“Entwurf und Umsetzung einer Softwarearchitektur für ein Flottenmanagementsystem im Inter-/Intranet mit mobiler Kommunikation”

Diplomarbeit

Studiengang Informatik, TU-Ilmenau, Institut für Praktische Informatik und Medieninformatik, Fachgebiet Telematik

verantwortlicher Hochschullehrer :	Prof. Dr.-Ing. habil. D. Reschke
Betreuer :	Dipl.-Inf. R. Döring
betrieblicher Betreuer :	Dipl.-Ing. T. Fuhrmann

Heiko Hüter
Matrikel M94
Studiennummer 24060

Für meine Eltern

Danksagung

Ich bedanke mich bei Prof. Dr.-Ing. habil. D. Reschke für die Betreuung dieser Arbeit.

Mein besonderer Dank gilt Dipl.-Inf. R. Döring und Dipl.-Ing. T. Fuhrmann für ihre konstruktive Kritik und wertvollen Hinweisen bei der Erstellung dieser Arbeit.

Bei den Mitarbeitern der Firma Technotrend möchte ich mich für die allgemeine Hilfe und Unterstützung bedanken.

Des weiteren bedanke ich mich bei den Mitarbeitern des Prüfungsausschusses der Fakultät IA für ihre Bereitschaft, mir über die Verwaltungstechnischen Klippen zu helfen.

Inhaltsverzeichnis

1. Einleitung

Der überwiegende Teil des Güterverkehrs wird heutzutage mit dem LKW bewältigt. Im Jahr 1996 wurden europaweit 73.5% aller Güter über die Straße transportiert (66.1% in Deutschland). Dies entspricht 1159.3 Milliarden tkm (281.3 Milliarden tkm in Deutschland, vgl. [TIF96]). Ausgehend von diesen Zahlen ist es recht einsichtig, daß eine Verminderung dieser Transportmengen vorteilhaft für die Umwelt und die Verkehrsbelastung ist. Das Erreichen dieses Zieles kann das hier zu entwerfende Flottenmanagementsystem unterstützen, indem es eine bessere Koordinierung der Fahrzeuge ermöglicht. Ebenfalls nicht ganz unwichtig ist eine mögliche Kostenersparnis für den Fuhrunternehmer.

Was kann ein solches System leisten ? Zuerst sollte geklärt werden, was Flottenmanagement ist. Eine mögliche Definition (aus [FLTMGMT]) lautet:

> [Flottenmanagement ist ein] Verfahren zur situationsgerechten Einsatzsteuerung (Disposition) und Überwachung (Monitoring) von Fahrzeugflotten, zunehmend unter Einbeziehung satellitengestützter Kommunikations- und Navigationssysteme (z.B. GPS - Global Positioning System) realisiert.

Ein Flottenmanagementsystem unterstützt den Anwender bei der Durchführung dieser Tätigkeit. Dazu müssen die relevanten Daten im Fahrzeug erfaßt und an die Zentrale übermittelt werden. Danach können dort verschiedene Auswertungen und Planungen durchgeführt werden.

Für diese Aufgabenstellung sind schon verschiedene Softwaresysteme im Einsatz. Diese bieten teilweise sogar noch erweiterte Funktionalitäten, wie z.B. eine Navigationsunterstützung im Fahrzeug. Warum gibt es also diese Arbeit ? Wenn man die am Markt befindlichen Lösungen betrachtet fällt auf, daß die meisten ihren Anwendungsfall gut lösen aber ansonsten recht eingeschränkt sind. Zum Beispiel ist ein großer Teil als monolithische Anwendung für ein spezielles Betriebssystem konzipiert. Der Einsatz auf anderen Plattformen erfordert teilweise recht große Anpassungen. Des weiteren bestehen bei solchen Systemen oft Probleme in Bezug auf die Skalierbarkeit. Man hat die Anwendung für ein bestimmtes Umfeld erstellt und kann nur ungenügend auf eine Steigerung der Anforderungen (z.B. eine größere Fahrzeugflotte) reagieren. Es besteht hier höchstens die Möglichkeit, den aktuellen

Rechner durch einen leistungsfähigeren zu ersetzen.

Das hier zu entwerfende System soll im Gegensatz dazu robust, einfach zu bedienen und leicht erweiterbar sein. Ein Anwender soll von seinem Arbeitsplatz aus auf die Daten zugreifen können, unabhängig davon wo sich dieser befindet oder welches Betriebssystem dort installiert ist. Um diese Anforderung zu erfüllen, muß eine Plattform ausgewählt werden, die auf möglichst vielen Rechnern zur Verfügung steht. Außerdem sollten die verwendeten Dienste und Protokolle zur Standardausstattung der meisten Betriebssysteme gehören, damit die Installation auf Seiten des Klienten vereinfacht wird. Neben diesen direkt für den Benutzer zugänglichen Eigenschaften, sollen aber auch bestimmte "innere Werte" berücksichtigt werden. Dazu gehört ein klarer und durchdachter Entwurf, um die Folgekosten zur Wartung zu minimieren. Des weiteren soll ein offenes und gut erweiterbares System geschaffen werden.

Bevor die Details der Anwendungsanalyse, -design und -umsetzung betrachtet werden, sollen im folgenden einige der grundsätzlichen Konzepte, auf denen diese Arbeit aufbaut, eingeführt werden.

1.1. komponentenbasierte Softwareentwicklung

Im Laufe der Entwicklung der Softwaretechnik hat sich gezeigt, daß eine gewisse Ordnung des Entwicklungsprozesses positive Auswirkungen auf die Gesamtkosten für die Entwicklung und Nutzung eines Softwaresystems hat. In einem ersten Schritt hat man versucht, immer wiederkehrende Algorithmen in Form von Funktionen und Prozeduren zu kapseln. Dadurch konnten diese von verschiedenen Programmteilen aus aufgerufen werden und es gab eine zentrale Stelle, die gewartet werden mußte. Dieses Vorgehen ist als *prozedurale* Programmierung bekannt.

In den 80er Jahren hat sich die Erkenntnis durchgesetzt, daß die treibende Kraft einer Anwendung die Daten sind. Schon in dem einfachen EVA Prinzip (Eingabe - Verarbeitung - Ausgabe) erkennt man, daß die Funktionen nicht im Vordergrund stehen. Vielmehr sind sie nur in Zusammenhang mit den zu verarbeitenden Daten von Interesse. Aus diesem Grundgedanken entwickelte sich die *objektorientierte Programmierung*. Hierbei steht die Modellierung der Daten in Form von abstrakten Datentypen im Vordergrund. Diese Typen definieren sowohl die Daten, als auch die Operationen die über ihnen ausgeführt werden können. In der Umsetzung bedeutet das, daß der Modellierer eines Softwaresystems Objekte in der jeweiligen Anwendungsdomäne identifizieren muß. Durch die weitere Spezifizierung der Interaktionen und Zusammenhänge zwischen diesen Objekten entsteht ein Modell der Anwendung, welches dann umgesetzt werden kann. Eine gute Einführung in die Objektorientierung bietet z.B. das Buch von Bertrand Meyer ([MEYER97]).

In den letzten Jahren hat es aber auch hier wieder Anpassungen gegeben. Diese wurden durch eine Änderung der Anforderungen an die Anwendungen hervorgerufen. In heutigen Problemstellungen steht nicht mehr ein einzelner Rechner im Mittelpunkt, auf denen ein Benutzer alle Berechnungen ausführt. Vielmehr gibt es meistens ein Netz von Rechnern, mit dem verschiedene Benutzer zusammen an einem Problem arbeiten. Dies erfordert auch, daß verschiedene Teile einer Anwendung über Rechnergrenzen hinweg zusammenarbeiten können. Solche verteilten Anwendungen stellen auch völlig neue Anforderungen an eine Softwarearchitektur. In einem *komponentenbasierten Softwareentwurf* werden verschiedene Teile einer Anwendung zu Funktionsgruppen, den sogenannten Komponenten, zusammengefaßt. Ein typischer Entwurf unterteilt eine Anwendung z.B. in Benutzeroberfläche, Geschäftslogik und Datenspeicherung (vgl. Abschnitt 2.1.2). Jede dieser Bausteine kann weiter unterteilt werden. Alle Komponenten sind voneinander getrennt und kommunizieren über genau festgelegte Schnittstellen und Protokolle (auch über Systemgrenzen hinweg). Eine komponentenbasierte Softwarearchitektur bedingt nicht unbedingt auch einen objektorientierten Entwurf dieser Komponenten. Es ist durchaus denkbar, eine Komponente als Sammlung von Prozeduren zu erstellen. Allerdings bietet sich eine objektorientierter Ansatz an. In diesem Fall ist die komponentenbasierte Entwicklung eine Erweiterung der Objektorientierung auf verteilte Systeme.

1.2. Mobile Kommunikation

Der Begriff Kommunikation bedeutet im Umfeld dieser Arbeit, daß zwei oder mehr verschiedene Komponenten, auf verschiedenen Rechnern und über klar definierte Protokolle Daten austauschen. Mobile Kommunikation bedeutet, daß wenigstens eine der beteiligten Komponenten nicht direkt über ein Netzwerk (im Sinne von Kabelverbindungen) an die anderen angeschlossen ist. Dies ermöglicht einen flexiblen, ortsungebundenen Einsatz. Allerdings müssen dazu auch andere Verbindungswege und Protokolle eingesetzt werden. In dieser Arbeit soll die mobile Kommunikation über einen Mobilfunkdienst (Globales System für Mobile Kommunikation - GSM, vgl. Abschnitt 2.2.1) erfolgen. Es sollen Ansätze aufgezeigt werden, wie eine solche mobile Komponente in eine verteilte Gesamtarchitektur eingebunden werden kann.

Zum Abschluß der Einleitung noch ein allgemeiner Hinweis. In dieser Arbeit wird ein gewisses Grundverständnis über den Aufbau und Arbeitsweise eines Netzwerkes vorausgesetzt. Insbesondere die Basistechnologien und Protokolle im Zusammenhang mit dem Internet sollten bekannt sein (HTTP, TCP/IP, ...). Für eine Einführung in diese Themen existieren verschiedene Quellen (siehe z.B. [COMER98]). Einige der neueren Entwicklungen, wie z.B. die Common Object Request Broker Architektur - CORBA oder Enterprise Java Beans - EJB, werden an den entsprechenden Stellen kurz vorgestellt.

2. Anwendungsanalyse und Architekturentwurf

2.1. objektorientierte Analyse

Bei einer objektorientierten Entwicklung eines Softwaresystems muß man in einem ersten Schritt untersuchen, welche Teile des Anwendungsumfeldes zusammengehören und welche Beziehungen zwischen diesen Teilen bestehen. Bei dieser *objektorientierten Analyse* geht es also um die Bestimmung von Objekten der Anwendungsdomäne und wichtiger Beziehungen zwischen diesen Objekten. Dies soll jetzt für die Flottenmanagementanwendung durchgeführt werden. Dazu muß zuerst das Anwendungsumfeld bestimmt werden. Dabei werden auch benachbarte Gebiete mit berücksichtigt. Dadurch soll das Ergebnis möglichst allgemeingültig und vielfältig einsetzbar sein.

2.1.1. Beschreibung verschiedener Einsatzgebiete und deren Umfeld

2.1.1.1. Flottenmanagement

Die Aufgabenstellung eines Flottenmanagementsystems ist die Verwaltung einer Fahrzeugflotte. In jedem Fahrzeug werden wichtige Informationen (z.B. Position des Fahrzeuges, aktuelle Geschwindigkeit, aktuelle Ladung, ...) gesammelt und in gewissen Abständen zur Zentrale übertragen. Dort werden alle eingehenden Informationen sortiert und in einer Datenbank abgelegt. Diese Daten können dann von verschiedenen Arbeitsstationen abgerufen werden. In der hier vorgestellten Umsetzung soll der Zugriff über handelsübliche Internetzugriffsprogramme (sogenannte *Browser* erfolgen). Ein späterer Zugriff über spezielle Einzelanwendungen soll aber, mit möglichst wenig Aufwand, möglich sein.

Die Anwendung soll verschiedene Auswertungen der Daten ermöglichen. Die einfachste Auswertung ist das Anzeigen der aktuellen Position eines Fahrzeuges. Dazu muß das entsprechende Kartenmaterial in der Zentrale vorhanden sein und eine Möglichkeit der Abbildung der eingegangenen Positionsdaten auf das Straßennetz vorhanden sein. Wenn mehrere Positionsdaten vorhanden sind, soll eine Fahrstrecke rekonstruierbar sein. Neben dieser reinen Überwachung der Fahrzeugflotte ist als Erweiterung das Verwalten von Aufträgen

denkbar. Hierbei sollte für einen gegebenen Auftrag das beste verfügbare Fahrzeug ermittelt und die Vergabe des Auftrags unterstützt werden. Als Erweiterung wäre jetzt denkbar, daß die eingehenden Positionsdaten im Zusammenhang mit dem vergebenen Auftrag ausgewertet werden. Dadurch sind z.B. Statusmeldungen bei Erreichen bestimmter Zwischenstationen oder Warnungen bei Verspätungen denkbar. Falls die aktuelle Ladung bekannt ist, können so auch in Abhängigkeit von deren Haltbarkeit Warnmeldungen an den Fahrer oder die Zentrale übermittelt werden.

2.1.1.2. Fahrplanüberwachung

Bei diesem Einsatzgebiet gibt es ebenfalls eine Fahrzeugflotte, deren Mitglieder Daten an die Zentrale übermitteln. Im Gegensatz zum Flottenmanagement können sich die Fahrzeuge nicht auf individuellen Strecken bewegen. Vielmehr gibt es einen Fahrplan, der die möglichen Fahrstrecken vorgibt. Eine komplette Strecke besteht dabei i.d.R. aus mehreren Zwischenstationen. Zu jeder Zwischenstation gibt es eine Zeitangabe. Diese ist entweder absolut vorgegeben oder relativ zum Anfang der Fahrstrecke. Wenn ein Fahrzeug eine Zwischenstation erreicht, muß die aktuelle Durchfahrtszeit mit der vorgegebenen Zeit verglichen werden. Danach können z.B. bei Bedarf Warnmeldungen erzeugt werden. Ein denkbares Einsatzgebiet ist hierbei die Überwachung von Eisenbahn- oder Busunternehmen.

2.1.1.3. Einsatzplanung und Abrechnung

Dieses Einsatzgebiet ähnelt stark dem Flottenmanagement. Es gibt eine Fahrzeugflotte deren Daten in der Zentrale abgelegt werden. Jedes einzelne Fahrzeug hat einen eigenen Auftrag (ein individuelles Fahrziel). Der Unterschied besteht jetzt allerdings darin, daß nicht nur der Zeitpunkt wann das Ziel erreicht wird von Interesse ist. Bei diesem Anwendungsgebiet wird die vorhandene Karte in Zonen eingeteilt. Für die Abrechnung einer Fahrstrecke ist deren Aufteilung auf die verschiedenen Zonen interessant. Ein denkbares Einsatzgebiet ist dabei die Planung und Abrechnung von Rettungsdiensteinsätzen.

2.1.1.4. Transportgutüberwachung

Im Gegensatz zu den vorherigen Einsatzgebieten gilt es in diesem Szenario nicht eine Fahrzeugflotte zu überwachen. Die Aufgabenstellung hierbei ist, den Weg einer bestimmten Ware zu verfolgen. Über gewisse Teilstrecken wird der Weg dieser Ware identisch mit dem Weg eines Fahrzeuges sein. Allerdings besteht hier auch die Möglichkeit, daß das Fahrzeug gewechselt wird. In der Zentrale werden weiterhin nur die Positionsdaten von Fahrzeugen eingehen. Die Anwendung muß eine Zuordnung dieser Fahrzeugdaten zu Transportgütern unterstützen. Bei einer Auswertung müssen außerdem bestimmte Teilstrecken, ggf. von

verschiedenen Fahrzeugen, zu einer Gesamtstrecke für eine Ware zusammengesetzt werden. Dies entspricht in etwa einem System das UPS (United Parcel Service) im Einsatz hat. Damit kann ein Kunde überwachen, wo die von ihm aufgegebene Sendung sich gerade befindet.

2.1.1.5. Informationssysteme

Die bisher vorgestellten Einsatzgebiete zielen vorwiegend auf betriebswirtschaftliche Aufgaben ab. Es geht um das Management von Aufgaben rund um eine Fahrzeugflotte. Die Anwendung richtet sich vorwiegend an fachlich versierte Benutzer. Für einige Einsatzgebiete ist allerdings auch eine Version für eine größere Anwendergruppe denkbar. Dabei geht es dann darum, die Leistungen und Anstrengungen einer Firma nach außen darzustellen. Ein Fuhrunternehmen könnte dabei z.B. mit seiner guten Qualität und Transparenz werben. Unternehmen des öffentlichen (Nah-)Verkehrs können ihren Kunden die Abfrage und Visualisierung der Fahrstrecken und -pläne ermöglichen. Vielfältige Statistiken über Verspätungen und Auslastungen können immer aktuell bereitgestellt werden. Interessant dabei ist, daß dafür auf die selbe Datenbasis wie für die betriebswirtschaftlichen Anwendungen zugegriffen werden kann. Es ist hierbei eine eingeschränkte öffentliche Schnittstelle und eine komplexe Verwaltungsschnittstelle denkbar. Datenänderungen würden sofort für beide Benutzergruppen zugänglich sein. Durch Einsatz von Internettechnologien kann ein einfacher Zugriff für die verschiedensten Klienten ermöglicht werden.

2.1.2. allgemeine Anwendungsarchitektur

Wenn man eine Softwarearchitektur für ein gegebenes Problem entwerfen will, muß man zuerst einige allgemeine Anforderungen an das System definieren. Für das hier zu betrachtende System sind folgende Punkte entscheidend:

1. Die einmal entwickelte Anwendungslogik soll leicht wiederverwendbar sein. Insbesondere ist eine mögliche Anpassung an die beschriebenen benachbarten Einsatzgebiete mit zu berücksichtigen.

2. Die entstehende Anwendung soll zukunftsicher, gut erweiterbar und leicht wartbar sein

3. Das System soll sowohl von Internetklienten als auch von traditionellen Anwendungen aus benutzbar sein

4. Eine Skalierung passend zu den Einsatzbedingungen soll unterstützt werden. Dies betrifft einerseits die Aufteilung des Systems in verschiedene Komponenten in der Art, daß jeder Anwender sich sein gewünschtes System zusammenstellen kann. Des weiteren muß auch eine Anpassung an größer werdende Leistungsanforderungen (steigende Benutzerzahlen, größere Datenmengen, ...) möglich sein

Diese Anforderungen müssen im folgenden Entwurf und der späteren Umsetzung entsprechend berücksichtigt werden.

Nachdem diese grundsätzlichen Rahmenbedingungen bestimmt sind, muß die Aufgabenstellung auf zusammengehörige Funktionsgruppen untersucht werden. Bei den vorgestellten Anwendungsgebieten gibt es eine gemeinsame Basis von benötigten Diensten. Insbesondere lassen sich folgende Teilaufgaben identifizieren:

1. Kommunikation mit der Fahrzeugflotte
2. Zugriff auf Datenbank, Sammeln und Bereitstellen der Fahrzeugdaten
3. Visualisierung von geographischen Daten, Abbildung der Fahrzeugpositionen auf das Straßennetz
4. Auftragsverwaltung, Unterstützung bei der Einsatzplanung
5. verschiedene Auswertungen zur Auftragsabrechnung (Summe der gefahrenen Kilometer, gefahrene Strecken nach bestimmten Kriterien, Planung und Optimierung von Fahrstrecken, ...)

Wenn man diese gemeinsamen Eigenschaften geeignet modelliert und bereitstellt, lassen sich die verschiedenen Anwendungen schneller und einfacher umsetzen. Ein weiterer Vorteil ist die bessere Wartbarkeit des resultierenden Gesamtsystems, da eine Verbesserung der Umsetzung einer Komponenten Vorteile für alle Benutzer dieser Komponente bringt (es muß nur eine zentrale Stelle geändert werden). Diese Abhängigkeit aller Programme von bestimmten Teilen bedeutet aber auch, daß sich jeder Klient auf die Eigenschaften der benutzten Teile verlassen muß. Aus diesen allgemeinen Überlegungen ergibt sich der Vorteil einer komponentenbasierten und objektorientierten Entwicklung. Ein Programm wird hierbei aus mehreren Komponenten bzw. Objekten zusammengesetzt. Von einem Objekt ist nur seine Schnittstelle bekannt, d.h. dessen garantierte Funktionalität. Wie diese Funktionalität erbracht wird, ist für den Benutzer versteckt und kann deshalb jederzeit geändert werden. Diese Änderungen bedingen keine Änderungen der anderen Komponenten, da die Schnittstelle unverändert bleibt. Durch einen objektorientierten Ansatz wird insbesondere die Realisierung der ersten beiden Punkte der Anforderungsliste unterstützt.

