Gerätetyp der aufzurufenden Dienste
    * `servicePortType` - der ServicePorttype der aufzurufenden Aktionen
    * `scopes` - Scope zum Einschränken der Ergebnisse
    * `scopeMatchingRule` - Die Methode, mit dem die Scopes der Geräte verglichen werden soll
    * `actionName` - Name der Action, die auf den Geräten ausgeführt werden soll
    * `inputParameters` - Parameter der Aktion

---

- `public void` **addToANDLevel**`( )`
  - **Usage**
    * schließt die Definition des Literals ab und fügt es als Konjunktion einer konjunktiven Normalform hinzu

---

- `public void` **addToNewOrLevel**`( )`
  - **Usage**
    * schließt die Definition des Literals ab und fügt es als neue(!)Disjunktion einer konjunktiven Normalform hinzu

---

- `public void` **addToORLevel**`( )`
  - **Usage**
    * schließt die Definition des Literals ab und fügt es als Disjunktion einer konjunktiven Normalform hinzu

---

- `public void` **compareProperties**`( java.lang.String` **localPropertyName1**`, java.lang.S` **localPropertyName2**`)`
  - **Usage**

  * vergleicht eine Property mit einer zweiten. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setSeconPropertyToCompare zusammen, jedoch stehen hier nicht die Möglichkeiten zur Verfügung, den Wert der zweiten Poperty zu multiplizieren oder eine Abweichung zu erlauben

- **Parameters**
  * `localPropertyName1` -
  * `localPropertyName2` -

---

- `public void` **compareProperties**`( java.lang.String` **servicePortType**`, java.lang.String` **remotePropertyName**`, java.lang.String` **localPropertyName**`)`

  - **Usage**
    * vergleicht eine Property mit einer zweiten. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setSeconPropertyToCompare zusammen, jedoch stehen hier nicht die Möglichkeiten zur Verfügung, den Wert der zweiten Poperty zu multiplizieren oder eine Abweichung zu erlauben
  - **Parameters**
    * `servicePortType` -
    * `remotePropertyName` -
    * `localPropertyName` -

---

- `public void` **compareProperties**`( java.lang.String` **servicePortType1**`, java.lang.String` **remotePropertyName1**`, java.lang.String` **servicePortType2**`, java.lang.String` **remotePropertyName2**`)`

  - **Usage**
    * vergleicht eine Property mit einer zweiten. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setSeconPropertyToCompare zusammen, jedoch stehen hier nicht die Möglichkeiten zur Verfügung, den Wert der zweiten Poperty zu multiplizieren oder eine Abweichung zu erlauben
  - **Parameters**
    * `servicePortType1` -
    * `remotePropertyName1` -
    * `servicePortType2` -
    * `remotePropertyName2` -

---

- `public void` **compareProperty**`( java.lang.String` **propertyName**`, boolean` **valueToCompareWith**`)`

  - **Usage**
    * vergleicht eine Property mit einem Wert. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setValueToCompareWith zusammen
  - **Parameters**
    * `propertyName` -

    * valueToCompareWith -

---

- `public void` **compareProperty**`( java.lang.String` **propertyName**`, int` **valueToCompareWith**`)`
  - **Usage**
    * vergleicht eine Property mit einem Wert. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setValueToCompareWith zusammen
  - **Parameters**
    * propertyName -
    * valueToCompareWith -

---

- `public void` **compareProperty**`( java.lang.String` **propertyName**`, java.lang.String` **valueToCompareWith**`)`
  - **Usage**
    * vergleicht eine Property mit einem Wert. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setValueToCompareWith zusammen
  - **Parameters**
    * propertyName -
    * valueToCompareWith -

---

- `public void` **compareProperty**`( java.lang.String` **servicePortType**`, java.lang.String` **propertyName**`, boolean` **valueToCompareWith**`)`
  - **Usage**
    * vergleicht eine Property mit einem Wert. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setValueToCompareWith zusammen
  - **Parameters**
    * propertyName -
    * valueToCompareWith -

---

- `public void` **compareProperty**`( java.lang.String` **servicePortType**`, java.lang.String` **propertyName**`, int` **valueToCompareWith**`)`
  - **Usage**
    * vergleicht eine Property mit einem Wert. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setValueToCompareWith zusammen
  - **Parameters**
    * propertyName -
    * valueToCompareWith -

---

- `public void` **compareProperty**`( java.lang.String` **servicePortType**`, java.lang.String` **propertyName**`, java.lang.String` **valueToCompareWith**`)`
  - **Usage**
    - vergleicht eine Property mit einem Wert. Macht das gleiche wie die Methoden setFirstPropertyToCompare und setValueToCompareWith zusammen
  - **Parameters**
    - `propertyName` -
    - `valueToCompareWith` -

---

- `public void` **compareStringWithList**`( java.lang.String` **content**`, java.lang.String` **servicePortType**`, java.lang.String` **listName**`, boolean` **onlyOnPropertyChange**`)`
  - **Parameters**
    - `content` -
    - `servicePortType` -
    - `listName` -
    - `onlyOnPropertyChange` -

