ROBERT BOSCH MULTIMEDIA-SYSTEME
GMBH & CO. KG

BOSCH

SCHLUSSBERICHT

über die Arbeiten im Zeitraum
vom 01. 01. 2000 bis 12. 02. 2002

Mobilität im Ballungsraum Stuttgart
MOBILIST

- Arbeitspaket C3 -
Anschlussinformationssystem ANIS

Projektteil Bosch
Kennzeichen 19B0009

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen 19 B 0009 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Aufgabenstellung

Das Forschungsprojekt MOBILIST, Arbeitspaket C3, Anschlussinformationssystem ANIS (nachfolgend: MOBILIST-ANIS) wird vom BMBF als Verbund- und Umsetzungsvorhaben gefördert und wurde von der Robert Bosch Multimedia-Systeme GmbH & Co KG (nachfolgend: MU) mit den MOBILIST-ANIS-Partnern

- Verkehrs- und Tarifverbund Region Stuttgart GmbH
- Stuttgarter Straßenbahnen AG
- Daimler Chrysler AG
- DB Regio AG
- Caatoosee AG

gemeinsam und partnerschaftlich auf der Grundlage eines Kooperationsvertrages bearbeitet.

Das Ergebnis von MOBILIST-ANIS soll zu den generellen Zielen der Attraktivitätssteigerung des öffentlichen Nahverkehrs sowie der Mobilität in Ballungsräumen beitragen. MOBILIST ist gleichzeitig der Einstieg in die Ausstattung von Fahrzeugen und Haltestellen der Verkehrsbetriebe in Stuttgart und in der Region mit Geräten für die Bereitstellung dynamischer Anschlussinformation. Die weitere Ausstattung von Fahrzeugen und Haltestellen ist vorgesehen.

Die Versorgung der Bahnen und Busse sowie der Haltestellen in Stuttgart und in der Region mit Fahrgastinformation auf der Grundlage der Übertragungstechnik DAB/DMB und den darauf basierenden Infrastruktur- und Endgeräte-Produkten eröffnet neue Wege und neue Qualitäten für eine kunden- und marktgerechte Information der Fahrgäste.

DAB (Digital Audio Broadcasting) ist der europäische Standard für die beschlossene Einführung des digitalen Rundfunks und mit der multimedialen Erweiterung DMB (Digital Multimedia Broadcasting) in hervorragender Weise für den Einsatz in mobilen und stationären Fahrgastinformationssystemen geeignet.

Die breitbandige Übertragungstechnologie erlaubt es, neben statischen und dynamischen Fahrplaninformationen multimediale Zusatzinformationen (z. B. Werbung) sowie aktuelle Informationen (z. B. aus Politik und Wirtschaft) in Bahnen/Busse und an Haltestellen zu übertragen und verzugslos zu präsentieren.

Das Gesamtsystem des auf der DAB/DMB-Technik basierenden Anschlussinformationssystem besteht aus den Teilsystemen Infrastruktur und Endgeräte.

INFRASTRUKTUR

Aufgaben der Infrastrukturbausteine sind:

- **Baustein RBL** der Verkehrsunternehmen: Bereitstellung der Fahrplandaten
- **Baustein Fahrplanserver**: Die Aufgabenstellung des Fahrplanservers ist die Endgegennahme, Speicherung und Aufbereitung der Fahrplandaten der angeschlossenen Verkehrsbetriebe sowie die Versorgung des DAB/DMB-Datenstudios mit den aufbereiteten Fahrplandaten zur Aussendung über DAB/DMB. Für MOBILIST wurde eine Aufteilung in die Komponenten C3-Datenserver und ANIS-Infopool vorgenommen.
- **Baustein Content Provider**: Sammlung und Aufbereitung der redaktionellen Beiträge (Unterhaltung, Werbung)
- **Baustein Datenstudio**: Die Aufgabenstellung des Datenstudios ist die Entgegennahme, Speicherung und DAB/DMB-gerechte Aufbereitung der Sendebeiträge des Content Providers und der vom Fahrplanserver bereitgestellten Fahrplandaten sowie die Zuführung dieser Daten zur Sendeinfrastruktur. Von dort erfolgt im L-Band die Versorgung der mobilen und stationären Endgeräte über DAB/DMB. Dafür wird die Verfügbarkeit eines kompletten Ensembles (1,5 Mbit) vorausgesetzt.

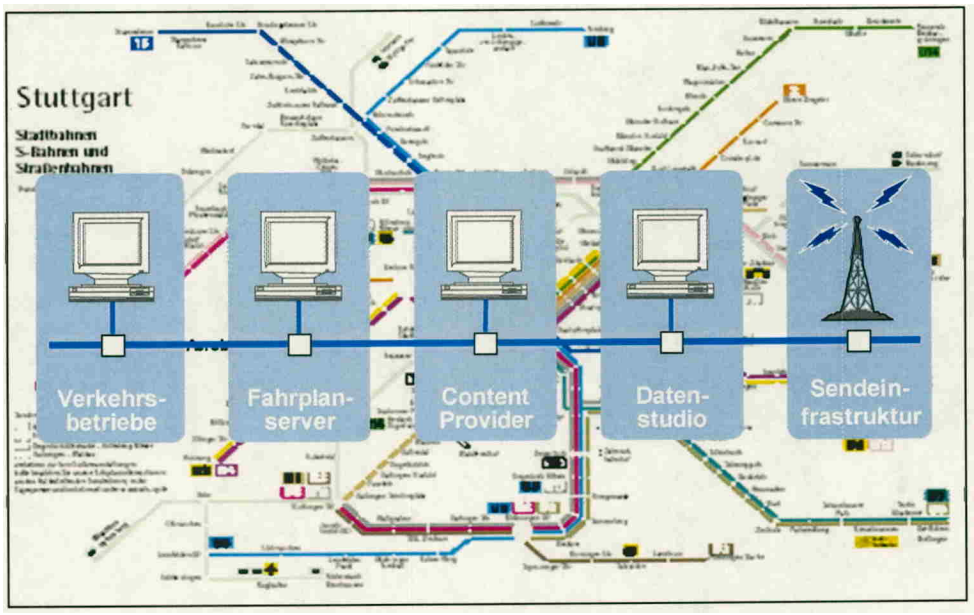


Abbildung 1: Systembausteine Infrastruktur

- **Baustein Sendenetz**: Verteilung aller Daten zwischen den Senderstandorten und Ausstrahlung über DAB/DMB.

ENDGERÄTE

Aufgaben der Endgeräte (Mobile Endgeräte in den Fahrzeugen, Stationäre Endgeräte an den Haltestellen) sind der Empfang, die Aufbereitung und Präsentation der über DAB/DMB zugeführten Fahrplan- und Zusatzinformationen.

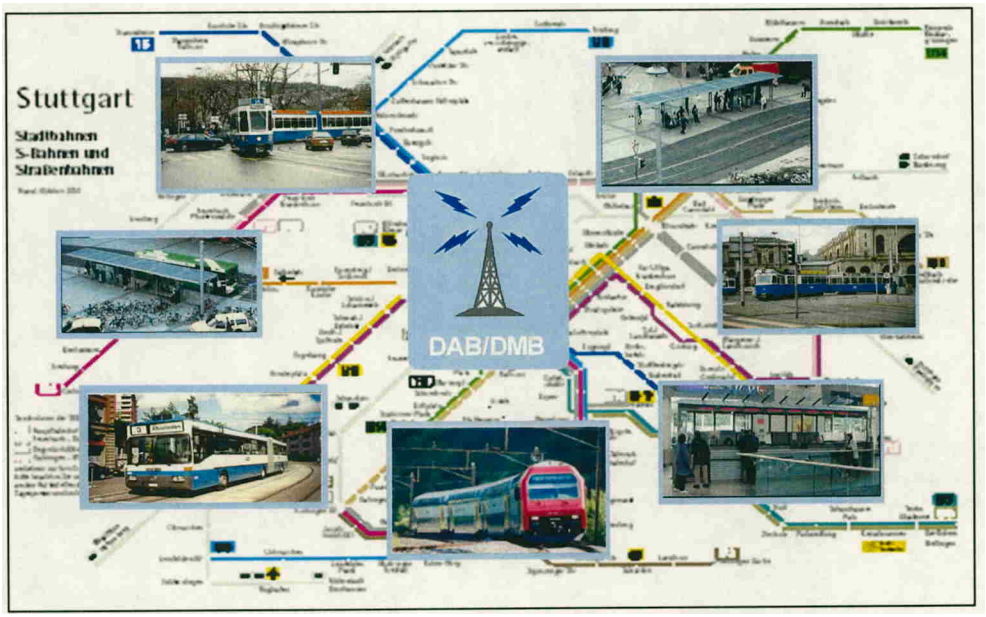


Abbildung 2: Endgeräte für Bahnen/Busse und Haltestellen/Infopunkte

Aufgabenstellung der MU war die Spezifikation, Entwicklung und Lieferung einer Anzahl von Endgeräten für Schienenfahrzeuge der SSB AG und der DB Regio AG sowie von Demonstratoren zur Erprobung der Haltestellenausstattung.

2. Voraussetzungen, unter denen das Vorhaben durchgeführt wurde

Zusätzlich zu den Arbeiten gem. BMBF-Zuwendungsbescheid (Aufbau der Infrastruktur, Inbetriebnahme der Endgerätedemonstratoren, Probebetrieb) sollte parallel zum Probebetrieb der Ausbau (durch Änderungen, Ergänzungen der MOBILIST-ANIS-Bausteine) im Hinblick auf die operationelle Nutzung erfolgen.
Dafür wurden im Projekt die erforderlichen Anwendungs- und Systemfunktionen für die Infrastrukturbausteine und insbesondere die stationären Endgeräte spezifiziert. Dazu hat die MU mit einem Partner Angebote zur Lieferung des ANIS-Infopools, des Datenstudios sowie zur DAB-gestützten Versorgung der Haltestellenendgeräte/-anzeiger unterbreitet.
Die sonst erheblichen Investitions- und Betriebskosten für die Netzinfrastruktur zur Versorgung der Endgeräte an den Haltestellen in der Region sollten durch die Mitnutzung der DAB/DMB-Bausteine (Datenstudio, Sendenetz) reduziert werden.

3. Planung und Ablauf des Vorhabens

Unter Berücksichtigung der Terminfestlegungen in den Zuwendungsbescheiden erfolgte gemeinsam und abgestimmt die detaillierte Planung der Arbeitspakete, Bearbeiter und Termine. Beschlüsse wurden durch Arbeitsgruppen vorbereitet und in den monatlichen Projektsitzungen unter der Leitung des VVS und des Projektkoordinators entschieden. Dabei wurden die Meilensteine des Gesamtvorhabens MOBILIST beachtet.
MU hat die Entwicklungsarbeiten Anfang Februar 2002 abgeschlossen und die Beendigung des Projektes zum 12. 02. 2002 beantragt.

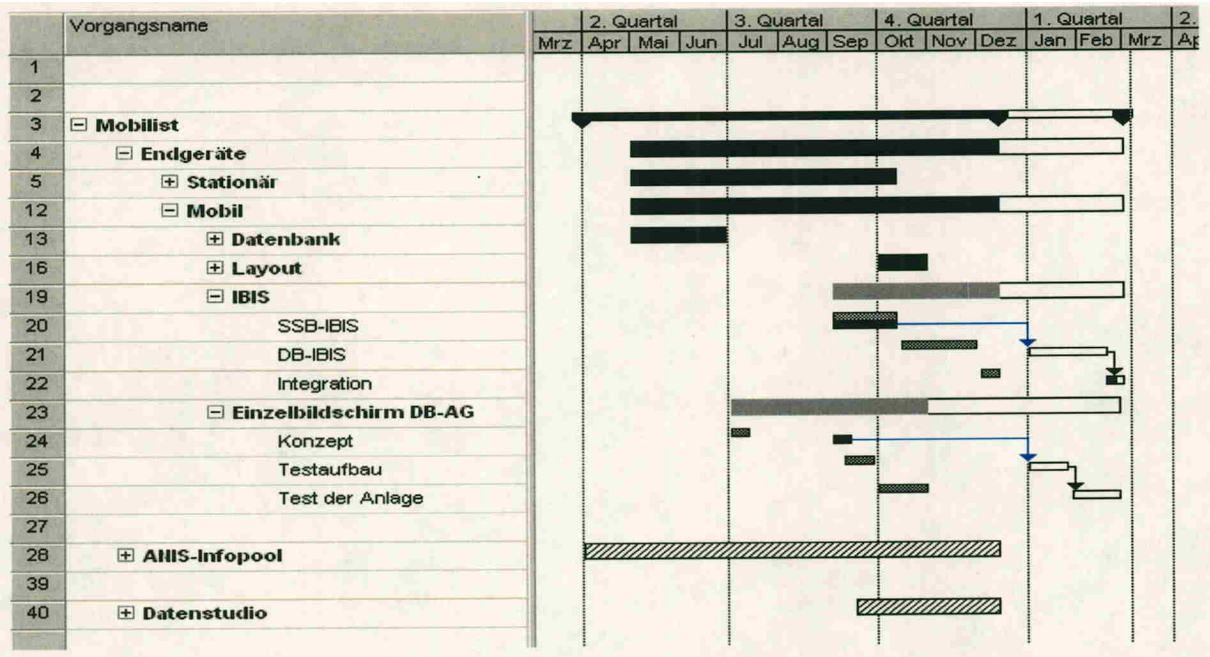


Abbildung 3: Mobilist Zeitplan

4. Wissenschaftlicher und technischer Stand zu Beginn des Vorhabens

Die für die DAB/DMB-gestützte breitbandige Versorgung der Endgeräte erforderlichen Kommunikationsstandards waren zu Beginn des Vorhabens vollständig spezifiziert und in diversen MU-Projekten realisiert. Ebenso die Datenübertragungsdienste und –verfahren der Schnittstellen zwischen dem Datenstudio und der Senderinfrastruktur.
Zu spezifizieren waren die Anwendungsschichten der Software auf den eingesetzten Servern der DAB/DMB-Endgeräte. Grundlagen dafür und für die durchgeführten Entwicklungsarbeiten (Anpassungen, Erweiterungen) waren die in diversen Projekten eingesetzten MU-Produkte.
Bearbeitet haben wir weiterhin die Schnittstellen zwischen den Rechnersystemen der Verkehrsbetriebe und dem ANIS-Infopool sowie zwischen dem ANIS-Infopool und dem Datenstudio. Die Ergebnisse wurden verbindlich festgeschrieben und sind in diverse Angebote eingeflossen.

Die Softwareentwicklung erfolgte nach dem Stand der Technik unter Beachtung folgender Entwicklungslinien:

- Einsatz von zukunftsweisenden Werkzeugen und Methoden aus dem Bereich der objektorientierten Softwaretechnologie
- Plattformunabhängige Implementierung
- Plattformübergreifende Kommunikation
- Skalierbarkeit

5. Grundlagen/Anforderungen

Neben der Eingangsfestlegung zur Nutzung des Datenübertragungsdienstes DAB/DMB waren zwei Festlegungen/Ergebnisse bestimmend für die Spezifikations- und Entwicklungsarbeiten:

Die Schnittstelle zwischen dem C3 Datenserver und dem ANIS-Infopool basiert auf der:

- Integrationsschnittstelle für betriebsübergreifende Anschlusssicherung
(Standardschnittstelle für betriebsübergreifende Anschlusssicherung,
Entwurf Dezember 2000, VDV-Schriften 453 12/00) -

Die darauf aufbauenden und für MOBILIST-ANIS gültigen Festlegungen sind festgeschrieben im Dokument

„Feinspezifikation Schnittstellen C3-Datenserver inkl. Anhang A – Anhang E
Konsolidierte Fassung vom 10.05.01

Die Verbindlichkeit wurde am 10.05.2001 in der Projektsitzung MOBILIST-ANIS von den Beteiligten beschlossen.

6. Erzielte Ergebnisse

6.1 Endgerätehardware

6.1.1 DAB/DMB-Server

Der DAB/DMB-Server ist die Schlüsselkomponente für die mobilen und stationären Fahrgastinformationsanzeiger in den Fahrzeugen und an den Haltestellen. Mit der integrierten Empfängerkarte übernimmt er den Empfang und die Decodierung des DAB/DMB-Signals sowie die Speicherung und Präsentation der Fahrgastinformation.

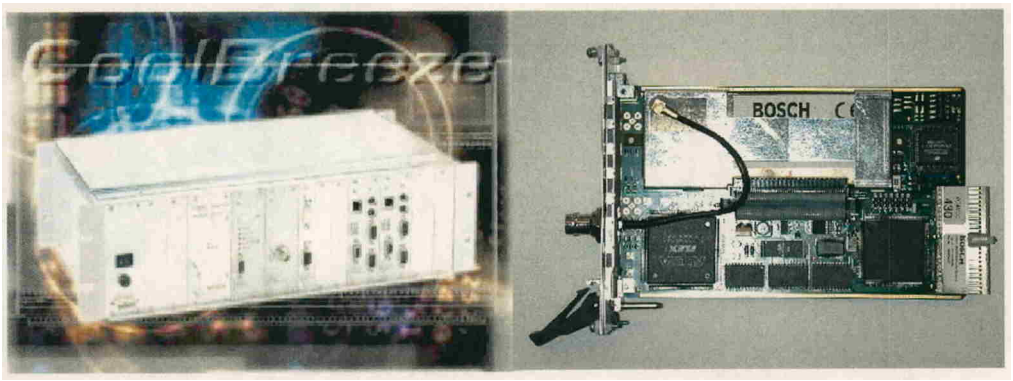


Abbildung 4: DAB/DMB-Server und -Empfängerkarte

Der DAB/DMB-Server kann für unterschiedliche Anwendungen (z. B. abhängig von der Displaytechnologie) konfiguriert werden, eine Auslegung zeigt die folgende Abb.:

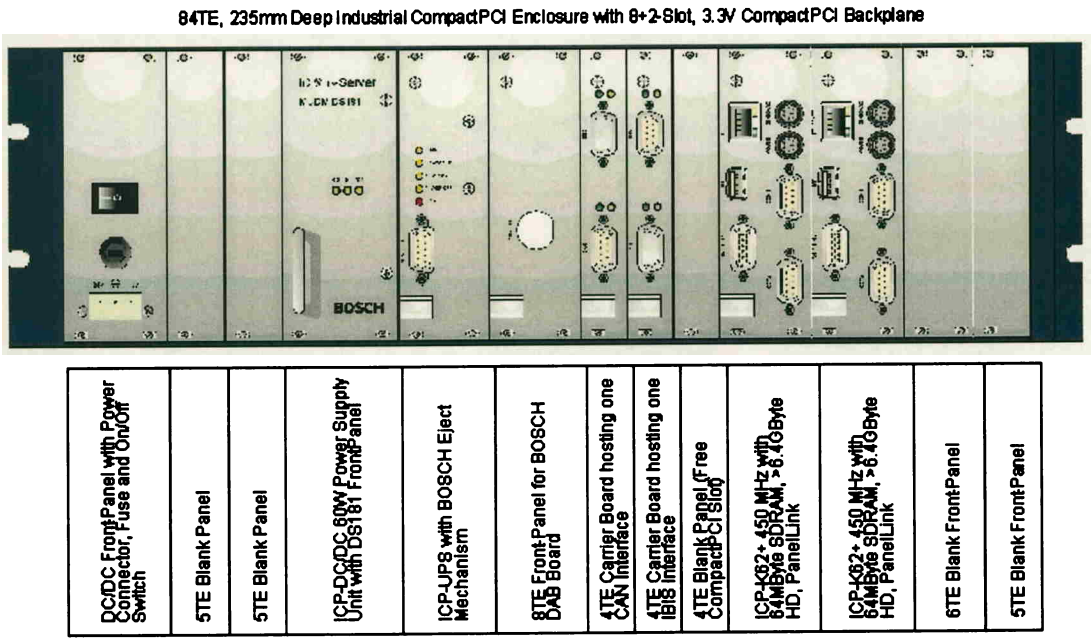


Abbildung 5: Beispielkonfiguration DAB/DMB-Server

6.1.2 Fahrzeugendgeräte

Jedes Fahrzeug ist mit einer kompletten Empfangs-, Verteil- und Präsentationseinrichtung ausgestattet, bestehend aus Antennen-Einheit, DMB-Server, kabelgebundener Signal- und Energieverteilung, PL-Repeater (eventuell), TFT-Displays und Fahrzeugbusanbindung (wenn vorhanden).

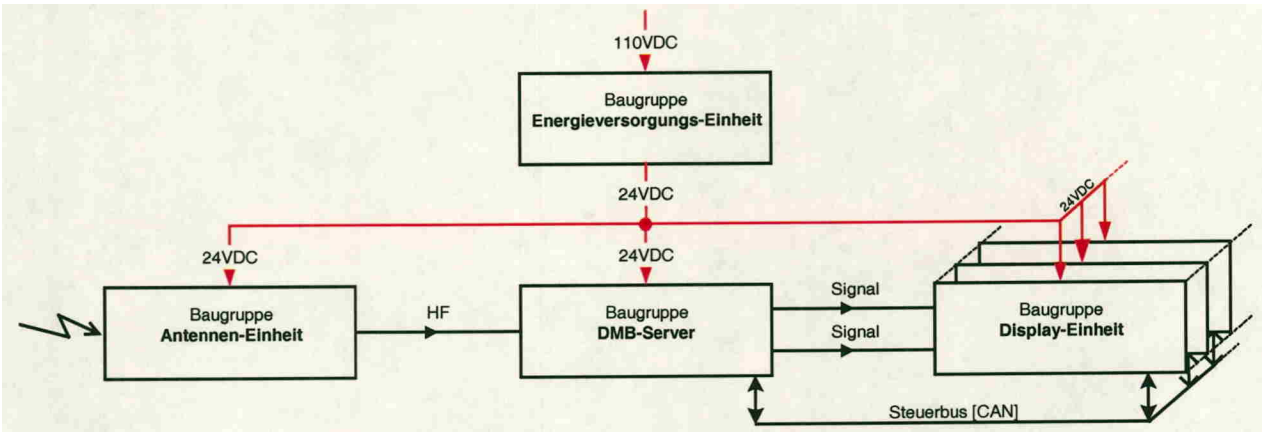


Abbildung 6: Baugruppen Fahrzeugausstattung

Je nach Fahrzeugkonstruktion und den Einsatzbedingungen sind Wagenübergänge (Kupplungen) zwischen den einzelnen Fahrzeugteilen zu installieren, über die die Stromversorgung, das Videosignal und die CAN-Steuerleitung überführt werden.
Ist dies nicht möglich (oder nicht gewollt) sind zusätzliche Geräte (z. B. 1 DMB-Server / Wagen) zu installieren.

Die Antenne befindet sich außerhalb des Fahrzeuges und empfängt den digitalen DMB-Datenfunk im Frequenzbereich L-Band (1452...1492 MHz).

Die auf den Displays gezeigten Bilder, Grafiken und Texte befinden sich als Dateien auf dem DMB-Server und werden durch den Datenfunk DMB laufend aktualisiert. Die Ablaufsteuerung der einzelnen Bilder und Filme erfolgt durch eine spezielle Steuer-Software, die Daten sowohl zeit- als auch ortsabhängig darstellen kann.

Abbildung 7: Displays und Anzeigen im Fahrzeug (Bsp.)

Ist ein Fahrzeug mit mehreren DMB-Servern ausgestattet (z. B. weil die einzelnen Wagen beliebig gekoppelt werden) und fällt ein DMB-Server in einem Wagen aus, ist nur dieser Wagen von der Störung betroffen. In allen anderen Wagen arbeitet das DMB-System ungestört weiter.

Die DMB-Server sind mit einer IBIS-Schnittstelle zum Mithören der IBIS-Meldungen ausgerüstet, um positionsabhängige Displayeinblendung sicherzustellen.

6.1.2.1 Merkmale

Allgemeines

Das System verfügt über zwei CPU-Karten mit je einer 450 MHz CPU und Onboard Graphikchip. Für Daten stehen dem Anwender mindestens 5 Gbyte Speicherraum zur Verfügung.
Der Videospeicher ist 4 Mbyte groß. Eine minimale Videoauflösung von 800 * 600 Pixel bei einer Farbpalette von 256 k Farben ist möglich.
Bei einer Auflösung von 800 * 600 Pixel und einer Farbtiefe von 32 k Farben wird mit dem MU-Benchmark MPGTOOLS Version 1.0.0.1 beim Abspielen des Videos Land1_3.mpg ca. 33 fps dekodiert.
Als Betriebssystem findet Microsoft Windows NT 4.0 Verwendung. Das offizielle Microsoft Service-Pack 5.0 ist installiert.
Alle Signale bzw. Schnittstellen, welche die Module für die Kommunikation/Datenübertragung untereinander benötigen, sind intern ausgeführt.

Eingangs- und Ausgangsschnittstellen (von außen zugänglich):
Das System hat zwei vollständig unabhängige digitale Videoausgänge. Das Ausgangsignal Panellink®, kann über ein bis zu 10 m langes Kommunikationskabel der Kategorie 7 (Patch-Kabel) getrieben werden.

Nach dem Starten des DMB-Servers wird ein Watchdog durch eine Software-Applikation aktiviert. Bei einem „Absturz" einer Applikation oder des OS („Blue Screen") führt der Watchdog zu einem sofortigen Reset des Systems. Der Reset durch den Watchdog führt zum Neustart des DMB-Servers, solange kein physikalischer Defekt vorliegt.

Um bei einem Einschaltvorgang oder Rebootvorgang keine Betriebssystem- oder treiberspezifischen Displayausgaben zu sehen, ist eine Applikations- und Systemgesteuerte Display ON/OFF Steuerung bereitgestellt. Nach dem Start des OS können die Video-Ausgänge aktiviert werden.

BIOS Einstellungen

Das Bios bietet einige Einstellungen, die für System-Debugging sinnvoll sind. In das Bios sind drei unterschiedliche Funktionen implementiert, welche sich im Menü Advanced <Videosettings> auswählen lassen:

a) Bootscreen enabled

Dieser Aufruf schaltet alle Videoausgänge aktiv. Dieses wird im besonderen für Systemdiagnose, Service und Wartung des Systems benötigt. Dieser Zustand wird auch eingenommen, wenn das Bios aufgerufen wird.

b) Bootscreen black

Bootscreen black gibt solange einen schwarzen Displayinhalt aus, bis die Applikation dieses beendet. Dieser Aufruf eignet sich im besonderen für analoge und digitale Displays, die über keine besonderen Funktionen hinsichtlich ihres Ein- und Ausschaltverhaltens (z.B. Powermanagement, Abschaltung der Backlights) verfügen.

c) Videosignal clock disabled

Dieser Aufruf eignet sich für die Bootscreenunterdrückung/Abschaltung bei digitalen Displays, die in der Lage sind, auf Grund des Zustandes des Videoclocks in einen definierten Zustand zu gehen.

Der voreingestellte Zustand ist b) Bootscreen black

Bei einer Spannungsunterbrechung > 5 s versorgt das Backup-Modul das System 55 s lang mit Energie. Erst nach diesen insgesamt 60 s schaltet auch das Backup-Modul ab. Da das System vollständig autark läuft, muß gewährleistet sein, daß nach einer längeren Spannungsunterbrechung das System automatisch wieder startet. Um aber Mehrfachstarts bei Spannungsunterbrechungen > 60 s zu verhindern, kann der Systemstart verzögert werden. Dieses erfolgt im Menü Advanced mit dem Menüpunkt Boot Delay. Hier läßt sich die Bootverzögerung auf 0 s, 30 s, 60 s oder 120 s einstellen.
Ab der HW-Rev. E1 läßt sich dieser Chip disablen. Der WD wird über die Southbridge realisiert und kann dann auch die Bootphase überwachen.
Da das System vollständig autark läuft, ist gewährleistet, daß nach einem Shutdown, Reset oder einer längeren Spannungsunterbrechung das System automatisch wieder startet, solange kein physikalischer Defekt vorliegt. Um ein Mehrfachstart bei einer Spannungsunterbrechungen zu verhindern, ist der Systemstart verzögert.
Nach einem Shutdown kann bei einem Neustart des Systems bis zu vier Minuten kein Videosignal ausgegeben werden.
Die DIN 50155 findet Anwendung und wird ohne Einschränkungen erfüllt.