Für eine genauere Einführung in die objektorientierte Entwicklung von Softwaresystemen (speziell objektorientierte Analyse/OOA und objektorientiertes Design/OOD) gibt es verschiedene Literaturquellen z.B. [BOOCH94] oder [OOMD].

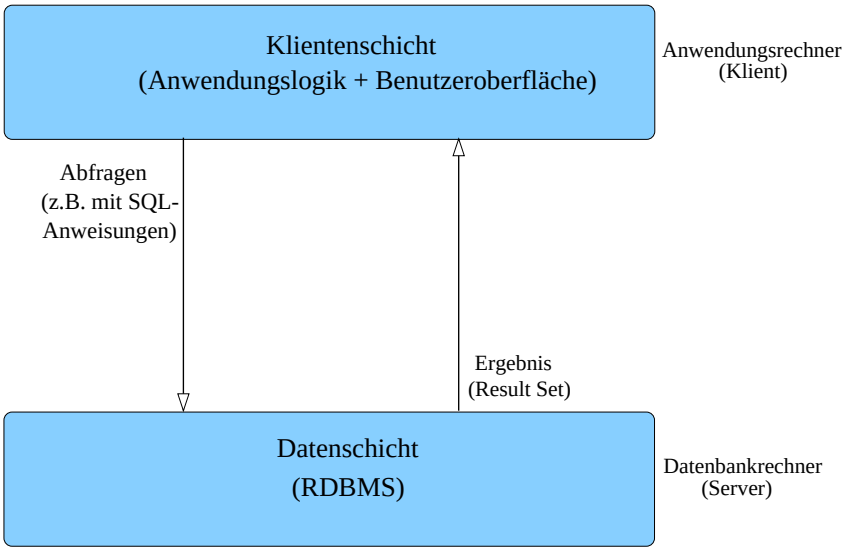


Abbildung 2.1.: schematische Darstellung einer typischen two-tier Architektur

In dieser Arbeit soll für den Architekturentwurf die Methodik der UML[1] benutzt werden. Dies ist eine Spezifikation die von der OMG[2] entwickelt wird. Zur Zeit aktuell ist die Version 1.1, beschrieben in [OMG97]. Der Standard definiert verschiedene Diagramme und Modelle zur Beschreibung eines OO-Softwaresystems.

Neben dem prinzipiellen Bekenntnis zu einem komponentenbasierten Entwurf gilt es aber noch weitere Architekturentscheidungen zu treffen. Dies betrifft u.a. die Aufteilung der Anwendung in verschiedene Ebenen (Schichten). Bis vor wenigen Jahren waren die meisten Anwendungen in einer sogenannten *two-tier Architektur* (auch *Klienten Server Architektur* genannt) umgesetzt. Diese Architektur wird durch zwei Schichten gekennzeichnet. Die erste Schicht ist dabei die *Klientenschicht*, d.h. die Schicht in der die gesamte Anwendung abläuft. In ihr ist sowohl die Anwendungslogik als auch die Benutzeroberfläche angesiedelt. Die Anwendungslogik kann dann auf Dienste der *Datenschicht* zugreifen, z.B. um die Persistenz der Anwendungsdaten zu erreichen (vergleiche Abbildung 2.1). Die Datenschicht besteht meistens aus einer relationalen Datenbank (RDBMS). Für den Entwickler ist dieses Modell recht einfach zu benutzen. Er kann sich auf die Umsetzung der ersten Schicht beschränken, d.h. er muß sich nur um die eigentliche Anwendungslogik und Benutzeroberfläche kümmern. Weiterführende Dienste wie Persistenz, Zugriffssteuerung, Sicherheit und Transaktionsmanagement werden durch die Datenbank zur Verfügung gestellt. Die Benutzung dieser Dienste erfolgt meist über SQL[3]-Anweisungen. Nachteile

[1] Unified Modelling Language

[2] Object Management Group, ein Zusammenschluß mehrerer Firmen die sich um die Entwicklung und Umsetzung von Technologien im Bereich der Objektorientierung bemühen

[3] Structured Query Language, Eine Abfragesprache zum Zugriff auf relationale Datenbanken, vgl. z.B. [SQL99]

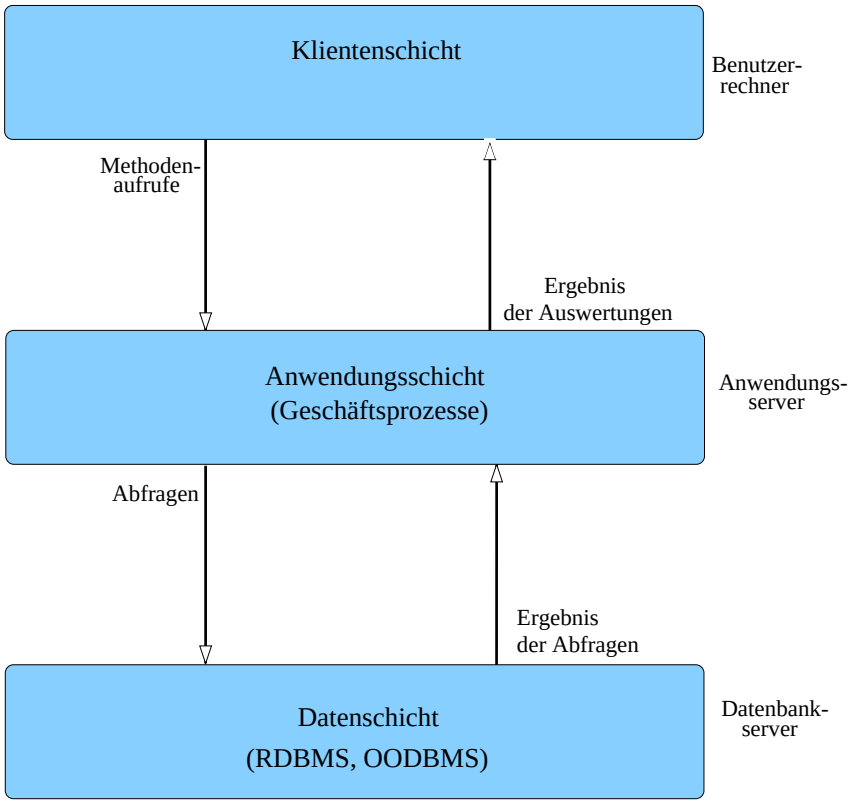


Abbildung 2.2.: Darstellung der umzusetzenden three-tier Architektur

bestehen im Bezug auf die Skalierbarkeit solcher Lösungen. Bei größer werdenden Anforderungen (höhere Anzahl von Benutzern, mehr Daten, ...) kann nur auf Seiten der Datenbank eingegriffen werden. Die Abarbeitung der Geschäftsprozesse[1] erfolgt auf Seiten des Anwenders.

Um die Probleme der two-tier-Architektur zu lösen, führt man eine weitere Schicht ein. Die neue Schicht dieser *three-tier Architektur* (drei Ebenen Architektur) wird zwischen der Klientenschicht und der Datenschicht eingefügt. Deshalb wird sie mittlere Schicht (*middle tier*) oder auch Anwendungsschicht genannt. Diese Schicht enthält die gesamte Anwendungslogik. Die Klientenschicht enthält nur noch die Benutzeroberfläche und leitet die eigentliche Verarbeitung der Aufträge an die Dienste der Anwendungsschicht weiter (vgl. Abbildung 2.2). Durch die Trennung der Verarbeitung von der Benutzerschnittstelle ist das Gesamtsystem leichter an die jeweiligen Bedürfnisse anpaßbar. Normalerweise läuft die gesamte Anwendungsschicht auf einem dedizierten Server (Anwendungsserver). Falls man während der Lebenszeit des Systems in eine Situation kommt, in der die Rechenleistung dieses Servers nicht mehr ausreicht, kann das System flexibel erweitert werden. Dies kann nicht nur durch Austausch des Servers durch einen leistungsfähigeren erfolgen. Es ist auch

[1] Ein Geschäftsprozeß (im folgenden auch Geschäftsobjekt genannt) ist ein bestimmter Teilablauf in einem Unternehmen. In dieser Arbeit ist dabei insbesondere auch deren Umsetzung in einer Softwarearchitektur gemeint, d.h. ein Geschäftsprozeß entspricht dann einer Menge von Softwareprozessen.

denkbar, nur einen Teil der Anwendungsdienste auf einen anderen Rechner auszulagern. Die gesamte Architektur unterstützt diese Aufteilung der Anwendungsschicht auf mehrere physikalische Rechner sehr gut (in diesem Zusammenhang spricht man dann auch von einer *n-tier* Architektur).

Der Anwender erhält dadurch also ein gut verfügbares, an seine Bedürfnisse leicht anpaßbares System. Für den Entwickler entstehen allerdings neue Herausforderungen. Bei der Klienten Server Architektur muß er sich nur um die Umsetzung seiner Anwendungslogik und um die Prozeß-Datenbank Kommunikation kümmern. Wichtige Dienste, die praktisch jede Anwendung benötigt (z.B. Transaktionsmanagement, Zugriffsschutz, ...), werden durch die Datenbank bereitgestellt. Durch Einführung der neuen Schicht muß ein Entwickler sich aber um die neuen Anforderungen durch die Prozeß-Prozeß Kommunikation zwischen Klientenschicht und Anwendungsschicht kümmern. Das betrifft nicht nur die Auswahl eines geeigneten Kommunikationsprotokolls. Vielmehr muß er sich um all die Dinge kümmern, die bei der Kommunikation zwischen Anwendungs- und Datenschicht das Datenbanksystem übernimmt. Der anfängliche Aufwand für den Entwickler steigt also. Allerdings verringern sich durch diese Architektur die Folgekosten für Wartung und Weiterentwicklungen merklich. Das System ist flexibel und gut erweiterbar. Deshalb soll in dieser Arbeit eine drei Ebenen Architektur umgesetzt werden. Darüber hinaus gibt es verschiedene Ansätze um den Entwickler von der Umsetzung der Basisfunktionalität zu befreien (EJB[1], MTS[2]). Dadurch kann der Anwendungsentwickler sich wieder damit beschäftigen womit er sich auskennt - der Umsetzung seiner Geschäftslogik.

Neben der Aufteilung der Anwendung auf mehrere Schichten muß auch festgelegt werden, wie diese Schichten miteinander kommunizieren. Dies betrifft als erstes die Auswahl eines geeigneten Kommunikationsprotokolls. Des weiteren muß man sich für ein Komponentenmodell entscheiden, welches dieses Protokoll gut unterstützt. Nachdem diese Wahl getroffen ist, geht es um die Definition der Schnittstellen der einzelnen Schichten. Die Wahl der Kommunikationsmethode wird zwischen Anwendungs- und Datenschicht meistens durch die zu benutzenden Datendienste bestimmt. Deshalb wird es im folgenden hauptsächlich um den Datenaustausch zwischen Klientenschicht und Anwendungsschicht gehen. Da die Anwendung aus objektorientierter Sicht entworfen wird, sollte das benutzte Komponentenmodell auch objektorientiert sein. Darüber hinaus besteht die Anforderung, daß das Softwaresystem möglichst plattformunabhängig ist. Deshalb sollte das benutzte Übertragungsprotokoll ein offener Standard sein, der auf vielen verschiedenen Plattformen zur Verfügung steht.

Aufgrund dieser grundlegenden Überlegungen ist also klar, daß die Anwendung in einer drei Ebenen Architektur entwickelt werden soll. Die für die Bereitstellung der Anwen-

[1] Enterprise Java Beans, ein Standard der von der Firma Sun entwickelt wird

[2] Microsoft Transaction Server, das entsprechende Gegenstück von Microsoft allerdings schon ein lauffähiges Produkt

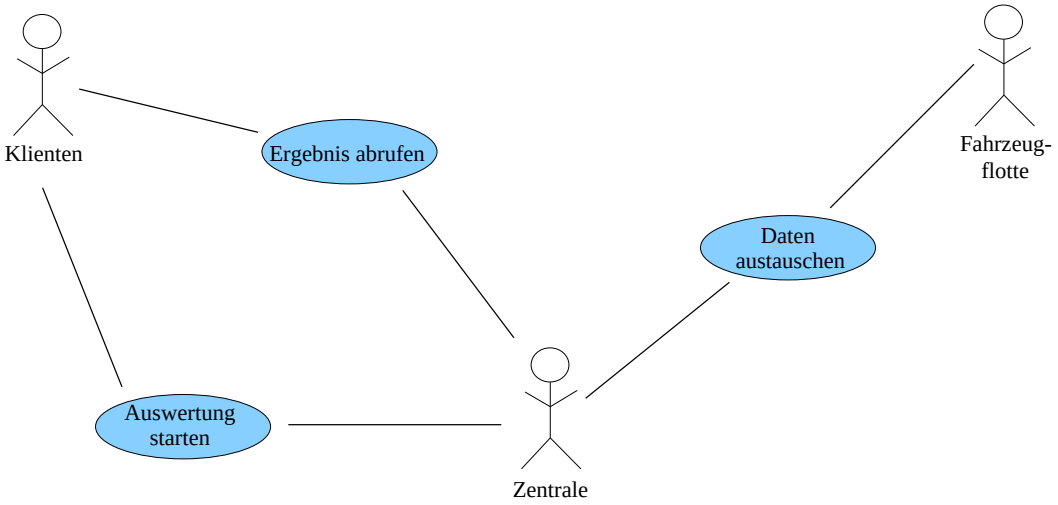


Abbildung 2.3.: Die globale Architektur der Anwendung, dargestellt mit einem Use Case Diagramm

dungsschicht benötigte Komponententechnologie ist eine Frage der Umsetzung. Deshalb wird sie in Kapitel 3 behandelt. Vorher wird es allerdings um eine genauere Spezifizierung der gesamten Architektur gehen. Dazu werden im nächsten Teil gemeinsame Komponenten gesucht. Darauf aufbauend wird dann der Entwurf einer Klassenhierarchie durchgeführt. Im Ergebnis liegt dann das Modell der Geschäftsobjekte vor. Dieses kann dann in verschiedenen Szenarien eingesetzt werden.

2.1.3. Globale Struktur und gemeinsame Komponenten der Anwendung

In einem ersten Schritt kann man für alle Anwendungsgebiete die drei im *Use Case* aus Abbildung 2.3 dargestellten Teilkomponenten identifizieren (zur Softwaremodellierung mit Use Case Diagrammen vgl. z.B. [JAC94] oder [OMG97]). Bei jedem Einsatzgebiet gibt es eine *Fahrzeugflotte*. In jedem einzelnen Fahrzeug werden relevante Daten (Position, Geschwindigkeit - abhängig von der konkreten Anwendung) erfaßt. Diese Daten werden gesammelt und in bestimmten Abständen zur *Zentrale* übertragen. Für die Übertragung soll in dieser Arbeit der GSM/SMS Dienst eines entsprechenden Anbieters verwendet werden. Wie solche Nachrichten in die Anwendung integriert werden können, wird in einem späteren Abschnitt untersucht. Im Moment wird davon ausgegangen, daß die Zentrale diese Daten empfängt und in einer Datenbank ablegt. Dabei werden für jeden eingehenden Datensatz bestimmte zusätzliche Informationen mit erfaßt. Dies betrifft z.B. das Kennzeichen des Fahrzeuges (nicht notwendiger Weise das amtliche Kennzeichen) sowie der Zeitpunkt, an dem die Daten erfaßt wurden. Nachdem die Daten empfangen und gespeichert sind, können verschiedene Auswertungen durch die *Klienten* ausgelöst werden. Die Klienten sind dabei die verschiedenen Anwendungen, die der Benutzer sieht (z.B. für das Flottenmanagement).

Diese Applikationen können dabei bestimmte Basisdienste[1] der Zentrale benutzen. Das Ergebnis der Auswertung wird auf dem Server bereitgestellt und kann vom Klienten abgeholt werden.

Der erste Teil dieser Arbeit wird sich damit beschäftigen, sinnvolle Basisdienste zu separieren und zu modellieren. Hierbei ist eine Auswahl zu treffen, welche Dienste allgemeingültig sind (und deshalb in der Zentrale zur Verfügung gestellt werden) und welche Dienste nur für bestimmte Anwendungen von Interesse sind (und demzufolge auch dort umgesetzt werden). Allerdings sollte man dabei auch beachten, wie datenintensiv bestimmte Teilaufgaben sind. Es kann durchaus sinnvoll sein, einen bestimmten Auftrag, der eigentlich nur für einen oder wenige Anwendungsfälle interessant ist, in der Zentrale zur Verfügung zu stellen. Dies hat den Vorteil, daß die Netzbelastung verringert wird. Ein Prozeß der viele Daten anfordert läuft so schneller, da der Datenbankserver entweder mit dem Anwendungsserver identisch oder zumindest in dessen Nähe untergebracht ist. Wenn solche Auswertungen auf Seiten des Klienten erfolgen, ist ein großer Datenaustausch zwischen Klient und Server die Folge. Insbesondere in einem heterogenen Netzwerk wie dem Internet ist es allerdings oft nicht möglich, die Qualität der Übertragung zu garantieren. Die Kommunikation erfolgt durch Weitergabe von Paketen zwischen mehreren Stationen. Die Gesamtübertragungsrate hängt von der Anbindung jeder einzelnen Station ab (diese kann sehr stark zwischen einzelnen Stationen schwanken). Ein Anwender kann nicht beeinflussen, welchen Weg seine Pakete benutzen. Deshalb sollten so viele Daten wie möglich direkt auf dem Server bearbeitet werden. Auf Seiten des Klienten werden die Daten nur für die Darstellung aufbereitet und die Benutzeroberfläche zur Verfügung gestellt. Dies entspricht dann einer reinen drei Schichten Architektur mit sogenannten *leichten Klienten* (*thin Clients*)[2].

Die Abläufe im Fahrzeug sollen in dieser Arbeit nicht weiter betrachtet werden. Es wird davon ausgegangen, daß im Fahrzeug eine Komponente existiert, die wichtige Daten sammelt und überträgt. Im einfachsten Fall könnte diese “Komponente” der Fahrer sein, der die Nachrichten direkt über den SMS-Dienst überträgt. Im allgemeinen wird es allerdings eher ein Rechner sein, der diese Aufgabe automatisiert. Eine kurze Beschreibung der verwendeten Hardware bietet Abschnitt 4.3.4. Eine genauere Untersuchung erfolgte in der Arbeit von Andreas Bernklau [BERN97].

Die Auswertungen auf Seiten des Klienten sollen zu einem späteren Zeitpunkt betrachtet werden. Im folgenden wird es um eine Verfeinerung des Modells der Abläufe auf Seiten des Servers gehen. Das Ziel dieser Betrachtungen wird eine komplette Klassenstruktur für die Basisdienste sein.

[1] Diese Basisdienste sind ein zentraler Bestandteil der gesamten Architektur. Ein Basisdienst repräsentiert eine Gruppe zusammengehörender Operationen. Die Basisdienste sind die Umsetzung der Geschäftsprozesse, die den Klienten zugänglich sind.

[2] Als *thin-client* werden Klienten bezeichnet, die keine eigene Abarbeitung der Anfragen durchführen. Alle Aufträge werden auf den Server übertragen. Der Klient stellt nur die Benutzeroberfläche zur Verfügung.