---

- `public void` **createNewLiteral**`( int` **typeOfCondition**`)`
  - **Usage**
    - erstellt ein Literal für die Condition einer Policy
  - **Parameters**
    - `typeOfCondition` - Art der Condition

---

- `public void` **createNewLiteral**`( int` **typeOfCondition**`, boolean` **isNegated**`)`
  - **Usage**
    - erstellt ein Literal für die Condition einer Policy
  - **Parameters**
    - `typeOfCondition` - Art der Condition
    - `isNegated` - gibt an, ob das Literal negiert werden soll

---

- `public void` **createNewPolicy**`( int` **policyType**`, java.lang.String` **policyName**`)`
  - **Usage**
    - erstellt eine neue Policy
  - **Parameters**
    - `policyType` - Art der zu erstellenden Policy
    - `policyName` - Name der zu erstellenden Policy

---

- `public void` **createNewPolicyFromTemplate**`( java.lang.String` **template**`)`
  - **Usage**
    - erstellt eine Policy aus einer im PolicyPool vorhandenen Vorlage
  - **Parameters**
    - `template` - Name der Vorlage oder UUID. Wirft eine Exception, wenn keine Policy vorhanden ist.
  - **Exceptions**
    - `edu.udo.pg490.management.PolicyPoolException` -

---

- `public void` **createNewTimeLiteral**`( int` **startHour**`, int` **endHour**`)`
  - **Usage**
    - erstellt ein TimeLiteral, das zu der Condition der Policy hinzugefügt wird. Es beschreibt, zwischen welchen Uhrzeiten die Policy gültig ist. Achtung: diese Methode hat nichts mit der TimedPolicy zu tun.
  - **Parameters**
    - `startHour` - die Stundenzahl des Starts der Gültigkeitsdauer
    - `endHour` - die Stundenzahl des Endes der Gültigkeitsdauer

---

- `public void` **createNewTimeLiteral**`( int` **startHour**`, int` **startMinute**`, int` **endHour**`, int` **endMinute**`)`
  - **Usage**
    - erstellt ein TimeLiteral, das zu der Condition der Policy hinzugefügt wird. Es beschreibt, zwischen welchen Uhrzeiten die Policy gültig ist. Achtung: diese Methode hat nichts mit der TimedPolicy zu tun.
  - **Parameters**
    - `startHour` - die Stundenzahl des Starts der Gültigkeitsdauer
    - `startMinute` - die Minutenzahl des Starts der Gültigkeitsdauer
    - `endHour` - die Stundenzahl des Endes der Gültigkeitsdauer
    - `endMinute` - die Minutenzahl des Endes der Gültigkeitsdauer

---

- `public void` **createNewTimeLiteral**`( int` **startHour**`, int` **startMinute**`, int` **startSecond**`, int` **endHour**`, int` **endMinute**`, int` **endSecond**`)`
  - **Usage**
    - erstellt ein TimeLiteral, das zu der Condition der Policy hinzugefügt wird. Es beschreibt, zwischen welchen Uhrzeiten die Policy gültig ist. Achtung: diese Methode hat nichts mit der TimedPolicy zu tun.
  - **Parameters**
    - `startHour` - die Stundenzahl des Starts der Gültigkeitsdauer
    - `startMinute` - die Minutenzahl des Starts der GGültigkeitsdauer

  - `startSecond` - die Sekundenzahl des Starts der Gültigkeitsdauer
  - `endHour` - die Stundenzahl des Endes der Gültigkeitsdauer
  - `endMinute` - die Minutenzahl des Endes der Gültigkeitsdauer
  - `endSecond` - die Sekundenzahl des Endes der Gültigkeitsdauer

---

- `public void` **editPolicy**`( java.lang.String s)`
  - **Usage**
    - bearbeitet eine im PolicyPool vorhandene Policy
  - **Parameters**
    - `s` - Name oder UUID der Policy.Wird der Name angegeben und es existien mehrere Policies mit diesem Namen wird die erste gefundene genommen. Wirft eine Exception, falls keine Policy vorhanden ist.
  - **Exceptions**
    - `edu.udo.pg490.management.PolicyPoolException` -

---

- `public void` **finishPolicy**`( )`
  - **Usage**
    - Abschluss der erstellten Policy. Die Policy wird im PolicyPool gespeichert. Fehlen in der Policy notwendige Informationen, wird vom PolicyPool aus ein Exception geworfen.
  - **Exceptions**
    - `edu.udo.pg490.management.InvalidPolicyException` -

---

- `public void` **saveAsTemplate**`( )`
  - **Usage**
    - Die bis dahin erstellte Policy wird so wie sie ist, als Vorlage in dem PolicyPool gespeichert (eine Überprüfung, ob alle für eine gültige Policy notwendigen Attribute gesetzt worden sind, findet nicht statt)

---

- `public void` **setAccessContent**`( java.lang.String` **userID**`)`
  - **Usage**
    - generiert eine Policy, die den allgemeinen Zugriff auf ein Gerät für den angegebenen User erlaubt
  - **Parameters**
    - `userID` - die ID des PersonalDevices, das den User eindeutig identifiziert.