Spannungsversorgung

Der Server wird mit typisch 24 V versorgt. Die Versorgung erfolgt von der Vorderseite. Es wird ein Backup-Modul eingesetzt, das dem Betriebssystem Zeit gibt, das System geordnet herunterzufahren falls ein Spannungseinbruch erfolgt. Das Backup-Modul stellt sicher, daß das System beim nächsten Power-Up sicher starten kann und die Datenintegrität auf der Festplatte gewährleistet ist.
Ist der Ladezustand des Backup-Moduls nicht ausreichend für die Überbrückung einer Spannungsunterbrechung, wird der Ausgang von der Versorgungsspannung genommen. Das Backup-Modul lädt sich so lange auf, bis wiederum eine Spannungsunterbrechung fehlerfrei überbrückt werden kann. Während dieser Ladezeit kann eine Spannungsversorgung auf der Secundärseite während einer Unterbrechung der Versorgungsspannung auf der Primärseite des Systems nicht erfolgen.

Das Backup-Modul kann folgende Zustände einnehmen

standby	Eine kurze Spannungsunterbrechung kann sicher überbrückt werden	
charge	die USV lädt	
discharge	Die Primärspannung ist unterbrochen und die USV hat die Spannungsversorgung des Systems übernommen. Dauert der Zustand länger als 5s wird ein Power-Fail signalisiert.	Geringfügige Änderungen jederzeit erlaubt, solange DIN 50155 nicht berührt wird.
power-off	Die Spannungsversorgung steht auf der Primärseite nicht zur Verfügung, Power fail wurde signalisiert und die Spannungsversorgung des Systems durch die USV wird für einen Systemshutdown aufrecht erhalten und danach abgeschaltet oder der minimal Ladezustand der USV wurde erreicht.	
Error	Fehler der USV	Nicht erlaubter Zustand

Tabelle 1: Zustände des Backup-Moduls

Diese Systemzustände werden durch Dioden signalisiert. Andere Schnittstellen. z.B. für die Statuserkennung, stehen nicht zur Verfügung.

Temperaturüberwachung

Eine Temperaturüberwachung arbeitet im Temperaturbereich von –25°C bis +70°C. Die Temperaturüberwachung überprüft ständig die Temperatur im **inneren** des Systems. Das folgende Systemverhalten wird durch die Temperaturüberwachung gewährleistet, wobei eine Toleranz von +2,5°C erlaubt ist:

Temperatur t in °C	Einschaltverhalten	Betriebsverhalten
-2,5 <= T <= 55	System darf eingeschaltet werden	Normalbetrieb
T < -2,5; T > 55	Einschalten unterlassen	Herunterfahren des Systems einleiten

Tabelle 2: Temperaturüberwachung

Gehäuse

Das 19" Gehäuse ist geschlossen ausgeführt und mit Haltewinkeln ausgerüstet.

Gehäuseabmessungen ohne Haltewinkel:

Höhe: 3 U + 5 mm
Breite: 84 TE ohne Montagewinkel
Tiefe: 235 mm

Das Systemgewicht beträgt ca. 18,59 Kg.

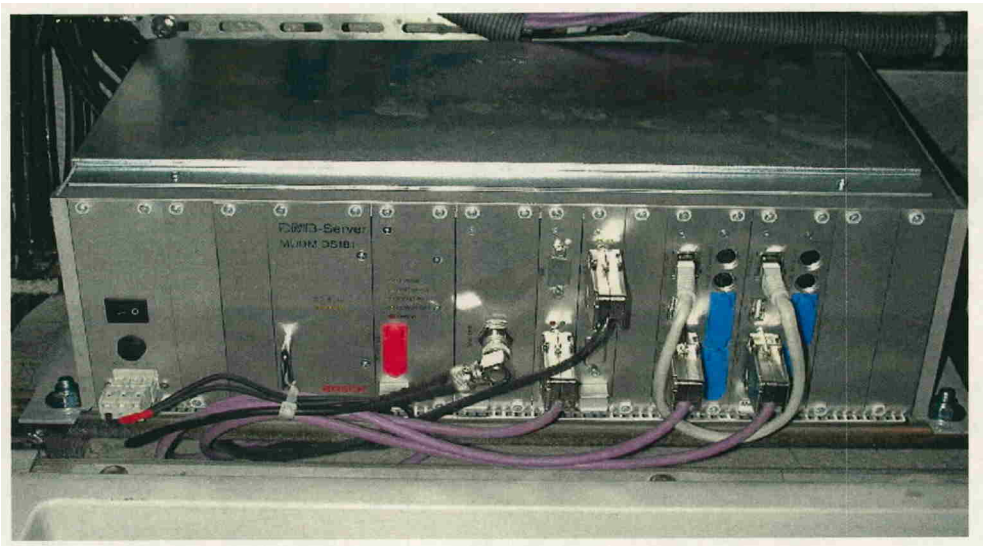


Abbildung 8: Servermontage im Fahrzeug

6.1.2.2 Ablaufsteuerung

Alle nicht unmittelbar darzustellenden Objekte (alles außer Echtzeit-Übertragung) müssen in einer geordneten und vom Redaktionsteam festzulegenden Reihenfolge und Darstellungsart auf den Bildschirmen in den Fahrzeugen angezeigt werden. Hierfür ist auf den Fahrzeug-Geräten (DMB-Server) eine Ablaufsteuerung realisiert, die diesen Anforderungen gerecht wird.

Aufgaben der Ablaufsteuerung

Die Aufgaben der Ablaufsteuerung sind:
- Interpretation der Ablauf- und Container(Beitrags)-Skripte.
- Aufruf der Beiträge in Reihenfolge der im Ablaufskript angegebenen Beiträge.
- Bearbeitung gesonderter Beiträge für die Laufschrift
- Aufruf der Beiträge in Abhängigkeit des Ortes
- Präsentation während der im Beitrags-Skript angegebenen Gültigkeitsdauer
- Kontrolle der präsentierten Objekte, Kontrolle der Displaymanger
- Kommunikation mit den Displaymanager auf den Rechnereinheiten Master und Slave.
- Steuerung der Präsentation auf einem Doppeldisplay

Beitragskontrolle

Die Darstellung der einzelnen Beiträge wird über Ablauf- und Beitragsskripte gesteuert.
Die Beitrags- und Ablaufskripte sind in der Skriptsprache MUBIS kodiert.
Die Ablaufskripte legen den sukzessiven Ablauf der einzelnen Beiträge fest.
Die maximale Anzahl von Containern in einem Ablaufskript beträgt 50.
Für die einzelnen Beitrags- und Ablaufskripte ist eine Gültigkeitsdauer einstellbar.
Existiert kein gültiges Ablaufskript wird das Skript default.ept im Verzeichnis c:\share\default ausgeführt.
Ungültige Beitragsskripte werden nicht ausgeführt. Existiert kein gültiger Beitrag für ein Ablaufskript, wird ebenfalls das Ablaufskript default.ept ausgeführt.
In den Beitragsskripten ist Zeitpunkt und Dauer der Darstellungsobjekte beschrieben. Weiter wird hier der Ausgabebereich für die Displays festgelegt (rechtes oder linkes Display). Dies gilt für:
- Standbilder
- MPEG-Video

Jeder Dienst für die Fahrgastinformation und –unterhaltung besteht in der Regel aus einer Vielzahl von Einzelobjekten. Diese Darstellungsobjekte können sich wiederum aus unterschiedlichen Daten- bzw. Dateiformaten zusammensetzen.

Folgende Daten-/Dateiformate und Einschränkungen sind hierbei zu berücksichtigen:

Objektkategorie	Format, Typ	Darstellung links / rechts möglich ?	Parameter / Einschränkungen
Grafik-Objekte	JPEG, Bitmap	Links und rechts	Max. Auflösung 800 x 600 Bildpunkte, bis zu 24 Bit Farbtiefe, JPEG: nicht progressiv, RGB-codiert Bitmap: MS-Windows BMP-Format
Offline-Video-Objekte	MPEG2-ES	Alternativ, entweder rechtes oder linkes Display	SIF-Auflösung (352 x 288 Bildpunkte), max. 2,5 Mbit/s;
Online-Video-Objekte	MPEG2-TS204	Nur rechtes Display	SIF-Auflösung (352 x 288 Bildpunkte), mit FEC (Paketlänge 204 Byte), max. 1 Mbit/s;
System-Objekte	ASCII, Binär-Dateien (SW-Updates)	Keine visuelle Darstellung	MU vorbehalten

Tabelle 3: Objektklassen

Displaymanager

Der Displaymanager verwaltet die Anzeige auf dem Bildschirm.
Für jeden Bildschirm steht ein Displaymanager zur Verfügung.
Die Ablaufsteuerung startet den Displaymanager auf dem Master.
Die Slave-Shell starte den Displaymanager, wenn die Ablaufsteuerung das Startzeichen dafür gibt.
Die Darstellung der Viewer-Objekte wird durch die Displaymanager veranlasst, wenn die Displaymanager durch die Ablaufsteuerung eine entsprechende Anweisung empfangen.
Der Displaymanager kontrolliert:

- die Präsentation von MPEG-Video.
- die Präsentation der Laufschrift.
- die Präsentation von BMP- und JPG-Dateien bei Standbildern.

Präsentation von Standbildern

Es können JPG- und BMP-Formate wiedergegeben werden (s. Tabelle 3: Objektklassen). Standbilder können auf beiden Bildschirm in der Zielauflösung 800x600 ausgegeben werden, es erfolgt eine automatische Skalierung auf diese Zielgröße. Die entsprechenden Angaben dazu befinden sich im Container-Skript.

Der Standbildübergang kann durch die Effekte:
- Dissolve
- Wipe
- Dip to Color

verknüpft werden.

Ein Bildübergang ohne Effekt ist möglich.

Für die Einstellung der Effekte sind folgende Parameter möglich:
- Dissolve
- Blenddauer (Frames)
- Verzögerung der Blende (Frames)
- Wipe
- Blenddauer (Frames)
- Verzögerung der Blende (Frames)
- Breite der Blendkante (% der Bildschimbreite)
- Blendrichtung
- Dip to Color
- Blenddauer (Frames)
- Einblend-(Dip) Farbe (R,G,B)

Präsentation von MPEG-Video

Die Wiedergabe für in SIF-Qualität codierten MPEG-Dateien ist möglich. Die gleichzeitige Darstellung von MPEG-Videos auf 2 Bildschirmen wird nicht unterstützt.

Die MPEG-Videos besitzen eine Originalauflösung, die durch die Objektdatei gegeben ist. In den Container-Skripten wird durch die Größenangabe die Zielauflösung von 800x600 definiert. Die MPEG-Videos werden entsprechend der Angabe im Container-Skript automatisch skaliert.

6.1.2.3 DMB-Anbindung

Die DMB-Anbindung hat die Aufgaben:
- Empfang der Video- und Bildobjekte
- Empfang der Ablauf- und Containerskripte und der Containerlisten.
- Empfang von Dateien zum SW-Update

Die DMB-Anbindung erfolgt über die 4 Schichten:
- DAB-Empfangskarte, DAB-CPCI
- PCI-Treiber zur Anbindung der Software an die DAB-Empfangskarte
- DAB-Agent, Software zur Steuerung der DAB-Empfangskarte und der DMB-Anbindung
- MOT-DLL, Decodierung, Speicherung und Verwaltung der MOT-Empfangsdaten (Schnittstelle zum MPEG-Decoder für Live-TV).

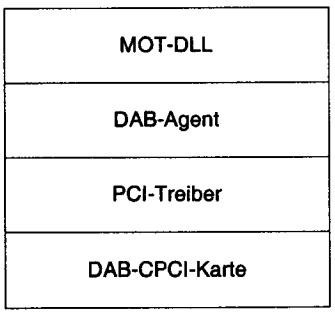


Abbildung 9: DAB-Schichtenmodell

DAB-Agent

Der DAB-Agent ist eine Applikation, um DAB Daten, die von der CPCI Karte empfangen werden, zu decodieren bzw. weiterzuleiten.

Alle notwendigen Einstellungen des DAB-Agenten können in einer INI-Datei eingetragen werden. Folgende Einstellungen sind möglich:
- Setzen der Empfangsfrequenz
- Aktivieren der Synchronisation der Systemzeit mit der DAB-Zeit
  Automatische Verwaltung für Sommer/Winterzeit
- Steuerung des Log-Prozesses
- Minimierung des DAB-Agenten

Der DAB-Agent erzeugt Log-Files, die im Verzeichnis c:\log abgelegt werden

MOT-DLL

Die MOT-DLL dekodiert die gelieferten DAB-Packet-Mode MOT Daten und erzeugt die vom DMB-Server benötigten Files (Ablaufskripte, Beitragskripte und Multimedia-Dateien)
Aktualisierte Daten werden erkannt und überschrieben.

Archivliste

Die empfangenen Daten werden mit Hilfe einer Archivliste verwaltet. Dazu werden die komplett empfangenen und abgespeicherten Dateien zusammen mit der Versionsnummer und der Gültigkeitsdauer in die Archivliste eingetragen. Die Liste dient der Vermeidung von doppelten Empfang und zur Identifikation ungültiger Dateien. Diese Archivliste wird bei jedem neu empfangenen Objekt auf der Festplatte aktualisiert und wird bei einem Neustart des DAB-Agenten wieder angelegt.

Eigenschaften der DAB-Empfangskarte

Die Aufgaben der DAB-CPCI-Card sind:

- Empfang und die Dekodierung von DAB-Signalen.
- Dekodierung von Datengruppen und Paketen.

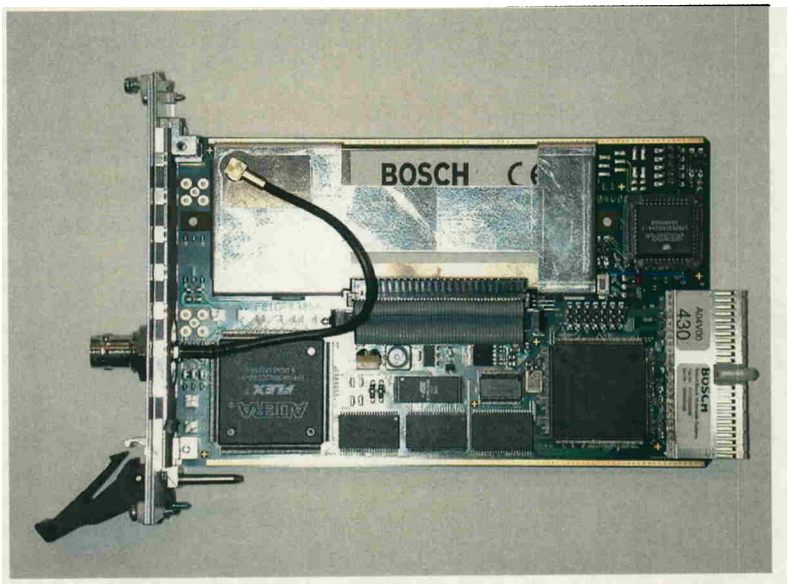


Abbildung 10: DAB-Empfängerkarte

6.1.2.4 Display-Steuerung über CAN-Bus

Die Displays werden durch den CAN-Bus kontrolliert:
- Ein/Ausschalten der Displays
- Einstellen, Kontrolle der Temperaturschwellen
- Einstellen der Leuchtdichten
- Fehlerkontrolle

Zur Anbindung der CAN-Software existiert das Schichtenmodell:

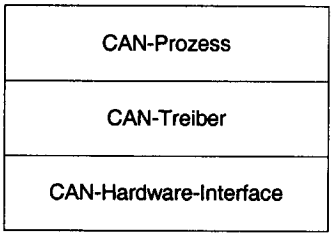


Abbildung 11: CAN-Schichtenmodell

Der CAN-Prozess wird durch die Ablaufsteuerung gestartet und beendet. Startet der CAN-Prozess werden alle Displays initialisiert und eingeschaltet. Erhält der CAN-Prozess das Signal zum Beenden von der Ablaufsteuerung, stellt der Prozess sicher, dass alle Displays dunkel geschaltet sind. Somit ist sichergestellt, dass nur dann eine Display-Anzeige vorhanden ist, wenn entsprechende Inhalte dargestellt werden können.

Die Informationen zur Initialisierung der Displays sind in einer Initialisierungsdatei des CAN-Prozesses vorgegeben. Initialisiert werden:
- Leuchtdichte der Hintergrundhelligkeit ca. 170 cd/mm^2
- Temperaturschwellen,
- Min-Temperatur: 10°C
- Max-Temperatur: 65°C
- Display- Konfiguration, Display-Anzahl

Die Displays melden auf Anfrage des CAN-Prozesses ihren Status. Im Fehlerfall (Überschreiten der Temperaturgrenzen, Display ist über den CAN-Bus nicht mehr sichtbar) wird das fehlerhafte oder sich temporär im Fehlerzustand befindliche Display dunkel geschaltet, wenn es über den CAN-Bus ansprechbar ist. Es ist sichergestellt, dass das dem defekten Display zugeordnet Display ebenfalls abgeschaltet wird (paarweise Abschaltung).

Der Status der Displays wird in einer Log-Datei gesichert.

6.1.2.5 IBIS-Anbindung

Nutzung von IBIS-Informationen in den Fahrzeugen

Das im Fahrzeug installierte Integrierte Bordinformationssystem, kurz IBIS genannt, dient der digitalen Datenverarbeitung und der seriellen Übertragung von Daten zur Information
der Fahrgäste und der Geräte der Fahrgastbedienung über Fahrzeugort, Zeit und Fahrtziel
der Leitstelle über Fahrzeugort, Fahrtziel und Besetzung,
des Fahrpersonals über Fahrplanabweichungen, betriebliche Anweisungen und technische Störungen.

Durch die Verknüpfung der Daten des DMB-Systems mit den IBIS-Informationen ist es möglich, aktuelle betriebsrelevante Informationen auf den Doppel-Display-Einheiten anzuzeigen und eine Ortinformation zur Steuerung der Anzeige zu erhalten. Der Zugang zu den unten beschriebenen Informationen auf dem IBIS-Wagenbus wird durch reine Mithörfunktion des DMB-Servers erreicht.

Interface IBIS-ZG <> DMB-Server

Übertragungsformat: nach VDV-Schrift 300 7/91 „Integriertes Bordinformationssystem (IBIS)"

Folgende fahrtbezogene Daten werden im Fahrzeug auf dem IBIS-Wagen-BUS bereitgestellt und vom DMB-Server ausgewertet (Syntax: Z = Dezimale Ziffer; C = Character; H = Hexadezimale Ziffer; n = n Stellen):

Bezeichnung	Inhalt	Codierung	Beispiel
DS 001	Liniennummer	LZZZ	l016
DS 002	Kursnummer	kZZ	K01
DS 003c	Innenanzeige: Haltestellenname der nächsten zu erreichenden Haltestelle	ZIHnC	zI4FELSENKELLER
DS 003	Zielnummer	zZZZ	z036
DS 008	Fehlererfassung, Wagenadresse	NHHH	Beispiel LVB: N12<_A3 entspricht Fahrzeugnummer 300

Tabelle 4: Inhalte, IBIS-Bus

Kommunikation mit der Ablaufsteuerung:

Das Tramposmodul wird durch die Ablaufsteuerung gestartet. Danach baut die Ablaufsteuerung TCP/IP-Verbindung mit der Ablaufsteuerung mit dem Tramposmodul auf. Die TCP/IP-Verbindung dient zur Socketkommunikation.

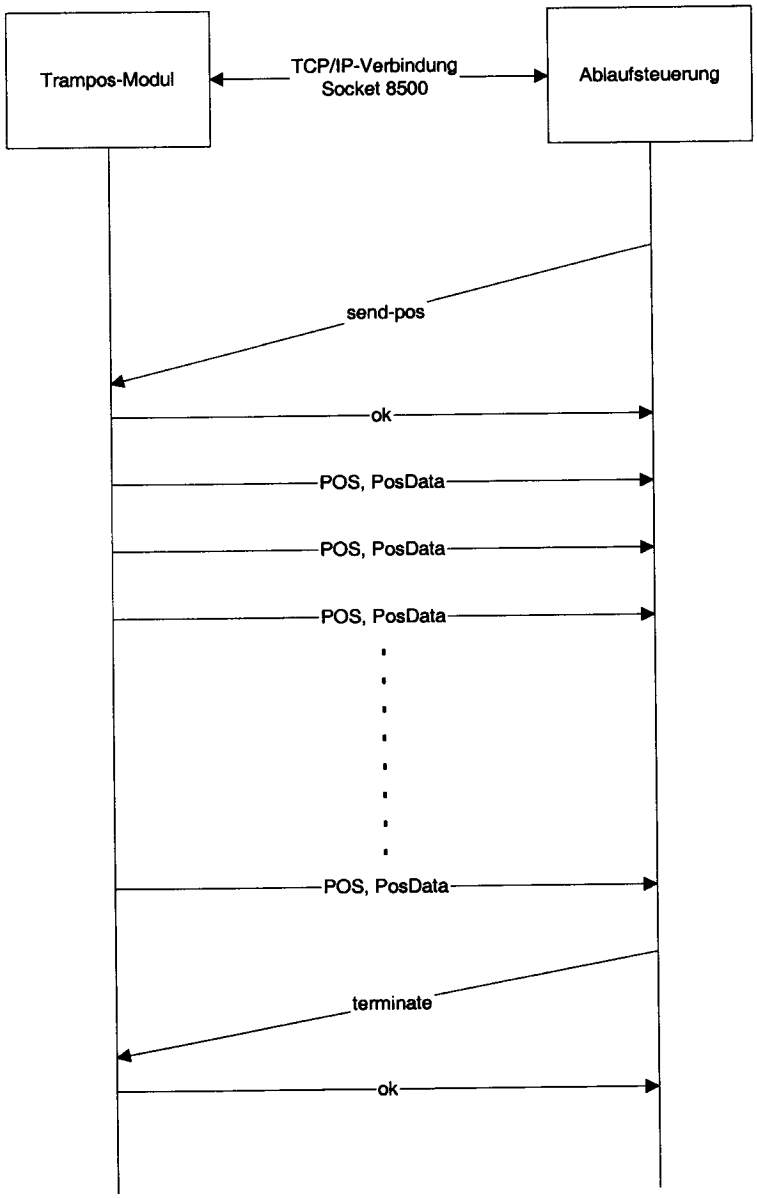


Abbildung 12: Sequenzdiagramm, Ablaufsteuerung <> Tramposmodul

Die Anfrage send-pos der Ablaufsteuerung initiiert eine permanente Positionsübertragung des Trampos-Moduls.
Das Tramposmodul erkennt eine Positionsänderung und übermittelt diese Informationen an die Ablaufsteuerung.. Diese Positionsdaten enthalten:

- Information über die nächste Haltestelle
- Information über die aktuell befahrene Linie
- Die aktuelle Wagen- (Fahrzeug) nummer

Die aktuelle Position wird einmal kurz nach dem Kommando und später bei jeder Änderungen an die Ablaufsteuerung übertragen, ohne dass dieser weitere Kommandos senden muss; solche Positionsupdates sind mit dem Schlüsselwort POS gekennzeichnet.

Die Schnittstelle zum IBIS-Treiber ist logisch als COM-Port realisiert.

Die Basisdaten (Linie, Wagennummer) und aktuelle Position des Fahrzeug wird durch Abhören der IBIS-Nachrichten durch das Trampos-Modul kontinuierlich erfasst. Kurzfristige Störungen auf dem IBIS-Bus beeinträchtigen das System nicht. Fällt die Datenerfassung für längere Zeit aus oder kann sie gar nicht erst aufgenommen werden, ist ein eingeschränkter Betrieb möglich

Das Trampos-Modul erkennt die Positionsänderung durch Auswertung der IBIS-Information und den nach MU-Standard übertragenen Tabellen. Bei Änderung wird die gesamte Informationen über die aktuelle Position und der Wagennummer im Objekt PosData an die Ablaufsteuerung übertragen.

Das Tramposmodul verwaltet den Datenbestand. Es können mehrere Instanzen des Datenbestandes existieren.
Basis- und Positionsdaten können in einer Dump-Datei protokolliert werden.

6.1.3 Demonstratoren für stationäre Fahrgastinformation

Die Komponenten für die stationären Demonstratoren sind Labormuster. Sie bestehen aus dem unter 6.1.1 aufgeführten DAB/DMB-Server ergänzt um ein 230V Netzteil, unterschiedlichen Displaytypen (TFT, Plasma) und den unter 6.2 beschriebenen Softwarekomponenten.
Die Layouts der Ankunfts- und Abfahrtsanzeiger sind ebenfalls in den unter 5. aufgeführten Grundlagen spezifiziert.

6.2 Endgerätesoftware

6.2.1 Einleitung

Die Software ***IPODIPSI*** *(INPUT, PROCESSING AND DISPLAY SOFTWARE FOR PASSENGER INFORMATION)* hat ihren Ursprung in der FuX-AIH/AIL-Anwendungssoftware. Sie diente dazu vom einem Infopool-Server generierte HTML-Seiten, die per DMB übertragen vorliegen, zu verarbeiten und korrekt anzuzeigen. IPODIPSI ist für die Java 2 Plattform implementiert und wurde für Mobilist wie folgt erweitert:

Erweiterungen
Vollständig überarbeitete Software nach neuem Konzept: • Verwendung eines einfachen ASCII-Formats für den Empfang von Ist-Daten (Tagesfahrplan, neue Fahrt- und Sondermeldungen). • Verwendung eines einfachen ASCII-Formats für den Empfang von Soll-Daten (Saisonfahrplan). • Lokale Datenbank auf dem Endgerät. • Lokale Aufbereitung der Daten zu einer Anzeige in HTML-Format. • Einsatz im mobilen Betrieb: Anzeige der Anschlussinformation für die jeweils nächste Haltestelle • Umschalten auf Basis der Haltestelleninformation über den IBIS Bus

6.2.2 Paketstruktur

Die verschiedenen Anforderungen an die Software lassen sich in insgesamt 6 Bereiche gliedern, denen im Design 6 Pakete entsprechen:

Anforderungen	Paket
Globale Anwendungskonfiguration, Start der verschiedenen Programmteile, globaler Kontrollfluss (Starten der Threads etc.). Reaktion auf global relevante Ereignisse (online- oder offline-Betrieb, neu selektierter AIH im Monitor-Betrieb, neue Haltestelle im Mobilist-Betrieb) Auffangen nicht behandelter Exceptions	ipodipsi
Datenstrukturen für anzuzeigenden Fahrplanausschnitt mit Fahrt- und Sondermeldungen Aktualisierung eines Fahrplanausschnitts (Abgleich mit allen aktuellen Fahrt- und Sondermeldungen in der Datenbank). Datenstruktur für Positionsinformationen, die im Mobilist-Betrieb von TramPos geliefert werden (IBIS).	ipodipsi.data
Erzeugung einer HTML-Darstellung eines aktuellen Fahrplans. Erzeugung einer Default-HTML-Seite, falls keine aktuellen Daten vorhanden sind.	ipodipsi.output
Erzeugung des benötigten Datenbankschemas (Tabellen, Spalten, Bedingungen) aus SQL-Code. Import der benötigten Initialdaten (Haltstellen, Haltepunkte, AIHs) aus ASCII-Quelltext.	ipodipsi.dbsetup
Import der per DMB empfangenen Tagesfahrpläne und Saisonfahrpläne in die lokale Datenbank. Import neuer Fahrt- und Sondermeldungen für einen Tagesfahrplan (Updates).	ipodipsi.dbimport
Löschen überflüssiger Dateien aus Importverzeichnissen. Löschen veralteter Daten aus online- und offline-Datenbank.	ipodipsi.cleanup
Allgemein verwendbare Hilfsklassen, z.B. für File- und Stringhandling.	ipodipsi.util

Die Abhängigkeiten zwischen den Paketen sehen im Überblick wie folgt aus:

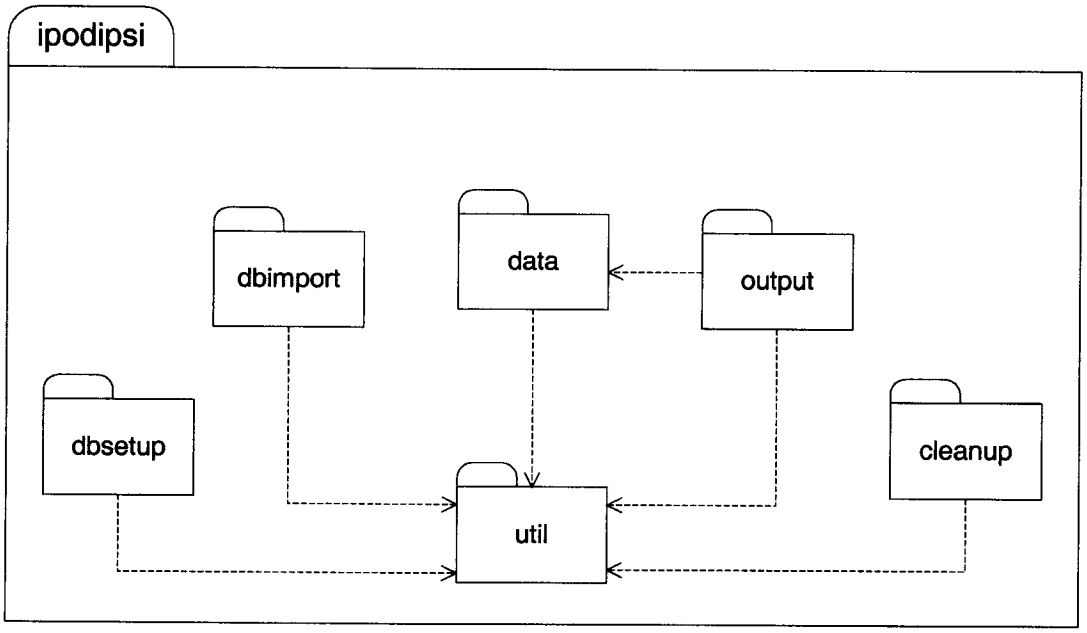


Abbildung 13: Pakete und Paketabhängigkeiten

Das Paket ipodipsi.util wird also von allen anderen Paketen benötigt. Außerdem benötigt output Klassen aus data, da es Datenstrukturen in eine Ausgabe überführen soll.