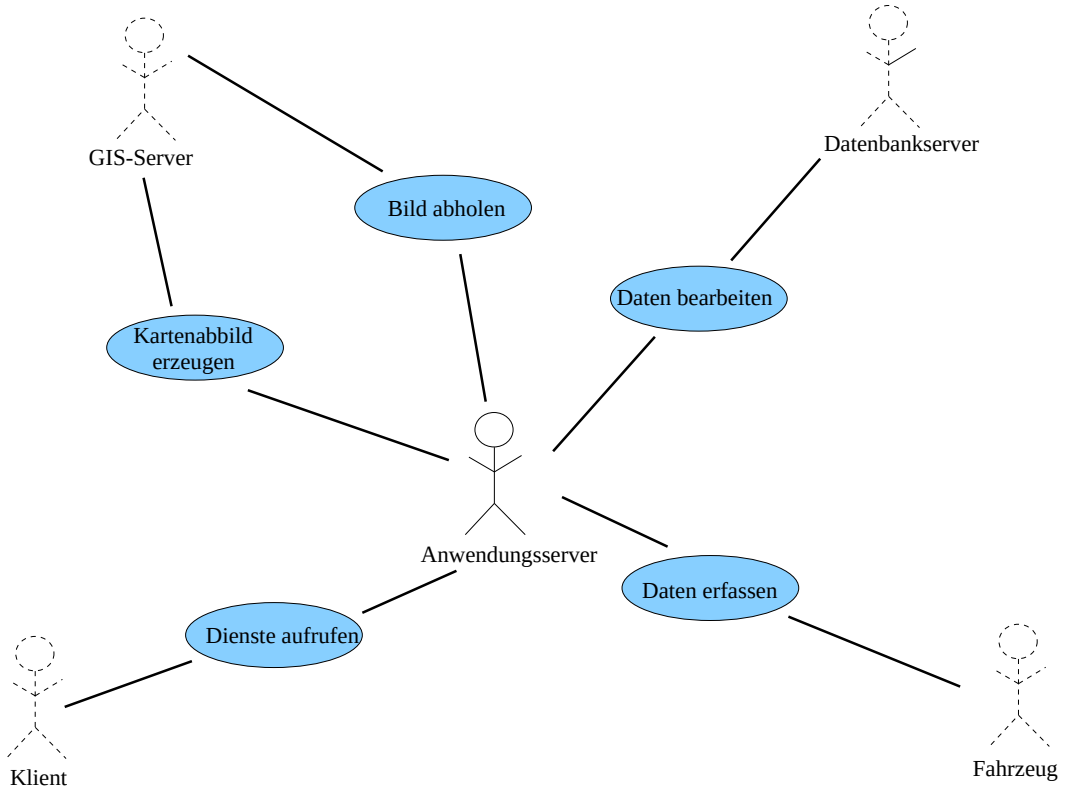


Abbildung 2.4.: die erste Verfeinerung der Serverarchitektur

Die erste Verfeinerung des Servers ist im Use Case aus Abbildung 2.4 dargestellt. Die gestrichelt gezeichneten Aktoren (Klient, Fahrzeug, ...) sollen dabei das Umfeld des Use Case Diagrammes besser kennzeichnen, sie gehören nicht zum Server. Dadurch ist es allerdings möglich aufzuzeigen, welche Dienste von außerhalb des Servers ansprechbar sind. In dieser Abbildung ist die zugrundeliegende drei Ebenen Architektur recht gut erkennbar (vergleiche Abbildung 2.5). Der Aktor *Klient* (die Klientenschicht) interagiert nur mit Diensten des *Anwendungsservers* (der mittleren Schicht). Dienste des Datenbankservers, GIS[1]-Servers und Fahrzeugschnittstelle (in der dritten oder Datenschicht) sind nicht direkt ansprechbar. Sie werden nur durch die Dienste des Anwendungsservers zur Erfüllung der jeweiligen Aufgabe benutzt (Use Cases *Kartenausschnitt rendern* und *Bild abholen*). So bietet der GIS-Server die Möglichkeit, einen vorher festgelegten Kartenausschnitt in einer Bilddatei abzulegen. Dieses Bild wird dann vom Anwendungsserver weiterverarbeitet. Im einfachsten Fall kann der Klient das Bild abfragen und darstellen. Wenn der Benutzer jetzt dieses Bild manipulieren will (z.B. hineinzoomen), muß der Klient diese Anforderung erfassen. Sie wird dann an den Anwendungsserver weitergeleitet, der wiederum dafür sorgt, daß der GIS-Server dieses neue Bild berechnet.

Die Dienste des Anwendungsservers benötigen zur Erbringung ihrer Leistung unter-

[1]Geographisches Informations System, in dieser Arbeit hauptsächlich ein System zur Visualisierung von geographischen Daten

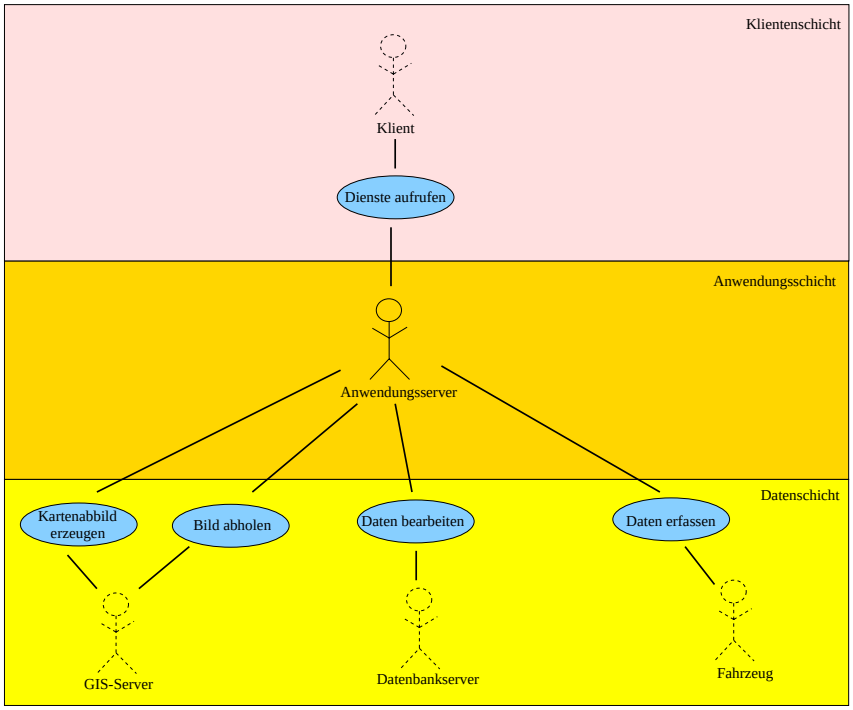


Abbildung 2.5.: das Server Use Case eingebettet in die drei Ebenen Architektur

schiedlichste Daten. Die erste Gruppe sind natürlich die bereits erwähnten Daten der Fahrzeugflotte. Darüber hinaus sind aber noch andere Daten von Interesse. Für eine Verfolgung und Planung der Fahrstrecken reichen die Daten des GIS-Servers im allgemeinen nicht aus. Hierfür benötigt man weitere Daten, wie z.B. die Breite einer Fahrbahn, die Belastbarkeit einer Brücke oder eventuelle Beschränkungen auf Grund der Höhe. Eine von der eigentlichen Darstellung abgelöste Repräsentation des Straßennetzes ist auch für die Analyse besser. Eine Verfolgung oder Berechnung von Routen kann durch geeignete Wahl der Repräsentation unterstützt werden. Diese und weitere denkbare Daten werden auf dem Datenbankserver abgelegt und können so durch die Dienste des Anwendungsservers manipuliert und abgefragt werden (Use Case *Daten bearbeiten*). Auch die Erfassung der Fahrzeugdaten erfolgt durch Verwendung der entsprechenden Geschäftsprozesse (Use Case *Daten erfassen*).

Der Anwendungsserver ist der zentrale Bestandteil dieser Architektur. Deshalb soll im folgenden dieser Teil weiter verfeinert werden. Diese Verfeinerung ist im Use Case aus Abbildung 2.6 dargestellt. Im Mittelpunkt dieses Modells stehen die *Basisdienste*. Des weiteren sind die Ressourcen dargestellt, auf die sie zur Erbringung ihrer Leistung zurückgreifen können. Dies betrifft im wesentlichen die Leistungen des GIS-Systems (z.B. *Karte laden*, *Einstellungen festlegen* und *Karte darstellen*) sowie der Datenbank (z.B. *Position abrufen* und *Straßennetz abfragen*). Außerdem haben die Basisdienste die Möglichkeit, eine Aktualisierung der Positionsdaten eines Fahrzeuges zu erzwingen. Dies geschieht über die

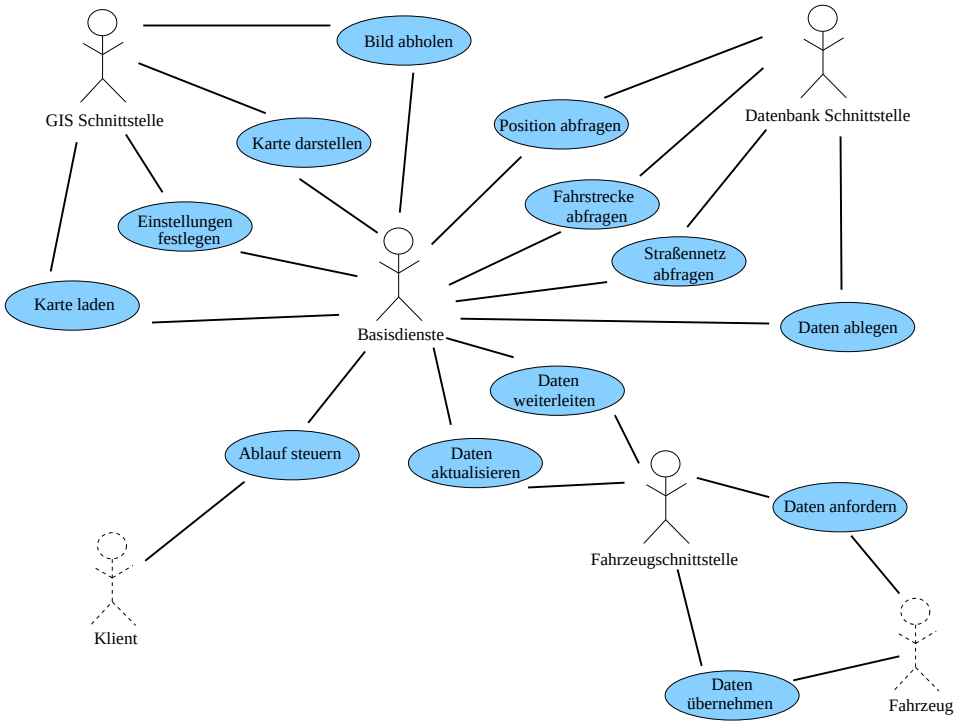


Abbildung 2.6.: die Erweiterte Darstellung der Architektur des Anwendungsservers

Fahrzeugschnittstelle die dann ihrerseits die Kommunikation mit dem Fahrzeug initiiert. Die Aktoren *GIS Schnittstelle* und *Datenbankschnittstelle* sind nicht die Standardschnittstellen dieser Komponenten. Vielmehr handelt es sich um eigens erstellte, anwendungsspezifische Schnittstellen (innerhalb der mittleren Schicht). Dadurch soll auch innerhalb des Servers ein Austausch dieser Komponenten durch andere Produkte erleichtert werden. Zur Anpassung an eine neue (eventuell proprietäre) Schnittstelle des jeweiligen Hersteller, müssen nur an einer Stelle Änderungen erfolgen. Alle anderen Komponenten werden nicht beeinträchtigt (und damit die Anwendungslogik). Dies soll insbesondere die Umsetzung des zweiten und vierten Punktes der Anforderungsliste aus Abschnitt 2.1.2 unterstützen.

2.2. objektorientiertes Design

Nachdem im vorherigen Abschnitt eine grobe Aufteilung der Anwendung vorgenommen wurde, soll im folgenden eine detaillierte Struktur entwickelt werden. Als Ziel steht dabei eine vollständig spezifizierte Klassenhierarchie. Zur Beschreibung dieser Klassenarchitektur sollen wieder die Möglichkeiten der UML benutzt werden (insbesondere die Klassendiagramme). Die Reihenfolge in der die einzelnen Komponenten betrachtet werden soll dabei grob den tatsächlichen Abläufen in der Praxis entsprechen. Das heißt, im ersten Schritt wird die Fahrzeugschnittstelle entwickelt. Hierbei werden auch verschiedene Möglichkeiten der mobilen Kommunikation zwischen Fahrzeug und Zentrale untersucht und

gegenübergestellt. Danach wird die Ablage der eingegangenen Daten in der Datenbank betrachtet. Es werden die Leistungen der Datenbankschnittstelle auf Seiten des Anwenders definiert. Die tatsächliche Umsetzung dieser (und aller anderen) Schnittstellen wird in Kapitel 3 erläutert. Diese Trennung ist sinnvoll, da diese internen Details nicht zur allgemeinen Architektur zählen, sondern von der jeweiligen Komponente (z.B. der zugrundeliegenden Datenbank) abhängen. Nachdem nun die Daten empfangen und abgelegt sind, können sie durch die Basisdienste verarbeitet werden. Zuvor werden allerdings noch die Voraussetzung zur Visualisierung dieser Daten geschaffen (GIS-Schnittstelle).

2.2.1. Die Fahrzeugschnittstelle zur mobilen Kommunikation

Für ein Flottenmanagementsystem ist es wichtig, daß in der Zentrale die Daten der Fahrzeugflotte vorhanden sind. Für einige Anwendungsfälle müssen die Daten möglichst aktuell sein (z.B. für die Fahrplanüberwachung). Bei anderen können die Daten gesammelt und erst später ausgewertet werden. Diese verschiedenen Anforderungen müssen auch bei der Wahl des Übertragungsdienstes berücksichtigt werden.

In dieser Arbeit soll für die Kommunikation zwischen Zentrale und Fahrzeug ein GSM[1]-Dienstanbieter verwendet werden. Der GSM-Standard ist seit Anfang der 90er Jahre etabliert. Es gibt mehrere verschiedene Anbieter weltweit. Die dafür nötige Ausrüstung (Mobiltelefone, Modems) sind in größerer Stückzahl verfügbar und damit relativ kostengünstig. Diese weite Verbreitung und geringe Kosten sind günstig für einen kommerziellen Einsatz einer darauf basierenden Anwendung. Außerdem eignet sich ein digitales mobiles Kommunikationsnetz gut für eine Datenübertragung, da hier eine Umwandlung der digitalen Daten in analoge Form entfällt. Für einige Anwendungen ist die geringe Bandbreite des Netzes ein Problem. Für ein Flottenmanagementsystem ist dies allerdings nicht relevant, da die zu übertragenden Datenmengen klein sind. Oftmals reichen bereits die 160 Zeichen, die mit Hilfe des SMS[2] Dienstes übertragen werden können aus. Des weiteren haben auch die Mobilfunkbetreiber dieses Problem erkannt und werden es wohl in nächster Zeit lösen (siehe z.B. [GPRS]).

Der prinzipielle Aufbau eines Mobilfunknetzes nach dem GSM-Standard zeigt Abbildung 2.7 (vergleiche hierzu und den folgenden [SCO]). Es gibt prinzipiell drei Teilsysteme. Das erste ist die mobile Station des Benutzers (also z.B. das Handy). Hier gibt es einerseits die *mobile Ausrüstung* (*Mobile Equipment*, ME) die zur Übertragung der Nachrichten eingesetzt wird. Zur Identifizierung des Nutzers dient das *Benutzer Identifizierungsmodul* (*Subscriber Identity Module*, SIM). Dadurch wird die Identität eines Benutzers vom eigentlichen Gerät getrennt. Ein Benutzer kann verschiedene Geräte einsetzen und trotzdem auf seine persönlichen Einstellungen zugreifen. Als nächstes kommt das *Basisstations Teilsy-*

[1] Globales System für mobile Kommunikation, ein Standard für ein digitales Mobilfunknetz

[2] Short Message Service, ein Dienst zur Übertragung kurzer Nachrichten der von den meisten GSM-Anbietern bereitgestellt wird

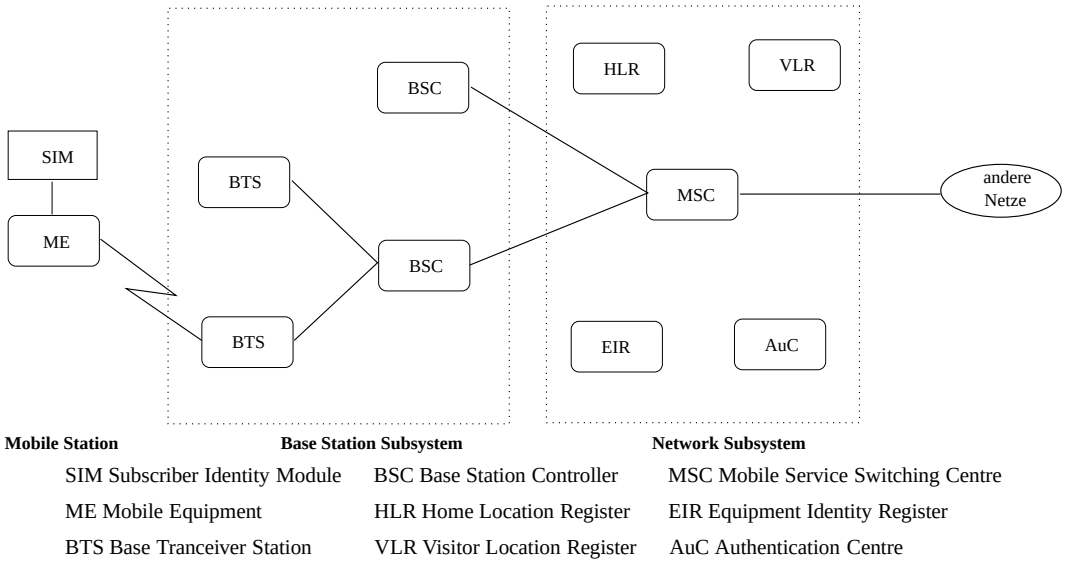


Abbildung 2.7.: Struktur eines GSM-Mobilfunknetzes

stem (*Base Station Subsystem*). Hier erfolgt die Koordinierung der Datenübertragung mit der Mobilen Ausrüstung. Die *Basis Übermittlungsstation* (*Base Tranceiver Station*, BTS) dient dabei direkt der drahtlosen Kommunikation. Die Signale mehrerer BTS' werden zur *Basisstations Steuerungseinheit* (*Base Station Controller*, BSC) weitergeleitet. Hier erfolgt die Steuerung der Übertragung (z.B. Frequenzauswahl, Stationsauswahl, ...). Mehrere BSC' sind mit einem *Mobilen Dienstauswahl Zentrum* verbunden (*Mobile Service Switching Centre*, MSC). In dieser Station sind die verschiedenen Benutzer registriert und eine Authentifizierung wird durchgeführt. Außerdem wird hier ein Anschluß des Mobilfunknetzes an andere Netze (z.B. an das normale Telefonnetz) ermöglicht.

Für diese Arbeit sind vor allem die Leistungen des MSC interessant. Für die Anwendungsfälle, bei denen die zeitliche Verzögerung zwischen Datensammlung und -übertragung nicht kritisch ist, kann die Übertragung mittels elektronischer Post (e-mail) erfolgen. Im MSC werden dabei eingehende SMS-Nachrichten an eine vorgegebene e-mail Adresse weitergeleitet. Die Anwendung kann dann diese Nachrichten abrufen und auswerten. Falls die dadurch entstehende Verzögerung nicht akzeptabel ist, muß ein direkter Empfang der SMS-Nachrichten möglich sein. Dies geschieht mittels eines auf dem Server installierten GSM-Modems (Dieses ist sowieso erforderlich, wenn es möglich sein soll die Daten vom Fahrzeug anzufordern). Dadurch kann eine SMS-Nachricht schnellstmöglich ausgewertet werden. Für den Entwurf bedeutet das, daß es eigentlich zwei Fahrzeugschnittstellen gibt. Die Unterschiede sollten allerdings möglichst transparent für andere Komponenten sein. Der Benutzer wählt seinen gewünschten Übertragungsweg aus und die entsprechende Fahrzeugschnittstelle wird initialisiert. Alle anderen Komponenten müssen mit dieser Schnittstelle zusammenarbeiten, egal welche konkrete Umsetzung dahintersteckt.