---

- `public void` **setAccessContent**`( java.lang.String` **userID**`, java.lang.String` **servicePortType**`)`

  - **Usage**
    * generiert eine Policy, die den Zugriff auf einen Service des Gerätes für den angegebenen User erlaubt
  - **Parameters**
    * `userID` - die ID des PersonalDevices, das den User eindeutig identifiziert
    * `servicePortType` - der Name des Porttypes des Services, für den der Zugriff erlaubt wird

---

- `public void` **setAccessContent**`( java.lang.String` **userID**`, java.lang.String` **servicePortType**`, java.lang.String` **actionName**`)`
  - **Usage**
    * generiert eine Policy, die den Zugriff auf ein Service des Gerätes für den angegebenen User erlaubt
  - **Parameters**
    * `userID` - die ID des PersonalDevices, das den User eindeutig identifiziert
    * `servicePortType` - der Name des Porttypes des Services, für den der Zugriff erlaubt wird
    * `actionName` - der Name der Action, für den der Zugriff auf den jeweiligen ServicePorttype erlaubt wird

---

- `public void` **setCondition**`( edu.udo.pg490.management.policies.Condition c)`
  - **Usage**
    * Setzt eine schon vorhandene Condition für die Policy ein, so dass man keine eigene definieren muss
  - **Parameters**
    * `c` - die bereits vorhandene Condition

---

- `public void` **setConfigurationContent**`( int` **ambit**`, java.lang.String` **name**`, java.util` **properties**`)`
  - **Parameters**
    * `ambit` - spezifiziert, ob ein bestimmtes Gerät, ein bestimmter Gerätetyp oder allgemein ein Servicetyp konfiguriert werden soll
    * `name` - die UUID eines Gerätes, der DevicePorttype eines Gerätes oder der Porttype eines Services
    * `properties` - eine Hastable, die die zu konfigurierenden Properties als ServieProperty-Objekt enthält

---

- `public void` **setDifferenceForSecondProperty**`( int` **difference**`)`
  - **Usage**

  - wenn die erste Property mit einer zweiten Property verglichen wird, wird mit dieser Methode ein Differenzwert um den Wert der ersten Properties definiert, um dem der Werte der zweiten Property abweichen darf und die Auswertung denoch positiv ausfällt.
- **Parameters**
  - `difference` - der Wert, den das zweite Property nach oben oder nach unten abweichen darf

---

- `public void` **setEventName**`( java.lang.String` **eventName**`)`
  - **Usage**
    - setzt für den Fall, dass es sich bei der Policy um eine EventPolicy handelt den Namen des Events,bei dessen auftreten die Policy ausgewertet werden soll
  - **Parameters**
    - `eventName` - Name des relevanten Events

---

- `public void` **setFirstPropertyToCompare**`( java.lang.String` **propertyName**`)`
  - **Usage**
    - Setzt in dem aktuellen Literal das erste Property für den Vergleich
  - **Parameters**
    - `propertyName` - Name des Properties

---

- `public void` **setFirstPropertyToCompare**`( java.lang.String` **servicePortType**`, java.lang.String` **propertyName**`)`
  - **Usage**
    - Setzt in dem aktuellen Literal das erste Property für den Vergleich
  - **Parameters**
    - `servicePortType` - servicePorttype, zu dem das Property gehört
    - `propertyName` - Name des Properties

---

- `public void` **setFirstPropertyToCompare**`( java.lang.String` **servicePortType**`, java.lang.String` **propertyName**`, boolean` **onlyOnPropertyChange**`)`
  - **Usage**
    - Setzt in dem aktuellen Literal das erste Property für den Vergleich
  - **Parameters**
    - `servicePortType` - servicePorttype, zu dem das Property gehört
    - `propertyName` - Name des Properties
    - `onlyOnPropertyChange` - gibt an, ob das Property nur verglichen werden soll, wenn dieses Property gerade geändert werden soll (true) oder wenn es nicht geändert werden soll(false)

---

- `public void` **setLastModifier**`( java.lang.String` **lastModifier**`)`
  - **Usage**
    - setzt in der Policy den Namen des letzten Bearbeiters
  - **Parameters**
    - `lastModifier` - Name des letzten Bearbeiters

---

- `public void` **setListToCompareWith**`( java.lang.String` **servicePortType**`, java.lang.S`
  **listName**`, boolean` **onlyOnPropertyChange**`)`
  - **Parameters**
    - `servicePortType` -
    - `listName` -
    - `onlyOnPropertyChange` -

---

- `public void` **setListToCompareWith**`( java.lang.String` **namespace**`, java.lang.String`
  **servicePortType**`, java.lang.String` **listName**`, boolean` **onlyOnPropertyChange**`)`
  - **Parameters**
    - `namespace` -
    - `servicePortType` -
    - `listName` -
    - `onlyOnPropertyChange` -