Aus ipodipsi selbst kommen in den Unterpaketen die folgenden Klassen und Interfaces zum Einsatz:

Klasse/Interface	Verwendung
ipodipsi.APP ipodipsi.C	Lesen und ggf. Setzen von anwendungsweiten Parametern, Lesen von anwendungsweiten Konstanten.
ipodipsi.ExceptionPollable	Interface: Klassen, deren Exceptions abgefragt werden können (Threads)
ipodipsi.ModeAdaptable ipodipsi.DeviceAdaptable ipodipsi.LocationAdaptable	Interface: Klassen, die auf einen Wechsel online/offline reagieren. Interface: Klassen, die auf eine fixe Haltestelle reagieren Interface: Klassen, die auf eine neue Haltestelle im Fahrzeug reagieren

6.2.3 Beschreibung der Pakete

Die Beschreibung der Pakete orientiert sich an den folgenden Fragen:

1. Wie werden die lokalen Datenbanken angelegt?
2. Welche Datenklassen entsprechen einem aktuellen Fahrplanausschnitt (aktuelle Anzeige)? Wie ist deren Beziehung zu online- und offline-Datenbank?
3. Wie wird ein aktueller Fahrplanausschnitt angezeigt?
4. Wie werden die lokalen Datenbanken mit per DMB empfangenen Daten aktualisiert?
5. Wie wird mit überflüssigen Dateien und veralteten Fahrtmeldungsdaten verfahren?
6. Wie wird auf wichtige Ereignisse reagiert? Wie ist die globale Ablaufsteuerung gestaltet?
7. Welche anwendungsspezifischen Hilfsklassen werden verwendet?

Dabei werden die globale Ablaufsteuerung und die Hilfsklassen (Punkte 5 und 6) – soweit notwendig – bereits im Vorfeld erwähnt und ggf. näher erläutert. In eigenen Abschnitten werden sie noch einmal ausführlicher dargestellt.

6.2.3.1 Das Einrichten der lokalen Datenbanken (ipodipsi.dbsetup)

Zum Einrichten der Infopool-Datenbanken für Online- und Offline-Daten müssen folgende Voraussetzungen gegeben sein:

1. Auf der Zielplattform läuft ein DBMS (hier eingesetzt: InterBase)
2. Für das DBMS ist ein JDBC-Treiber[1] installiert und aktiv. Die entsprechende Java-Klassenbibliothek für die Anwendung (Bei InterBase das Archiv interclient.jar) ist bei der Erzeugung des Ziel-Archivs ipodipsi.jar eingeschlossen worden.
3. Auf dem DBMS sind zwei leere Datenbanken eingerichtet (hier: infopool_online und infopool_offline). Dies ist notwendig, weil JDBC keine Verbindung zu einem Datenbankserver an sich, sondern nur zur einer bereits existierenden Datenbank kennt.
4. Auf der Zielplattform befinden sich die Definition des Datenbankschemas und seiner Integritätsbedingungen in Form von SQL-Code. Ebenso sind die Initialdaten im ASCII-Format verfügbar.

Für das Anlegen der lokalen Datenbanken muss nun die Schemadefinition gelesen und ausgeführt werden. Daran schließt sich der Import der Initialdaten aus den Dateien in die Datenbanktabellen an. Es ist zu beachten, dass das System hierzu sinnvolle Werte in der Konfigurationsdatei (mobilist.cfg) vorfindet.

```
#----------------------------------------
# DATABASE SETUP FILE HIERARCHY:
#----------------------------------------

db.setup.dir              = infopool              # Database definition directory
db.setup.file.tables      = infopool_schema.sql   # File containing database schema
db.setup.file.constraints = infopool_constraints.sql # File containing database constraints
db.setup.dir.data         = data                  # Directory for initial data
db.setup.dir.data.suffix  = export                # Suffix for each data file
db.setup.data.separator   = ;                     # Column separator character in each line
```

Angegeben sind also das Hauptverzeichnis für alle Datenbankinformationen, die Dateien mit der Schemadefinition und den Integritätsbedingungen, das Unterverzeichnis für die Initialdaten, die Endungen der entsprechenden ASCII-Dateien und schließlich das Trennzeichen für die Spaltenwerte.

6.2.3.1.1 Die Hilfsklasse DBInterface

Alle Klassen der Anwendung, die auf einer Datenbank operieren, verwenden hierfür die Hilfsklasse DBInterface aus dem Paket ipodipsi.util. DBInterface ermöglicht eine Verbindung zu einer Datenbank und das einfache Abschicken von SQL-Kommandos.

[1] Es werden keine speziellen Eigenschaften von JDBC 2.0 wie scrollable ResultSet benötigt, daher reicht bereits ein Treiber aus, der JDBC 1.1 implementiert.

Dabei ist sichergestellt, dass es von DBInterface nur maximal zwei Exemplare geben kann: Ein Objekt für die Verbindung zur Online-Datenbank und eines für die Verbindung zur Offline-Datenbank. Dies wird dadurch realisiert, dass der Konstruktor nicht sichtbar ist und nur über die statischen Operationen getOnlineInstance() und getOfflineInstance() ein DBInterface geholt werden kann. Damit ist die Klasse eine Variante des Singleton-Patterns[2]:

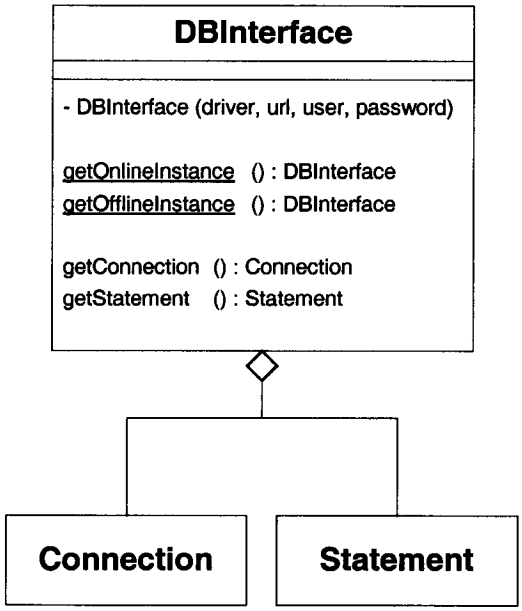


Abbildung 14: DBInterface aggregiert java.sql.Connection und java.sql.Statement

Anderen Teilen der Anwendung wird es hierdurch ermöglicht, auf einfache Weise Online- oder Offline-Datenbank anzusprechen und dabei stets die selben, eindeutigen Verbindungen zu verwenden.

[2] Dieses Pattern beschreibt eine Klasse, von der es nur ein einziges Exemplar gibt. Siehe E. Gamma et al.: *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley.

6.2.3.1.2 Die Datenbankeinrichtung mit DBCreator und SQLReader

Für das Einrichten einer Datenbank wird die Klasse DBCreator verwendet, unterstützt von DBInterface (s.o.) und SQLReader[3]:

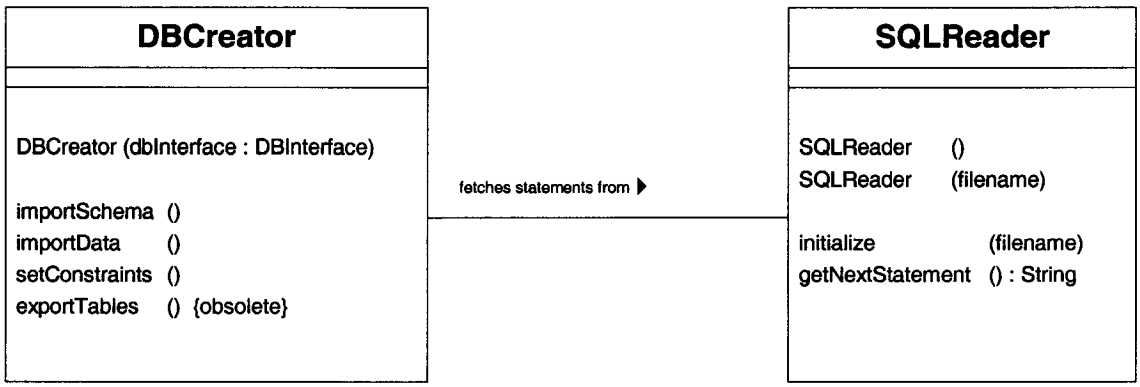


Abbildung 15: Klassen zum Einrichten von online- und offline-Datenbank

Die Operationen von DBCreator entsprechen dabei unmittelbar den einzelnen Schritten, die für das Einrichten einer Datenbank notwendig sind: importSchema() zum Einrichten des Schemas, importData() zum Importieren der Initialdaten und schließlich setConstraints() zum Setzen der Integritätsbedingungen[4].

Die SQL-Kommandos, die DBCreator für das Schema und die Integritätsbedingungen benötigt, werden aus Dateien gelesen. Diese Aufgabe übernimmt die Klasse SQLReader, die aus einer ASCII-Datei das jeweils nächste SQL-Statement extrahiert (getNextStatement()) und dabei Kommentare und überflüssige Leerzeichen entfernt.

[3] Hier wie in den folgenden Klassenbeschreibungen sind zunächst die Konstruktoren und die von außen sichtbaren Operationen und Attribute beschrieben. Falls weitere Operationen und Attribute für das Verständnis wichtig sind, werden sie mit einem Sichtbarkeitszeichen für „private" bzw. „package/protected" (-,#) hinzugefügt.

[4] Die Methode exportTables() wird für die Anwendung nicht benötigt. Sie dient dazu, Daten aus dem ursprünglichen FuX-Infopool in ASCII-Dateien zu exportieren, um über Initialdaten zu verfügen, falls solche nicht zur Verfügung stehen.

Das folgende Sequenzdiagramm beschreibt die Nachrichtenabfolge beim Einrichten des Schemas für die Online- Datenbank (Offline-Datenbank analog):

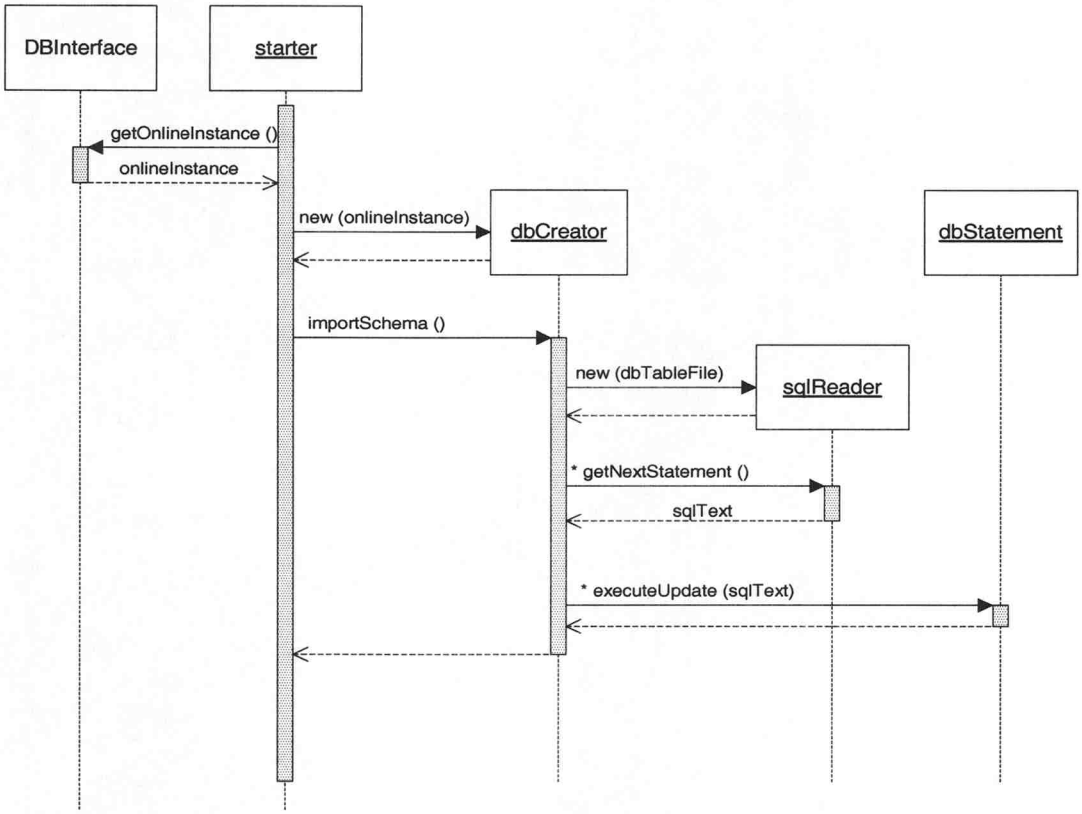


Abbildung 16: Einrichten von Datenbanktabellen mit Hilfe von DBCreator und SQLReader

Dabei ist dbStatement ein Exemplar der Klasse java.sql.Statement, und dbConnection stammt aus java.sql.Connection.

Das Setzen der Integritätsbedingungen für Online- und Offline-Datenbank basiert wie das Einrichten der Tabellen auf der Interpretation von SQL-Code. Daher verläuft das Senden der Nachrichten bei der Ausführung der Operation DBCreator.setConstraints() analog zur Beschreibung oben.

Für das Importieren der Referenzdaten gelten folgende Bedingungen:

- Die Daten für die Tabellen liegen in ASCII-Dateien, die den gleichen Namen wie die zugehörige Tabelle haben.
- Dateiendungen und Trennzeichen müssen in der Konfigurationsdatei der Anwendung eingestellt sein (siehe oben: Voraussetzungen).
- In einer Zeile einer Datei stehen die Werte in der gleichen Reihenfolge wie die zugehörigen Spalten in der Tabelle.

Die Operation importData() greift auf die folgenden, privaten Operationen von DBCreator zurück:

ImportFile	(file, tableName) : int	Importiert die Initialdaten aus einer Datei in eine Tabelle. Liefert die Anzahl der importierten Zeilen.
GetColumnTypes	(tableName) :int[]	Holt die SQL-Typen für alle Spalten einer Tabelle (Werte aus java.sql.Types).
PrepareInsertStatement	(tableName, numOfColumns)	Bereitet ein INSERT-Kommando vor, um eine Zeile mit Spaltenwerten in einer Tabelle einzufügen.
GetInsertValues	(line, numOfColumns) : String[]	Teilt eine Zeile aus einer Datei in Spaltenwerte auf.
setInsertParameters	(statement, values, columnTypes)	Konvertiert für ein INSERT-Kommando Spaltenwerte in die benötigten SQL-Typen und setzt die Parameter für das Kommando.

Die Nachrichtenabfolge beim Import der Daten für die online-Datenbank gestaltet sich dann wie folgt (offline-Datenbank wieder analog):

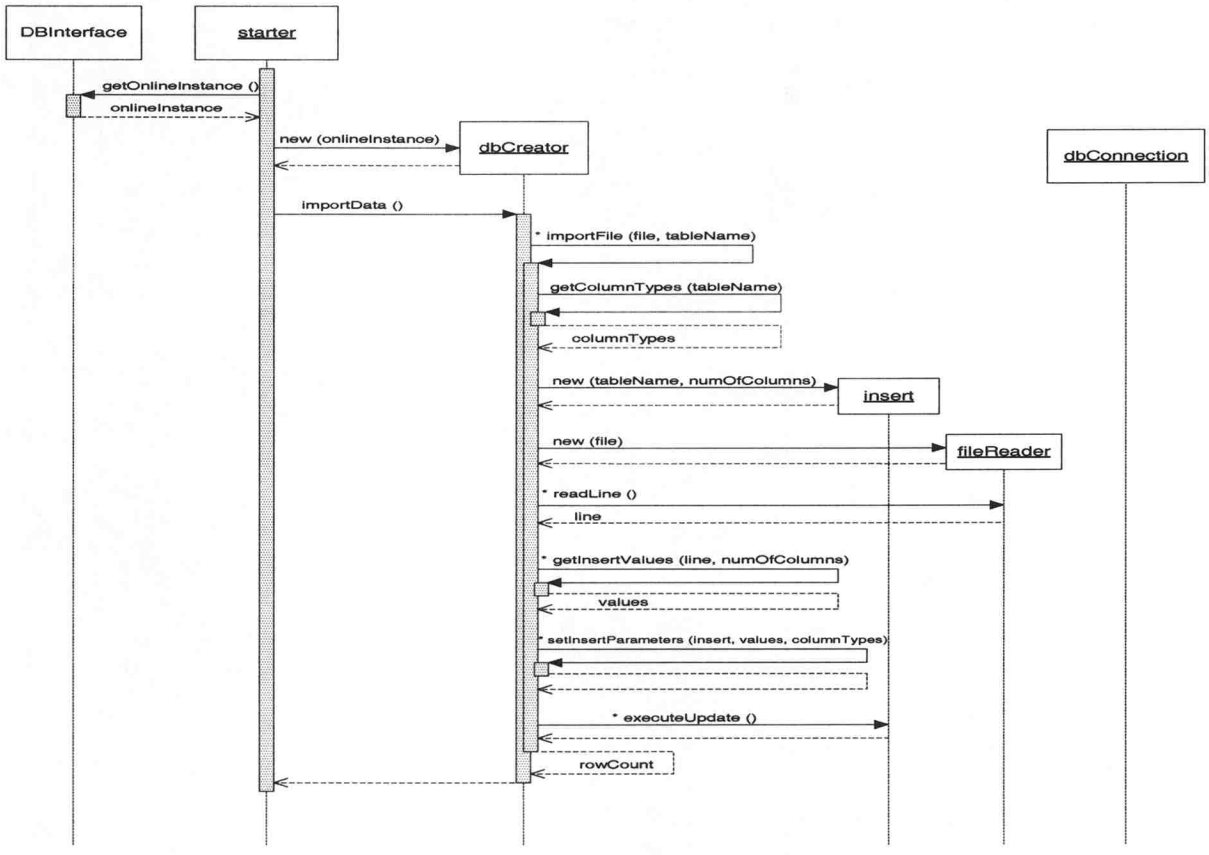


Abbildung 17: Importieren von Initialdaten mit Hilfe von DBCreator

Dabei ist insert ein Exemplar der Klasse java.sql.PreparedStatement. Das Objekt fileReader gehört zu java.io.FileReader, und dbConnection gehört zu java.sql.Connection.

6.2.3.2 Fahrplanausschnitte (ipodipsi.data)

Ein Fahrplanausschnitt ist diejenige Datenstruktur, die im System die zentrale Bedeutung hat, denn vereinfacht gesagt besteht die Aufgabe der Anwendungssoftware lediglich darin, solch einen Ausschnitt regelmäßig zu aktualisieren und anzuzeigen.

6.2.3.2.1 Verschiedene Ausprägungen von Fahrplanausschnitten

Die folgende Abbildung ist ein Beispiel als Demonstrator für eine Anzeige in der Betriebsart Abfahrtanzeiger:

Abbildung 18: Möglicher Fahrplanausschnitt für Abfahrten

Die Daten zum Fahrplanausschnitt bestehen hier aus den 10 Fahrtmeldungen in der Tabelle und aus der Sondermeldung in der Fußzeile. Der Titel mit dem Haltestellensymbol bleibt im Betrieb unverändert. Die Datums- und Uhrzeitanzeige wird ebenfalls unabhängig von den Meldungsdaten erzeugt. Für die Seriengeräte ist hier eine LED Anzeige mit ähnlicher Aufteilung und Inhalt geplant.
Die Fahrtmeldungen stehen für die nächsten 10 Verbindungen von dieser Haltestelle (Ernst-August-Platz) ab dem aktuellen Zeitpunkt.

In der Betriebsart Mobilist Fahrzeuganzeige kann eine Anzeige wie folgt aussehen:

Abbildung 19: Anzeige der nächsten Haltestelle mit Umsteigemöglichkeiten

Hier erscheint zunächst in einem Kopfbereich die Uhrzeit und der Name der Haltestelle, auf die das Fahrzeug zufährt. Neben der geplanten Ankunft- und Abfahrtzeit werden eventuelle Verspätungen angezeigt – im Beispiel oben kommt das Fahrzeug um 10:05 Uhr statt laut Plan um 10:03 Uhr an der Haltestelle Ludwigsburg ZOB an. Die Weiterfahrt bleibt mit 10:08 Uhr jedoch unverändert.

Unter dem Kopfbereich erscheinen Fahrtmeldungen, die den Fahrgast auf Umsteigemöglichkeiten (Verbindungen) hinweisen. Sondermeldungen sind hier nicht vorgesehen.

Für die Betriebsart Fahrtanzeiger liegt also ein Spezialfall eines normalen Fahrplanausschnitts vor: Neben den gewohnten Fahrtmeldungen ist im Kopfbereich diejenige Fahrtmeldung hervorgehoben, die sich auf die Linie bezieht, auf der das Fahrzeug selbst fährt. Außerdem ändern sich während der Fahrt laufend die angezeigte Haltestelle und die Umsteigemöglichkeiten. Daher lässt sich diese Betriebsart auch als „fahrender AIH“ auffassen.

6.2.3.2.2 Die Klasse *Schedule* und ihre Unterklassen

Es wird zunächst zusammengefasst, welche Eigenschaften für einen Fahrplanausschnitt in allen oben erläuterten Betriebsarten gelten:

Jeder Fahrplanausschnitt (*Schedule*) enthält in einem Rumpfteil eine beliebige aber feste Anzahl geordneter Fahrtmeldungen (BodyDepartureMessage).
Ein Fahrplanausschnitt lässt sich mit Fahrtmeldungen schrittweise aufbauen, bis die maximale Anzahl von Fahrtmeldungen erreicht ist[5].
Ein Fahrplanausschnitt lässt sich aus dem Bestand in der lokalen Datenbank auffrischen (aktualisieren[6]), und es ist feststellbar, ob sich dabei tatsächlich Daten geändert haben.
Es lässt sich prüfen, ob ein Fahrplanausschnitt überhaupt Daten enthält.
Für die Ausgabe eines Fahrplanausschnitts muss auf die Fahrtmeldungen im Rumpf zugegriffen werden können.
Ebenso muss man auf die Fahrtmeldung im Kopfbereich und auf die Sondermeldung zugreifen können, falls sie vorhanden sind (falls keine vorhanden sind, wird null zurückgeliefert).

Diese Eigenschaften sind mit der abstrakten Datenklasse *Schedule* wie folgt realisiert:

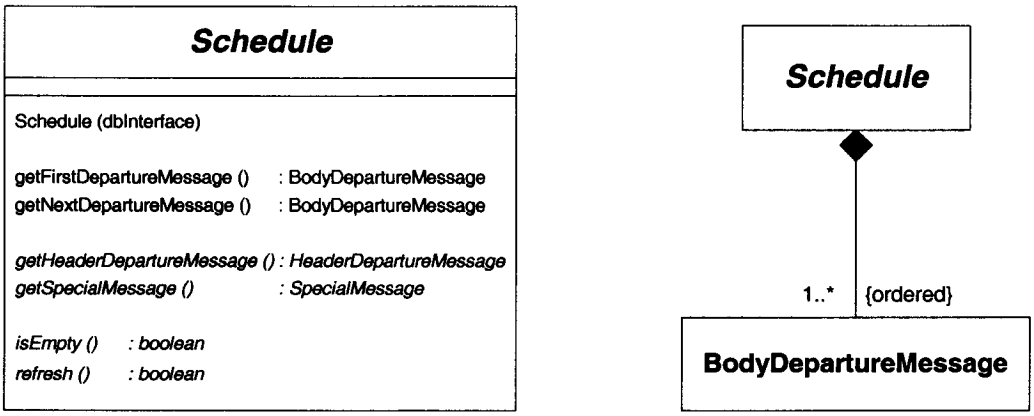


Abbildung 20: Ein Fahrplanausschnitt als abstrakte Klasse Schedule

[5] Diese Anzahl ist abhängig von der Anzahl der Zeilen die ein Gerät darstellen kann. Sie ist in der Konfigurationsdatei definiert.

[6] Aktualisierung bedeutet: Hole die nächsten n Fahrtmeldungen, die ab jetzt gültig sind, und hole die erste gültige Sondermeldung, die gefunden wird (n == Anzahl der möglichen Zeilen auf dem Endgerät). Aktualisiere dann das Objekt der Klasse Schedule mit diesen Daten.

Für die Aktualisierung mit der online- oder offline-Datenbank (refresh) benötigt *Schedule* eine entsprechende Verbindung (DBInterface), die im Konstruktor anzugeben ist[7]. Dieser Konstruktor kann zwar nicht direkt genutzt werden, weil *Schedule* abstrakt ist, jedoch liefert er die Basisfunktionalität für die Unterklassen von *Schedule*.
Beim Start von IPODIPSI wird je nach gewünschter Betriebsart entschieden, welche konkrete Implementation (Unterklasse) von *Schedule* verwendet wird: NormalSchedule, MonitorSchedule oder MobiSchedule:

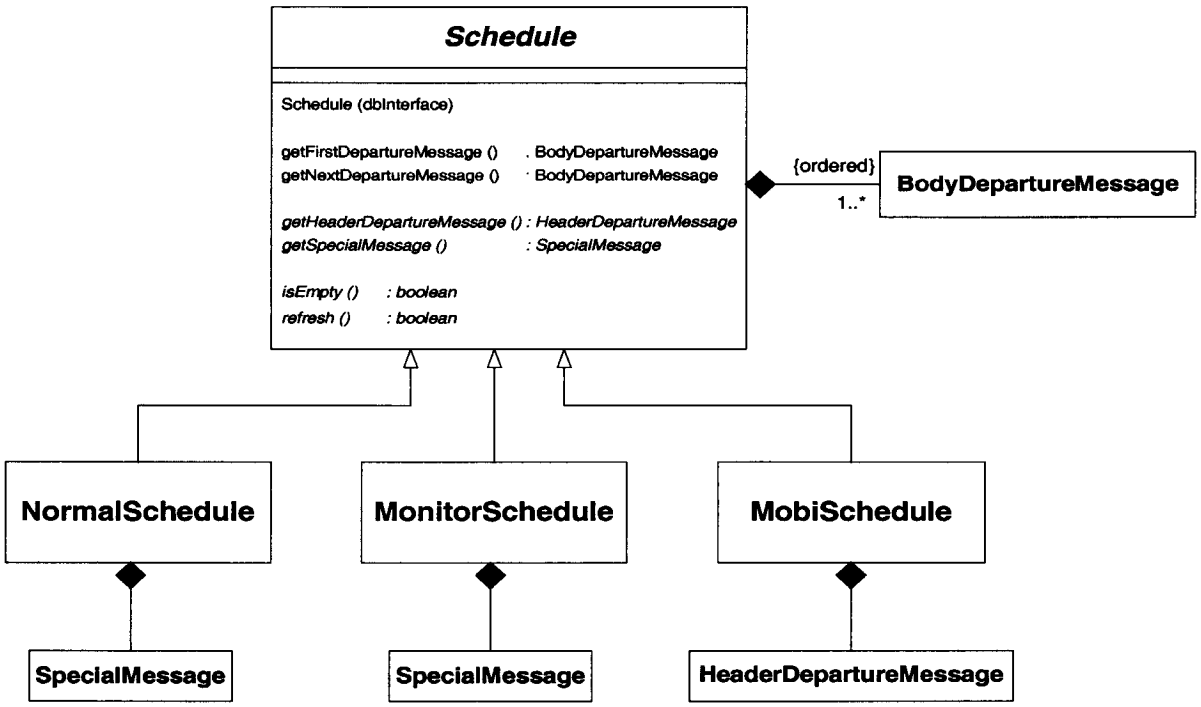


Abbildung 21: Die Klasse Schedule mit ihren konkreten Ausprägungen

In den Methoden der Unterklassen ist dann berücksichtigt, wie der Fahrplanausschnitt strukturiert ist (HeaderDepartureMessage und/oder SpecialMessage vorhanden?), wie die Aktualisierung mit der Datenbank mit refresh() erfolgt und wann der Fahrplanausschnitt mit isEmpty() als leer anzusehen ist. Unabhängig davon, welche Unterklasse von *Schedule* zur Verwendung kommt, wird das erzeugte Objekt immer abstrakt als *Schedule* angesprochen.