Unterschiede existieren aber nicht nur auf Seiten der Schnittstelle zum Fahrzeug. Nach-

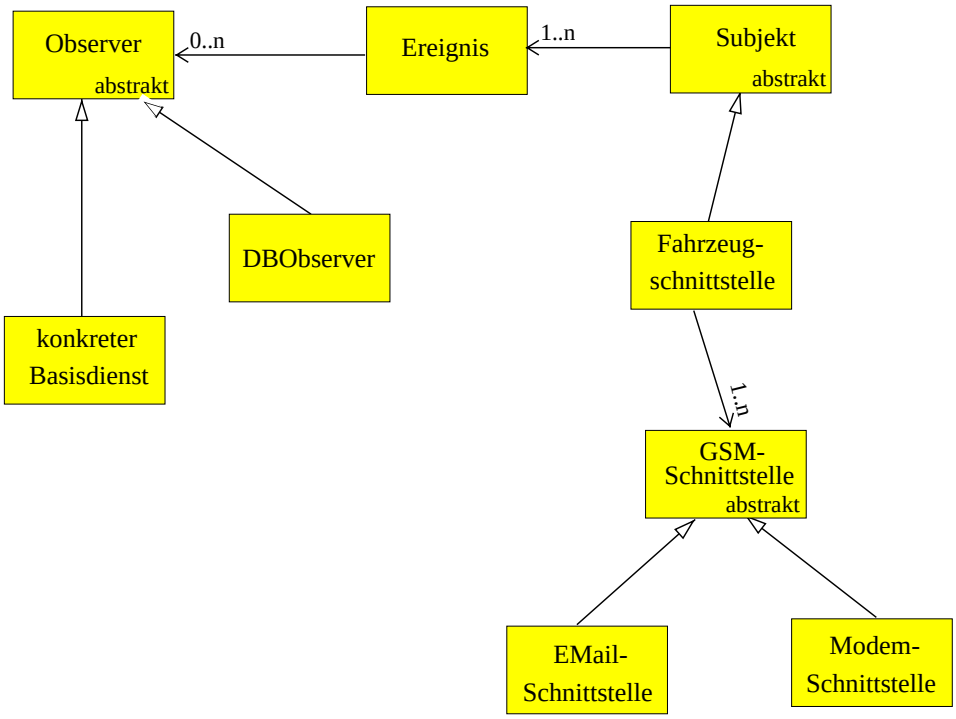


Abbildung 2.8.: die Fahrzeugschnittstelle als Klassendiagramm

dem die Daten empfangen und untersucht sind, gibt es verschiedene Möglichkeiten der Weiterverarbeitung. Im allgemeinen Fall werden die Daten in einer Datenbank abgelegt und können dann von anderen Komponenten über die Datenbankschnittstelle abgerufen werden. Für einige spezielle Anwendungsfälle reicht dies allerdings nicht aus. Insbesondere wenn die direkte Kommunikation mit dem Fahrzeug gewählt wurde, um die Daten möglichst schnell zu erhalten, muß die benutzende Komponente möglichst schnell erfahren, daß neue Daten da sind. In diesem Fall ist es nicht akzeptabel, daß der Benutzer die Daten aktiv abfragen muß. Es würde hier zu einem Polling Ansatz kommen, also dem permanenten Anfragens ob neue Daten da sind. Für diese Aufgabe besser geeignet ist ein Ereignis getriebener Ansatz (Interruptmechanismus). Die Fahrzeugschnittstelle benachrichtigt den Benutzer wenn neue Daten eingetroffen sind. Diese Daten können dann direkt (ohne Zwischenspeicherung in der Datenbank) weitergeleitet werden. Für die meisten Anwendungsfälle werden die Daten parallel dazu für spätere Auswertungen persistent abgelegt.

Die Benachrichtigung einer Komponente über Änderungen in einer anderen Komponente wird in verteilten Architekturen öfters benötigt. Die Kopplung zwischen den beteiligten Modulen sollte möglichst locker sein. Am besten wäre es, wenn eine Komponente nichts von der anderen voraussetzt. Für solche immer wieder auftretenden Problemfälle gibt es schon fertige Lösungen (sogenannte *Entwurfsmuster* oder auch *Design Patterns*). Ein Beispiel für den Einsatz eines Entwurfsmusters ist in dieser Arbeit schon vorgekommen. Der Object Request Broker (ORB) der CORBA baut auf dem allgemeinen *Brokerpattern* auf.

Das Broker oder Vermittler Entwurfsmuster definiert eine Art der Kommunikation zwischen Klienten und Servern. Diese erfolgt nicht direkt sondern über eine Vermittlerkomponente. Die Server melden sich beim Vermittler an und stellen dadurch ihre Dienste zur Verfügung. Die Klienten können dann diese Dienste über eine klar definierte Schnittstelle abfragen. Eine gute Einführung in dieses und andere Entwurfsmuster bietet das Buch *Design Patterns - Elements of Reusable Object-Oriented Software* ([GOF]).

Für das Problem der Benachrichtigung von unabhängigen Komponenten über eingetretene Ereignisse existiert das *Beobachterentwurfsmuster (Observerpattern*). Dabei gibt es *Subjekte*, *Ereignisse* und *Beobachter* (*Observer*). Die Subjekte sind die Komponenten deren Zustand durch die Observer überwacht werden soll. Die Änderung des Zustands wird durch Eintreten eines Ereignisses gemeldet. Ein Subjekt kann dabei mehrere Ereignisse unterstützen. Ein Observer kann sich bei einem oder mehreren Ereignissen anmelden. Bei relevanten Zustandsänderungen des Subjektes werden dann alle registrierten Beobachter benachrichtigt.

Für den Anwendungsfall der Fahrzeugschnittstelle ist diese das Subjekt. Die Observer können verschiedene Komponenten sein, die ein Interesse daran haben, über neue Daten informiert zu werden (d.h. über das Eintreten dieses Ereignisses). Ein denkbarer Observer ist z.B. der Datenbankobserver zum Speichern eingetroffener Daten.

Diese verbal formulierten Überlegungen und Anforderungen sind im Klassendiagramm aus Abbildung 2.8 dargestellt. Der Observermechanismus wird durch zwei abstrakte Basisklassen (*Observer, Subjekt*) unterstützt. Ein konkreter Observer dient dazu, neue Daten sofort in die Datenbank zu schreiben (*DBObserver*). Ansonsten kann sich z.B. jeder Basisdienst, der die Daten möglichst sofort braucht, an die Fahrzeugschnittstelle (bzw. an ein unterstütztes Ereignis) anmelden. Die Fahrzeugschnittstelle hat mindestens eine Instanz der *GSMSchnittstelle* als Attribut. Diese Klasse dient als abstrakte Basisklasse für konkrete Umsetzungen der Kommunikation mit einem Fahrzeug. Hier sind als Strategien die Übertragung mittels elektronischer Post (*EMailSchnittstelle*) bzw. direkte Modem-zu-Modem Kommunikation (*ModemSchnittstelle*) vorgesehen. Bei Bedarf lassen sich allerdings sehr einfach neue Strategien umsetzen, indem weitere Klassen von der *GSMSchnittstelle* abgeleitet werden (vgl. hierzu das *Strategy* Entwurfsmuster). Dies fördert eine gute Erweiterbarkeit des Systems (Punkt 2 der Anforderungsliste aus Abschnitt 2.1.2).

Ein zu beachtendes Problem gibt es im Zusammenhang mit der Klasse *Fahrzeugschnittstelle* und deren Instanziierung. Wenn mehrere Instanzen zur selben Zeit existieren, kann es zu Problemen mit der Konsistenz der Gesamtarchitektur kommen. Ein Observer müßte z.B. aufpassen, an welche dieser Schnittstellen er sich anmeldet. Wenn er sich nur an einer anmeldet, gehen eventuell Daten die nur die anderen Schnittstellen erhalten haben verloren. Wenn er sich an mehrere anmeldet, bekommt er eventuell mehrere Benachrichtigungen über ein eingetretenes Ereignis (eine Datenänderung). Dies würde über den Datenbankobserver eventuell auch Auswirkungen auf die Integrität der Datenbank haben. Um diese Probleme

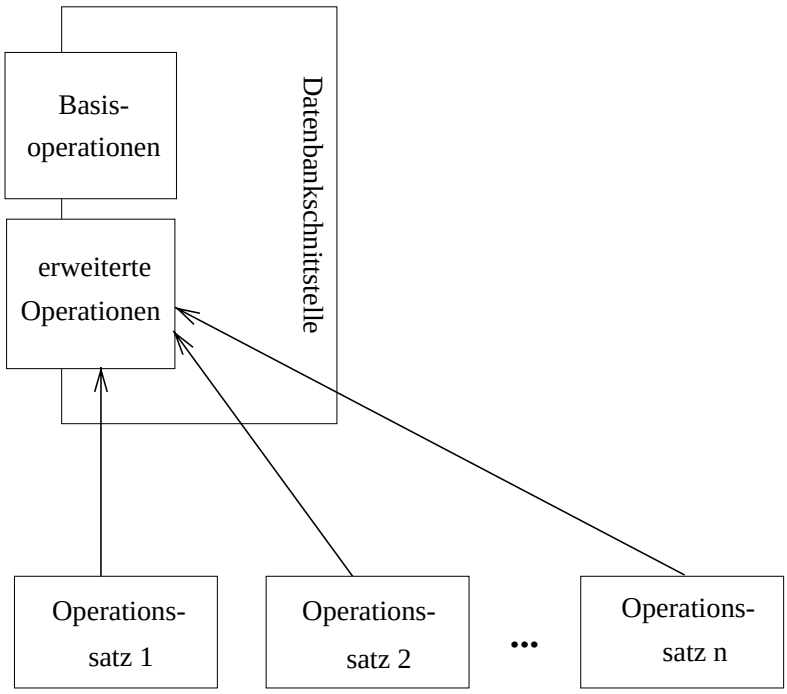


Abbildung 2.9.: Datenbankschnittstelle - prinzipieller Aufbau

zu verhindern und die nötige Synchronisierung zu vermeiden wird festgelegt, daß immer nur eine Instanz der Klasse *Fahrzeugschnittstelle* zu einem Zeitpunkt existieren darf (Singleton Entwurfsmuster). Wie dies mit der zu wählenden Komponentenarchitektur umgesetzt werden kann wird in Kapitel 3 behandelt.

2.2.2. Die Datenbankschnittstelle

Die Datenbankschnittstelle hat im wesentlichen zwei Funktionsgruppen zu unterstützen. Dies betrifft einerseits die Ablage der Daten, die von der Fahrzeugschnittstelle geliefert werden. Andererseits sind Leistungen für die Basisdienste zur Verfügung zu stellen. Dies betrifft das Auslesen der gespeicherten Daten unter Berücksichtigung verschiedener Randbedingungen. Außerdem sollte eine künftige Erweiterung um neue Funktionsgruppen einfach realisierbar sein. Dies läßt sich am besten berücksichtigen, wenn nicht eine Datenbankschnittstelle definiert wird. Vielmehr sollte diese Schnittstelle aus einem Satz von Basisoperationen bestehen, die bei Bedarf durch beliebige andere Funktionssätze erweitert werden kann (vgl. *Decoratorpattern*). Dieses Prinzip ist in Abbildung 2.9 dargestellt.

Wie läßt sich jetzt dieser allgemeine Ansatz in einer Klassenstruktur ausdrücken? Insbesondere unter der Prämisse, daß diese Klassenstruktur allgemeingültig und in den meisten OO-Programmiersprachen umsetzbar sein soll. Der direkte Weg, eine Änderung der Klassenschnittstelle zur Laufzeit, wird von den wenigsten Sprachen unterstützt. Eine etwas abgeschwächtere Form ist, daß die einzelnen Operationssätze zur Verfügung gestellt

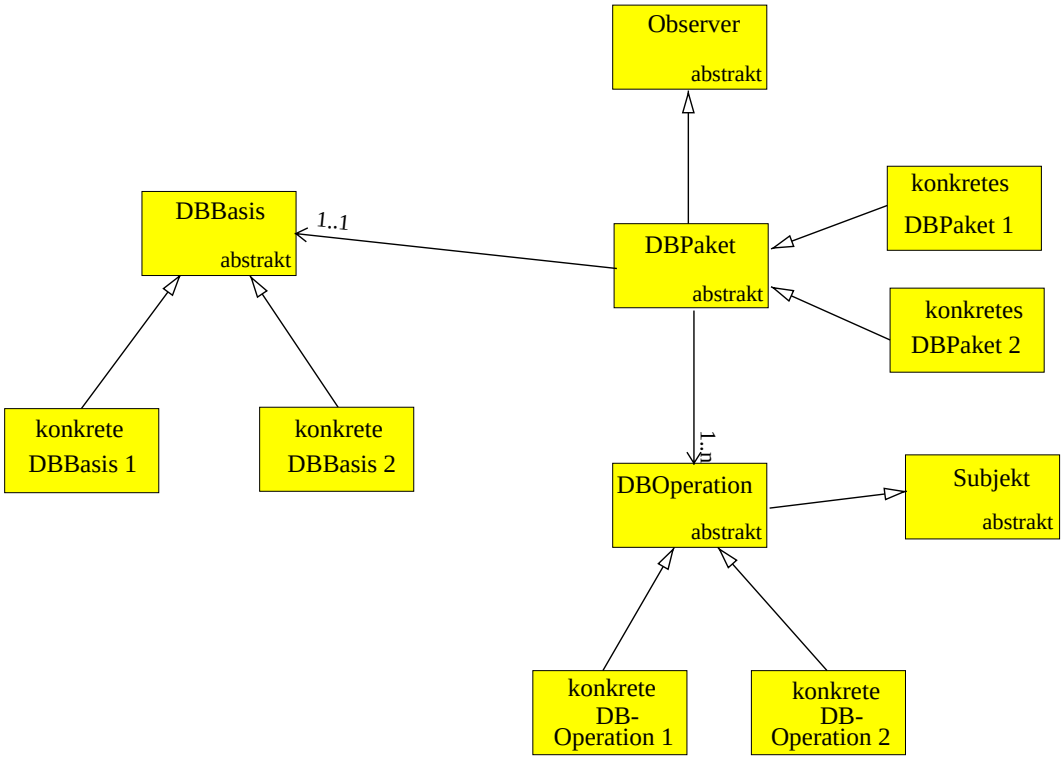


Abbildung 2.10.: Klassenstruktur der Datenbankschnittstelle

werden. Ein Klient würde dann eine eigene Klasse erzeugen, die von einer ausgewählten Untermenge dieser Klassen erbt. Mit dieser Lösung sind allerdings auch zwei große Probleme verbunden. Einerseits setzt sie Mehrfachvererbung voraus, welche nicht unbedingt immer unterstützt wird. Des weiteren besteht in der gewählten Architektur ein prinzipielles Problem darin, daß Klienten ihre eigenen Geschäftsklassen erzeugen. Diese würden zwangsweise innerhalb der Klientenschicht laufen, was dem drei Ebenen Entwurf widerspricht. Deshalb soll auch dieser Ansatz nicht verfolgt werden. Der nächste Schritt ist dann, die einzelnen Komponenten nicht zu einer Klasse zusammenzuführen, sondern getrennt zu benutzen. Die Datenbankschnittstelle ist dann nicht mehr eine einzelne Klasse, sondern vielmehr eine Sammlung von beliebig vielen. Dies ist der hier zu verfolgende Ansatz und wird im Klassendiagramm aus Abbildung 2.10 dargestellt.

Zentraler Punkt dieser Struktur ist die Klasse *DBPaket*. Sie ist die abstrakte Basisklasse für alle konkreten Operationssätze aus Abbildung 2.9 (*DBPaket 1*, *DBPaket 2*, ...). Solch ein Operationssatz stellt 1 bis n konkrete Operationen (*DBOperation 1*, *DBOperation 2*, ...) zur Verfügung, die eine gemeinsame Basisklasse *DBOperation* haben (einfache Operationen können auch direkt als Methoden eines Paketes bereitgestellt werden). Die einzelnen Operationssätze können auf die grundsätzlichen Leistungen der *DBBasis* zurückgreifen. Diese Datenbankbasis kapselt allgemeine Operationen, die von der tatsächlich zugrundeliegenden Datenbank abhängen. Dazu gehört z.B. der Auf- und Abbau der Verbindung, Ausführung

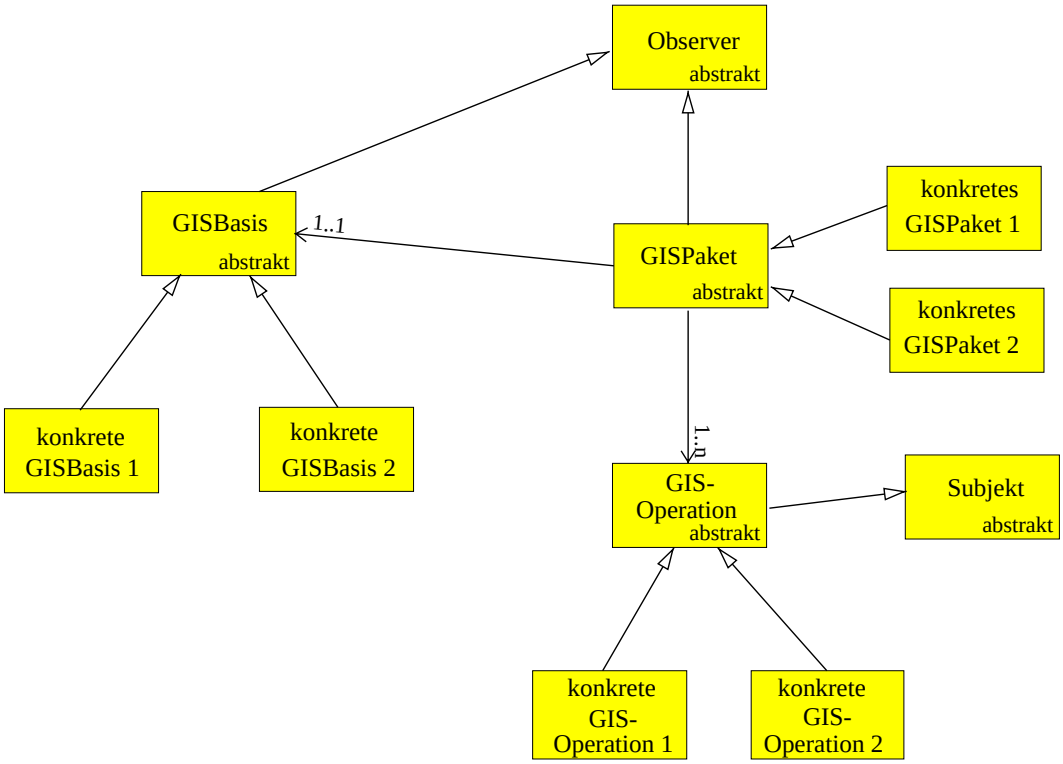


Abbildung 2.11.: Klassenstruktur der GISSchnittstelle

von Auswahlkommandos, Einfügen neuer Datensätze, ... Die Klassen *DBBasis 1* und *DBBasis 2* sind Stellvertreter für solche konkreten Basisoperationen (z.B. für Oracle, MSSQL, ...).

In dieser Struktur ist bereits eine Fähigkeit implizit enthalten, die bis jetzt nicht erwähnt wurde. Die Durchführung einer Datenbankoperation kann ggf. eine lange Zeit in Anspruch nehmen. Während dieser Zeit wäre der aufrufende Prozeß blockiert. Aus diesem Grund soll die Klassenstruktur eine asynchrone Abarbeitung solcher Prozesse unterstützen. Der Aufrufer könnte eine Operation starten und danach weitere Operationen ausführen. Wenn die Operation beendet ist, wird der Aufrufer darüber informiert und kann das Ergebnis weiterverarbeiten. Zur Realisierung dieser asynchronen Kommunikation wird wiederum das bereits eingeführte *Observerpattern* benutzt. Eine Operation ist dabei das Subjekt während das aufrufende Paket der Observer ist. Dieser Mechanismus setzt allerdings eine Unterstützung der Entwicklungsumgebung für die Erstellung mehrere Programmfäden (*Threads*) voraus. Diese Konsequenzen sind allerdings im wesentlichen Teil der konkreten Umsetzung und werden deshalb in Kapitel 3 behandelt. Die konkrete Klassenstruktur der Datenbankschnittstelle, mit den hier umgesetzten Paketen, Operationen und Basisoperationen, wird ebenfalls dort besprochen bzw. im Anhang A aufgeführt.