---

- `public void` **setMultiplierForSecondProperty**`( int` **multiplier**`)`
  - **Usage**
    - wenn das erste Property mit einem zweiten Property verglichen wird, wird mit dieser Methode für einen Vergleich der Wert des zweiten Property multipliziert
  - **Parameters**
    - `multiplier` -

---

- `public void` **setNegated**`( )`
  - **Usage**
    - neigiert das Literal

---

- `public void` **setPreferenceContent**`( java.lang.String` **servicePortType**`, java.util.Vec`
  **preferences**`)`
  - **Usage**

    * setzt für den Fall, dass es sich bei der Policy um eine PreferencePolicy handelt für einen ServicePorttype die zu präferierenden Geräte
  - **Parameters**
    * `servicePortType` - Porttype, für den die Präferenzen festgelegt werden sollen
    * `preferences` - Ein Vektor, bestehend aus PreferenceObjetcs, welche nähere Informationen zu den Geräten enthalten
  - **See Also**
    * `edu.udo.pg490.management.PreferenceObject`

---

- `public void` **setPriority**`( int` **priority**`)`
  - **Usage**
    * setzt die Priorität der Policy. Wird nicht bei allen Policies benötigt, da bei manchen Policies diese über andere Methoden implizit gesetzt werden
  - **Parameters**
    * `priority` - Priorität der Policy

---

- `public void` **setResult**`( edu.udo.pg490.management.policies.result.PolicyResult` **result**`)`

---

- `public void` **setResult**`( java.util.Vector` **results**`)`

---

- `public void` **setSecondPropertyToCompare**`( java.lang.String` **propertyName**`)`
  - **Usage**
    * setzt für das aktuelle Literal eine zweite Property zum Vergleich fest
  - **Parameters**
    * `propertyName` - Property, das verglichen werden soll

---

- `public void` **setSecondPropertyToCompare**`( java.lang.String` **servicePortType**`, java.lang.String` **propertyName**`)`
  - **Usage**
    * setzt für das aktuelle Literal eine zweite Property zum Vergleich fest
  - **Parameters**
    * `servicePortType` - Porttype der Property, die vergleichen werden soll
    * `propertyName` - Property, das verglichen werden soll

---

- `public void` **setTimeContent**`( java.util.Date` **start**`, java.util.Date` **end**`, long` **period**`)`

- **Usage**
  - setzt für den Fall, dass es sich bei der Policy um eine TimedPolicy handelt die relevanten Zeitdaten. Das Enddatum kann null sein,dann gibt es keine Befristung für die Policy
- **Parameters**
  - `start` - Startzeitpunkt, an dem angefangen wird, die Policy auszuwerten
  - `end` - den Endzeitpunkt, an dem mit der Auswertung der Policy aufgehört wird. Ist der Wert null, gibt es keinen Endzeitpunkt.
  - `period` - das Intervall, mit dem nach dem Startzeiptunkt die Policy erneut ausgewertet wird

---

- `public void` **setTimeContent(** `int` **startYear,** `int` **startMonth,** `int` **startDate,** `int` **startHourOfDay,** `int` **startMinute,** `int` **periodInMinutes)**
  - **Usage**
    - setzt für den Fall, dass es sich bei der Policy um eine TimedPolicy handelt die relevanten Zeitdaten. Die Policy erhält keine Zeitbeschränkung
  - **Parameters**
    - `startYear` - Jahr, an dem die Policy startet
    - `startMonth` - Monat, an dem die Policy startet
    - `startDate` - Tag des Monats, an dem die Policy startet
    - `startHourOfDay` - Stunde, in dem die Policy startet
    - `startMinute` - Minute, in dem die Policy startet
    - `periodInMinutes` - Interval zwischen 2 Überprüfungen in Minuten

---

- `public void` **setTimeContent(** `int` **startYear,** `int` **startMonth,** `int` **startDate,** `int` **startHourOfDay,** `int` **startMinute,** `int` **periodInMinutes,** `int` **endYear,** `int` **endMonth,** `int` **endDate,** `int` **endHourOfDay,** `int` **endMinute)**
  - **Usage**
    - setzt für den Fall, dass es sich bei der Policy um eine TimedPolicy handelt die relevanten Zeitdaten. Die Policy erhält eine Zeitbeschränkung
  - **Parameters**
    - `startYear` - Jahr, an dem die Policy startet
    - `startMonth` - Monat, an dem die Policy startet
    - `startDate` - Tag des Monats, an dem die Policy startet
    - `startHourOfDay` - Stunde, in dem die Policy startet
    - `startMinute` - Minute, in dem die Policy startet
    - `periodInMinutes` - Interval zwischen 2 Überprüfungen in Minuten
    - `endYear` - Jahr, an dem die Policy endet, kann null sein
    - `endMonth` - Monat, an dem die Policy endet, kann null sein
    - `endDate` - Tag des Monats, an dem die Policy endet, kann null sein
    - `endHourOfDay` - Stunde, zu der die Policy endet, kann null sein
    - `endMinute` - Minute, zu der die Policy endet, kann null sein