In den folgenden Abschnitten werden nun die in *Schedule* benutzten Klassen für Fahrt- und Sondermeldungen beschrieben (*DepartureMessage*, HeaderDepartureMessage, BodyDepartureMessage, SpecialMessage).

[7] Hieraus wird deutlich, dass es im laufenden Betrieb stets zwei Objekte aus Unterklassen von Schedule gibt: Eines, das die Aktualisierung aus der online-Datenbank durchführt und ein entsprechendes für die offline-Datenbank. Ist ein Wechsel von online nach offline oder umgekehrt notwendig, so wird bei der Ausgabegenerierung das aktuelle Schedule-Objekt gewechselt.

6.2.3.2.3 Die Klasse *DepartureMessage* und ihre Unterklassen

Fahrtmeldungen im Kopfteil unterscheiden sich von Fahrtmeldungen im Rumpf lediglich dadurch, dass sie anders dargestellt werden. Ihre sonstigen Eigenschaften sind identisch und in der abstrakten Oberklasse *DepartureMessage* zusammengefasst.
Die folgende Tabelle beschreibt die Zuordnung der Spaltennamen zu Attributen der Klasse *DepartureMessage*:

Spalte in **FAHRTMELDUNG**	**Attribut in** DepartureMessage
FAHRTMELDUNG_ID	id
HALTEPUNKT_ID	
LINIENNUMMER	routeID
LINIENZIEL	routeDestination
(LINIENZIEL_ID)	routeDestinationID
LINIENVARIANTE	routeVariant
ZUSATZTEXT1	viaInfo
ZUSATZTEXT2	
VERKEHRSBETRIEB_ID	
	transpCompany
VERKEHRSMITTEL_ID	
	transpTypeLogo
ANKUNFTSZEIT	arrival
ABFAHRTSZEIT	departure
	arrivalDelay
	departureDelay
DELTA_ANKUNFTSZEIT	actualArrival
DELTA_ABFAHRTSZEIT	actualDeparture
STRG	

Normalerweise enthalten Klassen keine expliziten IDs, da eine ID keine Eigenschaft eines Objektes ist[8]. Bei neuen Fahrtmeldungen kann jedoch auf vorangegangene Fahrtmeldungen über eine ID Bezug genommen werden (z.B. bei einem DELETE). Daher ist hier das Mitführen einer ID notwendig.
Zu HALTEPUNKT_ID gibt es in *DepartureMessage* keine Entsprechung.

[8] Zwei Objekte unterscheiden sich, „weil sie existieren" bzw. auf Implementationsebene durch verschiedene Orte im Speicher. In Java sind zwei Objekte verschieden, falls für zwei Objektreferenzen a und b gilt: a = = b.

Auf welchen Haltepunkt sich eine Fahrtmeldung bezieht, ergibt sich in der jeweiligen Betriebsart wie folgt:

Abfahrtanzeiger In der Datenbank gibt es eine Zuordnung von Geräte-ID zu Haltepunkt-ID. Die Geräte-ID wird mit der Installation eingestellt und ändert sich zur Laufzeit nicht mehr.

Fahrzeuganzeige In der Datenbank gibt es eine Zuordnung von Haltestellen-ID zu Haltepunkt-ID. Die aktuelle Haltestellen-ID (und Linienziel-ID) wird zur Laufzeit über den extern laufenden TramPos-Socketserver ermittelt.

Die Haltepunkt-ID ist also immer nur ein Selektions-Parameter.

Die Spalte LINIENZIEL_ID wurde für die Betriebsart Fahrtzeuganzeige hinzugenommen. Sie wird benötigt, weil TramPos bzw. IBIS keine Namen sondern Kürzel verwenden (18, 36, 2a, ...).

Die Spalten ZUSATZTEXT1 und ZUSATZTEXT2 enthalten Via-Informationen (oder ggf. fahrtbezogene Sondertexte[9]). Diese wurden im Attribut viaInfo zusammengefasst.

Zu einer Fahrtmeldung soll das Verkehrsmittel und der Verkehrsbetrieb angezeigt werden können, jedoch nicht in Form von abkürzenden IDs. *DepartureMessage* hat eigene Attribute transpCompany und transpTypeLogo (Name des Verkehrsbetriebs, Pfad der Logo-Datei), deren Werte über die Tabellen FAHRTMELDUNG, VERKEHRSMITTEL und VERKEHRSBETRIEB ermittelt werden.

Neben den geplanten und den eigentlichen Ankunftzeiten (arrival, actualArrival) und Abfahrtzeiten (departure, actualDeparture) sind in *DepartureMessage* die Verspätungen der Ankunft und Abfahrt enthalten, die ebenfalls für eine Anzeige relevant sind (arrivalDelay, departureDelay).

Die Spalte STRG wird für Steuerungszwecke auf der Sendeseite verwendet und ist hier bedeutungslos.

[9] Im Pflichtenheft ist beschrieben, dass für die Via-Information ggf. auf eine Wegtabelle mit den Spalten VIA1 und VIA2 zugegriffen wird. Gibt es solch eine Weginformation, so kann sie auch für DepartureMessage genutzt werden (variable Erzeugung der Via-Information, siehe Pflichtenheft 2.1.1.2.1). Der Aufbau der Klasse DepartureMessage bleibt jedoch unverändert – nur die Abbildung der Datenbank-Informationen auf ein Objekt (Schedule.refresh()) muss angepasst werden.

Die abstrakte Klasse *DepartureMessage* ist nun wie folgt definiert:

DepartureMessage		
# DepartureMessage ()		
# DepartureMessage (defaultsRequired)		
getID	()	: String
getTranspTypeLogo	()	: String
getTranspCompany	()	: String
getRouteID	()	: String
getRouteVariant	()	: String
getRouteDestinationID	()	: String
getRouteDestination	()	: String
getViaInfo	()	: String
getArrival	()	: Date
getDeparture	()	: Date
getArrivalDelay	()	: long
getDepartureDelay	()	: long
getActualArrival	()	: Date
getActualDeparture	()	: Date

Als Datenklasse verfügt sie lediglich über Operationen, die die Werte derjenigen Attribute liefern, die für die Anzeige von Bedeutung sein können. Das Setzen der Attributwerte erfolgt, wenn *Schedule* die Nachricht refresh() erhält.

Die Klasse und ihre Konstruktoren sind nur im Paket ipodipsi.data selbst sichtbar, da eine Fahrtmeldung immer nur als Bestandteil eines Fahrplans und nie eigenständig existieren soll (siehe oben: *Schedule* ist eine Komposition und nicht nur eine Aggregation). Die Erzeugung von *DepartureMessage*-Objekten geschieht also ausschließlich über die Klasse *Schedule*.

Für *DepartureMessage* stehen die Implementationen HeaderDepartureMessage und BodyDepartureMessage zur Verfügung, aus denen Objekte erzeugt werden können. Ihre Schnittstelle nach außen ist identisch. Intern haben sie einen eigenen Konstruktor und eine eigene Copy-Methode. Für die Weiterverarbeitung ist lediglich von Bedeutung, dass es *DepartureMessage*-Objekte gibt, die unterschiedlich behandelt werden müssen:

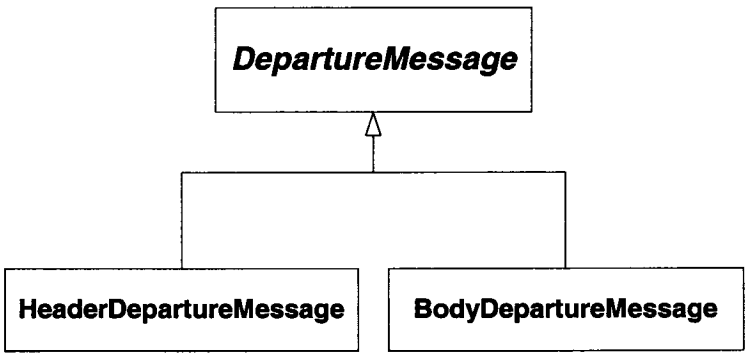


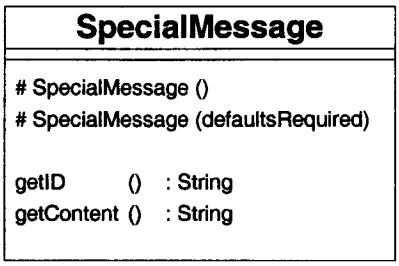
Abbildung 22: Die abstrakte Klasse DepartureMessage und mögliche Unterklassen

6.2.3.2.4 Die Klasse SpecialMessage

Die Zuordnung der Spaltennamen zu den Attributen der Klasse SpecialMessage ist wie folgt:

Spalte in SONDERMELDUNG	Attribut in SpecialMessage
SONDERMELDUNG_ID	id
HALTEPUNKT_ID	
SONDERTEXT	content
VERKEHRSBETRIEB_ID	
GUELTIG_VON	
GUELTIG_BIS	
SM_GUELTIG	

Für die Spalten SONDERMELDUNG_ID und HALTEPUNKT_ID gilt das Gleiche wie oben in den Erläuterungen zu DepartureMessage für FAHRTMELDUNG_ID und HALTEPUNKT_ID. Zur Spalte VERKEHRSBETRIEB_ID gibt es keine Entsprechung, da dies kein Bestandteil der Anzeige einer Sondermeldung ist.
Für die Gültigkeit einer Sondermeldung ist in der Infopool-Datenbank gegenüber der Feinspezifikation die Spalte SM_GUELTIG hinzugekommen, die ein Flag mit den Werten „0" oder „1" darstellt. Eine Sondermeldung ist nun erst dann als gültig anzusehen, wenn sie im Zeitraum zwischen GUELTIG_VON und GUELTIG_BIS liegt und außerdem „eingeschaltet" ist (SM_GUELTIG = „1").
Die Spalten zur Gültigkeit haben in der Klasse SpecialMessage keine Entsprechung, da sie nur für das Heraussuchen, jedoch nicht für die Präsentation einer Sondermeldung relevant sind (nur Selektions-Parameter, siehe auch oben HALTEPUNKT_ID bei DepartureMessage).



Auch SpecialMessage ist eine Datenklasse, und die Bedingungen zu *DepartureMessage* gelten analog. Klasse und Konstruktoren sind wieder nur im Paket ipodipsi.data sichtbar.

Die weiteren Datenklassen in ipodipsi.data (PositionData und PositionState) werden im Zusammenhang mit der Klasse ipodipsi.MobiController erläutert.

6.2.3.3 Die Anzeige eines Fahrplanausschnitts (ipodipsi.output)

Im Paket ipodipsi.output befinden sich diejenigen Klassen, die die Darstellung eines Fahrplanausschnitts realisieren. Hierfür liegt folgender Ansatz zu Grunde:

- Der Fahrplanausschnitt wird als HTML-Seite dargestellt.
- Basis der HTML-Seite ist ein Muster (Schablone), das feste und variable Bestandteile hat (Layout und Daten). Das Layout wird durch normalen HTML-Code festgelegt. Die Daten sind durch spezielle Platzhalter (Marker) gekennzeichnet.
- Neben einfachen Daten-Markern gibt es solche, die strukturierte Daten beschreiben, z.B. „hier beginnt/endet der Daten-Kopfteil". Eine Schablone ist also eine normale HTML-Datei, die an einigen Stellen Verweise auf elementare und auf strukturierte Daten enthält.
- Zur Laufzeit werden die Daten-Marker einer Schablone durch die aktuellen Daten eines Fahrplanausschnitts ersetzt. Die Struktur-Marker fallen weg. Andere Bestandteile bleiben unverändert.
- Der so erzeugte, fertige HTML-Text wird als aktuelle Seite in eine Datei geschrieben. Ein extern laufender Browser wird dazu veranlasst, diese Seite darzustellen.

Für diesen Ansatz sind also neben den Klassen für die Datenhaltung (realisiert in ipodipsi.data) solche notwendig, die eine Schablone realisieren, die mit diesen Daten abgeglichen werden kann.

6.2.3.3.1 Die Struktur einer Schablone

Zum besseren Verständnis für die Struktur einer Schablone betrachte man das folgende Beispiel. Es handelt sich hier um einen Ausschnitt[10] für eine Mobilist-Anzeige (siehe Abbildung auf S. 38). Die Daten-Marker sind zur Verdeutlichung **fett** notiert und stehen in geschweiften Klammern, z.B. als **{LINIE}**. Die Struktur-Marker stehen in eckigen Klammern, z.B. als **[SCHEDULE_BEGIN]**:

[10] <body>-Teil, bei dem zur besseren Lesbarkeit einige HTML-Tags weggelassen sind (font, color).

```
<table  width="100%"  border="0"  cellspacing="0"  cellpadding="0"  bgcolor="#000000">

  <tr>

    <td  width="140"  align="center"  bgcolor="#000000">
    <embed
      type      ="application/x-java-applet;version=1.3"
      codebase  = "."
      code      = "uhr_applet.bahn_uhr.class"
      name      = "TestApplet"
      width     = "140"
      height    = "140"
      hspace    = "0"
      vspace    = "0"
      align     = "Mitte"
      dimension = "140"
      autoSize  = "TRUE"

      bgcolor         = "000000"
      hourDotcolor    = "ffdd00"
      hour2Dotcolor   = "ffdd00"
      minuteDotcolor  = "ffdd00"
      centerDotcolor  = "ffdd00"

      hourHandcolor   = "ffdd00"
      minuteHandcolor = "ffdd00"
      secondHandcolor = "ffdd00"

      dial1Color    = "000000"
      dial2Color    = "000000"
      dial3Color    = "000000"
      dial4Color    = "000000"
    >
    </embed>
    </td>

    <td  align="center"  width="100%" >
    <table  width="660"  height="140"  border="0"  cellspacing="0"  cellpadding="1"
bgcolor="#000000">

    [HEADER_BEGIN]
      <tr>
        <td  colspan="4"  align="center"  width="660" > <font size=+4><b> Vaihingen Bahnhof
</b></font></td>
      </tr>
      <tr>
        <td align=center valign=bottom width="25%" height="20%"> Ankunft </td>
        <td align=left   valign=bottom width="25%" height="20%"> {ANKUNFTSZEIT}
</font></td>
        <td align=center valign=bottom width="25%" height="20%"> {ANKUNFT_HEUTE}
</font></td>
        <td align=left   valign=bottom width="25%" height="20%"> {DELTA_ANKUNFTSZEIT}
</font></td>
      </tr>
      <tr>
        <td align=center valign=top    width="25%" height="20%"> Abfahrt
</font></td>
         <td align=left   valign=top    width="25%" height="20%"> {ABFAHRTSZEIT}
</font></td>
        <td align=center valign=top    width="25%" height="20%"> {ABFAHRT_HEUTE}
</font></td>
        <td align=left   valign=top    width="25%" height="20%"> {DELTA_ABFAHRTSZEIT}
</font></td>
      </tr>
    [HEADER_END]

    </table>
    </td>

  </table>
    </br>
```

```
  <table width="800"  border=0  cellspacing=0  cellpadding=0  bgcolor="#ffffff" >

      <tr>
        <td colspan="2" align="left"   width="70%" height="5">     Umsteigen zu
</td>
        <td colspan="2" align="center" width="30%" height="5"> Abfahrt
</td>
      </tr>

  </table>
    <br>


  <table width="800"  border="0"  cellspacing="0"  cellpadding="0"  bgcolor="#330066" >

    [SCHEDULE_BEGIN]
    <tr>
      <td  align=center  width="10%"  height="5"> <b>{VERKEHRSMITTEL}     </td>
      <td  align=center  width="10%"  height="5"> <b>{LINIE}              </td>
      <td  align=left    width="55%"  height="5"> <b>{LINIENZIEL}         </td>
      <td  align=center  width="10%"  height="5"> <b>{ABFAHRT_HEUTE}      </td>
      <td  align=center  width="15%"  height="5"> <b>{DELTA_ABFAHRTSZEIT} </td>
    </tr>
    [SCHEDULE_END]

  </table>
```

6.2.3.3.1.1 Grundstruktur

Eine Schablone (*OutputPattern*) besteht aus 1..n Beschreibungen (*Description*). Jede Beschreibung besteht wiederum aus 1..n Elementen (*DescriptionElement*), die HTML-Text oder Daten-Platzhalter sind:

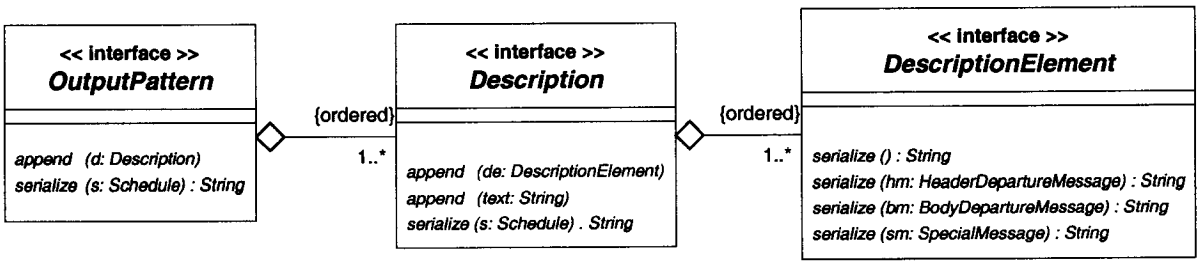


Abbildung 23: Grundstruktur einer Schablone (OutputPattern)

Eine Beschreibung ist entweder statisch (HTML-Text) oder dynamisch (HTML-Text mit Markierungen für Daten). Bei einer dynamischen Beschreibung wird zwischen Kopfteil (HeaderDescription), Anschlussteil (ScheduleDescription) und Sondermeldung (AnnouncementDescription) unterschieden:

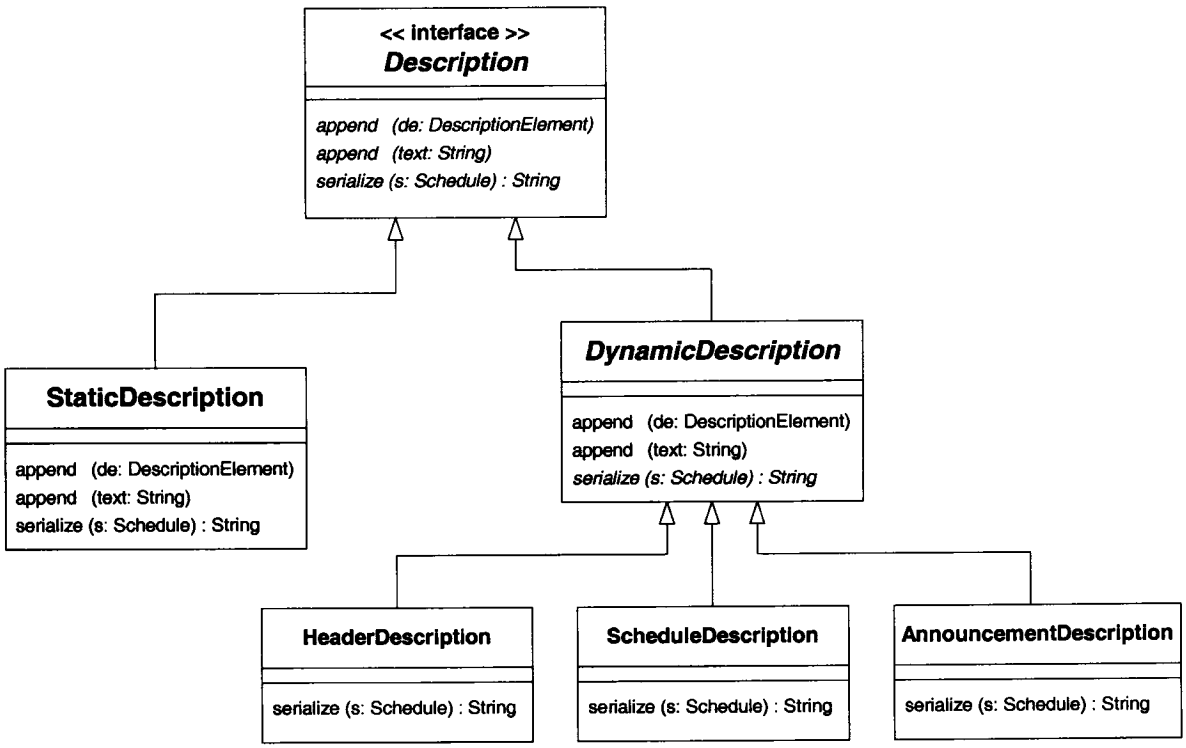


Abbildung 24: Mögliche Arten von Beschreibungen (Description) in einer Schablone

Diese Entwurfsentscheidung hat folgenden Hintergrund: Es kann Daten-Markierungen in einer Schablone geben, die zwar gleich lauten, aber verschieden interpretiert werden müssen. Man betrachte hierzu noch einmal die Anzeige für die Mobilist-Betriebsart (S. 38). Kopfbereich und Verbindungsbereich basieren auf Fahrtmeldungsdaten (*DepartureMessage*), die zwar unterschiedlich aus der Datenbank selektiert werden, aber die selben Attribute haben. Diese Attribute korrespondieren mit den Daten-Markern in einer Schablone.

Für den Kopfteil gilt:	Eine Ist-Abfahrtszeit soll nur dann ausgegeben werden, wenn sie von der Soll-Abfahrtszeit abweicht – es gibt tatsächlich eine Verspätung. Im abgebildeten Beispiel ist dies nicht der Fall.
Für den Verbindungsbereich gilt:	Die Ist-Abfahrtszeit ist *immer* anzuzeigen, unabhängig davon, ob eine Verspätung vorliegt oder nicht.

Das Attribut *DepartureMessage*.actualDeparture , das in der Schablone mit [DELTA_ABFAHRTSZEIT][11] markiert ist, muss also für Kopfbereich und Verbindungsbereich unterschiedlich interpretiert werden.
Die Methoden

HeaderDescription.serialize (s: *Schedule*)
ScheduleDescription.serialize (s: *Schedule*)

senden an ihre enthaltenen Elemente (*DescriptionElement*) die Nachricht, sich auf Basis von HeaderDepartureMessage bzw. ScheduleDepartureMessage zu serialisieren. In den Methoden zur Verarbeitung dieser Nachricht kommt dann die unterschiedliche Interpretation zur Anwendung.

[11] Die unterschiedlichen Namen für Attribute in Schablonen und in *DepartureMessage* sind historisch gewachsen. Zukünftig sollten sie gleich lauten.

6.2.3.3.1.2 Elemente

Bei einem Element einer Beschreibung handelt es sich entweder um HTML-Text, der unverändert ausgegeben werden soll, oder um einen Daten-Marker:

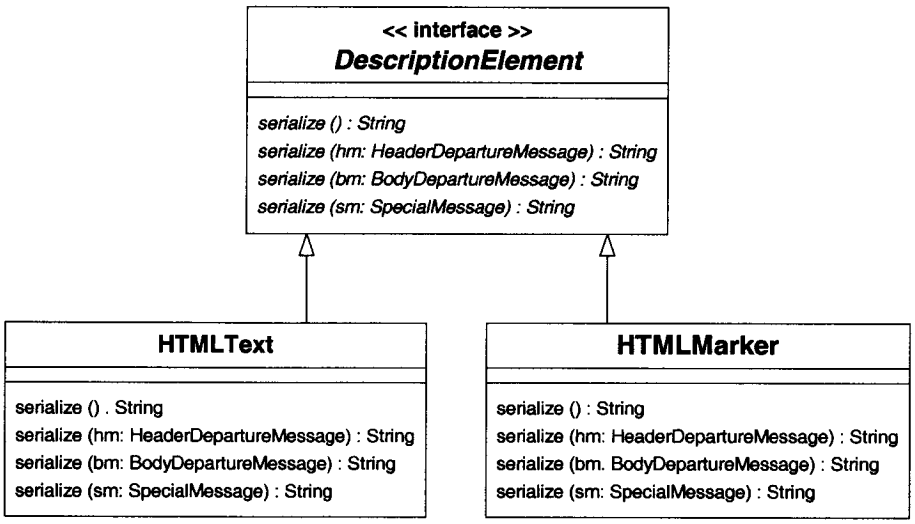


Abbildung 25: Elementare Bestandteile einer Schablone

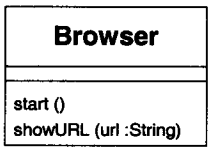
Jedes Element ist dazu in der Lage, sich selbst auszugeben (zu serialisieren), und zwar unverändert, also genau wie in der Schablone notiert (serialize ohne Parameter), oder abhängig von einem Bestandteil eines Fahrplanausschnitts (HeaderDepartureMessage, BodyDepartureMessage, SpecialMessage).

6.2.3.3.1.3 Sonstiges

Eine Schablone soll zunächst als HTML-Schablone angesehen werden, und daher gibt es für *Output-Pattern* bisher nur die Implementation HTMLOutputPattern. Es ließen sich auch andere Dateiformate erzeugen, die möglicherweise eine spezielle Handhabung erfordern, die vom Standard in HTMLOutputPattern abweicht.
Welche Marker für Daten und Strukturen definiert sind und in einer Schablone als solche erkannt werden, ist in der Klasse ipodipsi.output.M festgelegt.

6.2.3.3.2 Die Anzeige der HTML-Seite

Für die Anzeige wird ein extern laufender HTML-Browser eingesetzt. Für seine Ansteuerung kommt die Klasse Browser zum Einsatz. Sie verfügt lediglich über Methoden zum Starten des Browsers und zum Anzeigen einer URL[12].



6.2.3.3.3 Der Kontrollfluss

Die Controller-Klasse OutputCreator regelt die Aufbereitung eines aktuellen Fahrplanausschnitts und die Anzeige mit Hilfe eines Browsers. Die Fahrplandaten bezieht sie über *Schedule* – einmal für die online-, einmal für die offline-Daten (Umschaltprinzip, siehe Abschnitt 0). Ebenso werden zwei Exemplare von OutputPattern verwendet – damit ist es möglich, dass Ist-Daten (online-Modus) und Soll-Daten (Saisonfahrplan, offline-Modus) unterschiedlich präsentiert werden:

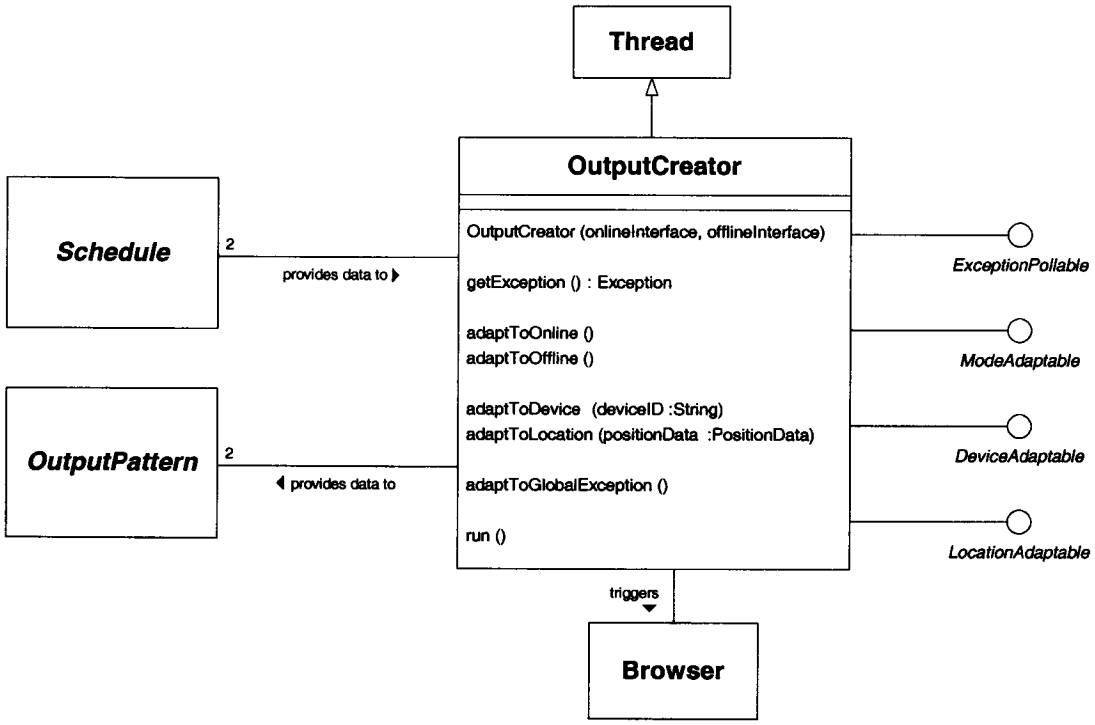


Abbildung 26: OutputCreator ist für die Steuerung der regelmäßigen Ausgabe zuständig.