2.2.3. Die GIS-Schnittstelle

Ein *geographisches Informationssystem - GIS* dient zur visuellen Darstellung und Manipulation von geographischen Daten. Geographische Daten sind dabei neben den reinen Kartendaten auch eventuelle Objektdaten, die auf dieses Kartennetz abgebildet werden. Im Kontext eines Flottenmanagementsystems geht es um die Darstellung des Straßennetzes und der Fahrzeuge. Des weiteren müssen auch bestimmte thematische Karten (z.B. Anzeige der Fahrtrouten oder -zeiten) unterstützt werden. Für die Schnittstelle ergeben sich ähnliche Anforderungen wie für die Datenbankschnittstelle. Es gibt einen Satz von Basisoperationen (Verbindung zum GIS-server herstellen, Kartenmaterial auswählen, ...) die von allen Anwendungen benötigt werden. Daneben gibt es bestimmte Funktionsgruppen (Auswertungen, thematische Karten, ...) aus denen ein Klient die benötigten auswählen kann. Eine Nebenläufigkeit geographischer Auswertungen erhöht die Flexibilität der Gesamtlösung. Aus diesen gemeinsamen Anforderungen ergibt sich die, bis auf eine kleine Änderung, gleiche Struktur der GIS-Schnittstelle wie die der Datenbankschnittstelle (vgl. Abbildung 2.11). Die *GISBasis* ist auch ein Beobachter (erbt von Observer). Dadurch können an dieser zentralen Stelle einige Einstellungen (z.B. aktuelles Darstellungsfenster) aktuell gehalten werden. Was sich natürlich ansonsten zwischen den beiden Schnittstellen ändert, ist deren konkrete Ausprägung (die eigentlich angebotenen Leistungen). Diese sind aus der kompletten Klassenstruktur in Anhang A ersichtlich.

2.2.4. Die Basisdienste

Nachdem in den vorherigen Abschnitten die grundlegenden Bausteine entwickelt wurden, sollen im folgenden die Basisdienste und deren Schnittstelle definiert werden. Ein Basisdienst entspricht dabei in der Umsetzung einem Objekt oder Objektgruppe die der Klient instanziieren kann, um deren Dienstleistungen in Anspruch zu nehmen. Im Gegensatz zu den vorherigen Diensten (Datenbank-, GIS- und Fahrzeugschnittstelle) werden diese Komponenten allerdings durch Klienten außerhalb des Anwendungsservers benutzt. Sie erfüllen komplexere Aufgaben (aufbauend auf den bereits beschriebenen Schnittstellen). Es gibt einen Zusammenhang mit den eigentlichen Geschäftsprozessen. Oft besteht eine 1:1 Beziehung zwischen einem Basisdienst und einem Geschäftsobjekt. Im allgemeinen erfolgt der Zugriff auf die Basisdienste von einem beliebigen Rechner mit Internetzugang. Zum Zugriff werden dabei z.B. HTML Dateien mit eingebetteten Java-Applets (dargestellt mit einem handelsüblichen Browser) oder spezielle Anwendungen verwendet. Die Details zur Umsetzung der Benutzerschnittstelle und möglicher Klienten wird im Kapitel 4 behandelt. Im folgenden sollen die zu erbringenden Leistungen definiert und eine dazu passende Klassenstruktur entwickelt werden.

In einem ersten Schritt müssen die verschiedenen Anwendungsfälle aus Abschnitt 2.1.1 untersucht und gemeinsame Prozesse gefunden werden. Diese benötigten Leistungen müs-

Basisdienst	Komponente
Initialisierung	Init
Visualisierung (Karte, Positionsdaten und thematische Karten)	Visual
Erzeugung von Reporten, Abfrage der Fahrzeugdaten	Report
analytische Auswertung des Straßennetzes, Ermittlung optimaler Fahrstrecken	RouteMgmt
Aufträge bearbeiten und planen	JobMgmt
Kommunikation mit Fahrzeugflotte	FleetCom

Tabelle 2.1.: einige denkbare Basisdienste mit den Namen der zugeordneten Komponente

sen dann entsprechend ihrer Funktion in Gruppen aufgeteilt und als Komponenten modelliert werden. Tabelle 2.1 zeigt einige wichtige Basisdienste. Jeder Basisdienst wird in der Umsetzung einer Komponente zugeordnet. Falls zu einem späteren Zeitpunkt eine größere Funktionalität benötigt wird, kann das System erweitert werden, ohne die bisherige Funktionalität zu beeinträchtigen (durch Bereitstellung neuer Komponenten).

Zwischen diesen Basisdiensten bestehen gewisse Abhängigkeiten. Die meisten Basisdienste bauen auf den bereits entworfenen Schnittstellen zur Datenbank, GIS und Fahrzeugflotte auf. Diese werden durch den *Init* Dienst initialisiert und bereitgestellt. Deshalb benötigen die Basisdienste eine Referenz auf eine Instanz der Init Komponente. Auch zwischen weiteren Modulen können Abhängigkeiten bestehen (z.B. zwischen Auftragsplanung und Straßennetzauswertung). Der Entwurf muß diese Abhängigkeiten berücksichtigen. Die Kopplung zwischen den Modulen sollte so locker wie möglich sein. Je weniger Bezüge zwischen den Komponenten untereinander existieren, um so leichter kann das System wiederverwendet werden. Bei zu engen Abhängigkeiten zwischen Modulen können diese nur zusammen eingesetzt werden. Eine zu enge Kopplung schränkt also die Wiederverwendung einzelner Komponenten ein. Insbesondere muß darauf geachtet werden, daß keine zyklischen Abhängigkeiten zwischen Komponenten bestehen. In einem solchen Fall müssen diese Abhängigkeiten durch geeignete Mittel aufgelöst werden (vergleiche dazu [OOSPEK]).

Abbildung 2.12 zeigt ein vereinfachtes Komponentenmodell der Basisdienste. Daraus sind die Abhängigkeiten erkennbar. Alle Basisdienste haben eine Referenz auf die *Init* Komponente. Diese Beziehung ist nötig, da nur diese Komponente Verbindungen zur Datenschicht (d.h. zur Datenbank, zum GIS-server und zur Fahrzeugflotte) hat. Die Komponenten zur Wegberechnung (*RouteMgmt*) und für Auswertungen (*Report*) können auf die Visualisierungskomponente (*Visual*) zugreifen, damit das Ergebnis geographisch dargestellt werden kann. Des weiteren ist für die Verwaltung von Aufträgen (*JobMgmt*) ein Zugriff auf die *RouteMgmt* Komponente zur Berechnung günstiger Fahrstrecken denkbar. Alle in diesem Modell enthaltenen Abhängigkeiten sollten als Rollenbeziehungen (*has-a* Beziehungen) modelliert werden. Dadurch ist sichergestellt, daß die Kommunikation zwischen den Komponenten nur über deren öffentliche Schnittstelle erfolgt. Gegenüber der Vererbungsbeziehung (*is-a* Beziehung) ist die Kopplung dadurch weniger stark. Eine Wie-

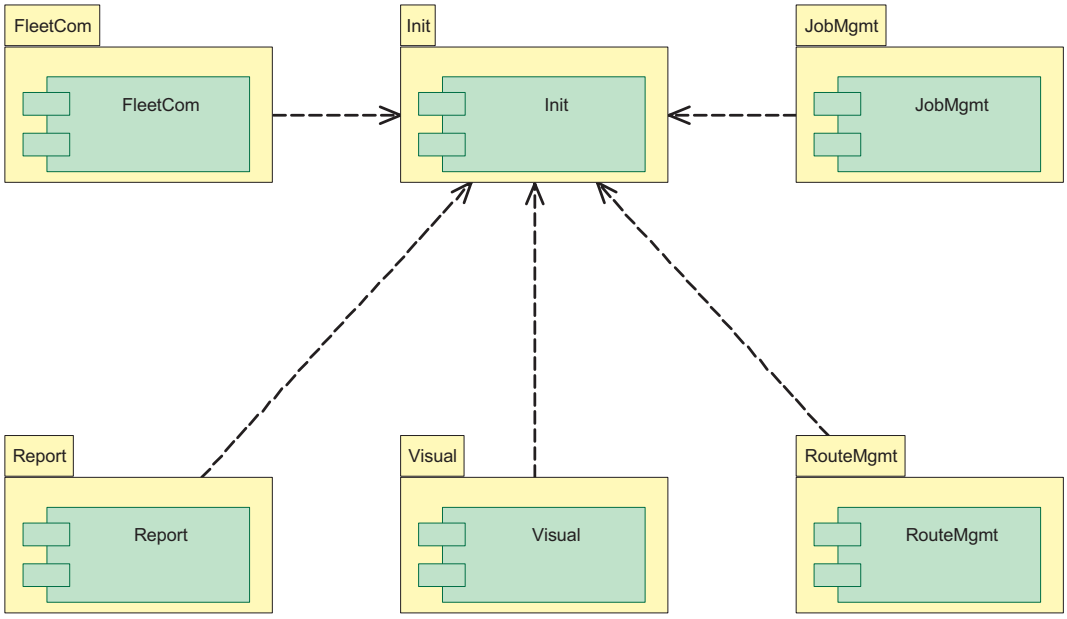


Abbildung 2.12.: mögliche Abhängigkeiten zwischen den Basisdiensten

derverwendung einzelner Komponenten wird besser unterstützt.

Die Verbindungen zwischen den einzelnen Komponenten und der Initialisierungkomponente (*Init*) sind unbedingt nötig. Allerdings sind die anderen Abhängigkeiten optional und sollten besser ganz eliminiert werden. Wenn z.B. das Modul für Reporte direkt auf das Visualisierungsmodul zugreift, wird eine Änderung dieser Komponenten erschwert. Eine Änderung der Visualisierungsschnittstelle hätte direkte Auswirkungen auf andere Module. Ein Weg diese Abhängigkeiten zu eliminieren liegt darin, zwischen den Komponenten nur Datenpakete auszutauschen. Auf diese Weise könnte z.B. das Reportmodul eine Auswertung erzeugen und bereitstellen. Das Ergebnis kann dann vom Visualisierungsmodul ausgewertet und dargestellt werden. Gegenüber der direkten Kommunikation der Komponenten ist diese Kopplung schwächer. Solange das Format der erzeugten und verbrauchten Datenpakete gleichbleibt, können die Komponenten beliebig geändert werden (auch deren Schnittstelle). Damit die erzeugten Daten möglichst auch an Module anderer Hersteller übermittelt werden können, sollte das verwendete Austauschformat ein gängiger Standard sein. Deshalb wird dafür die *Extensible Markup Language - XML* benutzt. Die Details dafür sind allerdings eine Sache der Umsetzung und werden deshalb in Kapitel 3 behandelt (insbesondere Abschnitt 3.2.3). Für den Entwurf interessant ist, daß durch Austausch von klar definierten Datenpaketen, die direkten Abhängigkeiten zwischen den Komponenten vermieden werden kann. Eine Komponente muß nur definieren, welche Datenpakete sie erzeugt und welche sie verbraucht. Es wird keinerlei Wissen über andere Komponenten benötigt (außer natürlich zur Initialisierung). Die dahingegen angepaßte Klassenstruktur ist in Anhang A aufgeführt. Aus diesen kompletten Diagrammen sind auch die einzelnen Leistungen (die angebotenen Methoden) der Komponenten erkennbar.

3. Umsetzung der Anwendungsschicht

Das Ergebnis des vorherigen Kapitels ist eine Klassenarchitektur für die Teilkomponenten einer Flottenmanagementanwendung. Im folgenden soll diese Klassenarchitektur in einer konkreten Programmierumgebung umgesetzt werden. Des weiteren werden verschiedene Technologien und Produkte zur Bereitstellung der Geschäftslogik in der Anwendungsschicht besprochen. Damit soll es möglich sein, daß mehrere Komponenten (eventuell auf verschiedenen Rechnern) zusammenarbeiten, um die Flottenmanagementanwendung zu realisieren.

3.1. Technologien zur Umsetzung der Geschäftsobjekte

Die Umsetzung der Geschäftslogik soll in zwei Schritten erfolgen. In einem ersten Schritt sollen die Geschäftsobjekte umgesetzt und getestet werden. In einem zweiten Schritt sollen diese Geschäftsobjekte dann die Grundlage für die verteilten Komponenten (Basisdienste) des Anwendungsservers bilden. Dies hat den Vorteil, daß die Geschäftslogik auch in anderen Szenarien einsetzbar ist. Des weiteren kann die Anwendungslogik entwickelt und getestet werden, ohne die erhöhte Komplexität einer verteilten Anwendung beachten zu müssen. Allerdings sollte man trotzdem an die angestrebte Architektur denken. Nur die Basisdienste sollen veröffentlicht werden. Für diese Komponenten sollten dann bestimmte Grundsätze der Schnittstelle berücksichtigt werden. Dies bedeutet z.B., daß die typischen Aktionen der Anwendung durch möglichst wenige Methodenaufrufe realisierbar sind (geringere Netzbelastung). Diese Anforderungen sollten beachtet werden, ohne daß gleich von Anfang an eine verteilte Architektur umgesetzt wird. Vielmehr soll im folgenden zuerst eine normale Klassenarchitektur entwickelt werden. Dafür wird im folgenden die Programmierumgebung ausgewählt. Danach werden verschiedene Komponentenmodelle vorgestellt und ein geeignetes ausgewählt. Nachdem diese beiden Entscheidungen getroffen sind, wird mit der Umsetzung der einzelnen Ebenen der Anwendungsschicht (vgl. Abbildung 3.1) begonnen.

3.1.1. Auswahl der Programmiersprache

Die entwickelte Klassenhierarchie läßt sich recht einfach und geradlinig in einer konkreten Programmiersprache umsetzen. In dieser Arbeit wird dazu die Sprache Java benutzt (konkret das JDK[1] 1.1.7). Dies dient der Unterstützung der Portabilität der Gesamtlösung, da diese Sprache einen binären Standard für erzeugte Dateien mit einschließt. Dadurch können damit geschriebene Programme auf allen Plattformen, die einen entsprechenden Interpreter[2] anbieten, ausgeführt werden (das *Write Once - Run Everywhere* Prinzip welches im Zentrum der Java Marketingstrategie steht). Im folgenden sollen einige der interessanteren Aspekte dieser Lösung betrachtet werden.

Eine Einschränkung von Java ist das Verbot von Mehrfachvererbungen. Allerdings gibt es Möglichkeiten, diese bei der Umsetzung zu ersetzen. Dabei gibt es im wesentlichen zwei Strategien. Wenn die Basisklassen wenig Funktionalität anbieten, besteht die Möglichkeit sie als Schnittstellen zu definieren. Eine Javaschnittstelle ist im Prinzip eine abstrakte Klasse (vollständig abstrakt, d.h. es werden keinerlei Methoden implementiert). Eine Subklasse kann jetzt mehrere verschiedene Schnittstellen unterstützen. Dies entspricht einer reinen Schnittstellenvererbung. Wenn die zu erbenden Klassen recht schwer sind, d.h. eine große Funktionalität anbieten, ist es nicht praktikabel, daß jede Subklasse diese Funktionalität erneut bereitstellt. In diesem Fall sollte man eine einzige Umsetzung der Schnittstelle erstellen. Klassen die diese Schnittstelle unterstützen, können dann die eigentliche Berechnung an diese Klasse delegieren. Mit diesen Hilfsmitteln kann also jede beliebige Klassenhierarchie in Java umgesetzt werden.

Allerdings ist dies eigentlich keine große Überraschung. Alle Sprachen der dritten Generation sind gleichmächtig, d.h. es lassen sich im Prinzip alle Sprachen zur Lösung eines Problems benutzen. Jede Sprache hat allerdings ihre Vorteile, die sie für ein bestimmtes Einsatzgebiet prädestiniert. Java ist die Sprache des Internets. Durch die Erzeugung von Bytekode, der dann von einem Interpreter ausgeführt wird, können Klassen leicht auf verschiedene Plattformen übertragen werden. Dies bietet im Internet den Vorteil, daß beliebige Klienten Programmdateien vom Server laden und ausführen können. Darüber hinaus bietet Java viele mächtige Standard- und Erweiterungs API[3]'s. Die Erzeugung von Threads, Benutzung von Socketverbindungen und Zugriff auf andere Netzressourcen sind fest mit der Sprache verbunden. Außerdem gibt es für die verschiedensten Problemstellungen schon frei verfügbare Lösungen. Diese Erweiterungen werden im Zusammenhang mit der sich gerade entwickelnden *Java 2 Enterprise Edition* festen Einzug in Java halten. Dadurch können Java Programme einfach auf Transaktionenssysteme, Mailserver, verschiedenste Dateisysteme u.a. zugreifen (vergleiche [JAVA2EE]). Dies macht die Java 2EE insbesondere für die Umsetzung von Geschäftsanwendungen interessant. Gegenüber anderen Sprachen er-

[1] Java Developement Kit, ein Übersetzer und andere Java Entwicklungswerkzeuge der Firma Sun
[2] die Java Virtual Machine - JVM
[3] Application Interfaces, Programmierschnittstellen für Bibliotheken

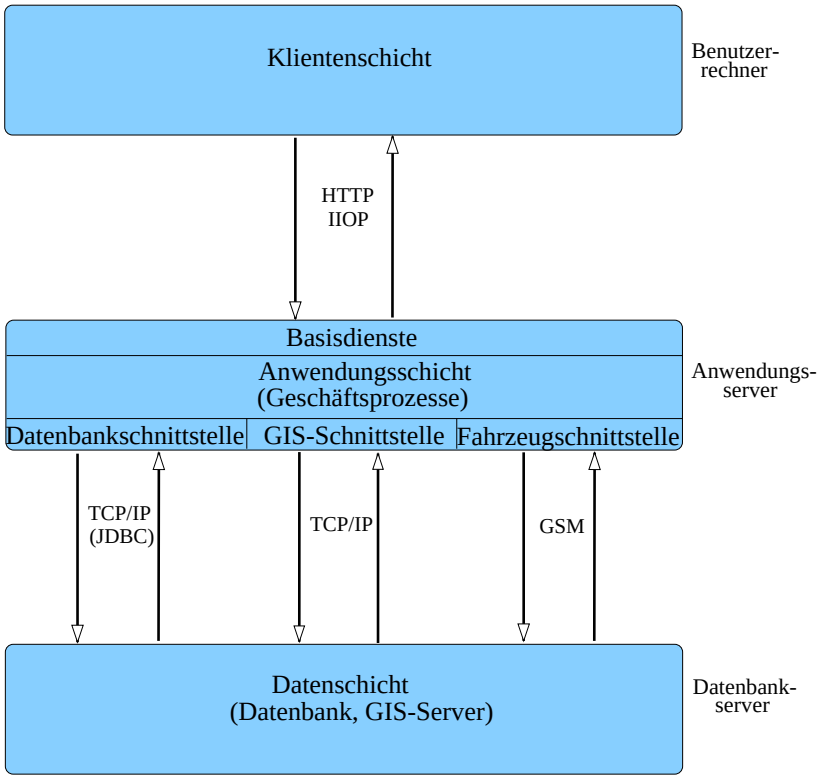


Abbildung 3.1.: Die erweiterte Darstellung der drei Ebenen Architektur

gibt sich eine höhere Produktivität, ohne verschiedene Bibliotheken integrieren zu müssen.

3.1.2. Auswahl der Komponententechnologie für eine verteilte Anwendung

Wenn die in Kapitel 2 entwickelte Klassenhierarchie in Java umgesetzt ist, könnte sie in eine Benutzeroberfläche eingebettet werden. Dadurch wäre die Entwicklung einer Einzelplatzanwendung oder Klienten Server Architektur möglich. Dies ist allerdings nicht das Ziel dieser Arbeit. Vielmehr soll die entworfene Funktionalität dazu benutzt werden, die Anwendungsschicht einer drei Ebenen Architektur umzusetzen. Die dabei denkbaren Ansätze und Technologien werden im folgenden kurz vorgestellt. Danach wird die Umsetzung mit einer gewählten Technologie exemplarisch durchgeführt.

Abbildung 3.1 zeigt eine erweiterte Version der besprochenen Architektur. In dieser Abbildung ist die Anwendungsschicht in weitere Teilschichten unterteilt. Die Klienten greifen auf die Anwendungsschicht über die Basisdienste zu. Die Funktionalität dieser Basisdienste wird durch die Geschäftsprozesse innerhalb der Anwendungsschicht bereitgestellt. Die Geschäftsprozesse können dabei wiederum auf die Dienste der einzelnen Schnittstellen zur Datenschicht zurückgreifen. Der Entwurf der Geschäftsprozesse und der Schnittstellen zwischen Anwendungsschicht und Datenschicht war Gegenstand des 2. Kapitels. Die Details der Umsetzung werden in Abschnitt 3.2 dargestellt. Im folgenden sollen zuerst verschiedene

Technologien zur Umsetzung verteilter Komponentenarchitekturen vorgestellt werden. Die hier hauptsächlich zu besprechenden Lösungen lassen sich grob in zwei Kategorien einteilen - entweder die Microsoft Windows Lösung oder die Vorschläge der OMG. Im Zentrum der Microsoftlösung steht COM[1] bzw. DCOM[2] (die Gesamtarchitektur wird auch *ActiveX*, *COM+* oder *Distributed Network Architecture - DNA* genannt). Dem gegenüber steht die *Common Object Request Broker Architecture - CORBA*. Beide Technologien bieten ähnliche Lösungen für das selbe Problem an. Mit ihnen ist es möglich, Objekte über Prozeß- und Systemgrenzen zu benutzen. Dadurch wird der Entwurf einer verteilten Anwendung aus objektorientierter Sicht unterstützt.