---

- `public void` **setTimedPolicyConditionType**`( int` **conditionType**`)`

---

- `public void` **setValueToCompareWith**`( boolean` **value**`)`
  - **Usage**
    - setzt für das aktuelle Literal einen boolean Wert zum Vergleich mit einer Property fest
  - **Parameters**
    - `value` - boolean Wert, der verglichen werden soll

---

- `public void` **setValueToCompareWith**`( int` **value**`)`
  - **Usage**
    - setzt für das aktuelle Literal einen int Wert zum Vergleich mit einer Property fest
  - **Parameters**
    - `value` - int Wert, der verglichen werden soll

---

- `public void` **setValueToCompareWith**`( java.lang.String` **value**`)`
  - **Usage**
    - setzt für das aktuelle Literal einen String Wert zum Vergleich mit einer Property fest
  - **Parameters**
    - `value` - String Wert, der verglichen werden soll

## E.6. Class PolicyEvaluation

Prüft, ob Conditions von Policies erfüllt sind.

### Declaration

public class PolicyEvaluation
**extends** java.lang.Object

### Methods

- `public Vector` **evaluate**`( java.util.Hashtable` **h2**`, int` **i**`)`
  - **Usage**

    * Wertet die Conditions von Policies aus. Der Typ der zu betrachtenden Policies wird durch den Parameter i angegeben. Hier werden nur die Ergebnisse gültiger Policies der jeweils höchsten Priorität zurückgeliefert.
  - **Parameters**
    * `h2` - Hashtable mit Parametern von Actions, etc. die zur Auswertung benötigt werden.
    * `i` - Policytyp als int. Gültige Konstanten sind in der Klasse ManagementConstants definiert.
  - **Returns** - Vector mit den Ergebnissen

---

- `public Vector` **evaluatePolicy**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**`)`
  - **Usage**
    * Wertet die Condition einer gegebenen Policy aus. Falls deren Condition erfüllt ist, werden ihre Ergebnisse in einem Vector zurückgeliefert. Wird genutzt für TimedPolicies.
  - **Parameters**
    * `policy` - TimedPolicy, deren Condition geprüft werden soll.
  - **Returns** - Vector mit den Ergebnissen

---

- `public static PolicyEvaluation` **getInstance**`( )`
  - **Usage**
    * Liefert eine Instanz von PolicyEvaluation. Sorgt auch dafür, dass es maximal eine Instanz von PolicyEvaluation auf einem Gerät gibt.
  - **Returns** - Instanz von PolicyEvaluation

## E.7. Class PolicyPool

Diese Klasse ist verantwortlich für das speichern von Policies und PolicyTemplates.

### Declaration

public class PolicyPool
**extends** java.lang.Object

### Methods

- `public void` **activatePolicies**`( java.util.Vector` **policyIdentifiers**`)`

- **Usage**
  - Aktiviert **alle** zu den übergebenen PolicyIdentifiern gehörenden Policies. Diese stehen danach der **Auswertung** durch eine entsprechende Managementkomponente zur Verfügung
- **Parameters**
  - `policyIdentifiers` - die eindeutigen PolicyIdentifier der zu aktiviernden Policies
- **Exceptions**
  - `edu.udo.pg490.management.PolicyPoolException` - wird geworfen,falls eine oder mehrere Policies bei den gespeicherten inaktiven Policies gefunden wird. Es werden **alle gefundenen** Policies auch **aktiviert**.

---

- `public void` **activatePolicy**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**`)`
  - **Usage**
    - Aktiviert die übergebene Policy. Die Policy steht danach der **Auswertung** durch eine entsprechende Managementkomponente zur Verfügung
  - **Parameters**
    - `policy` - Die zu aktivierende Policy
  - **Exceptions**
    - `edu.udo.pg490.management.PolicyPoolException` - wird geworfen, falls die übergebene Policy nicht bei den inaktiven Policies gefunden wurde

---

- `public void` **activatePolicy**`( java.lang.String` **policyIdentifier**`)`
  - **Usage**
    - Aktiviert eine Policy mit angegebenem Identifier. Diese Policy steht danach für die **Auswertung** durch eine Managementkomponente zur Verfügung
  - **Parameters**
    - `policyIdentifier` - Der Identifier der die Policy eindeutig identifiziert
  - **Exceptions**
    - `edu.udo.pg490.management.PolicyPoolException` - wird geworfen, falls es keine inaktive Policy mit angegebenem Identifier im PolicPool gibt

---

- `public void` **addInactivePolicies**`( java.util.Vector` **policies**`)`
  - **Usage**
    - Fügt dem Policy **mehrere** Policies aufeinmal in **inaktivem** Zustand hinzu
  - **Parameters**
    - `policies` - Die zu speichernden Policies