Erzeugung und Anzeige der aktuellen HTML-Seite geschehen nebenläufig. OutputCreator ist ein Thread, der in run() zyklisch sein Schedule-Objekt aktuell hält (Schedule.refresh()) und bei Änderungen sein OutputPattern-Objekt damit ausgibt (OutputPattern.serialize (schedule)). Es werden folgende Interfaces implementiert (sie sind im Zusammenhang mit dem Paket ipodipsi ausführlich beschrieben):

[12] Als Browser kommt der Netscape Navigator 4.7 zum Einsatz. Die Implementation ist z.Zt. spezifisch für MS Windows und wird dadurch realisiert, dass mit speziellen Bibliotheken das DDE-Interface zum Browser angesprochen wird. Das Steuern über die Kommandozeile wie unter UNIX ist hier nicht möglich.

ExceptionPollable	In OutputCreator könnten unerwartete Exceptions auftreten, die manuell durch Starter.main() geholt werden müssen (gilt für jeden Thread).
ModeAdaptable	OutputCreator muss auf einen Wechsel zwischen online- und offline-Modus reagieren (passendes Schedule- und OutputPattern-Objekt verwenden).
DeviceAdaptable	OutputCreator muss auf eine neue Geräte-ID reagieren können (AIH-Monitor).
LocationAdaptable	OutputCreator muss auf eine neue Haltestellen-ID reagieren können (Mobilist).

Zur Verdeutlichung des Ablaufs bei der Erzeugung der HTML-Seite betrachte man ein Objekt p der Klasse HTMLOutputPattern mit folgenden Bestandteilen:

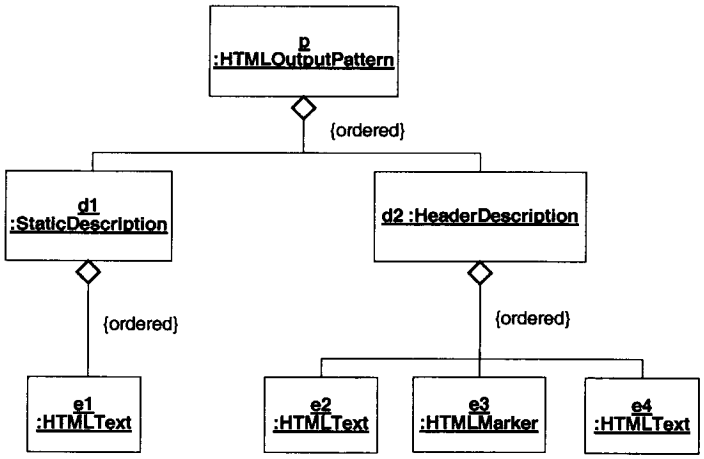


Abbildung 27: Beispiel für ein Exemplar von HTMLOutputPattern :OutputPattern

Schickt dann OutputCreator die Nachricht p.serialize (schedule), so ergibt sich folgender weiterer Verlauf von Nachrichten:

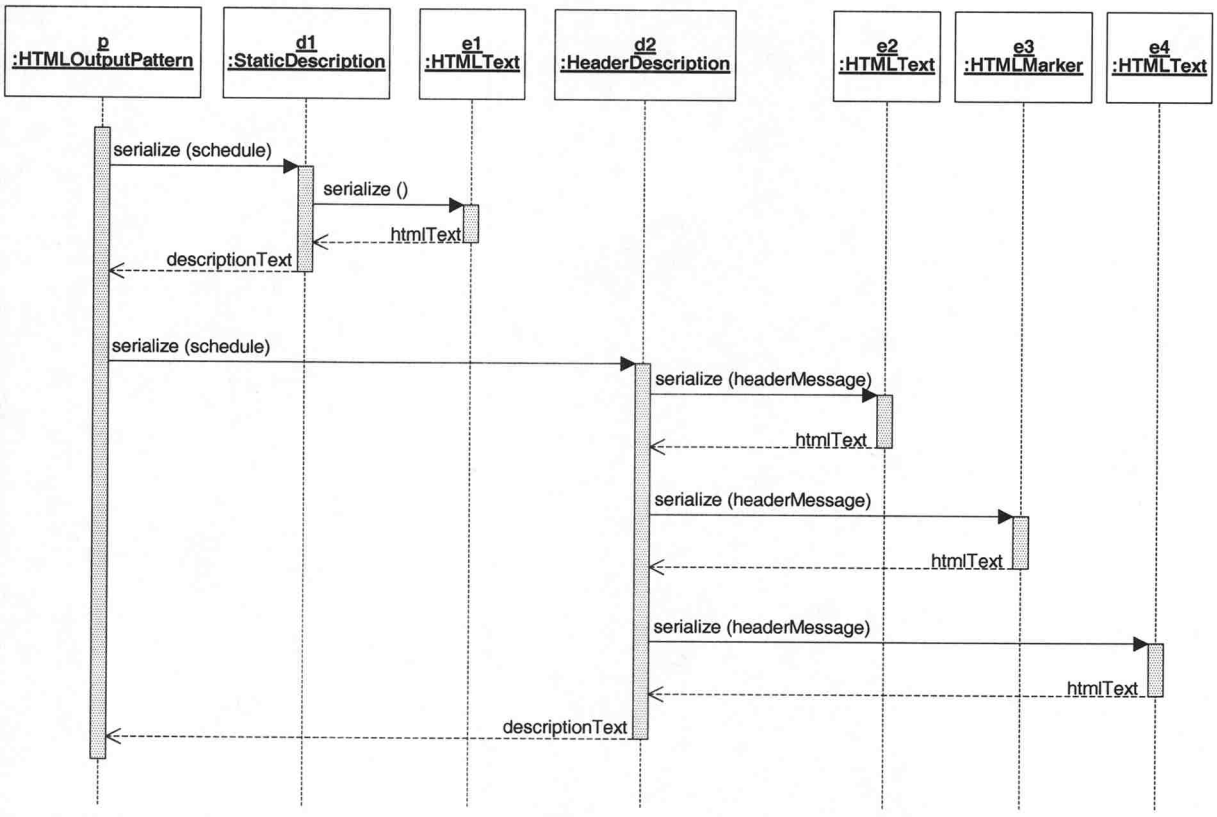


Abbildung 28: Typischer Nachrichtenverlauf bei der Erzeugung der aktuellen HTML-Seite

6.2.3.4 Der laufende Datenimport (ipodipsi.dbimport)

Zur Laufzeit muss von IPODIPSI kontinuierlich geprüft werden, ob neue Soll- oder Ist-Daten in den Datenimport-Verzeichnissen vorliegen.

6.2.3.4.1 Abfahrtanzeiger-Betrieb

Der einfachste Fall ist der Abfahrtanzeiger-Betrieb, bei dem in folgenden Schritten verfahren wird:

Online-Modus:
Prüfe, ob ein aktueller Tagesplan mit Daten vorliegt.
Falls der Plan existiert:
- Importiere ihn, falls er neu oder verändert ist.
- Führe Aktualisierungen auf diesen Plan durch, falls sie existieren.

Falls kein Plan existiert, tue nichts.
Warte und beginne von vorn.

Offline-Modus:
Prüfe, ob ein Saisonfahrplan für den heutigen Tag vorliegt.
Falls der Plan existiert:
Importiere ihn, falls er neu oder verändert ist.
Falls kein Plan existiert, tue nichts.
Warte und beginne von vorn.

Die Zustände, die in diesen Verfahren durchlaufen werden, lassen sich mit Hilfe von State-Charts darstellen. Für den Online-Betrieb sieht das State-Chart wie folgt aus:

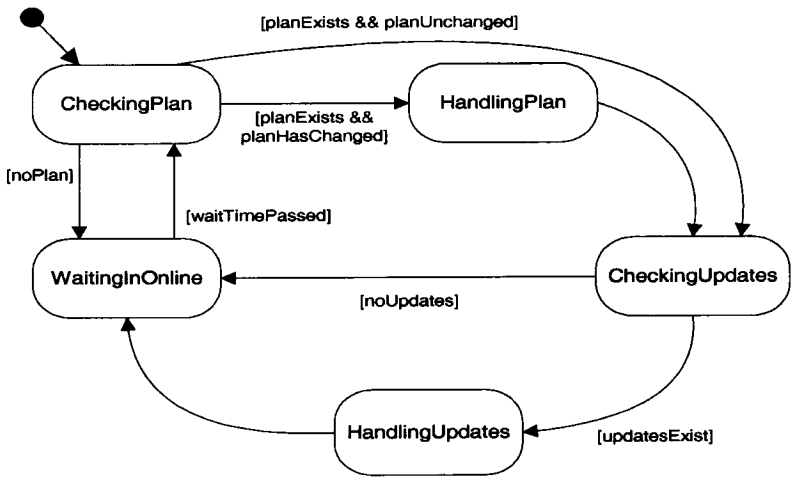


Abbildung 29: Zustände beim Datenimport im Online-Modus (Betriebsart Abfahrtanzeiger)

Für den Offline-Betrieb ergibt sich die folgende Darstellung:

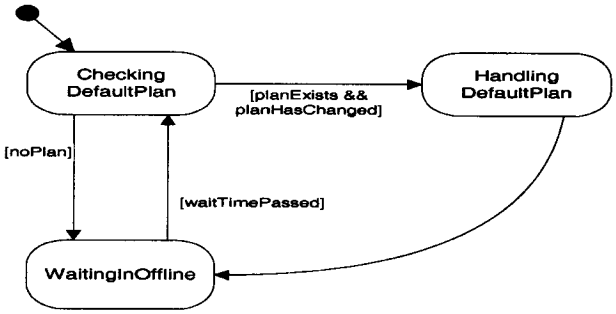


Abbildung 30: Zustände beim Datenimport im Offline-Modus (Betriebsart Abfahrtanzeiger)

6.2.3.4.2 Monitor-Betrieb

Beim Monitor-Betrieb ist das Verfahren ähnlich, jedoch werden hier die Import-Daten für mehrere Geräte verarbeitet. Die Idee dabei ist, dass es für jedes Gerät einen eigenen „Controller" gibt, der den Import durchführt. Kommt zur Laufzeit ein neues Datenverzeichnis für ein neues Gerät hinzu, so wird hierfür ein neuer Controller erzeugt. Fasst man dies mit dem Verfahren für den normalen Abfahrt-Betrieb zusammen, so ergibt sich im Online-Modus das folgende State-Chart:

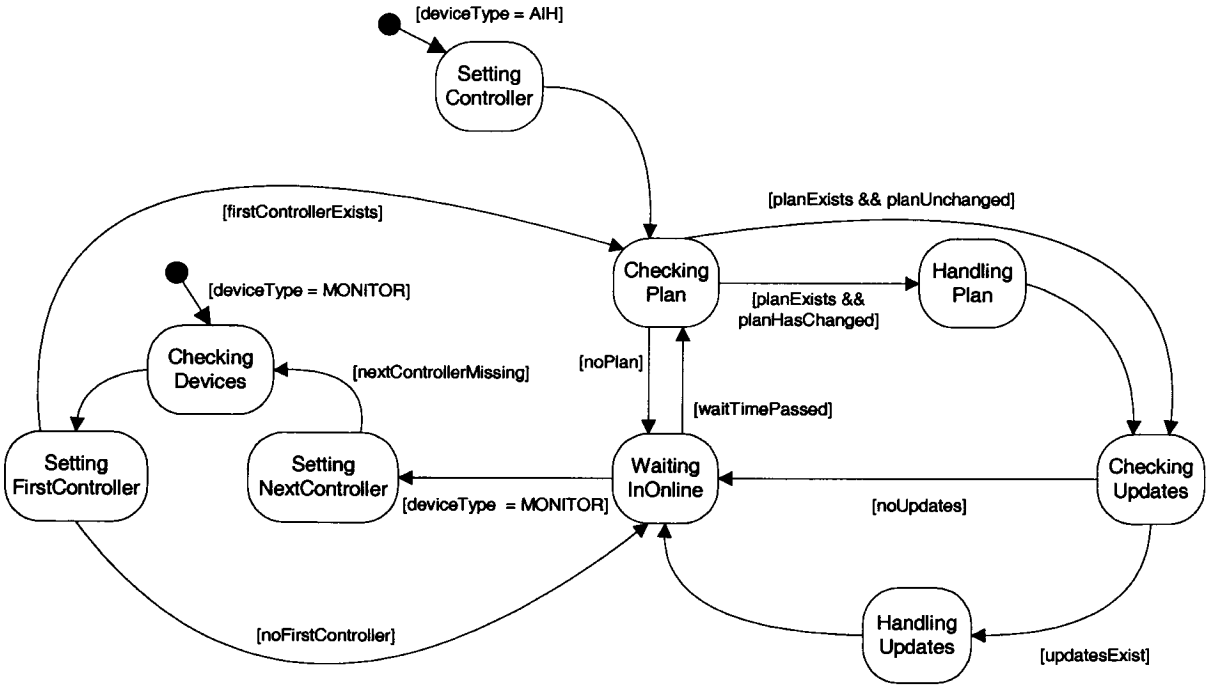


Abbildung 31: Zustände beim Datenimport im Online-Modus (Betriebsarten Monitor)

In der Betriebsart Abfahrt ist der Startzustand SettingController. Hier muss nur ein Gerät berücksichtigt werden; die Anzahl der Controller ist konstant 1. Sonstige mögliche Zustände wie in Abbildung 28

In der Betriebsart Monitor wird zuerst in CheckingDevices geprüft, für welche Geräte ein Monitoring durchgeführt werden muss. Die Anzahl dieser Geräte ist abhängig von den gefundenen Online-Datenimport-Verzeichnissen, von denen zur Laufzeit anfangs noch keine existieren und später, bei laufendem Datenempfang, mehrere hinzukommen können. Für jedes Gerät bzw. Datenimport-Verzeichnis wird ein eigener Controller angelegt. Für jeden dieser Controller wird dann der Datenimport ab CheckingPlan durchgeführt.
Das State-Chart für den Offline-Modus ergibt sich analog (Abbildung 29 mit den gleichen Erweiterungen).

6.2.3.4.3 Fahrzeuganzeige-Betrieb

Für die Mobilist-Betriebsart sind ähnliche Vorkehrungen zu treffen wie für den Monitor. Zur Laufzeit können ein oder mehrere Online-Datenimport-Verzeichnisse hinzukommen, von denen jedes die Daten für eine bestimmte Haltestelle enthält. Für die Prüfung dieser Verzeichnisse lässt sich ein eigener Zustand CheckingLocations einführen, mit dem ein State-Chart für den Online-Modus und sämtliche Betriebsarten wie folgt aussieht (für den Offline-Modus analog):

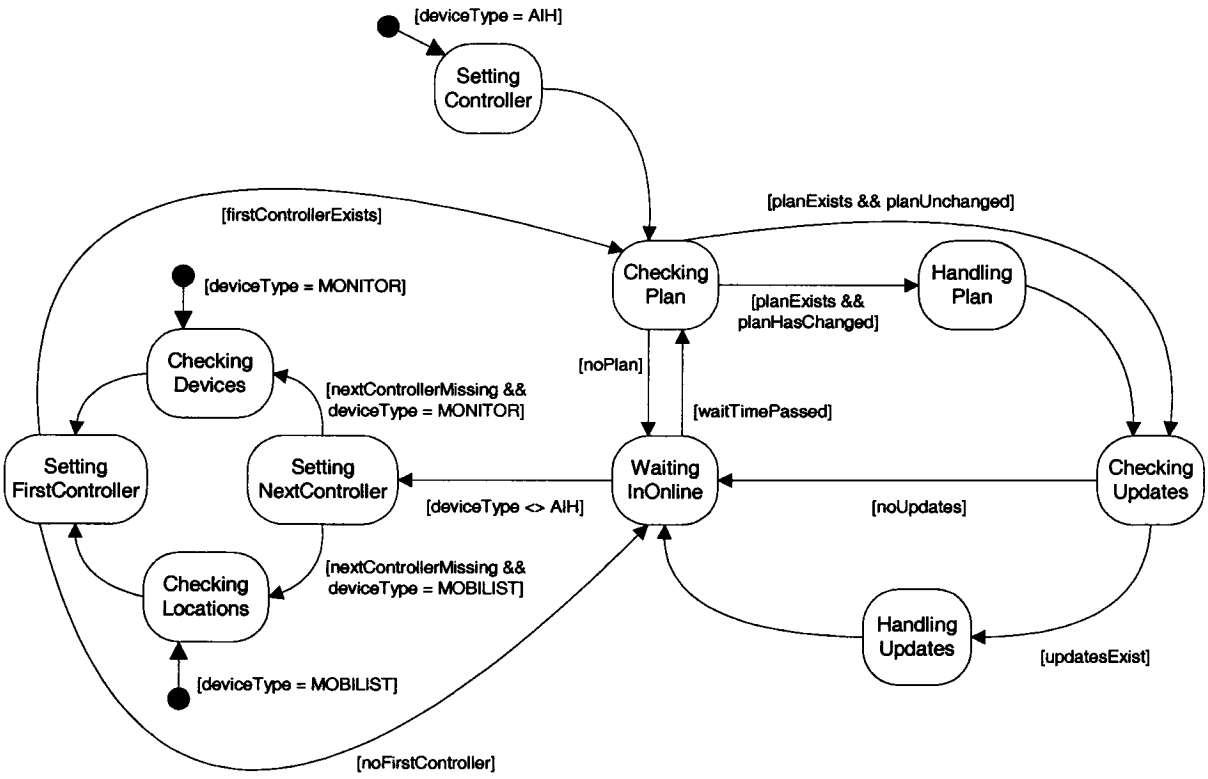


Abbildung 32: Zustände beim Datenimport im Online-Modus (sämtliche Betriebsarten)

6.2.3.4.4 Umsetzung: Zustände als Klassen

Das State-Chart, das den Kontrollfluss für alle Betriebsarten zusammenfasst (s.o.) wird so umgesetzt, dass ein Zustand einer Klasse entspricht. Dies folgt dem State-Pattern nach Gamma[13] Der State-Kontext, der das Durchschalten der Zustände veranlasst, ist für den Online-Modus mit OnlineUpdater realisiert:

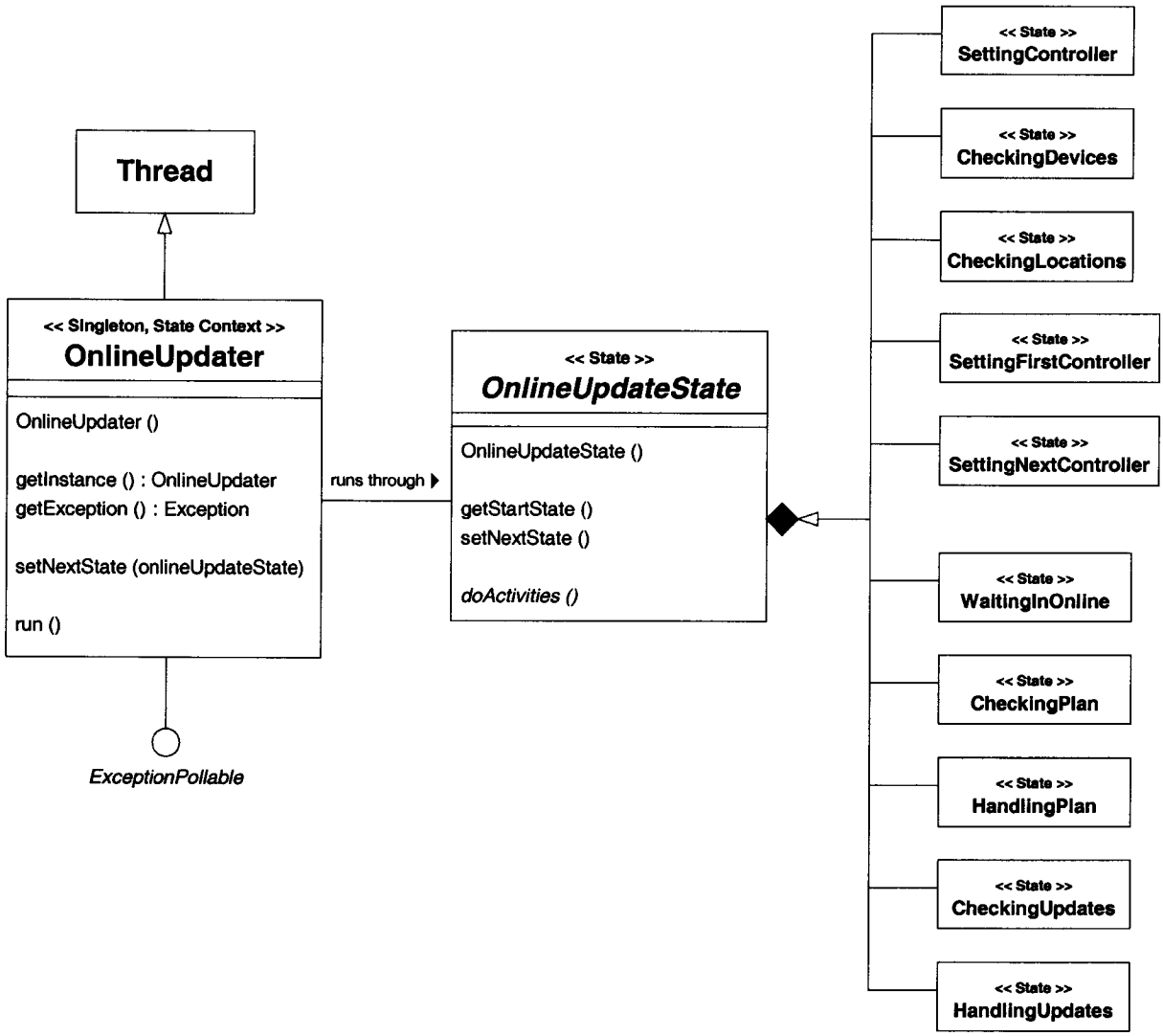


Abbildung 33: Umsetzung des Datenimports für den Online-Modus

OnlineUpdateState ist hier nicht nur Basisklasse für alle konkreten Zustände, sondern auch deren Komposition. Dies ist mit Hilfe von inneren Klassen realisiert. Dadurch ist es erlaubt, dass der entsprechende Offline-Teil mit *OfflineUpdateState* zum Teil die gleichen Namen für konkrete Zustände verwendet (z.B. CheckingDevices).
Der State-Kontext für den Offline-Modus ist ähnlich modelliert:

[13] E. Gamma et al.: Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley.

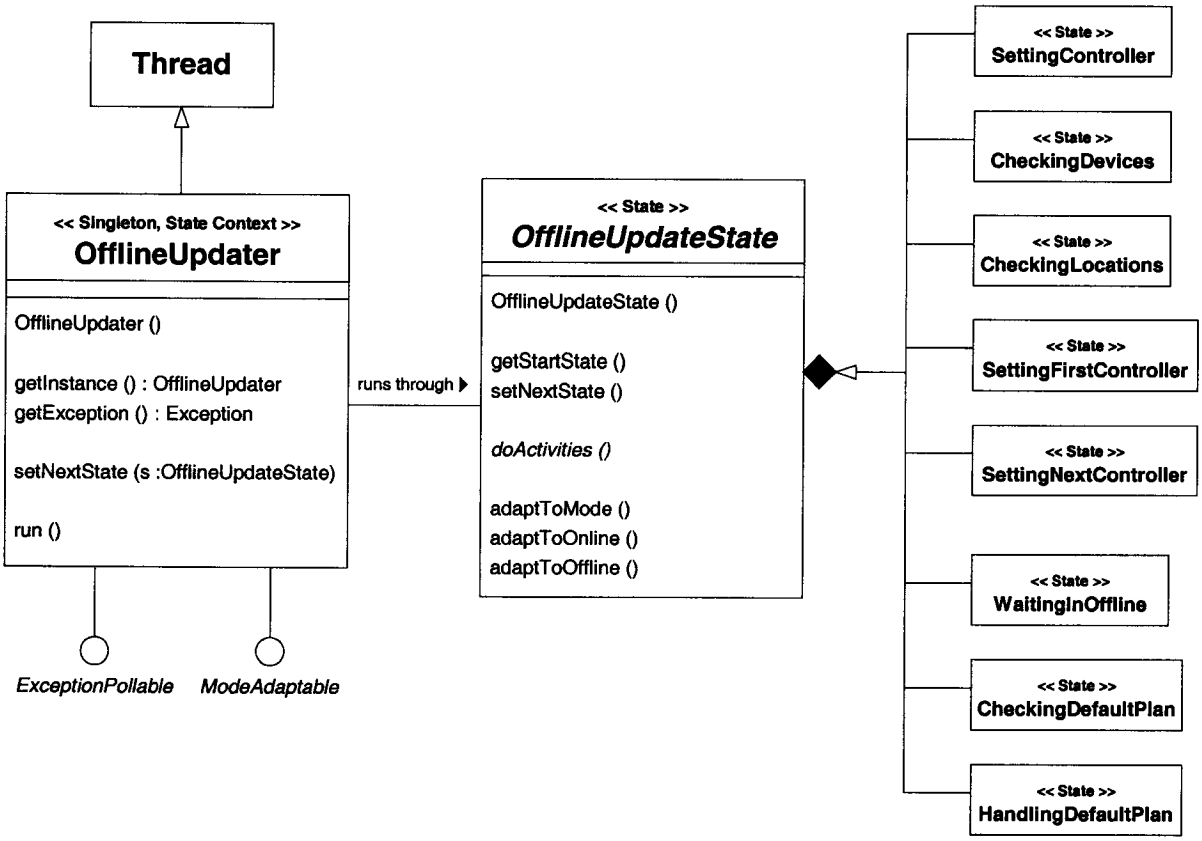


Abbildung 34: Umsetzung des Datenimports für den Offline-Modus

Im Unterschied zum OnlineUpdater soll der OfflineUpdater dazu in der Lage sein, auf einen Wechsel zwischen Online- und Offline-Modus zu reagieren: Solange IPODIPSI online ist, sollte der Import der Offline-Daten weniger Priorität (längere Wartezyklen) haben. Daher implementiert OfflineUpdater das Interface *ModeAdaptable*, und *OfflineUpdateState* verfügt über Methoden, dies umzusetzen.
Sowohl OfflineUpdater als auch OnlineUpdater sind Threads, bei denen unerwartete Exceptions auftreten könnten, die manuell durch Starter.main() geholt werden müssen. Daher implementieren beide das Interface *ExceptionPollable*.