Für komplexe Anwendungen reicht diese reine Objektverwaltung allerdings nicht aus. Im Geschäftsumfeld ist eine hohe Sicherheit, gute Performance und ständige Verfügbarkeit der Lösung wichtig. Dadurch ergeben sich bestimmte Dienste, die jede Anwendung benötigt. Dies betrifft z.B. die Bereitstellung eines Verzeichnisdienstes, persistente Objekte, Transaktionsmanagement und Zugriffssteuerung. Um den Entwickler von diesen immer wiederkehrenden Dingen zu entlasten, gibt es in jedem Lager weiterführende Technologien. Auf der Microsoftseite ist dies der *Microsoft Transaction Server* (MTS). Die OMG bietet dafür die *CORBA Services* (vgl. [OMG98/2]) bzw. die gerade in Entwicklung befindlichen *CORBA Components* (vgl. [OMG99]) an. Darüber hinaus gibt es noch andere (teilweise herstellerabhängige) Lösungen wie z.B. Sun's *Enterprise Java Beans Architektur* (EJB).

Im folgenden werden verschiedene dieser Möglichkeiten vorgestellt und auf ihre Tauglichkeit für die hier zu entwickelnde Anwendung untersucht.

3.1.2.1. COM/DCOM basierende Technologien

Das Microsoft Objektmodell COM definiert binärkompatible Objektschnittstellen. Diese können von verschiedenen Prozessen, mit DCOM auch über Systemgrenzen hinweg, benutzt werden. Dabei spielt es keine Rolle, mit welcher Programmiersprache sie erzeugt bzw. aufgerufen werden.

Das COM lehnt sich stark an dem *Distributed Computing Environment - DCE* der *Open Software Foundation - OSF* an. In diesem Standard wird eine Umgebung zum Zugriff auf entfernte Prozeduren[3] und der Entwicklung verteilter Anwendungen definiert. Wie der Name schon erkennen läßt, ist dieser Ansatz aber auf Prozeduren beschränkt. Deshalb ist er keine Alternative für die hier zu entwickelnde Anwendung. Allerdings sind Kenntnise über diese Spezifikation für ein Verständnis der COM Technologie von Vorteil (vgl. z.B. [OSF98]).

Eine COM-Schnittstelle wird eindeutig durch einen *Globally Unique Identifier - GUID*, einer weltweit eindeutigen 128-bit Integerzahl (basierend auf dem *Universally Unique Identifier-*

[1] Component Object Model
[2] Distributed Component Object Model
[3] Remote Procedure Calls - RPC

UUID der OSF), identifiziert. Zur Definition dieser Schnittstellen wird zumeist eine Schnittstellenbeschreibungssprache (*Interface Definition Language - IDL*) benutzt. In einem auf COM basierenden System kann ein Klient die Dienste (Schnittstellen) eines Servers benutzen (instanziieren). Für Server gibt es verschiedene Möglichkeiten ihre Dienste anzubieten (als sogenannter *in-process* Server oder als lokaler Server). Diese Unterschiede sind aber transparent für den Klienten. Jeder Server stellt eine Klassenfabrik (vgl. Factorypattern) zur Verfügung, mit dessen Hilfe die angebotenen Komponenten abrufbar sind. Um einen Schnittstellenzeiger für diese Klassenfabrik zu erhalten, stehen verschiedene Windows-API Funktionen zur Verfügung. Die Verknüpfung zwischen GUID und dem entsprechenden Server wird in der Systemdatenbank (Registry) hinterlegt.

Nachdem ein Klient einen Zeiger auf die Schnittstelle erhalten hat, kann er deren Methoden benutzen. Die Aufrufe werden dabei über ein Proxyobjekt an den Server weitergeleitet und dort abgearbeitet. Eine gute Einführung in COM und einige darauf aufbauenden Technologien bietet das Buch *Inside OLE* ([BROCK95]). Zu erwähnen bleibt noch, daß Microsoft versucht, diese Technologien auch außerhalb der Windowsplattform zu etablieren. Dazu ist die Weiterentwicklung der ActiveX Technologie 1996 an die Open-Group[1] übergeben worden (vgl. [MIPP]). Diese versucht das Objektmodell auch für andere Betriebssysteme anzubieten. Allerdings ist diese Entwicklung zum jetzigen Zeitpunkt noch in einer experimentellen Phase. Eine erste Vorabversionen einer Umsetzung für verschiedene Unix Betriebssysteme wurde erst Anfang dieses Jahres veröffentlicht.

Aufbauend auf COM gibt es verschiedene weiterführende Technologien. Die erste Erweiterung aus dem Jahre 1996 ermöglicht dabei die Instanziierung von Komponenten auch über eine Netzwerkverbindung. Diese Technologie wird *Distributed Component Object Model - DCOM* genannt. Dazu werden die Parameter und Rückgabewerte der Funktionen in Datenpakete verpackt[2] und dann über das Netzwerk übertragen. Dafür werden sogenannte *Proxy-* und *Stubobjekte* benutzt. Das *Proxyobjekt* nimmt die Anforderungen (Methodenaufrufe) des Klienten entgegen und wandelt sie in entfernte Methodenaufrufe[3] um. Diese Aufrufe werden auf Seiten des Servers durch das *Stubobjekt* entgegengenommen. Dieses leitet den Aufruf dann an die eigentliche Umsetzung weiterleitet.

Im selben Jahr wie DCOM wurde auch der *Microsoft Transaction Server - MTS* vorgestellt. Damit werden weiterführende Dienste, die für große, verteilte Anwendungen wichtig sind, angeboten. Dazu zählt z.B. ein implizites Transaktionsmanagement. In herkömmlichen Transaktionssystemen wird die Laufzeit einer Transaktion während der Anwendungsentwicklung festgelegt. Im MTS kann jeder Methodenaufruf innerhalb einer Transaktion ablaufen. Die Festlegung der Transaktionssicherheit erfolgt auf dem Server. Sie kann während der Laufzeit konfiguriert werden.

[1] Die Open Group ist u.a. auch Nachfolger der OSF

[2] dieser Vorgang wird *Marshalling* genannt

[3] basierend auf dem Remote Procedure Call Protokoll der DCE

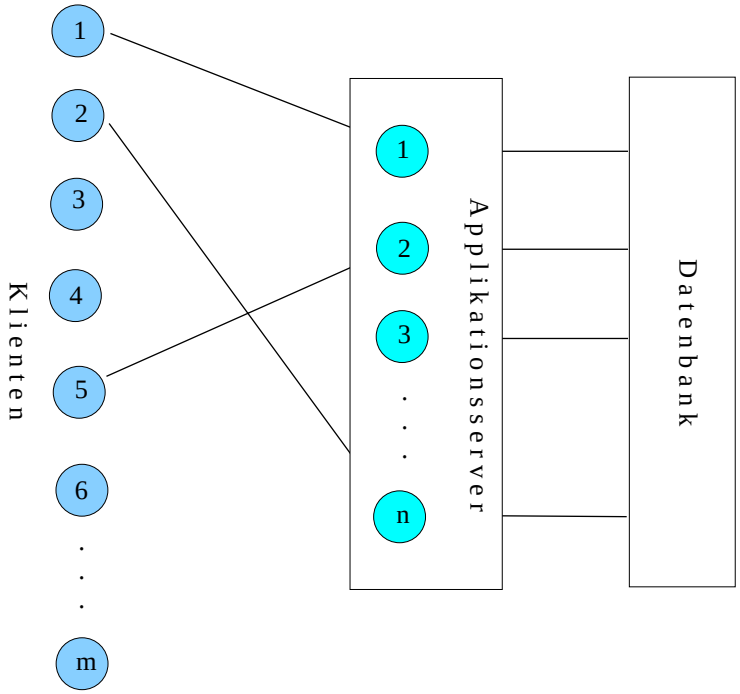


Abbildung 3.2.: Darstellung eines Ressourcenpools für Datenbankverbindungen

Für große Anwendungen, die von vielen Klienten benutzt werden, sind begrenzt verfügbare Ressourcen eine Leistungseinschränkung. Dies betrifft z.B. Datenbankverbindungen, Programmfäden (Threads) und Speicherplatz. MTS versucht diese Ressourcen optimal zu verwalten. Dafür legt er jeweils einen *Pool* für sie an. Eine Komponente kann z.B. eine Datenbankverbindung anfordern. Nach Benutzung wird sie wieder dem Pool zur Verfügung gestellt. Auf diese Weise können mehrere Komponenten sich diese Ressourcen teilen. Außerdem entfallen dadurch die Kosten für die ständige Erzeugung bzw. Zerstörung dieser Ressourcen. Abbildung 3.2 verdeutlicht dieses Prinzip am Beispiel von Datenbankverbindungen. Es gibt m Klienten die sich n Datenbankverbindungen teilen (im allgemeinen n << m). Der Applikationsserver initialisiert die Datenbankverbindungen und kann diese dann den Klienten schnell zur Verfügung stellen. Im dargestellten Szenario benutzt Klient 1 die Datenbankverbindung 1, Klient 2 die Datenbankverbindung n u.s.w.. Dieser Mechanismus beruht auf der Annahme, daß die Klienten die Datenbankverbindungen zu verschiedenen Zeitpunkten benötigen. Im Beispiel benutzt z.B. der Klient 3 keine Datenbankressourcen. In einem herkömmlichen Klienten Server System wäre trotzdem eine Datenbankverbindung durch diesen Klienten blockiert. Insgesamt erreicht man dadurch eine bessere Skalierbarkeit und höhere Verfügbarkeit der Gesamtlösung bei vielen gleichzeitig angemeldeten Benutzern.

Eine Reihe weiterführender Technologien beruhen darauf, daß eine Komponente be-

stimmte Schnittstellen unterstützt. Zum Beispiel definiert die OLE[1] Architektur eine Reihe von Schnittstellen, die das Anzeigen und Bearbeiten von Dokumenten ermöglichen. Objekte die diese Standardschnittstellen anbieten, können dann leicht in andere Dokumente eingefügt und von dort bearbeitet[2] werden. Solche zusammengesetzten Dokumente heißen *compound documents*. Eine neuere Version der OLE Technologie heißt *ActiveX*. Diese erweitert den allgemeinen Ansatz auf das Internet. Dokumente können so über das Netz übertragen werden. Eingebettete Objekte (auch *ActiveX controls* genannt) können bei Bedarf ebenfalls über das Netz übertragen und auf dem Klientenrechner installiert werden.

Ein großer Vorteil dieser Technologien ist zugleich deren größter Nachteil. Alle Ansätze sind eng mit der Firma Microsoft und deren Betriebssystemen verbunden. Dadurch können sie in diesem Umfeld leicht eingesetzt werden. Entwicklungswerkzeuge unterstützen diese Technologien recht gut, es gibt viele Komponenten von Drittanbietern. Allerdings ist es schwierig bis unmöglich eine darauf basierende Lösung auf andere Plattformen zu portieren. Es gibt zwar wie erwähnt Anstrengungen die COM Technologie auch auf anderen Plattformen bereitzustellen. Solange aber Microsoft mit jeder neuen Betriebssystemversion Änderungen und Erweiterungen an diesen Technologien einführt, ist eine vollkommene Portabilität kaum möglich. Unterstützungen für andere Betriebssysteme müssen die Änderungen in der Windowsversion nachahmen. Des weiteren sind die Technologien auch eng mit Windows als zugrundeliegende Plattform verbunden. Für andere Betriebssysteme entstehen so u.U. suboptimale Lösungen. Teile der Windowsarchitektur (z.B. die Systemdatenbank[3]) müssen dort emuliert werden.

Aus den genannten Gründen eignen sich diese Technologien nicht als Grundlage für das hier zu entwickelnde System. Für plattformunabhängige und leicht portierbare Anwendungen sind Objektmodelle ohne direkten Produktbezug besser. Dazu zählt z.B. die im folgenden vorgestellte *Common Object Request Broker Architecture - CORBA*.

3.1.2.2. CORBA basierender Ansatz

Neben den vorgestellten Microsofttechnologien gibt es verschiedene weitere Möglichkeiten für die Umsetzung einer verteilter Komponentenarchitektur. Die einfachste besteht in der direkten Verwendung des TCP/IP Transportprotokolls. Die Komponenten würden dann eigene Datenpakete zusammenstellen, übermitteln und auswerten. Dieser Ansatz ist aber weder objektorientiert noch werden weiterführende Dienste angeboten. Eine darauf basierende Lösung würde also einen großen Aufwand erfordern, um erstmal die benötigte Infrastruktur bereitzustellen. Diese Arbeit haben allerdings schon andere gemacht, so daß es besser ist auf solchen bereitstehenden Lösungen aufzubauen. Eine der ersten dieser Lösungen ist die Benutzung von Remote Procedure Call's, die im Rahmen der erwähnten DCE Architektur

[1] Object Linking and Embedding
[2] dieser Vorgang heißt *in-place-activation*
[3] auch *Registry* genannt

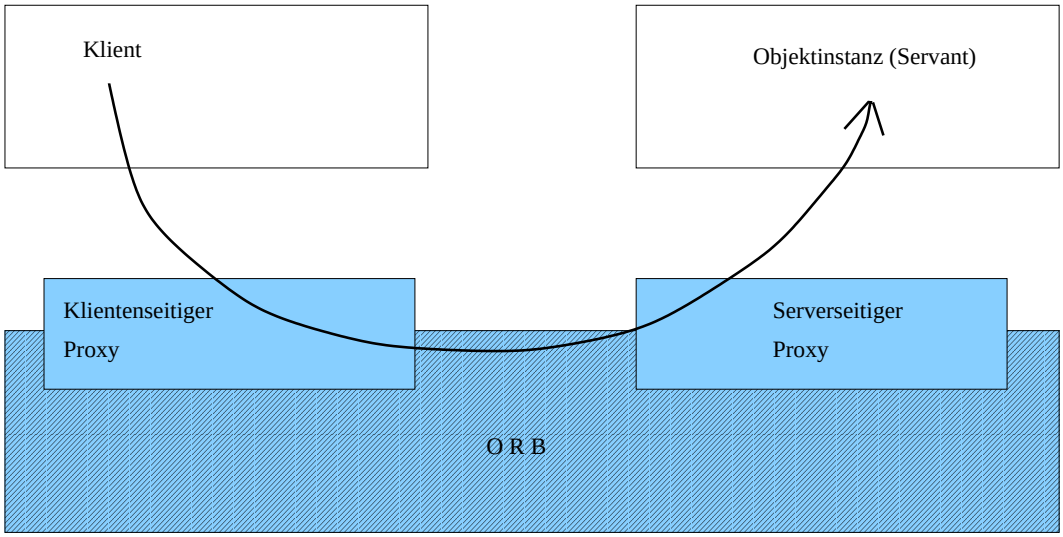


Abbildung 3.3.: Prinzip des Methodenaufrufs im CORBA Standard

definiert sind. Wie der Name schon sagt geht es im wesentlichen darum, wie Prozeduren auf anderen Rechnern benutzt werden können. Diese ersten Ansätze sollen hier aber nicht benutzt werden. Dies ist im wesentlichen damit zu begründen, daß die zu entwickelnde Architektur den Grundsätzen der Objektorientierung entsprechen soll. Deshalb ist es besser, wenn auch die Kommunikation der verteilten Komponenten zwischen Objekten erfolgt. Die zu benutzenden Technologien müssen deshalb nicht nur das Aufrufen entfernter Prozeduren sondern das Benutzen entfernter Objekte unterstützen. Diesen Anspruch erfüllt die Common Object Request Broker Architecture.

CORBA ist ein wichtiger Bestandteil der *Object Management Architecture - OMA* der Object Management Group. Im Zentrum dieser Architektur steht die Kommunikation verteilter Objekte über einen *Object Request Broker*. Dieser ORB ermöglicht die Bereitstellung von Objekten auf einem System. Diese können dann mit anderen Komponenten, auch über die Grenzen des Systems, zusammenarbeiten (vgl. Abbildung 3.3). Methodenaufrufe werden über *klientenseitige Proxyobjekte* an den ORB weitergeleitet. Dieser leitet den Aufruf dann über den *serverseitigen Proxy* an die eigentliche Objektinstanz weiter. Dabei ist denkbar, daß ein Aufruf auch an einen anderen ORB (eventuell auf einem anderen Rechner) weitergeleitet wird. Dabei wird als Protokoll das *Internet Inter ORB Protocol - IIOP* verwendet. Zur Umsetzung der Komponenten können verschiedene Programmierumgebungen eingesetzt werden.

Zur Definition eines Objektes wird seine Schnittstelle benutzt. Die eigentliche Implementierung wird davon losgelöst betrachtet. Die Definition der Schnittstellen erfolgt mittels einer Schnittstellenbeschreibungssprache (IDL). Damit verschiedene Programmiersprachen verwendet werden können, definiert der CORBA Standard eine Abbildung dieser IDL-Schnittstellen auf einige wichtige Programmiersprachen (Java, C++, COBOL, Smalltalk,

ADA). Für andere Sprachen existieren Firmenvorschläge (TCL/TK, Perl, ...). Neben der Festlegung der Programmierkonventionen ist auch die eigentliche Umsetzung der Komponenten zu berücksichtigen. Der Standard muß die Kommunikation zwischen Komponenten genau festlegen. Nur dadurch können ORB's verschiedener Hersteller zusammenarbeiten. Der CORBA Standard definiert dafür ein Übertragungsprotoll, das sogenannte *Internet Inter Orb Protocol - IIOP*. Erst dadurch wird eine offene und gut wiederverwendbare Lösung ermöglicht.

Neben dieser Grundfunktionalität gehört zur OMA auch die Definition einiger weiterführender Dienste. Diese *CORBA Services* erleichtern die Entwicklung robuster, wiederverwendbarer und skalierbarer Lösungen. Es werden u.a. Dienste zur Lokalisation von Objekten, zur Ereignisbehandlung und für Transaktionssysteme definiert (vgl. [OMG98/2]). Dadurch wird der Entwickler von der Umsetzung dieser immer wiederkehrenden Leistungen entlastet. Er kann sich voll auf die Lösung der Probleme konzentrieren mit denen er sich gut auskennt - der eigentlichen Geschäftslogik.

Gegenüber dem Mitbewerber COM fallen die großen Ähnlichkeiten der beiden Systeme auf. Beide Architekturen benutzen ähnliche Mechanismen zum Zugriff auf entfernte Objekte. Aus Sicht des Anwenders scheint es aber einen großen Unterschied zwischen beiden Modellen zu geben. Bei der Benutzung von COM Objekten sieht es so aus, als ob der Server keine Objekte sondern Klassen exportiert. Ein Anwender erzeugt ein lokales Proxyobjekt und kann damit genauso arbeiten wie mit einem lokalen Objekt. Insbesondere mit entsprechenden Entwicklungswerkzeugen, wie z.B. Visual Basic, besteht kein Unterschied zwischen der Benutzung lokaler und entfernter Klassen. Intern stellt der Server natürlich trotzdem nur Objekte zur Verfügung. Die Erzeugung dieser Objekte ist dabei allerdings für den Benutzer versteckt, da eine Klassenfabrik fester Bestandteil eines COM-Servers ist. Demgegenüber beschäftigt sich CORBA nur mit der Übergabe von Objektreferenzen. Es gibt keine Klassen oder Klassenidentifizierer. Die Implementierung einer Klassenfabrik ist optional. Ein CORBA Server stellt ein oder mehrere Objekte zur Verfügung. Ob diese Objekte eine Klassenfabrik zur Verfügung stellen ist kein Bestandteil der Spezifikation[1]. Dadurch gewinnt man eine gewisse Flexibilität. Allerdings wird es für den Anwender komplizierter (er muß sich selber darum kümmern eine Objektreferenz zu bekommen). Diese Komplexität läßt sich jedoch durch entsprechende Werkzeuge verstecken. Sowohl das COM als auch die CORBA bieten ein Objektmodell mit ähnliche Leistungen und Möglichkeiten. Für den Anwender besteht der größte Unterschied bei den verfügbaren Entwicklungswerkzeugen. Während die Komplexität der Entwicklung und Benutzung von COM basierenden Komponenten sehr gut durch Werkzeuge versteckt wird, beschränken sich ihre CORBA Pendants oft auf das Nötigste. Es besteht sicherlich noch ein gewisser Nachholebedarf bei

[1] Dadurch lassen sich sehr leicht CORBA Singleton Objekte umsetzen. Dies kann man erreichen, indem man genau eine Objektreferenz bereitstellt und keine Klassenfabrik zur Verfügung stellt.

der Integration in Entwicklungsumgebungen.