---

- `public void` **addInactivePolicy**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**)
  - **Usage**
    - Fügt dem Policy ein Policy in **inaktivem** Zustand hinzu
  - **Parameters**
    - `Policies` - Die zu speichernde Policy

---

- `public void` **addPolicies**`( java.util.Vector` **policies**)
  - **Usage**
    - Fügt dem Policy **mehrere** Policies aufeinmal in **aktivem** Zustand hinzu
  - **Parameters**
    - `policies` - Die zu speichernden Policies

---

- `public void` **addPolicy**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**)
  - **Usage**
    - Diese Methode fügt dem PolicyPool eine Policy im **aktiven** Zustand hinzu.
  - **Parameters**
    - `policy` - Die zu speichernde Policy

---

- `public void` **addPolicy**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**`, int` **policyType**)
  - **Usage**
    - Diese Methode fügt dem PolicyPool eine Policy im aktiven Zustand hinzu, der Parameter int policyTyp ist optional.
  - **Parameters**
    - `policy` - Die zu speichernde Policy
    - `policyType` - Der PolicyTyp der zu speichernden Policy

---

- `public void` **addPolicyTemplate**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**)
  - **Usage**
    - Fügt dem PolicyPool eine **PolicyVorlage** hinzu
  - **Parameters**
    - `policy` - Die PolicyVorlage

---

- `public void` **deactivatePolicies**`( java.util.Vector` **policyIdentifiers**)

- **Usage**
  - Deaktiviert **alle** zu den **PolicyIdentifiern** gehörenden Policies im PolicyPool.
- **Parameters**
  - `policyIdentifiers` - die PolicyIdentifier der zu deaktivierenden Policies
- **Exceptions**
  - `edu.udo.pg490.management.PolicyPoolException` - wird geworfen, falls eine oder mehrere Policies nicht bei den gespeicherten, aktiven Policies gefunden wird. Es werden aber alle gefundenen Policies auch deaktiviert.

---

- `public void` **deactivatePolicy**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**`)`
  - **Usage**
    - Deaktiviert die übergebene Policy. Diese Policy wird dann bei der **Evaluierung** durch eine Managementkomponente **nicht** mehr betrachtet.
  - **Parameters**
    - `policy` - Die zu deaktivierende Policy
  - **Exceptions**
    - `edu.udo.pg490.management.PolicyPoolException` -

---

- `public void` **deactivatePolicy**`( java.lang.String` **policyIdentifier**`)`
  - **Usage**
    - Deaktiviert eine Policy mit angegebenem policyIdentifier. Diese Policy wird dann bei der **Evaluierung** durch eine Managementkomponente **nicht** mehr betrachtet.
  - **Parameters**
    - `policyIdentifier` - Der Identifier, der die zu deaktivierende Policy eindeutig identifiziert
  - **Exceptions**
    - `edu.udo.pg490.management.PolicyPoolException` - wird geworfen, wenn eine Policy mit angegebenem Idenifier nicht im PolicyPool exisitiert

---

- `public Hashtable` **getInactivePolicies**`( )`
  - **Usage**
    - Gibt alle gespeicherten **inaktiven** Policies zurück. Dabei wird nicht zwischen den verschiedenen Policytypen unterschieden.
  - **Returns** - Eine Hashtable mit den inaktiven Policies. Die Policies sind unter ihrem eindeutigen PolicyIdentifier gespeichert.

---

- `public static PolicyPool` **getInstance**`( )`

  - **Usage**
    * Mit dieser Methode bekommt man eine Instanz des PolicyPools.Da mehrere Managementkomponenten jeweils auf einen Policypool zugreifen und die Speicherung der Policies zentral erfolgen soll, erhählt man jedesmal die gleiche Referenz auf das Objekt, anstelle dass jede Managementkompopnente ihren eigenen PolicyPool instanziiert.
  - **Returns** - eine Instanz des PolicyPools

---

- `public Vector` **getPolicies**`( int` **policyType**`)`
  - **Usage**
    * Gibt die gesamten **aktiven** Policies eines bestimmten **PolicyTyps** zurück
  - **Parameters**
    * `policyType` - Der Typ der Policies, der zurückgegeben werden soll
  - **Returns** - alle gespeicherten, **aktiven** Policies des angegebenen PolicyTyps

---

- `public AbstractPolicy` **getPolicy**`( java.lang.String` **policyIdentifier**`)`
  - **Usage**
    * Sucht im PolicyPool nach einer **aktiven** / **inaktiven** Policy, die den übergebenen PolicyIdentifier hat und gibt diese zurück.
  - **Parameters**
    * `policyIdentifier` - Der Identifier, der die zu suchende Policy eindeutig identifiziert
  - **Returns** - Gibt die Policy mit dem angegebenen Identifier zurück, falls vorhanden.
  - **Exceptions**
    * `edu.udo.pg490.management.PolicyPoolException` - wird geworfen,falls es keine Policy mit diesem Identifier im PolicyPool gibt.