6.2.3.4.5 Der Import-Vorgang

Der eigentliche Import erfolgt über einen ImportController. Für die Betriebsarten Abfahrt und onitor regelt er für jeweils eine Geräte-ID die Überwachung der Importdateien („Was ist vorhanden, was hat sich geändert?") und den Import in die Datenbank. Beim „Abfahrt" ist die Geräte-ID konstant, beim Monitor kann sie laufend wechseln. Für die Betriebsart Fahrzeuganzeige erfolgt dieser Wechsel für jeweils eine Haltestellen-ID, die sich mit jeder neu angefahrenen Haltestelle ändert.
Der ImportController verteilt seine Aufgaben auf einen ImportWatcher und einen ImportHandler:

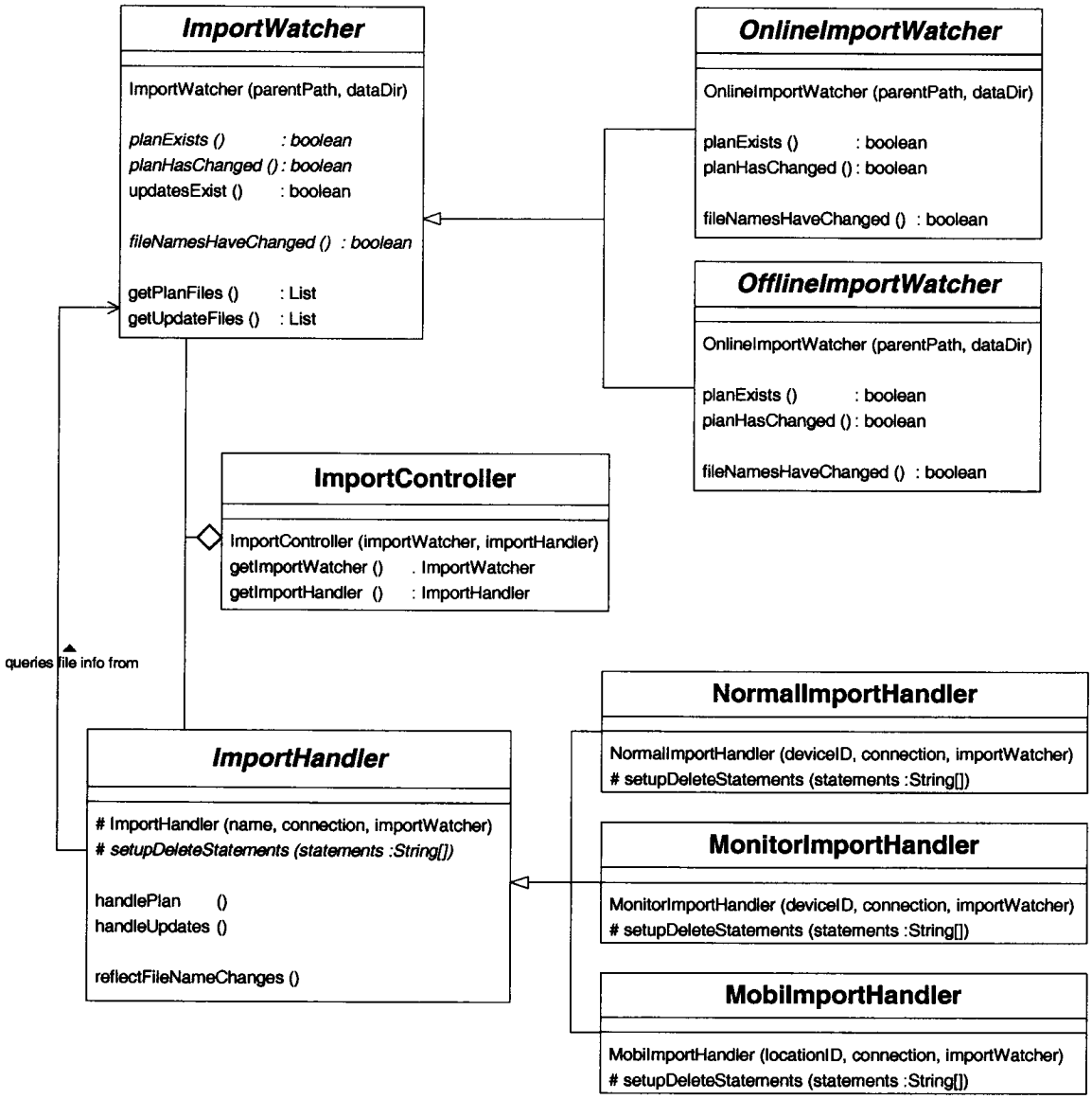


Abbildung 35: Aufgabenverteilung beim Import-Prozess

Ein ImportWatcher gibt Auskunft über den Status der Importdateien. Er ist für Online- und Offline-Modus implementiert und berücksichtigt dabei die unterschiedlichen Verzeichnisse und Regeln für Tagespläne. Über die Nachricht fileNamesHaveChanged() lässt sich feststellen, ob sich die Namen der Importdateien geändert haben. Dies kann z.Zt. dann der Fall sein, wenn im Offline-Modus ein neuer Tag begonnen hat, da sich der Name des Verzeichnisses für den Saisonfahrplan immer aus dem aktuellen Tag ergibt – das Muster ist <JJJJ-MM-TT>, also z.B. 2001-12-24.
Ein ImportHandler kann neue oder geänderte Tagespläne und deren Änderungen importieren. Er ist für die einzelnen Betriebsarten (Abfahrt, Monitor, Fahrzeuganzeige) speziell implementiert (NormalImportHandler, MonitorImportHandler, MobilImportHandler). Dabei wird berücksichtigt, dass unterschiedliche Delete-Statements zum Aufräumen in der Datenbank ausgeführt werden müssen, bevor ein neuer oder geänderter Tagesplan importiert werden kann.
Haben sich die Namen der Importdateien geändert (s.o.), so kann ImportHandler über die Nachricht reflectFileNameChanges() darüber informiert werden und dies entsprechend umsetzen. Er benötigt dazu eine neue Liste der Importdateien für Tagesplan und Änderungen und erhält diese über ImportWatcher.getPlanFiles() und ImportWatcher.getUpdateFiles().

6.2.3.4.5.1 Der eigentliche Import

Für den eigentlichen Import müssen ASCII-Dateien im Infopool-Format verarbeitet werden können[14]. Als Beispiel hier die ersten Zeilen eines typischen Beispiels (Zeilennummern links gehören nicht zum Inhalt):

```
001: ## ------------------------------------------------------------------------
002: ## esdToInfopoolKonvertierung der Datei: c:\fux\u3_vaihingen.esd
003: ## ------------------------------------------------------------------------
004: ## Datum der Konvertierung: 04.12.2001 13:43:48
005: ## Konvertiertag          : 12.11.2001 00:00:00
006: INSERT
007: FAHRTMELDUNG
008: ABFAHRTSZEIT,DELTA_ABFAHRTSZEIT,ANKUNFTSZEIT,DELTA_ANKUNFTSZEIT,LINIENNUMMER,VERKEHRSMITTEL_ID, ...
009: 2002-01-21 00:05:00,2002-01-21 00:05:00,2002-01-21 00:04:00,2002-01-21 00:04:00,U3,U, ...
010: 2002-01-21 00:25:00,2002-01-21 00:25:00,2002-01-21 00:24:00,2002-01-21 00:24:00,U3,U, ...
011: 2002-01-21 00:45:00,2002-01-21 00:45:00,2002-01-21 00:44:00,2002-01-21 00:44:00,U3,U, ...
012: 2002-01-21 04:13:00,2002-01-21 04:13:00,2002-01-21 04:12:00,2002-01-21 04:12:00,U3,U, ...
```

Erläuterung:

Zeilen 1-5	Kommentare zur Erzeugung der Datei.
Zeile 6	Durchzuführende Aktion (hier: INSERT, also neue Daten einfügen).
Zeile 7	Datenbanktabelle, auf die die Aktion durchzuführen ist (hier: FAHRTMELDUNG)
Zeile 8	Spalten der Tabelle, für die Werte angegeben sind.
Zeilen 9 ff.	Spaltenwerte in der Reihenfolge wie in Zeile 8 (NULL-Werte als NULL notiert).

Eine Importdatei besteht also aus Kommentaren (Zeilen 1-5), einer Konfiguration (Zeilen 6-8) und den eigentlichen Daten (ab Zeile 9). Für den Import werden folgende Schritte durchgeführt:

1. Der Kopfteil der Datei, in dem die Konfiguration steht, wird ausgelesen und in ein entsprechendes Konfigurationsobjekt übersetzt.

2. Je nachdem, welche Aktion gefordert ist (INSERT, UPDATE, DELETE), wird ein passender „Prozessor" erzeugt, der die Aktion durchführen kann.

Der erste Schritt wird durch SourceProcessorConfig realisiert (unterstützt von TableInfo und ColumnInfo):

[14] Siehe FuX-Infopool-Pflichtenheft, Abschnitt 2.1.1

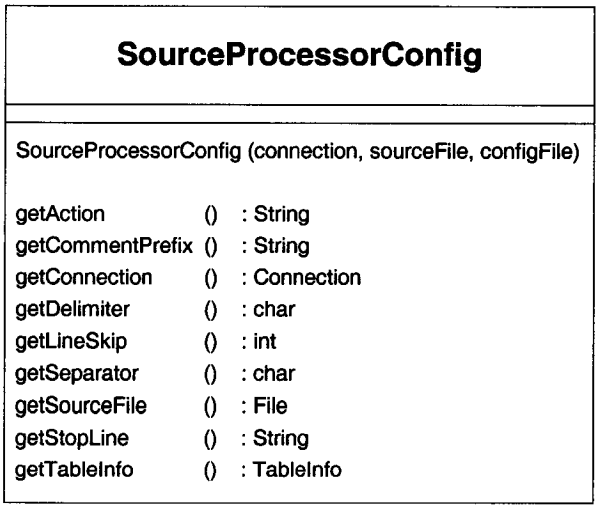


Abbildung 36: Konfigurationsklassen für einen Datenimport

Ein Exemplar von SourceProcessorConfig wird mit dem Konstruktor automatisch für eine gegebene Datenbankverbindung, eine Importdatei und ggf. eine Konfigurationsdatei erzeugt. Falls – wie im Beispiel oben – Konfiguration und Daten in einer Datei vorliegen, so müssen sourceFile und configFile identisch sein. SourceProcessorConfig liefert folgende Informationen:

getAction	durchzuführende Aktion (INSERT, UPDATE, DELETE).
getCommentPrefix	String, der einen Kommentar einleitet (z.Zt. fix "##") .
getConnection	JDBC-Verbindung zur betroffenen Datenbank.
getDelimiter	Begrenzungszeichen für Strings (z.Zt. keines bzw. 0).
getLineSkip	Anzahl der Zeilen vor den Nutzdaten (im Beispiel oben 8).
getSeparator	Trennzeichen für Attribute (z.Zt. fix " , ").
getSourceFile	Datei mit den Nutzdaten.
getStopLine	String, der eine „Stop-Zeile" einleitet, die ein Nutzdatenende vor EOF markiert.
getTableInfo	Informationen zur betroffenen Datenbanktabelle.

Mit der Hilfe von SourceProcessorConfig kann nun im zweiten Schritt ein Objekt der Klasse SourceProcessor erzeugt werden:

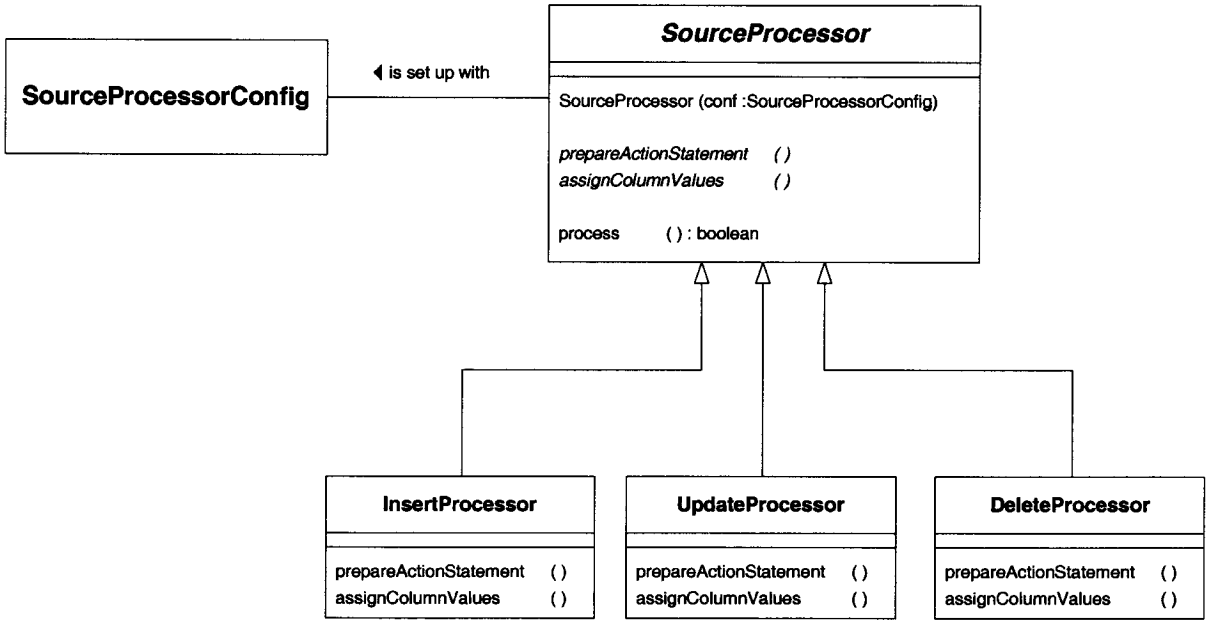


Abbildung 37: Spezifische Klassen zur Durchführung von UPDATE-, INSERT- und DELETE-Aktionen

Im Konstruktor von ImportHandler wird ein SourceProcessorConfig-Objekt aufgebaut und dann je nach geforderter Aktion entschieden, welche Art von *SourceProcessor* gebraucht wird. Jeder *SourceProcessor* ist dazu in der Lage, die zugehörige Importdatei mit process() zu verarbeiten. Dabei wird dann die konkret geforderte Aktion ausgeführt (INSERT, UPDATE, DELETE).

Tatsächlich ist aber der Kontrollfluss immer gleich und daher auch schon in *SourceProcessor* vollständig implementiert: Die Importdatei muss zeilenweise gelesen werden, bis EOF oder die spezielle Ende-Zeile erreicht ist. Für jede Zeile müssen die Spaltenwerte extrahiert und in das passende Kommando eingesetzt werden, das dann ausgeführt wird. Daher existieren die Methoden

prepareActionStatement	zum Einrichten des benötigten PreparedStatement-Objektes für ein INSERT-, UPDATE- oder DELETE-Kommando mit der benötigten Parameteranzahl.
assignColumnValues	zum Einsetzen gelesener Spaltenwerte in die vorbereiteten Parameter.

Für genau diese beiden Methoden sind Informationen über die betroffene Datenbanktabelle und ihre Spalten notwendig, die über SourceProcessorConfig.getTableInfo() geholt werden. Das resultierende Objekt ist von der Klasse TableInfo:

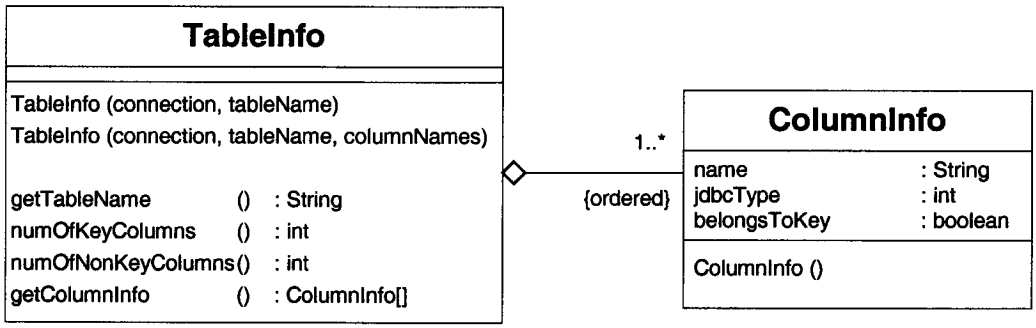


Abbildung 38: Meta-Information für eine Datenbanktabelle und ihren Spalten

Mit TableInfo und ColumnInfo lässt sich ein PreparedStatement für die Durchführung des Imports/Updates aufbauen. Hierfür werden benötigt:

- Tabellenname, Spaltennamen
- Schlüssel- und Nicht-Schlüssel-Spalten für die Zuordnung zu Parametern

INSERT : Alle Spalten enthalten die neu einzufügenden Werte.
UPDATE: Schlüssel-Spalten selektieren Zeilen, andere Spalten enthalten die neuen Werte.
DELETE : Schlüssel-Spalten selektieren.

Beim Lesen der Importdatei wird eine Zeile mit assignColumnValues () in Parameter-Werte umgesetzt. Hierfür werden benötigt:

- Schlüssel- und Nicht-Schlüssel-Spalten und deren Anzahl
- Geforderter JDBC-Typ einer Spalte, in den ein gelesener String konvertiert wird.

6.2.3.4.5.2 Überblick

Die in 6.2.3.4.4 bis 6.2.3.4.5.1 besprochenen Klassen stellen sich für den Online-Modus im Überblick wie folgt dar (für den Offline-Modus analog):

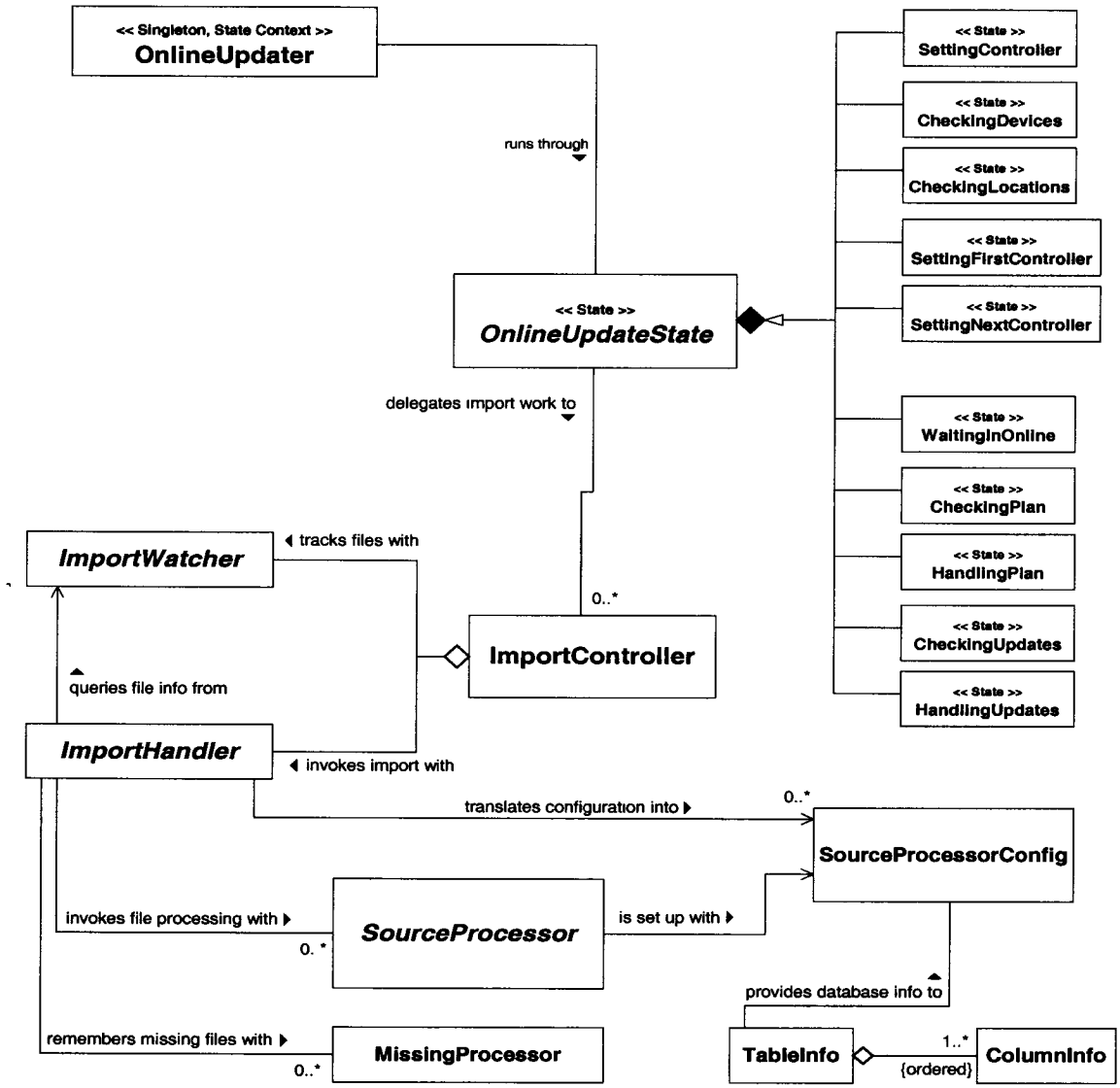


Abbildung 39: Zusammenhang der Klassen für den Datenimport (Online-Modus)

Die Hilfsklasse MissingProcessor wird dann benötigt, wenn eine Importdatei erwartet wird, die noch nicht existiert. Dies kann z.B. ein Tagesplan sein, der noch nicht versendet worden ist, obwohl der betreffende Tag bereits begonnen hat.

In diesem Fall wird ein MissingProcessor-Objekt als Platzhalter für einen *SourceProcessor* eingesetzt, wobei die betroffene Datei vermerkt ist. Trifft der *ImportHandler* bei einem Import-Zyklus auf einen MissingProcessor, wird versucht erneut, daraus einen *SourceProcessor* zu bauen. Gelingt dies (Datei existiert mittlerweile), so wird die Verarbeitung unmittelbar mit *SourceProcessor*.process() gestartet.

6.2.3.4.5.3 Anmerkungen

Die möglichen Zustände beim Datenimport wurden in den Abschnitten 6.2.3.4.1 bis 6.2.3.4.3 mit Hilfe von State-Charts beschrieben. Der Kontrollfluss wurde dann direkt über ein State-Pattern im Software-Design umgesetzt. Diese Vorgehensweise ist nicht falsch, muss jedoch wie folgt kritisch hinterfragt werden:

- Ist die Anzahl der Zustände immer gleich, d.h. wird zur Laufzeit jeder Zustand erreicht?
- Wovon hängen die Zustandsübergänge ab?
- Lässt sich der beschriebene Kontrollfluss auch anders umsetzen?

Das State-Chart in Abbildung 31 (S. 56) fasst die möglichen Zustände für alle Betriebsarten im Online-Modus zusammen. Zur Laufzeit kann es aber immer nur eine Betriebsart geben (z.B. AIH), für die sich dann die Importregeln und der resultierende Kontrollfluss vereinfachen wie beim State-Chart in Abbildung 28 (S. 54).
Außerdem hängen einige Zustandsübergänge (inkl. Startzustandsübergänge) ausschließlich davon ab, für welche Betriebsart die Anwendung gestartet wurde. Beispielsweise findet der Übergang von WaitingInOnline zu SettingNextController nur in den Betriebsarten AIH-Monitor und Mobilist statt (Bedingung [deviceType <> AIH]).
Es bietet sich an, den Kontrollfluss über Polymorphismus und einfache Konstrukte wie if...else und while umzusetzen. Die Klassenstruktur dafür könnte für den Online-Modus wie folgt aussehen (für Offline noch einmal analog):

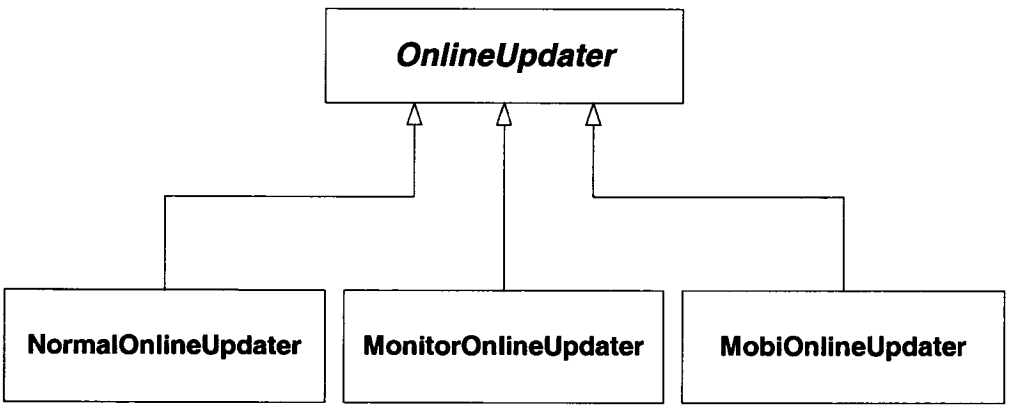


Abbildung 40: Alternative Design-Lösung für den Online-Update-Vorgang

Dies würde auch den sonstigen Design-Ansätzen für verschiedene Betriebsarten und Modi entsprechen (wie z.B. in *Schedule* → NormalSchedule, MonitorSchedule, MobiSchedule). Falls jedoch die Anzahl der möglichen Zustände und Übergänge mit neuen, spezielleren Anforderungen unabhängig von der jeweiligen Betriebsart steigt, ist der ursprüngliche Design-Ansatz über das State-Pattern vorzuziehen.

6.2.3.5 Aufräumarbeiten in Dateisystem und Datenbanken (ipodipsi.cleanup)

Im Laufe des Betriebs von IPODIPSI ist es möglich, dass sich überflüssige Dateien in den Eingangsverzeichnissen anhäufen. Dies können beispielsweise veraltete Tagesfahrpläne sein, oder auch Daten, die nicht für das Gerät bestimmt sind und durch einen Konfigurationsfehler in der Sendeschleife an der falschen Stelle stehen. Außerdem ist jede Fahrtmeldung in der Datenbank veraltet, deren Soll-Abfahrtszeit in der Vergangenheit liegt.
Um das Risiko einer Festplattenüberlaufs zu verkleinern, sind im Paket ipodipsi.cleanup Klassen definiert, die ein regelmäßiges Aufräumen im Dateneingangsverzeichnis und in den Datenbanken ermöglichen.

6.2.3.5.1 Aufräumen im Dateisystem

Hier sind zwei Fälle zu unterscheiden: Zum einen die Betriebsart AIH, bei der es ein Datenverzeichnis für das eine Gerät gibt, auf dem IPODIPSI läuft. Zum anderen die Betriebsarten AIH-Monitor und Mobilist, bei denen es mehrere Datenverzeichnisse gibt – entweder für mehrere zu beobachtende AIHs, oder für mehrere angefahrene Haltestellen.

Beispielverzeichnis Abfahrt:

```
input\infopool\export\A_0001
```

Beispielverzeichnisse Monitor:

```
input\infopool\export\M_0001
input\infopool\export\M_0002
input\infopool\export\M_0003
input\infopool\export\M_0004
...
```

Beispielverzeichnisse Fahrtzeuganzeige:

```
input\infopool\export\Haltestelle_355
input\infopool\export\Haltestelle_360
input\infopool\export\Haltestelle_6002
input\infopool\export\Haltestelle_6169
...
```

In allen Fällen gibt es von den Datenverzeichnissen abwärts folgende weitere Struktur für Importdateien[15]:

```
Tagesfahrplan\fahrtmeldung.insert

Tagesaenderungen\fahrtmeldung.insert
Tagesaenderungen\fahrtmeldung.update
Tagesaenderungen\fahrtmeldung.delete

...
Saisonfahrplan\2002-01-01\fahrtmeldung.insert
Saisonfahrplan\2002-01-02\fahrtmeldung.insert
Saisonfahrplan\2002-01-03\fahrtmeldung.insert
Saisonfahrplan\2002-01-04\fahrtmeldung.insert
Saisonfahrplan\2002-01-05\fahrtmeldung.insert
...
```

Es gibt also genau einen Tagesfahrplan mit Änderungen (Ist-Daten für einen Tag) und optional ein oder mehrere Saisonfahrpläne (Soll-Daten für jeweils einen Tag). Ein Saisonfahrplan eines Tages steht immer in einem Verzeichnis, das ein entsprechendes Datum als Namen hat, und zwar in der Form `<JJJJ-MM-TT>` (Jahr-Monat-Tag).

Unabhängig von der Betriebsart muss für das Löschen überflüssiger Dateien immer wie folgt vorgegangen werden:

- Gehe in ein Datenverzeichnis
- Lösche ab dort sämtliche Dateien, die keine gültigen Importdateien oder -verzeichnisse sind.

[15] Die Namen sind zwar in der Konfigurationsdatei von IPODIPSI änderbar, aber die Grundstruktur bleibt immer gleich.