Trotz dieses Nachteiles soll in dieser Arbeit die CORBA als zugrundeliegendes Komponentenmodell eingesetzt werden. Dies läßt sich im wesentlichen damit begründen, daß die Gesamtlösung plattformunabhängig sein soll. CORBA ist von Anfang an als reine Spezifikation ohne direkten Produktbezug entwickelt worden. Dadurch gibt es auch keine Abhängigkeiten zu einer bestimmte Plattform oder Entwicklungsumgebung. Mittlerweile gibt es einige gute Umsetzungen für die verschiedensten Einsatzgebiete. Dadurch ist eine flexible und wertbeständige Lösung möglich. Außerdem lassen sich die Schwierigkeiten bei der Entwicklung durch eigene Hilfsklassen recht gut kapseln. Dadurch können dann weitere Projekte schneller umgesetzt werden. Für die Zukunft bleibt nur zu hoffen, daß die entsprechenden Anbieter an einer besseren Einbindung der CORBA in ihre Produkte arbeiten.

Weiterführende Information zu CORBA, unter anderem auch die aktuellste Version der Spezifikation, kann man unter [OMG98] nachlesen.

3.1.2.3. weitere Ansätze

Neben diesen beiden programmiersprachenneutralen Modellen bzw. Architekturen gibt es noch andere Möglichkeiten. Insbesondere für den Java Entwickler bietet die Firma Sun verschiedene Lösungen an. Als Basisprotokoll zur Kommunikation zwischen Objekten dient die *Remote Method Invocation API - RMI*. Dadurch können Methodenaufrufe zwischen zwei verschiedenen Virtuellen Maschinen (auch über Rechnergrenzen hinweg) ausgelöst werden. Die Parameterübergabe erfolgt dabei als Wert (*call-by-value*). Zur Übermittlung von Objekten dient die *Serialisierung*. Ein Objekt implementiert eine Schnittstelle um anzuzeigen, daß es serialisierbar ist. Dadurch kann es in einen Ausgabestrom (*Outputstream*) geschrieben werden. An einer anderen Stelle kann der Status dieses Objektes dann wieder gelesen und deserialisiert werden. Dies erfolgt alles automatisch, ohne zusätzlichen Programmieraufwand. Objektserialisierung ist ein fester Bestandteil von Java. Dadurch können allerdings auch nur Java Komponenten miteinander kommunizieren. Ein Zugriff mit nicht Java Klienten ist praktisch nicht möglich. Deshalb kommt die RMI API für das hier zu entwerfende Projekt nicht in Frage. Es ist wichtig, nicht an eine bestimmte Programmiersprache oder Komponentenmodell gebunden zu sein. Dadurch können für ein zu lösendes Teilproblem die jeweils besten Werkzeuge verwendet werden.

Neben dieser reinen Kommunikation verteilter Objekte strebt Sun auch die Plazierung kompletter Komponentenmodelle an. Eine der neuesten Entwicklung dabei ist die *Enterprise Java Beans - EJB* Architektur. Dadurch können sehr einfach verteilte Java Komponenten entwickelt werden. Diese können dann, vergleichbar mit dem Microsoft Transaction Server, auf dem Server konfiguriert werden. Dies ermöglicht z.B., daß das Transaktionsverhalten ohne Änderung an den Komponenten beeinflußbar ist. Die Komponenten müssen sich nicht explizit darum kümmern. Allerdings ist diese Technologie noch zu jung, um die Entwicklung genau abschätzen zu können. Da EJB eine reine Spezifikation ist, muß man

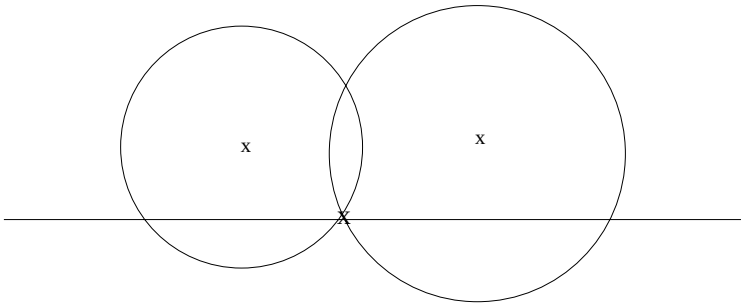


Abbildung 3.4.: GPS Prinzip in der Ebene

abwarten bis entsprechend ausgereifte Produkte zu Verfügung stehen. Im Moment würde deren Einsatz an Java und an einen bestimmten EJB Server gebunden sein. Dies ist für das zu entwerfende Projekt nicht akzeptabel. Allerdings können Enterprise Java Beans in Zukunft ein wichtiger Baustein werden. Jedoch nur, wenn eine Brücke zu anderen Programmiersprachen und Komponentenmodellen geschlagen wird. Eine reine Java Insellösung ist keine echte Konkkurenz zu DCOM und insbesondere auch CORBA.

3.2. Umsetzung der Datenschnittstellen

Ein Klient greift auf die Leistungen des Anwendungsservers über die Basisdienste zu. Die Basisdienste bauen dabei auf einer weiteren Teilschicht auf. Diese bietet verschiedene Schnittstellen zur Kommunikation mit der Datenschicht (vgl. Abbildung 3.1). Die Fahrzeugschnittstelle bietet alle Leistungen, die ausschließlich der Kommunikation mit der Fahrzeugflotte dienen. Genauso greifen die Datenbankschnittstelle und GIS-Schnittstelle genau auf den zugehörigen Teil der Datenschicht zu. Aufgabe der Basisdienste ist es, die angebotenen Leistungen dieser drei Schnittstellen geeignet zu kombinieren, um die geforderten Basisoperationen umzusetzen. Deshalb soll im folgenden zuerst die Umsetzung der grundlegenden Schnittstellen besprochen werden. Danach werden in Abschnitt 3.3 einige anwendungsspezifische Probleme dargestellt und die Basisdienste zu deren Lösung umgesetzt.

3.2.1. Fahrzeugschnittstelle

Die Fahrzeugschnittstelle dient dazu, Daten von verschiedenen Fahrzeugen zu sammeln und anderen Modulen zur Verfügung zu stellen. Die wichtigste Information über ein Fahrzeug ist die Position zu einem bestimmten Zeitpunkt. Diese ist Voraussetzung für alle Auswertungen. Die Positionsdaten werden im Auto durch einen GPS[1]-Empfänger bereitgestellt. Dieses System beruht darauf, daß durch (gleichzeitiges) Messen der Entfernung zu mehre-

[1]Global Positioning System, ein von der US-Regierung zu militärischen Zwecken entwickeltes Satelittennavigationssystem. Dieses steht auch für zivile Anwendungen zur Verfügung.

Nr.	Feld	Beschreibung	Typ	Beispiel
1	$GPRMC	Satz-/Erzeugerkennzeichen		$GPRMC
2	POS_UTC	Zeit	hhmmss	154232
3	POS_STAT	Status	A - kein Fehler V - Fehler	A
4	LAT	geographische Breite	ggmm.mmmm	2758.612
5	LAT_REF	Hemisphäre	N - Nord S - Süd	N
6	LON	geographische Länge	gggmm.mmmm	08210.515
7	LON_REF	Hemisphäre	W - West E - Ost	W
8	SPD	Geschwindigkeit	xxx.x	085.4
9	HDG	Richtung	xxx.x	084.4
10	DATE	Datum	ddmmyy	230394
11	MAG_VAR	Magnetische Abweichung	xxx.x	003.1
12	MAG_REF	Hemisphäre	W - West E - Ost	W

Tabelle 3.1.: Aufbau eines RMC Datensatzes des NMEA Standards

ren Satelliten die Position berechnet werden kann. Abbildung 3.4 stellt (sehr vereinfacht) das Prinzip in der Ebene dar. Durch Kenntnis der Entfernung von zwei Punkten kann genau ein Punkt auf der Geraden bestimmt werden. Für eine Position im Raum reicht im Prinzip die Entfernung zu drei Referenzpunkten.

Der eigentliche Aufbau eines solchen Systems übersteigt den Rahmen dieser Arbeit bei weiten. Außerdem existieren schon recht gute Arbeiten zu diesem Thema, z.B. [GPS]. Des weiteren beschäftigt sich die Diplomarbeit von Andreas Bernklau eingehender mit der technischen Seite der mobilen Übertragung und Auswertung von Positionsdaten ([BERN97]).

Allerdings kann man ein GPS verwenden, ohne die genauen internen Abläufe zu kennen. Es gibt verschiedene Systeme auf dem Markt, die verwendet werden können. Diese Geräte werden meistens an die serielle Schnittstelle angebunden und können dann darüber angesprochen werden. Die Kommunikation erfolgt dabei fast immer mit den Protokollen des NMEA0183 Standards (vgl. [NMEA0183]). Das Kernstück bildet dabei die Definition verschiedener Datensätze (als Zeichenketten) die zwischen den Geräten ausgetauscht werden. Dadurch kann z.B. ein solcher NMEA-kompatibler GPS-Empfänger aufgefordert werden, die aktuelle Position zu ermitteln. Diese liefert er dann als einen NMEA-Datensatz zurück. Dabei sind viele verschiedene Datensätze für die verschiedensten Datenarten definiert. Im Zusammenhang mit dieser Arbeit sind aber nur die Positionsdaten des GPS-Empfängers von Interesse. Dazu werden RMC[1]-Datensätze übermittelt und ausgewertet. Ein NMEA Datensatz besteht aus einer Liste von Datenfeldern. Diese werden durch Komma voneinander getrennt. Das erste Datenfeld bestimmt den Typ des Datensatzes. Am Ende eines Datensatzes kann noch eine Prüfsumme folgen (durch einen Stern * vom letzten Datensatz getrennt).

Der Aufbau eines RMC-Datensatzes ist in Tabelle 3.1 aufgeführt. Die wichtigsten In-

[1]Recommend Minimum Specific GPS/TRANSIT Data

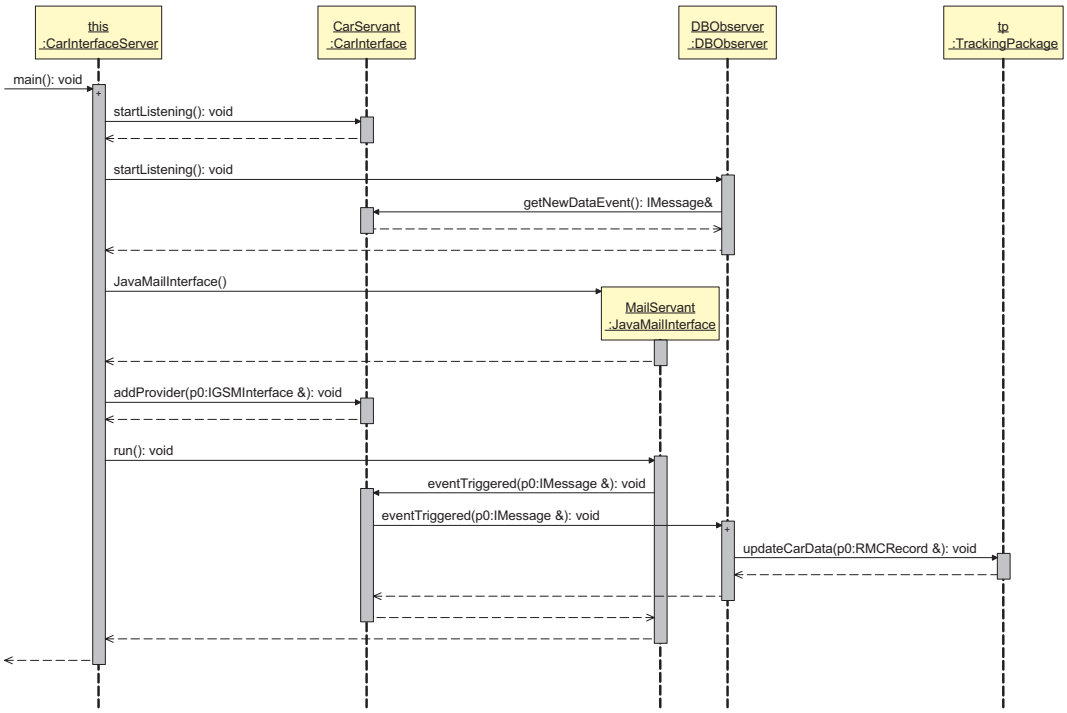


Abbildung 3.5.: Sequenzdiagramm für den Flottenserver

formationen sind die Positionsdaten und die Zeit/Datumsangaben. Dieser Datensatz wird direkt vom GSP-Empfänger erzeugt. Wenn er so direkt an die Zentrale übermittelt werden würde, wäre keine Zuordnung zu einem bestimmten Fahrzeug möglich. Deshalb wird dem Datensatz als Fahrzeugkennzeichen die Telefonnummer des Mobiltelefons vorangestellt. In der Zentrale muß dann eine Zuordnung dieser Telefonnummer zu einem Fahrzeug möglich sein. Dies wird durch die Fahrzeugdaten in der Datenbank ermöglicht (vgl. Abschnitt 3.2.3).

Die Fahrzeugschnittstelle übernimmt diese Daten und wertet Sie entsprechend aus. Normalerweise werden die Daten nur in einer Datenbank abgelegt. Dank dem eingebauten Beobachtermechanismus kann aber jede Komponente diese Daten empfangen und direkt auswerten. Ein Szenario zur Auswertung von Fahrzeugdaten ist im Diagramm aus Abbildung 3.5 zu sehen.

Zuerst werden die entsprechenden Schnittstellenobjekte initialisiert (im Diagramm sind im Interesse der Übersichtlichkeit einige unwesentliche Methodenaufrufe, z.B. Konstruktoren, nicht aufgeführt). Die Schnittstellenobjekte sind der eigentliche Fahrzeugserver (*CarServant*) und der Datenbankobserver (*DBObserver*). Der Datenbankobserver initialisiert für den Zugriff auf die Datenbank eine Instanz des Datenbank-Trackingpaketes (*tp*, vergleiche Abschnitt 3.2.3 für die Umsetzung der Datenbankschnittstelle). Außerdem meldet er sich an dem Ereignis der Fahrzeugschnittstelle für neue Daten an (stellvertretend durch *getNewDataEvent* dargestellt). Jetzt wird ein Datenlieferant initialisiert. Im dargestellten Fall

Paket	Methode	Beschreibung
StreetNetPackage	getNearestPointOnNet	liefert den nächstgelegenen Punkt auf dem Straßennetz
	getPositionFromPointOnNet	liefert für einen Punkt die analytische Beschreibung der Position
InteractorPackage	setCentre	setzt den Mittelpunkt des dargestellten Kartenausschnittes
	zoomBy	verändert die Größe des dargestellten Kartenausschnittes
	panBy	verschiebt die dargestellte Karte in 8 mögliche Richtungen (N,NE,E,SE,S,SW,W,NE)
ThemaPackage	visualize	dient zur Visualisierung einer Punktwolke

Tabelle 3.2.: Die Pakete der GIS Schnittstelle

sollen die Daten von einem e-mail Server gelesen werden (*JavaMailInterface*). Nachdem dieser Datenlieferant erzeugt ist, muß er der Fahrzeugschnittstelle bekanntgegeben werden. Dies geschieht durch den Aufruf der Methode *addProvider*. Danach wird er gestartet (in einem eigenen Programmfaden). Wenn der e-mail Server neue Daten erhält, wird dies dem Flottenserver durch Auslösen des entsprechenden Ereignisses bekanntgegeben (*eventTriggered*). Dieser leitet dieses Ereignis an die angemeldeten Beobachter weiter. Das betrifft im dargestellten Szenario den Datenbankobserver. Nachdem das Ereignis dort eingetroffen ist, benutzt dieser die Datenbankschnittstelle um die neuen Daten zu speichern (*updateCarData*).

Das dargestellte Szenario zeigt, wie flexibel die Fahrzeugschnittstelle ist. Dank des umgesetzten Beobachterentwurfsmuster können beliebig viele Datenlieferanten und -verbraucher eingebaut werden. Dies kann jederzeit während der Laufzeit eingestellt werden. Mit der Flottenschnittstelle steht eine Möglichkeit zur Verfügung, die Fahrzeugdaten zu sammeln und abzulegen. Im folgenden werden die Schnittstellen betrachtet, mit denen diese Daten abgerufen und dargestellt werden können.

3.2.2. Geographieschnittstelle

Bei einer GIS Anwendung in Klienten Server Architektur wird meist eine Visualisierungskomponente direkt in die Anwendungsoberfläche eingebunden. Diese ist für die Manipulation und Darstellung der Karte zuständig. Diese Komponente muß dafür natürlich auf jedem Klienten installiert sein. Bei einer drei Schichten Architektur ist dieser Ansatz nicht möglich. Ein Klient sollte möglichst wenig eigene Funktionalität haben. Die Installation einer zusätzlichen Bibliothek ist nicht immer praktikabel. Deshalb muß ein anderer Weg für die Einbindung einer GIS Komponente gewählt werden. Diese muß auf dem Server

laufen. Das bedeutet, daß ausgehend von dem Kartenmaterial und den getroffenen Einstellungen der aktuelle Ausschnitt in einem gängigen Format erzeugt werden muß. Dieses kann dann vom Klienten gelesen und dargestellt werden. Bei der hier verwendeten GIS Komponente werden Bilder im GIF Format erzeugt. Diese können dann z.B. direkt in eine HTML Seite eingebunden werden (vgl. Kapitel 4). Durch Bereitstellung der Bilder zum Abrufen über HTTP können die Daten auch in andere Anwendungen eingebunden werden. Die GIS-Schnittstelle ist jetzt nur eine dünne Schicht, um die Benutzeranfragen an die jeweilige herstellerabhängige Schnittstelle weiterzuleiten. Dies verringert die Kopplung der Anwendung an eine proprietäre Schnittstelle. Dadurch kann die GIS Komponente mit wenig Aufwand ausgetauscht werden.

Die GIS-Schnittstelle bietet die grundlegenden Dienste zur Visualisierung der Daten an. Neben den Basisoperationen bietet sie dabei verschiedene Pakete an. Welche Pakete angeboten werden, sowie deren genaue Ausprägung und die angebotenen Methoden, sind aus dem Klassendiagramm in Anhang A.2 bzw. der Tabelle 3.2 ersichtlich.

3.2.3. Datenbankschnittstelle

Im folgenden wird die Umsetzung der entworfenen Datenbankschnittstelle besprochen. Dabei sind insbesondere folgende Punkte von Interesse:

- direkte Schnittstelle zur Datenbank (wie lege ich Daten ab bzw. frage ich sie ab ?)
- Datenbankstruktur (wo werden die Daten abgelegt ?)
- XML[1](wie werden die Daten an andere Komponenten übergeben ?)
- Zugriff auf eine relationale Datenbank aus einer objektorientierten Sicht

Diese Punkte werden jetzt der Reihe nach behandelt.