---

- `public Vector` **getPolicyByName**`( java.lang.String` **name**`)`
  - **Usage**
    * Durchsucht sämtliche gespeicherten Policies (auch die PolicyTemplates) nach einer Policy mit angegebenem Namen und gibt diese, falls vorhanden zurück
  - **Parameters**
    * `name` - Name der Policy
  - **Returns** - Policy mit angegebenem Namen, null falls es keine Policy mit diesem Namen gibt.

---

- `public AbstractPolicy` **getPolicyTemplate**`( java.lang.String` **policyIdentifier**`)`

- **Usage**
  - Gibt ein PolicyTemplate mit angegebenem Identifier zurück
- **Parameters**
  - `policyIdentifier` - Der Identifier, der die zu suchende Vorlage eindeutig identifiziert
- **Returns** - Die PolicyVorlage mit angegebenem Identifier

---

- `public AbstractPolicy` **getPolicyTemplateByName**`( java.lang.String` **policyName**`)`

---

- `public void` **removePolicy**`( edu.udo.pg490.management.policies.AbstractPolicy` **policy**`)`
  - **Usage**
    - Entfernt eine **aktive** / **inaktive** Policy aus dem PolicyPool
  - **Parameters**
    - `policy` - Die aus dem PolicyPool zu löschende Policy

---

- `public void` **removePolicy**`( java.lang.String` **policyIdentifier**`)`
  - **Usage**
    - Entfernt eine **aktive** / **inaktive** Policy aus dem PolicyPool
  - **Parameters**
    - `policyIdentifier` - Der Identifier, der die zu löschende Policy eindeutig identifiziert

---

- `public void` **removePolicyTemplate**`( java.lang.String` **policyIdentifier**`)`
  - **Usage**
    - Entfernt eine **PolicyVorlage** aus dem PolicyPool
  - **Parameters**
    - `policyIdentifier` - Der Identifier, der die zu löschende PolicyVorlage eindeutig identifiziert.

## E.8. Class StateManagement

Diese Klasse beinhaltet die Managementfunktionalität für Devices. Sie wird bei jeder Property-Änderung befragt und prüft, ob Bedingungen über den Properties erfüllt sind.

Declaration

public class StateManagement
**extends** java.lang.Object

Methods

- `public static void` **activate**`( )`
  - **Usage**
    - Diese Methode aktiviert das StateManagement.

---

- `public static boolean` **isActive**`( )`
  - **Usage**
    - Diese Methode prüft, ob das StateManagement aktiviert ist.
  - **Returns** - true, wenn das Management schon aktiviert wurde

---

- `public static void` **onServicePropertyChange**`( edu.udo.pg490.services.AbstractDevic`
  **service,** `edu.udo.pg490.services.ServiceProperty` **property**`)`
  - **Usage**
    - onServicePropertyChange wird jedes mal aufgerufen, wenn sich ein Property in einem Device verändert hat. Falls das StateManagement vorher aktiviert wurde, werden Managementfunktionen basierend auf Policies durchgeführt. Die Propertyveränderung wird in jedem Fall durchgeführt nach diesem Aufruf (extern).
  - **Parameters**
    - `service` - Der Service in dem sich die Property geändert hat.
    - `property` - Das PropertyObjekt, das sich geändert hat.

# F. Gruppenfoto

**Abbildung 24:** Die PG 490

## Abbildungsverzeichnis



## Tabellenverzeichnis

## Quellcodeverzeichnis

## Literatur

[ABFG04] Austin, Daniel ; Barbir, Abbie ; Ferris, Christopher ; Garg, Sharad: *Web Services Architecture Requirements*. Version: Februar 2004. http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211/, Abruf: 16.04.2006. W3C Working Group Note

[ADLH+02] Atkinson, Bob ; Della-Libera, Giovanni ; Hada, Satoshi ; Hondo, Maryann ; Hallam-Baker, Phillip ; Kaler, Chris ; Klein, Johannes ; LaMacchia, Brian ; Leach, Paul ; Manferdelli, John ; Maruyama, Hiroshi ; Nadalin, Anthony ; Nagaratnam, Nataraj ; Prafullchandra, Hemma ; Shewchuk, John ; Simon, Dan: *Web Services Security (WS-Security)*. Version: April 2002. http://msdn.microsoft.com, Abruf: 23.04.2006

[Apf04] Apfel, Steffen: *Ein generisches Framework zur grafischen Anbindung von Web-Services*, Universität Kaiserslautern, Diplomarbeit, Dezember 2004

[Bel04] Bell, Jason: *Understand the autonomic manager concept*. Version: Februar 2004. http://www-128.ibm.com/developerworks/library/ac-amconcept/. IBM developerworks

[BL06] Booth, David ; Liu, Canyang K.: *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. Version: März 2006. http://www.w3.org/TR/wsdl20-primer/, Abruf: 16.04.2006. W3C Candidate Recommendation

[BRS03] Badach, Anatol ; Rieger, Sebastian ; Schmauch, Matthias: *Web Technologien - Architekturen, Konzepte, Trends*. 1. Auflage. Carl Hanser Verlag München, 2003. – 311–344 S. – ISBN 3–446–22149–2