Die Funktionalität hierfür wird in der abstrakten Klasse *ImportCleaner* zur Verfügung gestellt:

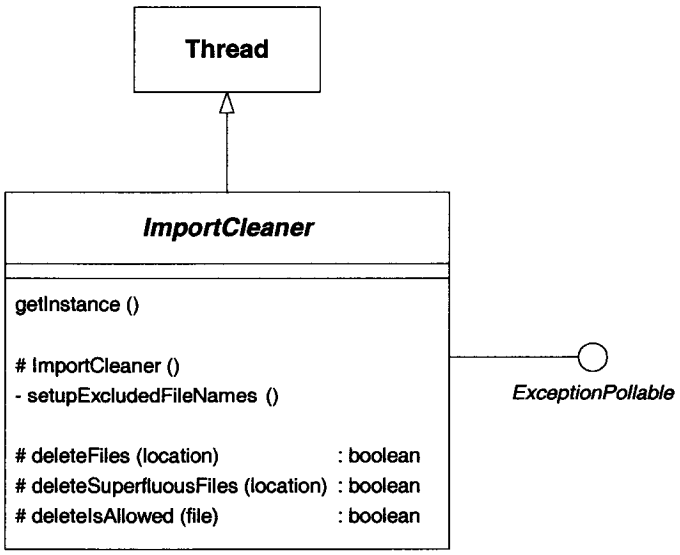


Abbildung 41: Basisfunktionalität für das Löschen in ImportCleaner

Der Konstruktor ImportCleaner(), der für abgeleitete Klassen zur Verfügung gestellt wird, nutzt die private Methode setupExcludedFileNames(), um aus der Konfigurationsdatei, mit der IPODIPSI gestartet wurde, die zu schützenden Importdateien bzw. Verzeichnisstrukturen zu ermitteln.
Als Basisfunktionalität werden außerdem deleteFiles und deleteSuperfluousFiles zur Verfügung gestellt, mit denen Dateien/Verzeichnisse rekursiv gelöscht werden können – einmal ohne weitere Bedingung, einmal nur für diejenigen Dateien/Verzeichnisse, die überflüssig sind. Die Entscheidung, ob eine bestimmte Datei oder ein Verzeichnis gelöscht werden darf oder nicht, ist in deleteIsAllowed implementiert.
In den konkreten Realisierungen von *ImportCleaner* werden im Kontrollfluss (run()-Methode) ein oder mehrere Datenverzeichnisse berücksichtigt. NormalImportCleaner ist für ein einziges Verzeichnis zuständig (AIH), MultiImportCleaner für mehrere (AIH-Monitor, Mobilist):

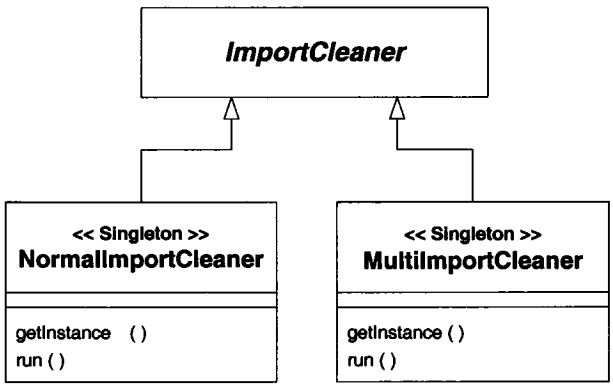


Abbildung 42: Realisierung unterschiedlicher Kontrollflüsse beim Löschen

6.2.3.5.2 Aufräumen in den Datenbanken

Das Aufräumen in online- und offline-Datenbank ist mit der Klasse DBCleaner realisiert und gestaltet sich vergleichsweise einfach: Es müssen lediglich diejenigen Fahrtmeldungen gelöscht werden, deren Ist-Abfahrtszeit in der Vergangenheit liegt. Dies ist mit einem einfachen, vorbereiteten SQL-Statement möglich, das im Konstruktor definiert und in run() regelmäßig an online- und offline-Datenbank geschickt wird:

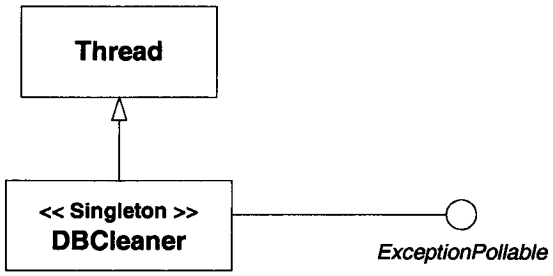


Abbildung 43: Regelmäßiges Aufräumen in der Datenbank mit einem einfachen Thread

DBCleaner erledigt wie ImportCleaner seine Aufgaben nebenläufig und ist daher als Thread realisiert. Beide Klassen implementieren das Interface ExceptionPollable, damit unerwartete Exceptions geholt werden können.

6.2.3.6 Globale Ereignisbehandlung und Ablaufsteuerung (ipodipsi)

6.2.3.6.1 Reaktion auf Ereignisse

IPODIPSI muss dazu in der Lage sein, wie folgt Ereignisse zu erfassen und darauf zu reagieren:

Betriebsarten	Modus	Ereignis	Reaktion	Mitteilung an
Alle	online	Keine Fahrtmeldungen mehr in der online-Datenbank.	Umschalten in offline-Modus	OutputCreator, OfflineUpdater
	offline	Fahrtmeldungen in der online-Datenbank.	Umschalten in online-Modus	OutputCreator, OfflineUpdater
Monitor	online, offline	Fahrtmeldungen für neues Gerät angefordert (neue Geräte-ID)	Neue Geräte-ID setzen	OutputCreator
Fahrtzeug	online, offline	Fahrtmeldungen für neue Haltestelle angefordert (neue Haltestelle-ID)	Neue Haltestellen-ID setzen	OutputCreator

Es kann also notwendig sein, den Modus zu wechseln (online/offline), eine angeforderte Geräte-ID oder eine neue Haltestellen-ID zu setzen. In allen Fällen müssen andere Programmteile eine Mitteilung über das Ereignis erhalten: OutputCreator muss sofort eine aktualisierte Darstellung liefern (siehe 6.2.3.3.3), OfflineUpdater muss die Wartezeit zwischen Update-Zyklen ändern (siehe 6.2.3.4.4).

6.2.3.6.1.1 Ereigniserfassung

Für die Ereigniserfassung existieren spezielle Klassen, die Threads sind[16] und jeweils eine spezifische Überwachungsfunktion erfüllen. Werden sie gestartet, so führen sie in ihren run()-Methoden zunächst einmal ihre spezifischen Prüfungen durch:

OnOffController prüft regelmäßig, ob in den online- oder offline-Modus gewechselt werden muss.

MonitorController prüft regelmäßig, ob vom Monitor-Frontend die Fahrtmeldungen für ein neues Gerät angefordert werden (neue Datei mit einer Geräte-ID).

MobiController prüft regelmäßig, ob die aktuelle Haltestelle verlassen und die nächste angefahren wird (neue Haltestellen-ID vom TramPos-Socket-Server).

[16] Sie implementieren wieder alle das Interface ExceptionPollable, siehe die Erläuterungen zu Interfaces in 6.2.3.3.3.

OnOffController ist eine abstrakte Klasse und wie folgt mit ihren Unterklassen definiert:

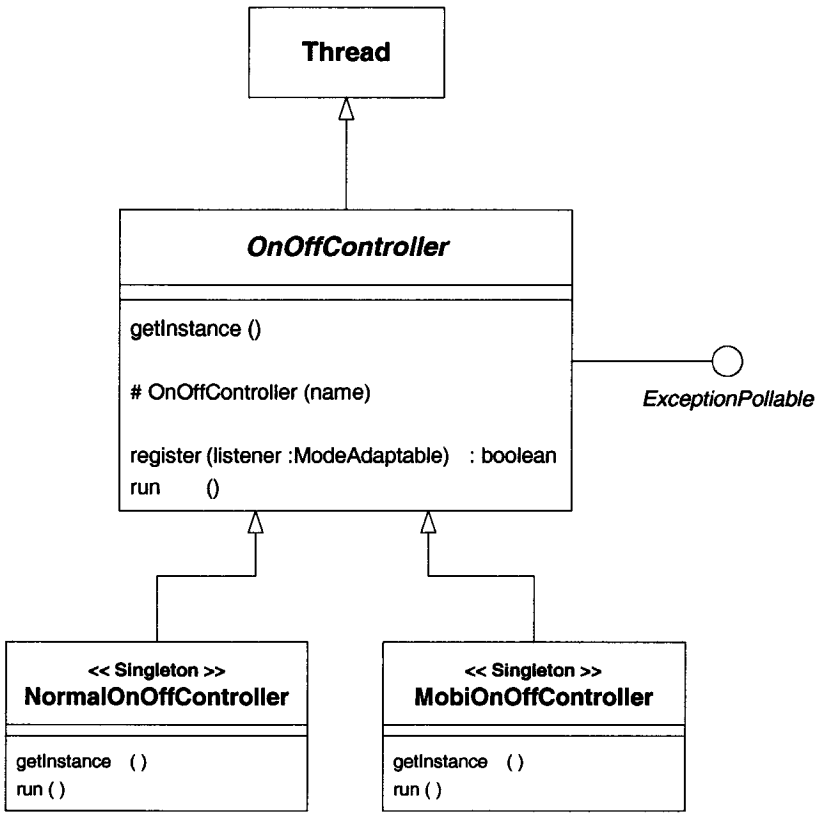


Abbildung 44: Überwachung für online/offline-Modus durch spezifische OnOffController

Die Unterklassen implementieren spezifische Abfragen mit verschiedenen Parametern für die Betriebsarten Abfahrt, Monitor (NormalOnOffController) und Fahrzeuganzeige(MobiOnOffController). Diejenigen Objekte, die durch *OnOffController* eine Mitteilung erhalten sollen, dass ein Wechsel von online nach offline erfolgt ist, können sich hierfür mit *OnOffController*.register anmelden. Dazu müssen ihre Klassen jedoch das Interface *ModeAdaptable* implementieren:

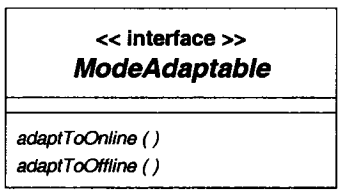


Abbildung 45: Registrierbarkeit für online/offline-Wechsel über das Interface ModeAdaptable

Mit Hilfe dieses Interfaces können im Prinzip beliebig viele Objekte unterschiedlicher Klassen für einen online/offline-Wechsel registriert werden und auf einfache Weise eine Mitteilung über den Wechsel erhalten. *OnOffController* durchläuft hierzu lediglich eine Liste der registrierten *ModeAdaptable*-Objekte und schickt je nach neuem Modus die Nachricht adaptToOnline() oder adaptToOffline().

Für die Überwachungsklasse MonitorController kommt dasselbe Prinzip zur Anwendung – hier sind jedoch keine spezifischen Unterklassen wie bei *OnOffController* notwendig:

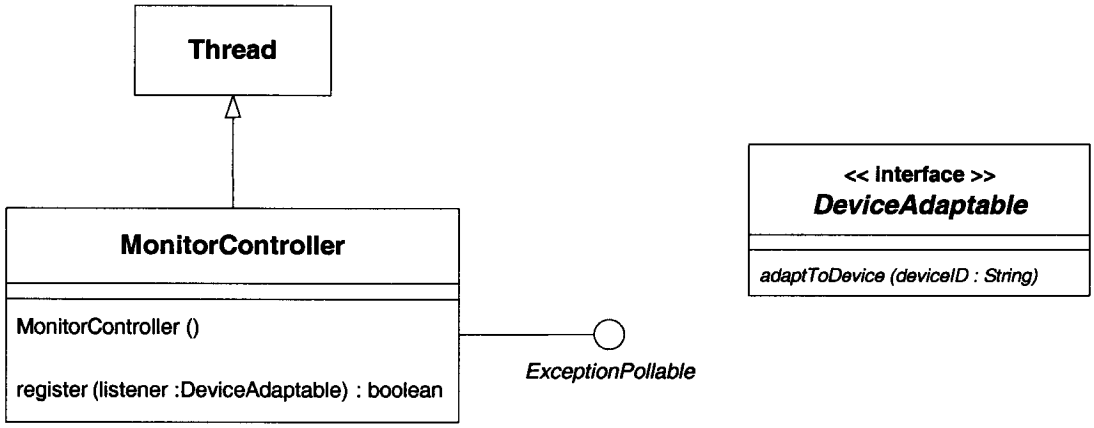


Abbildung 46: Registrierbarkeit für eine neue Geräte-ID über das Interface DeviceAdaptable

Für Objekte, die eine Mitteilung über eine neue Geräte-ID erhalten sollen, müssen sich hier also als *DeviceAdaptable* registrieren.

Die Überwachung auf neue Haltestellen mit Hilfe von MobiController ist analog realisiert, hier mit einem entsprechenden Interface *LocationAdaptable* :

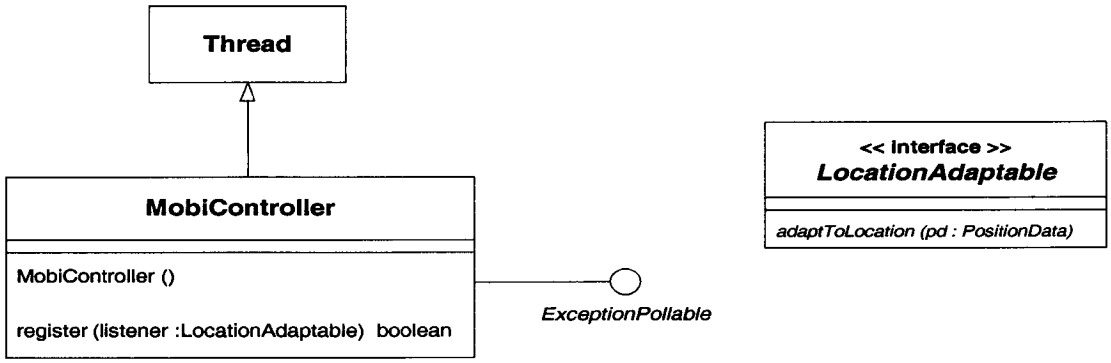


Abbildung 47: Registrierbarkeit für eine neue Haltestellen-ID über das Interface LocationAdaptable

Alle Controller-Klassen für Ereignisse sind also Threads, die bestimmte Objekte für das von ihnen überwachte Ereignis registrieren können.

6.2.3.6.1.2 Unterstützende Klassen für MobiController

Bei MobiController wird eine neue Haltestellen-ID über die die extern laufende Anwendung TramPos (trampos.exe) festgestellt. Sie stellt einen Socket-Server zur Verfügung, über den man neue Positionsdaten abfragen kann. TramPos liefert jedoch wesentlich mehr als nur eine Haltestellen-ID, und daher ist hierfür im Paket ipodipsi.data eine eigene Datenklasse PositionData definiert. Sie wird unterstützt vom Interface PositionState, das einen Aufzählungstyp für mögliche Positionszustände realisiert:

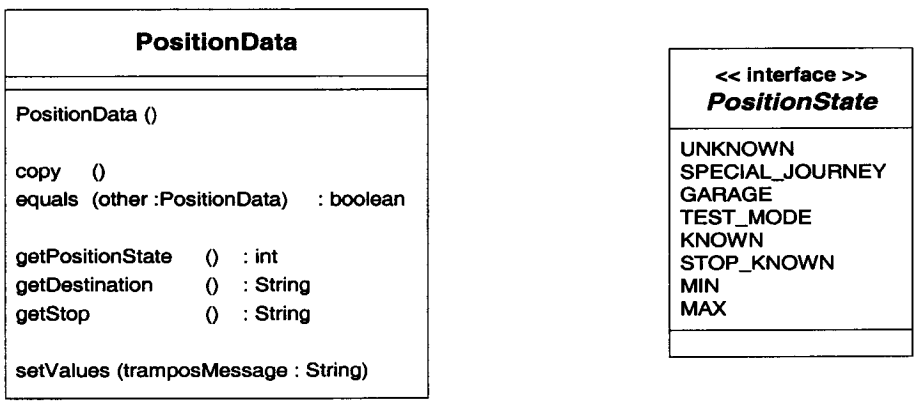


Abbildung 48: Anbindung an die TramPos-Daten mit Hilfe von PositionData

Sie entspricht in großen Teilen der C++-Klasse PosData, die von TramPos gesendet wird. Näheres findet sich in der TramPos-Dokumentation.

6.2.3.6.1.3 Anmerkungen

Für die Erfassung von Ereignissen und Mitteilungen an betroffene Objekte wurde der folgende Design-Ansatz gewählt, der auch in anderen Anwendungen verwenden finden kann:

- Es gibt für die Erfassung eines Ereignisses eine spezifische Controller-Klasse. Sie verlangt, dass sich ein Objekt, das an diesem Ereignis interessiert ist, bei ihr anmeldet (register-Methode). Dazu muss die Klasse dieses Objektes ein Interface (*XAdaptable*) implementieren, mit dem man ihr eine Nachricht schicken kann, dass ein Ereignis X stattgefunden hat (adaptToX-Methode).

- Unabhängig von seiner Funktion ist das Objekt also als „anpassbar an ein bestimmtes Ereignis" gekennzeichnet, egal von welcher Klasse es nun genau ist (Geheimnisprinzip). Mehr ist bei der Controller-Klasse nicht bekannt.

Dies resultiert in einem einfachen Kontrollfluss für das Controller-Objekt: Es muss lediglich eine Liste aller registrierten Objekte durchlaufen und jedem von ihnen die Nachricht adaptToX schicken.

6.2.3.6.2 Reaktion auf unerwartete Exceptions

Zur Laufzeit können in Java eine ganze Reihe von Exceptions auftreten, die sich nicht unmittelbar aus dem Code selbst ergeben (siehe java.lang.RuntimeException). Darüber hinaus können weitere Exceptions nicht sinnvoll handhabbar sein, so dass sie ein (sauberes) Beenden der Anwendung erzwingen. In den vorhergehenden Abschnitten wurde deutlich, dass sich die verschiedenen Aufgaben in IPODIPSI auf voneinander unabhängige Threads verteilen, die durch die Hauptklasse ipodipsi.Starter erzeugt und gestartet werden. Tritt in einem Thread eine RuntimeException auf, so geht diese im Normalfall für die Hauptklasse verloren. Sie kann nicht automatisch „nach oben weitergereicht" werden, weil es keinen eindeutigen Rücksprungpunkt gibt (Problem der Asynchronität).

Für das Auffangen von Exceptions aus Threads gibt es daher einen speziellen Mechanismus. Jeder Thread muss das Interface ExceptionPollable implementieren:

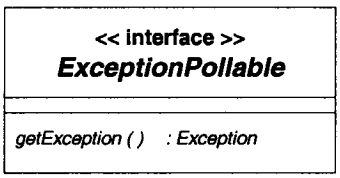


Abbildung 49: Kennzeichnung von Threads, die sich unerwartete Exceptions merken

Ist in der Hauptklasse ipodipsi.Starter ein Thread erfolgreich erzeugt und gestartet, so wird er als ExceptionPollable registriert. Sobald alle Threads laufen, werden von ihnen regelmäßig die Exceptions abgefragt (Endlosschleife mit einstellbarem Wartezyklus, typisch 5s). Gibt es eine Exception, die nicht null ist, so wird die Anwendung beendet. Dies geschieht wie folgt:

- Die Exception wird ins Logfile geschrieben (Klasse, Methode, Exception-Nachricht).
- Das OutputCreator-Objekt wird dazu veranlasst, die Default-Seite darzustellen.
- Es wird eine Meldung auf stdout geschrieben.
- IPODIPSI wird mit einem Fehlercode verlassen.

6.2.3.6.3 Globale Ablaufsteuerung

Beim Start von IPODIPSI werden zunächst die übergebenen Argumente ausgewertet, die Anwendung konfiguriert und das anwendungsweite Logging vorbereitet. Je nach gewünschtem Modus werden dann die erforderlichen Prozesse bzw. Threads gestartet. Während IPODIPSI läuft, werden Ereignisse überwacht wie z.B. unerwartete Exceptions oder Änderungen von Geräte-IDs.

IPODIPSI wird wie folgt von der Kommandozeile aus aufgerufen[17]:

```
java -jar ipodipsi.jar -c <configFile> -m <mode> -f <fileSystemType>
```

`<configFile>`	ist die Konfigurationsdatei für die Anwendung (→ ApplicationProps)
`<mode>`	erlaubt dbsetup oder run (Datenbankeinrichtung oder normaler Start)
`<fileSystemType>`	erlaubt unix oder windows explizit zu setzen

Die Hauptklasse Starter liest zunächst die in der Kommandozeile angegebene Konfigurationsdatei (z.B. aih.cfg). Die enthaltenen Werte werden in einem ApplicationProps-Objekt abgelegt. Dann wird ein Logger-Objekt aufgebaut und global zur Verfügung gestellt. Die Utility-Klassen ApplicationProps und Logger sind im Abschnitt zum Paket ipodipsi.util beschrieben. Die Hilfsklasse APP ermöglicht einen einfachen Zugriff auf diese Klassen (APP.props, APP.logger), ohne dass eine Initialisierung notwendig wäre. Dies dient jedoch lediglich als Schreibabkürzung im Quelltext.

Je nach gewähltem Modus (`<mode>`) schließen sich nun unterschiedliche Abläufe an.

[17] Für den vereinfachten Aufruf existieren Batch-Dateien für die jeweiligen Betriebsarten (aih.bat, monitor.bat, mobilist.bat).

6.2.3.6.4 Start für die Datenbankeinrichtung (dbsetup)

Für die Einrichtung der Datenbanken (online+offline) geschieht folgendes:

- Es wird eine Datenbankverbindung geholt (DBInterface).
- Damit werden die benötigten Tabellen erzeugt und mit Initialdaten gefüllt (DBCreator, SQLReader). Dies ist im Abschnitt 6.2.3.1 beschrieben.
- IPODIPSI wird beendet.

6.2.3.6.5 Normaler Start (run)

Bei einem normalen Start von IPODIPSI geschieht folgendes:

- Es wird eine Liste für die zu überwachenden Threads (*ExceptionPollable*-Objekte) angelegt.
- Je ein Objekt der folgenden Klassen wird erzeugt, gestartet und als ExceptionPollable registriert:
  - ImportCleaner
  - DBCleaner
  - OnlineUpdater
  - OfflineUpdater
  - OutputCreator
  - OnOffController
- Falls die gewünschte Betriebsart Monitor ist, wird ein MonitorController erzeugt, gestartet und als ExceptionPollable registriert.
- Falls die gewünschte Betriebsart Fahrzeuganzeige ist, wird ein MobiController erzeugt, gestartet und als ExceptionPollable registriert.
- In einer Endlosschleife werden alle Threads auf unerwartete Exceptions geprüft.

6.2.3.6.6 Zusammenfassung

Die Beziehung der Hauptklasse Starter zu den anderen Klassen aus ipodipsi stellt sich im Überblick wie folgt dar:

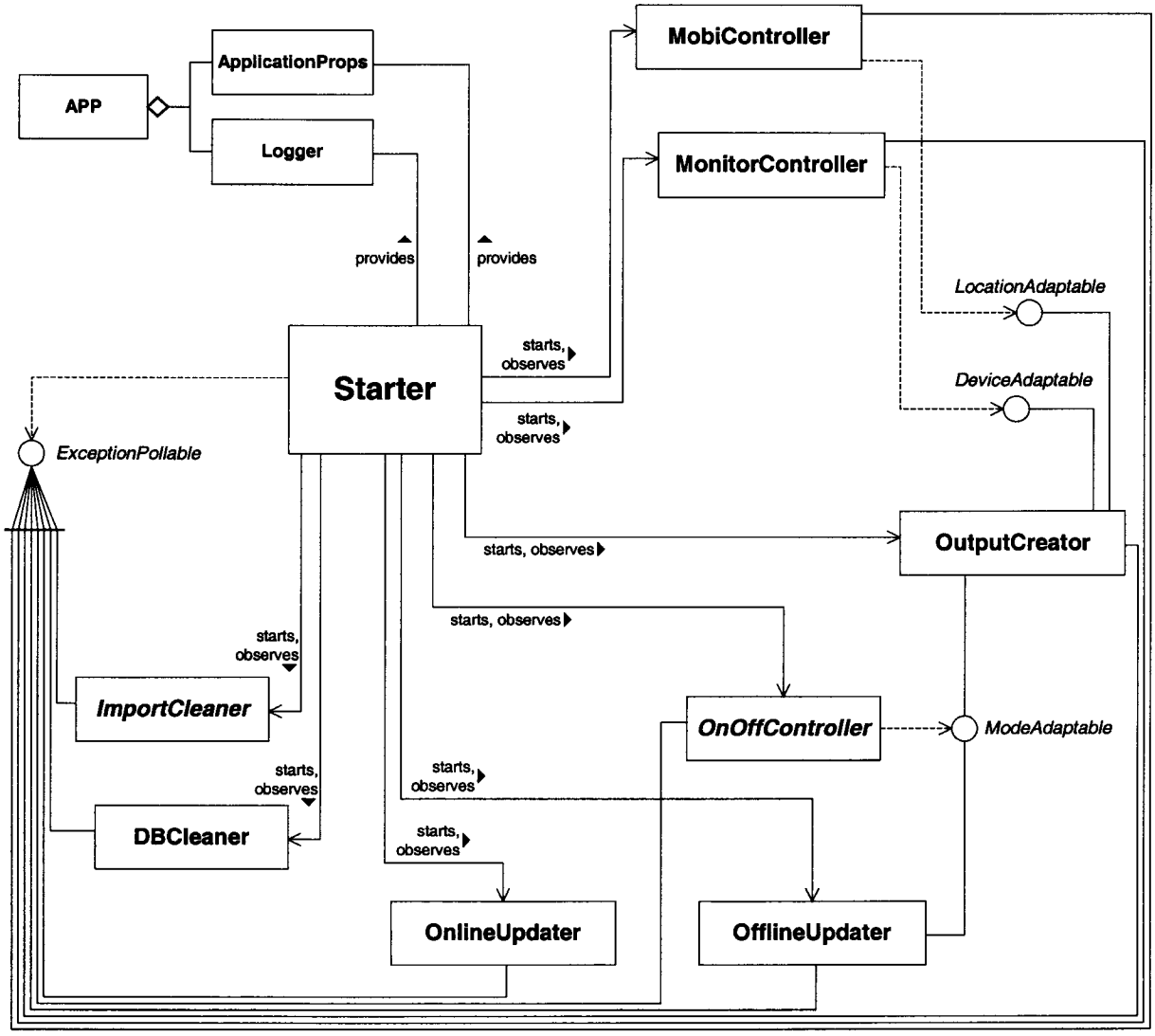


Abbildung 50: Hauptklasse ipodipsi.Starter mit assoziierten Klassen

6.2.3.7 Paket ipodipsi.util

In IPODIPSI kommen eine Reihe von Klassen zur Anwendung, die allgemeine Hilfsaufgaben erfüllen und daher im Paket ipodipsi.util zusammengefasst sind. Sie sind in der folgenden Übersicht kurz beschrieben.

Klasse	Aufgabe	Verwendung in
ApplicationProps	Fasst alle einstellbaren Eigenschaften der Anwendung zusammen, die zu Beginn aus einer Konfigurationsdatei gelesen werden. Verwendet java.util.Properties, lässt aber nicht das Setzen/Lesen von Werten zu, die nicht in der Konfigurationsdatei definiert sind.	ipodipsi.*
Logger	Ermöglicht die Ausgabe von Log-Meldungen in ASCII-Dateien, die mit den Zahlen eines Monats durchnummeriert werden (1.log, 2.log, 3.log, ...) und protokolliert so immer das Verhalten von IPODIPSI in den maximal 31 letzten Tagen (danach Überschreiben, Neubeginn bei 1.log). Kann entweder einen String oder einen String mit einem Parameter ausgeben, wobei dann im String der Platzhalter „$" markiert, wo der Parameterwert stehen soll.	ipodipsi.*
DBInterface	Ermöglicht eine einheitliche Anbindung an online- und offline-Datenbank (siehe Abschnitt 6.2.3.1.1)	dbsetup dbimport cleanup data output ipodipsi
StringDelimiter	Ermöglicht – analog zu java.util.StringTokenizer – ein Parsing eines Strings, jedoch mit Strings als Trennern, nicht nur mit einzelnen Zeichen (char).	output.HTMLOutputPattern
StringSplitter	Teilt einen String in ein Array aus Strings auf.	dbimport.SourceProcessor dbimport.SourceProcessorConfig
WatchedFile	Ermöglicht das Überwachen einer Datei auf Änderungen. Abgeleitet aus java.io.File mit zusätzlicher Methode WatchedFile.hasChanged().	dbimport.ImportWatcher dbimport.OnlineImportWatcher dbimport.OfflineImportWatcher dbimport.ImportHandler
FileStringBuffer	Puffert den Inhalt einer Datei in einem String.	output.OutputCreator output.HTMLOutputPattern
FileFilterName	java.io.FileNameFilter, der nur die Dateien akzeptiert, die einen bestimmten Namen haben.	cleanup.NormalImportCleaner
FileFilterNoName	java.io.FileNameFilter, der nur die Dateien akzeptiert, die nicht einen bestimmten Namen haben.	cleanup.NormalImportCleaner
FileFilterPrefix	java.io.FileNameFilter, der nur die Dateien akzeptiert, deren Name mit einem bestimmten Präfix beginnt.	cleanup.MultiImportCleaner
FileFilterNoPrefix	java.io.FileNameFilter, der nur die Dateien akzeptiert, deren Name nicht mit einem bestimmten Präfix beginnt.	cleanup.MultiImportCleaner
FileFilterSuffix	java.io.FileNameFilter, der nur die Dateien akzeptiert, deren Namen mit einem bestimmten Suffix enden.	dbsetup.DBCreator

7. Nutzen, Verwertbarkeit der Ergebnisse

Das technische System Endgeräte ist ausbaufähig und erweiterbar. Durch die Verwendung von Normen und Standards ist das Softwaresystem ein offenes System. Hinsichtlich der Anwendungsfunktionen sind Reserven in der Hardware vorhanden.