[Can] *CANDELA – Content Analysis and Network DELivery Architectures*. http://www.hitech-projects.com/euprojects/candela/

[CCK+06] Chan, Shannon ; Conti, Dan ; Kaler, Chris ; Kuehnel, Thomas ; Regnier, Alain ; Roe, Bryan ; Sather, Dale ; Schlimmer, Jeffrey ; Sekine, Hitoshi ; Thelin, Jorgen ; Walter, Doug ; Weast, Jack ; Whitehead, Dave ; Wright, Don ; Yarmosh, Yevgeniy: *Devices Profile for Web Services*. Version: February 2006. http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf

[CCMW01] Christensen, Erik ; Curbera, Francisco ; Meredith, Greg ; Weerawarana, Sanjiva: *Web Services Description Language (WSDL) 1.1*. Version: März 2001. http://www.w3.org/TR/wsdl, Abruf: 16.04.2006. W3C Note

[CKK+05] Chan, S. ; Kaler, C. ; Kuehnel, T. ; A.Regnier ; Roe, B. ; Sather, D. ; Schlimmer, J. ; Sekine, H. ; Walter, D. ; Weast, J. ; Whitehead, D. ; Wright, D. ; Yarmosh, Y.: Devices Profile for Web Services. In: *Microsoft*

*Developers Network Library* 115 (2005), 05. http://specs.xmlsoap.org/ws/2005/05/devprof/devicesprofile.pdf

[DGH03] Dustdar, Schahram ; Gall, Harald ; Hauswirth, Manfred: *Software-Architekturen für Verteilte Systeme*. 1. Auflage. Springer Verlag, Berlin, 2003. – 113–142 S. – ISBN 3–540–43088–1

[ECL] *Eclipse - an open development platform*. Webseite. http://www.eclipse.org/

[F.J05] F.Jammes, A.Mensch und H.: Service-Oriented Device Communications Using the Devices Profile for Web Services. In: *ACM International Conference Proceeding Series* 115 (2005), 11

[HB06] Haas, Hugo ; Brown, Allen: *Web Services Glossary*. Version: Februar 2006. http://www.w3.org/TR/ws-gloss/, Abruf: 16.04.2006. W3C Working Group Note

[IBM05a] *Policy Management for Autonomic Computing: Developer's Guide and Reference*. Third Edition. IBM Tivoli, 2005 http://dl.alphaworks.ibm.com/technologies/pmac/PMDevGuide121.pdf

[IBM05b] IBM: An architectural blueprint for autonomic computing. Third Edition (2005), June. http://www-03.ibm.com/autonomic/pdfs/ACBlueprintWhitePaperV7.pdf

[IEE02] IEEE_802.15_WPAN_Task_Group_1: *IEEE 802.15.1 - A Wireless Personal Area Network standard based on the Bluetooth v1.1 Foundation Specifications*. http://www.ieee802.org/15/pub/TG1.html. Version: 06 2002

[JME] *Java ME Technology*. Webseite. http://java.sun.com/javame/technology/index.jsp

[KKSS03] Keidl, Markus ; Kemper, Alfons ; Seltzsam, Stefan ; Stocker, Konrad: Web Services. In: Rahm, Erhard (Hrsg.) ; Vossen, Gottfried (Hrsg.): *Web & Datenbanken*. 1. Auflage. dpunkt, 2003. – ISBN 3–89864–189–9, Kapitel 10, S. 293–331

[KL04] Kossmann, Donald ; Leymann, Frank: Web Services. In: *Informatik Spektrum* 27(2) (2004), S. 117–128

[Kuh] Kuhrmann, Dipl. Inf. M.: *Serviceorientierte Architekturen – Überblick*. http://www.software-kompetenz.de. Fraunhofer IESE

[L.F04] L.F.Cabrera, D.Box C.Kurt: *An Introduction to the Web Service Architecture and its Specifications*. Version 2.0. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/introWSA.asp. Version: 10 2004

[Mic05] Microsoft_Corporation: *Discovery, Plug and Play Integration, and Web Services for Network Connected Device*. WinHEC 2005 Version. http://www.microsoft.com/whdc/. Version: 04 2005

[Mit03] Mitra, Nilo: *SOAP Version 1.2 Part 0: Primer*. Version: Juni 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/, Abruf: 16.04.2006. W3C Recommendation

[Sir] *SIRENA – Service Infrastructure for Real-Time Embedded Networked Applications*. http://www.sirena-itea.org

[Slo94] Sloman, Morris: Policy driven Management for Distributed Systems. In: *Journal of Network and Systems Management* 2 (1994), Nr. 4

[STK02] Snell, James ; Tidwell, Doug ; Kulchenko, Pavel: *WebService-Programmierung mit SOAP*. 1. Auflage. O'Reilly, Köln, 2002. – 13–37 S. – ISBN 3–89721–159–9

[UPn] *UPnP Forum*. http://www.upnp.org