Dies gilt insbesondere für die Endgeräte im stationären Einsatz hinsichtlich:
- Integration zusätzlicher Displays
- Präsentation von Zusatzinformation (z. B. Werbung)
- Rückkanalfähigkeit (z. B. Notruf, Videoüberwachung).

Insgesamt ist es möglich, den Verbund mehrerer Verkehrsbetriebe herzustellen und eine gemeinsame integrierte Datenbasis und gemeinsam nutzbare Übertragungstechnik zu implementieren. Eine Integration weiterer, benachbarter Verkehrsbetriebe ist systemtechnisch möglich und wünschenswert.

Es ist festzustellen, dass ein begrenzter Markt für mobile und stationäre Fahrgast-informationssysteme entsteht. Entsprechende Aktivitäten im In- und Ausland sind angelaufen.
Ebenso sehen wir Chancen und Gelegenheiten, Produkte und Erfahrungen in nationale und internationale Forschungsvorhaben einzubringen.

Insgesamt ist ein hohes Maß an Portierbarkeit und Übertragbarkeit in andere Städte und Ballungsgebiete vorhanden.

Weiterhin sind mit der Beendigung von MOBILIST-ANIS die Grundlagen für ein übergreifendes Anschlussinformationssystem sowie ein weitgehend automatisches verkehrsträgerübergreifendes Anschlusssicherungs-System vorhanden.
Ebenso zeigt sich die Notwendigkeit und Chance, die vielen entstandenen und vorhandenen Systeme datentechnisch und im Zusammenwirken zu integrieren. Eine wirkliche Integration der IV- und ÖV-Systeme ist noch zu leisten.

Es wurden keine der Arbeiten als Erfindungen gekennzeichnet oder Schutzrechtsanmeldungen gemacht oder in Anspruch genommen oder Schutzrechte erteilt oder verwertet. Dennoch ist natürlich erhebliches Know-how entstanden, aus dem künftig ggf. im Rahmen weiterer Projekte Geschmacksmuster oder ggf. auch noch Erfindungsmeldungen entstehen könnten.

8. Erfolgte oder geplante Veröffentlichungen der Ergebnisse

Von den Arbeiten in MOBILIST werden die Öffentlichkeit und die Verantwortlichen überwiegend durch den Projektkoordinator und mit einer eigenen MOBILIST-Schriftenreihe (mobilist news letter) kontinuierlich informiert.
Veröffentlichungen durch die MU sind nicht erfolgt.

Erfolgskontrollbericht zum Projekt

Mobilität im Ballungsraum Stuttgart
MOBILIST

- Arbeitspaket C3 -
Anschlussinformationssystem ANIS
Projektteil Bosch

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie gefördert
Förderkennzeichen: 19B0009

1 Beitrag des Ergebnisses zu förderpolitischen Zielen

Das Forschungsprojekt MOBILIST, Arbeitspaket C3, Anschlussinformationssystem ANIS (nachfolgend: MOBILIST-ANIS) wird vom BMBF als Verbund- und Umsetzungsvorhaben gefördert, das von der Robert Bosch Multimedia-Systeme GmbH & Co KG (nachfolgend: MU) und den MOBILIST-ANIS-Partnern bearbeitet wurde.

Die Ergebnisse von MOBILIST-ANIS tragen zu den generellen Zielen der Attraktivitätssteigerung des öffentlichen Nahverkehrs sowie der Mobilität in Ballungsräumen bei.

Die Versorgung der Bahnen und Busse sowie der Haltestellen in Stuttgart und in der Region mit Fahrgastinformation auf der Grundlage der Übertragungstechnik DAB/DMB und den darauf basierenden Infrastruktur- und Endgeräte-Produkten eröffnet neue Wege und neue Qualitäten für eine kunden- und marktgerechte Information der Fahrgäste.

DAB (Digital Audio Broadcasting) ist der europäische Standard für die beschlossene Einführung des digitalen Rundfunks und mit der multimedialen Erweiterung DMB (Digital Multimedia Broadcasting) in hervorragender Weise für den Einsatz in mobilen und stationären Fahrgastinformationssystemen geeignet.
Die breitbandige Übertragungstechnologie erlaubt es, neben statischen und dynamischen Fahrplaninformationen multimediale Zusatzinformationen (z. B. Werbung) sowie aktuelle Informationen (z. B. aus Politik und Wirtschaft) in Bahnen/Busse und an Haltestellen zu übertragen und verzugslos zu präsentieren.

MOBILIST-ANIS ist daher gleichzeitig der Einstieg in die Ausstattung der Fahrzeuge und Haltestellen der Verkehrsbetriebe in Stuttgart und in der Region mit Geräten für die streckenbezogene und flächendeckende Bereitstellung dynamischer Anschlussinformation und Zusatzinformation. Die weitere Ausstattung von Fahrzeugen und Haltestellen über die MOBILIST-ANIS-Demonstratoren hinaus ist vorgesehen.

2 Wissenschaftlicher und technischer Erfolg, erreichte Nebenergebnisse, gesammelte Erfahrungen

2.1 Projektpartner

MOBILIST-ANIS wurde von der MU und u. a. den folgenden Partnern gemeinsam und partnerschaftlich auf der Grundlage eines Kooperationsvertrages bearbeitet:

- Verkehrs- und Tarifverbund Region Stuttgart GmbH
- Stuttgarter Straßenbahnen AG
- Daimler Chrysler AG
- DB Regio AG
- Caatoosee AG.

2.2 Technische Aufgabenstellung/Lösung

Neben der Eingangsfestlegung zur Nutzung des Datenübertragungsdienstes DAB/DMB waren zwei Festlegungen/Ergebnisse bestimmend für die Spezifikations- und Entwicklungsarbeiten:
Die Schnittstelle zwischen dem C3 Datenserver und dem ANIS-Infopool basiert auf der:

Integrationsschnittstelle für betriebsübergreifende Anschlusssicherung (Standardschnittstelle für betriebsübergreifende Anschlusssicherung, Entwurf Dezember 2000, VDV-Schriften 453 12/00)

Die darauf aufbauenden und für MOBILIST-ANIS gültigen Festlegungen sind festgeschrieben im Dokument

„Feinspezifikation Schnittstellen C3-Datenserver inkl. Anhang A – Anhang E Konsolidierte Fassung vom 10.05.01

Die Layouts der Ankunfts- und Abfahrtsanzeiger sind ebenfalls in diesen Grundlagen spezifiziert.
Die Verbindlichkeit der Dokumente wurde am 10.05.2001 in der Projektsitzung MOBILIST-ANIS von den Beteiligten beschlossen.

Das Gesamtprojekt umfasst die Entwicklung und Inbetriebnahme eines Informations- und Kommunikationssystems zur Versorgung der Fahrgäste des öffentlichen Nahverkehrs In Bahnen und an Haltestellen mit aktueller und dynamischer Fahrgastinformation sowie beliebiger Zusatzinformation.
Das Gesamtsystem MOBILIST-ANIS besteht aus den Teilsystemen:

- INFRASTRUKTUR
- ENDGERÄTE.

Aufgaben der Infrastrukturbausteine sind:

- **Baustein RBL der Verkehrsunternehmen:** *Bereitstellung der Fahrplandaten*
- **Baustein Fahrplanserver:** *Endgegennahme, Speicherung und Aufbereitung der Fahrplandaten der angeschlossenen Verkehrsbetriebe sowie die Versorgung des DAB/DMB-Datenstudios mit den aufbereiteten Fahrplandaten zur Aussendung über DAB/DMB. Für MOBILIST-ANIS wurde eine Aufteilung in die Komponenten C3-Datenserver und ANIS-Infopool vorgenommen.*

- **Baustein Content Provider:** *Sammlung und Aufbereitung der redaktionellen Beiträge (Unterhaltung, Werbung)*
- **Baustein Datenstudio:** *Entgegennahme, Speicherung und DAB/DMB-gerechte Aufbereitung der Sendebeiträge des Content Providers und der vom Fahrplanserver bereitgestellten Fahrplandaten sowie die Zuführung dieser Daten zur Sendeinfrastruktur. Von dort erfolgt im L-Band die Versorgung der mobilen und stationären Endgeräte über DAB/DMB.*
- **Baustein Sendenetz:** *Verteilung aller Daten zwischen den Senderstandorten und Ausstrahlung über DAB/DMB.*

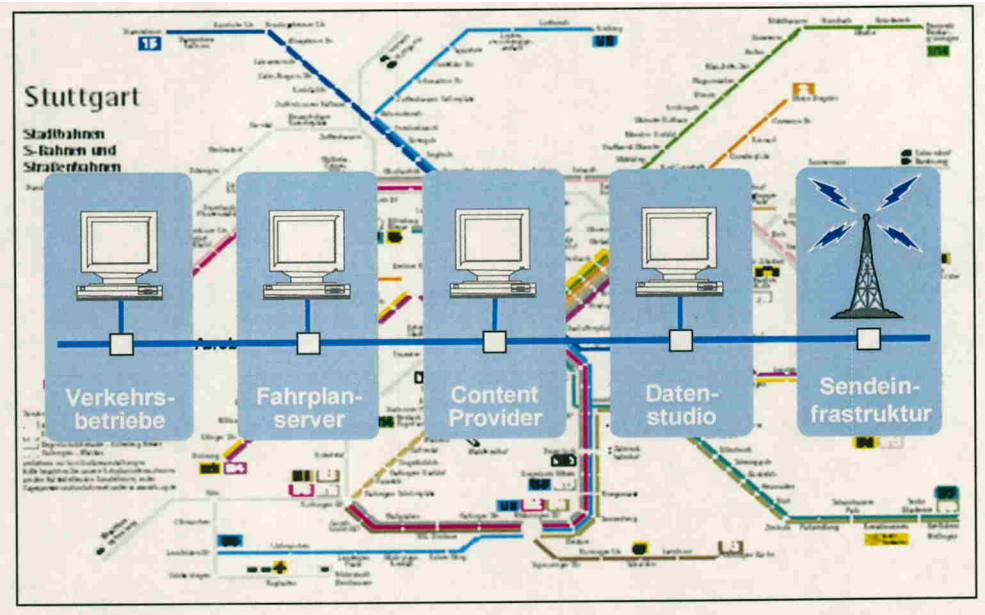


Abbildung 1: Systembausteine Infrastruktur

Aufgaben der Endgeräte (Mobile Endgeräte in den Fahrzeugen, Stationäre Endgeräte an den Haltestellen) sind der Empfang, die Aufbereitung und Präsentation der über DAB/DMB zugeführten Fahrplan- und Zusatzinformationen.

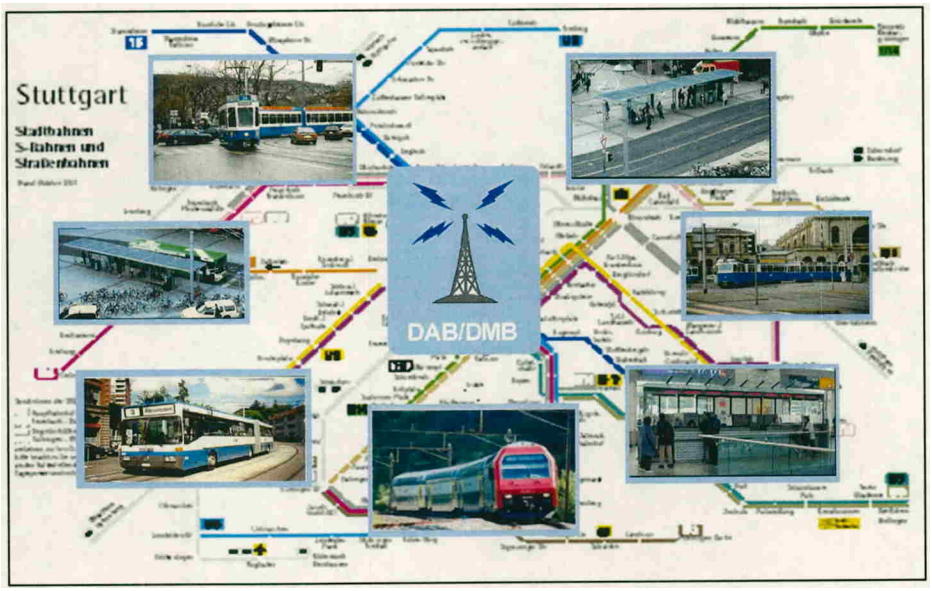


Abbildung 2: Endgeräte für Bahnen/Busse und Haltestellen/Infopunkte

Abbildung 3: Displays und Anzeigen im Fahrzeug (Bsp.)

Aufgabenstellung der MU war die Spezifikation, Entwicklung und Lieferung einer Anzahl von Endgeräten für Schienenfahrzeuge der SSB AG und der DB Regio AG sowie von Demonstratoren zur Erprobung der Haltestellenausstattung.

2.3 Erzielte Ergebnisse

Eine ausführliche Darstellung der DAB/DMB-gestützten Endgeräte für den mobilen und stationären Einsatz hinsichtlich der Geräte- und Softwarefunktionen ist im Abschlussbericht MOBILIST. Pkt. 6. enthalten.

3 Ausbaufähigkeit/Erweiterbarkeit

Das technische System Endgeräte ist ausbaufähig und erweiterbar. Durch die Verwendung von Normen und Standards ist das Softwaresystem ein offenes System. Hinsichtlich der Anwendungsfunktionen sind Reserven in der Hardware vorhanden.

Dies gilt insbesondere für die Endgeräte im stationären Einsatz hinsichtlich:

- Integration zusätzlicher Displays
- Präsentation von Zusatzinformation (z. B. Werbung)
- Rückkanalfähigkeit (z. B. Notruf, Videoüberwachung).

Insgesamt ist es möglich, den Verbund mehrerer Verkehrsbetriebe herzustellen und eine gemeinsame integrierte Datenbasis und gemeinsam nutzbare Übertragungstechnik zu implementieren. Eine Integration weiterer, benachbarter Verkehrsbetriebe ist systemtechnisch möglich und wünschenswert.

4 Einhaltung des Finanzierungs- und Zeitplans

Unter Berücksichtigung der Terminfestlegungen in den Zuwendungsbescheiden erfolgte gemeinsam und abgestimmt die detaillierte Planung der Arbeitspakete, Bearbeiter und Termine. Beschlüsse wurden durch Arbeitsgruppen vorbereitet und in den monatlichen Projektsitzungen unter der Leitung des VVS und des Projektkoordinators entschieden. Dabei wurden die Meilensteine des Gesamtvorhabens MOBILIST beachtet.

MU hat die Entwicklungsarbeiten bereits Anfang Februar 2002 abgeschlossen und die vorzeitige Beendigung des Projektes zum 12. 02. 2002 beantragt.

5 Verwertung der Ergebnisse

Es ist festzustellen, dass ein begrenzter Markt für mobile und stationäre Fahrgast-informationssysteme entsteht. Entsprechende Aktivitäten einer Reihe von Firmen sind im In- und Ausland angelaufen.
Ebenso sehen wir Chancen und Gelegenheiten, Produkte und Erfahrungen in nationale und internationale Forschungsvorhaben einzubringen. Zu nennen sind z. B. die diversen anderen Projekte im Rahmen des Programms „Mobilität in Ballungsräumen".
Insgesamt ist ein hohes Maß an Portierbarkeit und Übertragbarkeit in andere Städte und Ballungsgebiete vorhanden.

6 Wissenschaftl. und wirtschaftl. Anschlussfähigkeit für die nächsten Schritte

Mit MOBILIST-ANIS die Grundlagen für ein übergreifendes Anschlussinformationssystem sowie ein weitgehend automatisches verkehrsträgerübergreifendes Anschlusssicherungs-System vorhanden.

Ebenso zeigt sich die Notwendigkeit und Chance, die vielen in MOBILIST entstandenen und vorhandenen Systeme datentechnisch und im Zusammenwirken zu integrieren. Eine wirkliche Integration der IV- und ÖV-Systeme ist noch zu leisten.

7 Arbeiten, die zu keiner Lösung geführt haben

- entfällt -

8 Gemachte oder in Anspruch genommene Erfindungen/Schutzrechtsanmeldungen, erteilte Schutzrechte sowie deren Verwertung

Es wurden keine der Arbeiten als Erfindungen gekennzeichnet oder Schutzrechtsanmeldungen gemacht oder in Anspruch genommen oder Schutzrechte erteilt oder verwertet. Dennoch ist natürlich erhebliches Know-how entstanden, aus dem künftig ggf. im Rahmen weiterer Projekte Geschmacksmuster oder ggf. auch noch Erfindungsmeldungen entstehen könnten.

9 Präsentationsmöglichkeiten

Durch Präsentationen, Vorträge und multimediale Präsentationsmittel.

Berichtsblatt

1. ISBN oder ISNN	2. Berichtsart Schlussbericht

3a. Titel des Berichts
MOBILIST (Mobilität im Ballungsraum Stuttgart), Arbeitspaket C3-, Anschlussinformationssystem ANIS, Projektteil Bosch
Schlussbereicht über die Arbeiten im Zeitraum vom 01. 01. 2000 bis 12. 02. 2002

3b. Titel der Publikation

4a Autoren des Berichts (Name, Vorname(n))
Holger Schwark, Hans-Joachim Kalkbrenner, Dieter Stöckl

5. Abschlussdatum des Vorhabens
12. 02. 2002

4b Autoren des Publikation (Name, Vorname(n))

6. Veröffentlichungsdatum
Juni 2002

7. Form der Publikation
Schlussbericht für BMBF

8. Durchführende Institution (Name, Adresse)

Robert Bosch Multimedia-Systeme GmbH & Co. KG
Postfach 77 77 77

31132 Hildesheim

8. Ber.Nr. Durchführende Institution

10. Förderkennzeichen
19B0009

11a. Seitenzahl Bericht
82

13. Fördernde Institution (Name, Adresse)

Bundesministerium für Bildung und Forschung

53170 Bonn

11b. Seitenzahl Publikation

12. Literaturangaben

14. Tabellen: 4

15. Abbildungen: 48

16. Zusätzliche Angaben

17. Vorgelegt bei (Titel, Ort, Datum)

18. Kurzfassung

Das Forschungsprojekt MOBILIST, Arbeitspaket C3, Anschlussinformationssystem ANIS (nachfolgend: MOBILIST-ANIS) wurde vom BMBF als Verbund- und Umsetzungsvorhaben gefördert und von der Robert Bosch Multimedia-Systeme GmbH & Co KG (nachfolgend: MU) und den MOBILIST-ANIS-Partnern, u. a. Verkehrs- und Tarifverbund Region Stuttgart GmbH, Stuttgarter Straßenbahnen AG, Daimler Chrysler AG, DB Regio AG, Caatoosee AG gemeinsam und partnerschaftlich auf der Grundlage eines Kooperationsvertrages bearbeitet .

Das Ergebnis von MOBILIST-ANIS soll zu den generellen Zielen der Attraktivitätssteigerung des öffentlichen Nahverkehrs sowie der Mobilität in Ballungsräumen beitragen. MOBILIST ist gleichzeitig der Einstieg in die Ausstattung von Fahrzeugen und Haltestellen der Verkehrsbetriebe in Stuttgart und in der Region mit Geräten für die Bereitstellung dynamischer Anschlussinformation. Die weitere Ausstattung von Fahrzeugen und Haltestellen ist vorgesehen.

Die Versorgung der Bahnen und Busse sowie der Haltestellen mit Fahrgastinformation auf der Grundlage der Übertragungstechnik DAB/DMB und den darauf basierenden Infrastruktur- und Endgeräte-Produkten eröffnet neue Wege und neue Qualitäten für eine kunden- und marktgerechte Information der Fahrgäste.

DAB (Digital Audio Broadcasting) ist der europäische Standard für die beschlossene Einführung des digitalen Rundfunks und mit der multimedialen Erweiterung DMB (Digital Multimedia Broadcasting) in hervorragender Weise für den Einsatz in mobilen und stationären Fahrgastinformationssystemen geeignet.
Die breitbandige Übertragungstechnologie erlaubt es, neben statischen und dynamischen Fahrplaninformationen multimediale Zusatzinformationen (z. B. Werbung) sowie aktuelle Informationen (z. B. aus Politik und Wirtschaft) in Bahnen/Busse und an Haltestellen zu übertragen und verzugslos zu präsentieren.

Aufgabenstellung der MU war die Spezifikation, Entwicklung und Lieferung einer Anzahl von Endgeräten für Schienenfahrzeuge der SSB AG und der DB Regio AG sowie von Demonstratoren zur Erprobung der Haltestellenausstattung.
MU hat die Entwicklungsarbeiten Anfang Februar 2002 abgeschlossen und die Beendigung des Projektes zum 12. 02. 2002 beantragt.

19. Schlagwörter
DAB/DMB, MOBILIST, Mobile und Stationäre Fahrgastinformation, Multimedia, Stuttgart

20. Verlag	21. Preis

Document Control Sheet

1. ISBN or ISSN	2. Type of Report Schlussbericht
3a. Report Title MOBILIST (Mobilität im Ballungsraum Stuttgart), Arbeitspaket C3-, Anschlussinformationssystem ANIS, Projektteil Bosch Schlussbereicht über die Arbeiten im Zeitraum vom 01. 01. 2000 bis 12. 02. 2002	
3b. Title of Publication	
4a. Author(s) of the Report (Family Name, First Name(s)) Holger Schwark, Hans-Joachim Kalkbrenner, Dieter Stöckl	5.End of Project 12.02.2002
4b. Author(s) of the Publication (Family Name, First Name(s))	6. Publication Date Juni 2002
	7. Form of Publication Schlussbericht für BMBF
8. Performing Organization(s) (Name, Address) Robert Bosch Multimedia-Systeme GmbH & Co. KG Postfach 77 77 77 31132 Hildesheim	9. Originator's Report No.
	10. Reference No. 19B0009
	11a. No. of Pages Report 82
	11b. No. of Pages Publication
	12. No. of References
13. Sponsoring Agency (Name, Address) Bundesministerium für Bildung und Forschung (BMBF) 53170 Bonn	14. No. of Tables 4
	15. No. of Figures 48
16. Supplementary Notes	
17. Presented at (Title, Place, Date)	

18. Abstract

Das Forschungsprojekt MOBILIST, Arbeitspaket C3, Anschlussinformationssystem ANIS (nachfolgend: MOBILIST-ANIS) wurde vom BMBF als Verbund- und Umsetzungsvorhaben gefördert und von der Robert Bosch Multimedia-Systeme GmbH & Co KG (nachfolgend: MU) und den MOBILIST-ANIS-Partnern, u. a. Verkehrs- und Tarifverbund Region Stuttgart GmbH, Stuttgarter Straßenbahnen AG, Daimler Chrysler AG, DB Regio AG, Caatoosee AG gemeinsam und partnerschaftlich auf der Grundlage eines Kooperationsvertrages bearbeitet .

Das Ergebnis von MOBILIST-ANIS soll zu den generellen Zielen der Attraktivitätssteigerung des öffentlichen Nahverkehrs sowie der Mobilität in Ballungsräumen beitragen. MOBILIST ist gleichzeitig der Einstieg in die Ausstattung von Fahrzeugen und Haltestellen der Verkehrsbetriebe in Stuttgart und in der Region mit Geräten für die Bereitstellung dynamischer Anschlussinformation. Die weitere Ausstattung von Fahrzeugen und Haltestellen ist vorgesehen.

Die Versorgung der Bahnen und Busse sowie der Haltestellen mit Fahrgastinformation auf der Grundlage der Übertragungstechnik DAB/DMB und den darauf basierenden Infrastruktur- und Endgeräte-Produkten eröffnet neue Wege und neue Qualitäten für eine kunden- und marktgerechte Information der Fahrgäste.

DAB (Digital Audio Broadcasting) ist der europäische Standard für die beschlossene Einführung des digitalen Rundfunks und mit der multimedialen Erweiterung DMB (Digital Multimedia Broadcasting) in hervorragender Weise für den Einsatz in mobilen und stationären Fahrgastinformationssystemen geeignet.
Die breitbandige Übertragungstechnologie erlaubt es, neben statischen und dynamischen Fahrplaninformationen multimediale Zusatzinformationen (z. B. Werbung) sowie aktuelle Informationen (z. B. aus Politik und Wirtschaft) in Bahnen/Busse und an Haltestellen zu übertragen und verzugslos zu präsentieren.

Aufgabenstellung der MU war die Spezifikation, Entwicklung und Lieferung einer Anzahl von Endgeräten für Schienenfahrzeuge der SSB AG und der DB Regio AG sowie von Demonstratoren zur Erprobung der Haltestellenausstattung.
MU hat die Entwicklungsarbeiten Anfang Februar 2002 abgeschlossen und die Beendigung des Projektes zum 12. 02. 2002 beantragt.

19.Keywords
DAB/DMB, MOBILIST, Mobile und Stationäre Fahrgastinformation, Multimedia, Stuttgart

20. Publisher	21. Price

Document Control Sheet

1. ISBN or ISSN	2. Type of Report Schlussbericht
3a. Report Title MOBILIST (Mobilität im Ballungsraum Stuttgart), Arbeitspaket C3-, Anschlussinformationssystem ANIS, Projektteil Bosch Schlussbereicht über die Arbeiten im Zeitraum vom 01. 01. 2000 bis 12. 02. 2002	
3b. Title of Publication	
4a. Author(s) of the Report (Family Name, First Name(s)) Holger Schwark, Hans-Joachim Kalkbrenner, Dieter Stöckl	5.End of Project 12.02.2002
4b. Author(s) of the Publication (Family Name, First Name(s))	6. Publication Date Juni 2002
	7. Form of Publication Schlussbericht für BMBF
8. Performing Organization(s) (Name, Address) Robert Bosch Multimedia-Systeme GmbH & Co. KG Postfach 77 77 77 31132 Hildesheim	9. Originator's Report No. 10. Reference No. 19B0009 11a. No. of Pages Report 82 11b. No. of Pages Publication
13. Sponsoring Agency (Name, Address) Bundesministerium für Bildung und Forschung (BMBF) 53170 Bonn	12. No. of References 14. No. of Tables 4 15. No. of Figures 48
16. Supplementary Notes	
17. Presented at (Title, Place, Date)	

18. Abstract

The research project MOBILIST, Workpackage C3, Anschlussinformationssystem ANIS (that is: MOBILIST-ANIS) was sponsert by BMBF as a interconnection - und realisation project The Robert Bosch Multimedia-Systeme GmbH & Co KG (that is: MU) works on this package with partners. (e g Verkehrs- und Tarifverbund Region Stuttgart GmbH, Stuttgarter Straßenbahnen AG, Daimler Chrysler AG, DB Regio AG, Caatoosee AG)

MOBILIST-ANIS should enhance the attractiveness of the public local traffic MOBILIST is the Entry in the configuration of Busses and Stops of the Municipal Transport Services of Stuttgart with Displays for connection informations.

The supply of trains and Busses as well as Stops with information on the basis of transfer technology DAB/DMB and on the Infrastructure and endproducts based on it opens new ways and new qualities for a custemer and market oriented information of the passengers.

DAB (Digital Audio Broadcasting) is the Europe standard of digital radio. The add on DMB (Digital Multimedia Broadcasting) is very qualified for the mobile application The large bandwidth enabled to send multimedia Informations (e g. advertisement) as well as up to date informations (e g policy and economy) in trains, busses and Stops

The task of the MU was the specification, development und delivery of end products for the railed vehicles from the SSB AG und der DB Regio AG as well as preproduction models for test the stop configurations.
The MU has completed the development work Februar 2002 and has request the completion of the project to the 12 02. 2002.

19.Keywords
DAB/DMB, MOBILIST, Mobile und Stationäre Fahrgastinformation, Multimedia, Stuttgart

20. Publisher	21. Price

01. Mai 03