Datenbank API

Zur Datenspeicherung soll eine relationale Datenbank verwendet werden (vgl. Abschnitt 4.3.2). Der Zugriff auf ein solches System erfolgt durch die *Structured Query Language - SQL* (vgl. z.B. [SQL99]). Dadurch können verschiedene relationale Datenbanken über eine einheitliche Schnittstelle angesprochen werden. Neben diesem Standard SQL gibt es aber einige weiterführende Operationen die jede der zu unterstützenden Datenbanken anbietet (z.B. für Typumwandlungen). Allerdings bestehen Unterschiede in der Syntax dieser Erweiterungen. Dies ist in der Klassenstruktur für die Datenbankschnittstelle berücksichtigt (vgl. Abschnitt 2.2.2 bzw. Anhang A). Die produktspezifischen Anteile werden in der

[1]Extensible Markup Language, eine Metasprache mit der verschiedenste Daten in einer einheitlichen Form abgelegt werden können

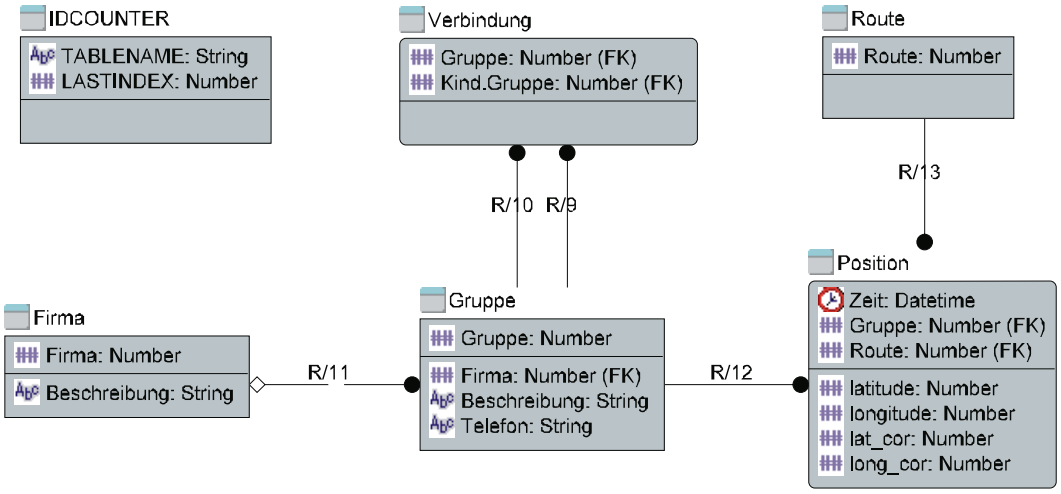


Abbildung 3.6.: Struktur der Flottenmanagementdatenbank

Datenbankbasis gekapselt (für jede unterstützte Plattform muß also eine eigene Datenbankbasis erstellt werden). Dadurch wird die anwendungsspezifische Funktionalität der Datenbankpakete von der zugrundeliegenden Plattform getrennt. Bei der zu benutzenden Sprache Java gibt es die *Java Database Connectivity API - JDBC* (die aktuelle Spezifikation und Informationen dazu sind unter [JDBC] zu finden). Über diese Schnittstelle können die SQL Anweisungen an die Datenbank übergeben und die Ergebnisse abgerufen werden. Dadurch wird es möglich, einen Großteil der Funktionalität in einer abstrakten JDBC Datenbankbasis umzusetzen. Die Umsetzung einer konkreten Basis für eine bestimmte Datenbank kann dadurch recht schnell erfolgen. Es muß nur das absolute Minimum an Funktionalität neu definiert werden (vgl. Anhang A.3).

Datenbankstruktur

Neben dieser reinen Zugriffsmöglichkeit ist die Struktur für die Datenablage wichtig. Die minimale Datenbankstruktur für die Flottenmanagementanwendung ist in Abbildung 3.6 dargestellt. Diese ermöglicht eine hierarchische Ablage der Fahrzeugdaten. Zentraler Bestandteil dabei ist die Tabelle *GRUPPE*. Diese bildet zusammen mit der Tabelle *VERBINDUNG* eine Baumstruktur der Fahrzeugflotte. Eine Gruppe ohne Untergruppen (ohne Kinder) ist ein Fahrzeug. Zu einer Gruppe gibt es verschiedene zugeordnete Daten. Dies betrifft einerseits Verwaltungsdaten (wie die zugeordnete Firma) oder auch Positionsdaten (Tabellen *FIRMA* und *POSITION*). Positionsdaten wiederum werden in verschiedene *ROUTEN* aufgeteilt. Die Tabelle *IDCOUNTER* dient Verwaltungsaufgaben bei der Erzeugung der benötigten Primärschlüssel. Diese Struktur definiert die minimal nötigen Daten. Sie stellt somit den Kern einer Datenstruktur für eine Flottenmanagementanwendung dar. Bei einem kommerziellen Einsatz sind noch einige weitere Daten zu erheben. Dies soll aber bei der hier zu untersuchenden Beispielarchitektur vernachlässigt werden.

Extensible Markup Language (XML)

Für die Speicherung und Übertragung von Daten muß ein Datenformat festgelegt werden. Damit Daten von unterschiedlicher Herkunft bearbeitet und eingebunden werden können, ist ein standardisiertes Format von Vorteil. Dieser Standard muß aber flexibel genug sein, um an viele verschiedene Anwendungsfälle angepaßt werden zu können. Zur Lösung dieses Problems schlägt das *World Wide Web Consortium - W3C* die *Extensible Markup Language - XML* vor (vgl. [XML10]). XML ist eine Metasprache mit der jeder Anwender seine eigenen Dokumentenstruktur umsetzen kann. Diese Definition erfolgt allerdings in einer einheitlichen Form, so daß eine automatische Verarbeitung unterstützt wird. Durch diese einheitliche Notation können verschiedene Bibliotheken den Anwender bei dem Einsatz von XML unterstützen. Neben der Basisunterstützung durch einen Parser[1] sind vor allem die Möglichkeiten zur Änderung der Struktur wichtig. Dafür werden vom W3C zur Zeit die *Extensible Stylesheet Transformation (XSLT)* Anforderungen definiert. Dadurch können XML Dokumente mit einer bestimmten Struktur durch Einsatz von sogenannten *Stylesheets* in andere XML Dokumente transformiert werden (siehe [XSLT]).

Wie sieht das jetzt in der Praxis aus ? Die Datenbankschnittstelle stellt das Ergebnis einer Abfrage in einem XML Dokument zur Verfügung. Zur Anzeige innerhalb eines Browsers könnte jetzt mit Hilfe eines entsprechenden Stylesheets eine XML konforme HTML Datei (Stichwort: XHTML) erzeugt werden. Die selbe XML Datei könnte aber auch an einen Reportgenerator oder eine weiterführende Auswertung übergeben werden. Die Datensicht kann einmal erzeugt und dann beliebig weiterverarbeitet werden. Durch ein einheitliches Datenformat gibt es weniger Probleme bei der Einbindung von Fremdkomponenten, z.B. zum Erzeugen von Reporten (ggf. ist vorher höchstens eine Transformation nötig).

Objekt-Relationale Abbildung

Die bisher besprochenen Komponenten stellen die Basisunterstützung für einen Zugriff auf die Datenbank dar. Sie sind auf der untersten Ebene angesiedelt. Dadurch erreicht man größtmögliche Flexibilität allerdings auf Kosten der Einfachheit. Beim Einsatz dieser Schnittstelle muß ein Anwender:

- den Sql Befehl zusammenbauen,
- die Datenbankverbindung öffnen und den Sql Befehl ausführen,
- die Ergebnismenge aufzählen,
- Daten für jede Spalte einlesen und konvertieren sowie

[1] ein Programm, welches einen Datenstrom untersucht und entsprechen aufbereitet, sodaß ein strukturierte Zugriff möglich wird

- die Datenbankverbindung schließen und benutzte Ressourcen freigeben

Dies ist ein großer Aufwand, der zudem zu einer unübersichtlichen Programmstruktur führt. Bei einer objektorientierten Anwendung möchte man eigentlich nicht auf einzelne Zeilen und Spalten einer Ergebnismenge zugreifen. Vielmehr sollte es möglich sein, Abfragen der Form “gib mir alle Objekte der Klasse X” zu stellen. Da aber die Randbedingungen der Anwendung eine relationale Datenbank erfordern (vgl. Abschnitt 4.3.2), muß eine neue Abstraktionsebene eingeführt werden. Aufbauend auf der bisher entwickelten Datenbankschnittstelle ermöglicht diese Schicht einen objektorientierten Zugriff auf die Daten. Im folgenden wird die Umsetzung dieser Schicht bei der Flottenmanagementanwendung kurz vorgestellt.

Die zu entwerfende Schicht soll folgenden Anforderungen genügen:

- Darstellung einer Zeile einer Relation als Objekt
- Lesen der Daten und Zugriff über Methoden und Attribute der ORE[1]Hüllobjekte
- Berücksichtigung von Fremdschlüsseln als ORE Hüllobjekt Attribute
- Möglichkeiten für Insert, Select und Update mit objektorientierten Mitteln (ohne Sql-Kenntnisse)

Das Lesen von mehreren Datensätzen und Ablegen dieser in geeigneten Datenstrukturen soll nicht unterstützt werden. Vielmehr sollen in diesem Fall Möglichkeiten geschaffen werden, eine Ergebnismenge der Abfrage zeilenweise abzufragen. Jeweils eine Zeile wird dann als ORE Hüllobjekt dargestellt. Ein komplettes Lesen der Ergebnismenge in den Hauptspeicher erscheint nicht sinnvoll. Ausgehend von diesen Anforderungen ist folgende Lösung umgesetzt worden:

- Es gibt eine Schnittstelle die alle ORE Hüllobjekte unterstützen müssen. Diese dient zur Abfrage der verfügbaren Spalten und Lesen der Daten
- Zusätzlich gibt es eine abstrakte Basisklasse, die diese Funktionalität für einfache Hüllobjekte bereitstellt. Einfache Hüllobjekte sind dabei Objekte, die genau einer Datenbanktabelle zugeordnet sind.
- Letztendlich gibt es noch eine allgemeine Hüllobjektumsetzung, die beliebige Zeilen einer Ergebnismenge repräsentieren kann.

Diese verbale Beschreibung ist im Klassendiagramm aus Abbildung 3.7 dargestellt. *IOREWrapper* ist die Basisschnittstelle. *OREWrapperBase* und *OREWrapper* stellen jeweils verschiedene Basisumsetzungen für ORE Objekte bereit. Von der Klasse *OREWrapper*

[1]Objekt Relationalen

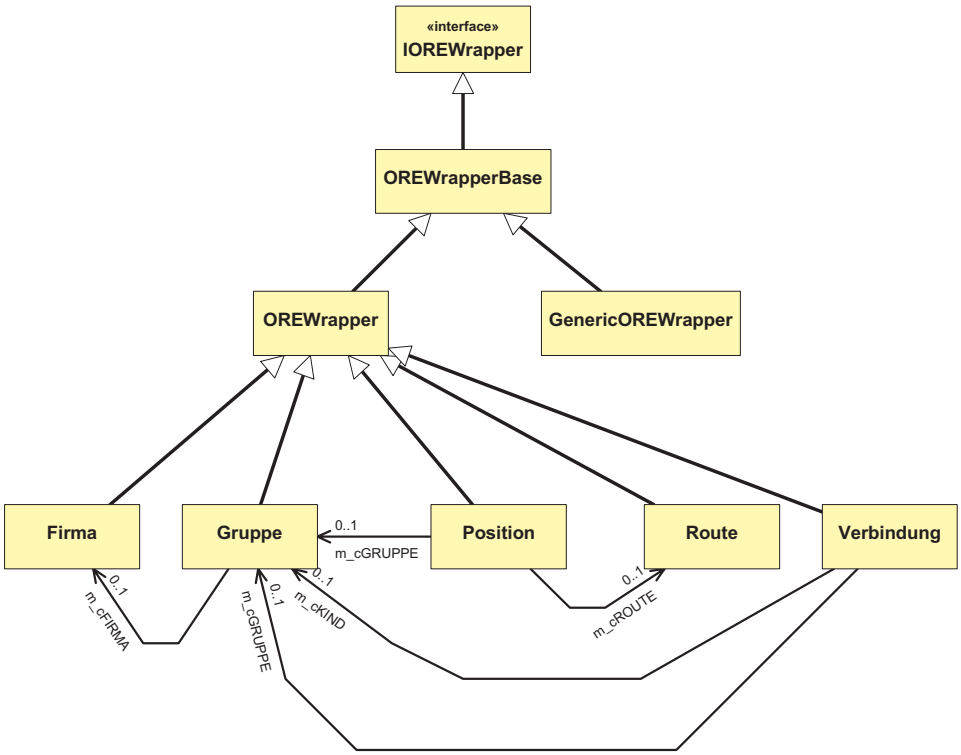


Abbildung 3.7.: Klassenstruktur für die ORE Schicht der Flottenmanagementanwendung

erben jeweils die konkreten Klassen (eine pro Datenbanktabelle). Die Klasse *GenericORE-Wrapper* dient dazu, Relationen mit einzubeziehen, die nicht direkt einer Datenbanktabelle zugeordnet werden können. Dies betrifft z.B. die Ergebnismenge von Abfragen über mehr als eine Tabelle. Das komplette Klassendiagramm mit allen öffentlichen Methoden ist im Anhang A.3.1 aufgeführt.

Aufbauend auf dieser grundlegenden Infrastruktur bietet die Datenbankschnittstelle verschiedene Reporte und einfache Abfragen an. Die einfachen Abfragen sind z.B. *liefere mir die letzte bekannte Position des Fahrzeuges x.* Reporte stellen umfangreichere Abfragen dar. Sie sind eine Erweiterung der *DBOperation* Klasse aus dem Klassendiagramm in Abbildung 2.10. Einer dieser Reporte liefert z.B. eine bestimmte Fahrstrecke für ein Fahrzeug. Alle angebotenen Leistungen und Methoden sind aus dem kompletten Klassendiagramm der Datenbankschnittstelle in Anhang A.3 ersichtlich.

3.3. anwendungsspezifische Probleme

Die bisher besprochenen Komponenten stellen die Grundlage für eine Datenbank und GIS Anwendung mit mobiler Kommunikation dar. Sie sind größtenteils noch nicht speziell für eine Flottenmanagementanwendung. Diese applikationsabhängigen Komponenten sollen im folgenden behandelt werden. Dafür werden zuerst einige allgemeine Probleme und

Lösungsvorschläge besprochen. Danach wird die Umsetzung der einzelnen Basisdienste betrachtet. In diesem Zusammenhang wird danach auch die Anbindung an andere Komponentenmodelle untersucht (Abschnitt 3.5). Dadurch können bereits vorhandene Vorarbeiten (z.B. zur Berechnung von optimalen Fahrstrecken) benutzt werden.

3.3.1. Bereitstellung des Kartenmaterials

Zentraler Bestandteil für eine Flottenmanagementanwendung sind die benötigten Daten. Die richtige Erfassung und Aufbereitung erfordert mehr Aufwand, als die Entwicklung der Softwarearchitektur. Den Hauptteil dabei bilden die Straßendaten. Diese werden einerseits für die Visualisierung aber auch für die Analyse von Fahrstrecken benötigt. Dabei sind nicht nur die digitalisierten Straßenverläufe sondern auch bestimmte Attribute (Straßenausbau, Breite, Steigungen, besondere Hindernisse) von Interesse. Diese Daten zu erfassen und aktuell zu halten ist ein großer Aufwand. Allerdings gibt es dieses Kartenmaterial bereits fertig zu erwerben.

Für diese Arbeit sollen Daten der Firma Teleatlas Verwendung finden. Diese werden im *Geographic Data File - GDF* Format geliefert. Dieses definiert ein weltweit anerkanntes Format, welches u.a. auch von der ISO übernommen wird (vgl. [GDF30]). In einer GDF Datei werden drei verschiedene geographische Grundobjekte abgelegt: Punktwolken, Linienzüge und Polygongebiete. Darüber hinaus können sogenannte komplexe Objekte aus diesen Grundtypen gebildet werden. Neben diesen rein geographischen Informationen gibt es auch die Möglichkeit, Attribute und Relationen abzulegen. Attribute sind an ein bestimmtes Objekt gebunden. Für einen Straßenzug sind z.B. häufig anzutreffende Attribute die Straßenklasse (*Route Number - RN*) und der Name (*Official Name - ON*). Relationen definieren Beziehungen zwischen Objekten. Um eine realistische Navigation auf dem Straßennetz zu ermöglichen, müssen z.B. die Relationen zur Festlegung möglicher Fahrtrichtungen ausgewertet werden. Dabei wird zwischen verbotenen Teilstrecken (*prohibited manouvers*) und vorgeschriebenen Teilstrecken (*restricted manouvers*) unterschieden. Die Notation erfolgt dabei als Relation zwischen Streckenabschnitten und Knoten.

Im GDF Format sind viele verschiedene Attribute und Relationen definiert. Damit können Straßendaten für die verschiedensten Anwendungsgebiete definiert werden. Ein Nutzer wird sich von einem Anbieter genau das Paket auswählen, daß die für ihn interessanten Daten enthält. Die für eine Flottenmanagementanwendung interessanten Attribute und Relationen sind in Tabelle 3.3 aufgeführt. Dies sind allerdings nur die wichtigsten. Je nach Anwendungsfall sind auch andere Attribute interessant (z.B. für Geschwindigkeitsbeschränkungen oder Durchschnittsgeschwindigkeiten).

Das GDF Format ist für eine kompakte und standardisierte Übermittlung geeignet. Für eine Visualisierung und Auswertung ist dies allerdings nicht unbedingt optimal. Hierfür ist eine vorherige Umwandlung in andere Formate besser. Für eine Visualisierung mit der benutzten Kartierungssoftware (basierend auf einem Produkt der Firma Mapinfo, vgl. Ab-

Kennzeichen	Name	Beschreibung
ON	Official Name	der offizielle Name für dieses Objekt
RN	Route Number	Straßennummerierung und -klassifizierung des jeweiligen Landes (z.B. A4, B9, ...)
FC	Functional Road Class	Straßenklassifizierung (0...9)
BP	Blocked Passage	Durchfahrt gesperrt (ja/nein)
DF	Direction of Traffic Flow	Fahrtrichtung (z.B. für Einbahnstraßen wichtig)
VT	Vehicle Type	zugelassenen Fahrzeugtypen
2102	Restricted Maneouvre	vorgeschriebene Fahrtrichtung
2103	Prohibited Maneouvre	verbotene Fahrtrichtung

Tabelle 3.3.: minimale GDF Attribute und Relationen für das Flottenmanagement

schnitt 4.3.2) müssen die Daten in das proprietäre Mapinfo Format konvertiert werden. Der Austausch erfolgt dabei über das *Mapinfo Interchange Format - MIF*. Der Konverter nimmt die GDF Daten als Eingabe und erzeugt MIF Daten als Ausgabe. Diese können dann in Mapinfo importiert werden. Die eigentliche Umsetzung dieses Konverters soll hier nicht weiter berücksichtigt werden. Für diese Arbeit soll er als externes Produkt betrachtet werden.

3.3.2. Abbildung der GPS-Koordinaten auf ein gegebenes Straßennetz

Zum jetzigen Zeitpunkt stehen also die Daten für das Straßennetz und die Positionsdaten der einzelnen Fahrzeuge zur Verfügung. Was noch fehlt, ist eine Abbildung der Positionen auf das Straßennetz. Nur dadurch kann eine sinnvolle Auswertung und Visualisierung der Fahrzeugdaten erfolgen. Diese Abbildung ist leider nicht trivial. Es gibt verschiedene Probleme zu beachten. Diese entstehen z.B. durch fehlerbehaftete Daten bei der Digitalisierung des Kartenmaterials oder bei den ermittelten GPS Koordinaten. Bei dem Kartenmaterial kann man nur darauf achten, daß die Qualität des Anbieters möglichst gut ist und eine regelmäßige Aktualisierung der Daten erfolgt. Bei den GPS Koordinaten ist eine Abweichung von mehreren hundert Metern durch die benutzte Technik bedingt. Der Einsatz eines Differentiellen GPS (DGPS) Empfängers kann diesen Fehler zwar drastisch verringern (vgl. [BERN97]), allerdings zu Ungunsten der entstehenden Kosten.

Deshalb ist ein Verfahren zu finden, daß auch bei großen Ungenauigkeiten der Positionsdaten deren wahrscheinlichste Abbildung auf das Straßennetz findet. Eine exakte Abbildung ist bei diesen Randbedingungen nicht zu gewährleisten. Dies ist insbesondere im Stadtbereich, mit vielen dicht beieinander liegenden Straßen und Kreuzungen, einsichtig. Allerdings wird sich zeigen, daß trotzdem recht gute Abbildungsergebnisse erreichbar sind. Zur Abbildung eines einzelnen Punktes auf das Straßennetz stehen nur wenige Informationen zur Verfügung. Die sinnvollste Wahl ist hier, den nächstgelegenen Punkt auf dem Straßennetz auszuwählen. Ein möglicher Algorithmus hierfür ist in [BERN97] untersucht

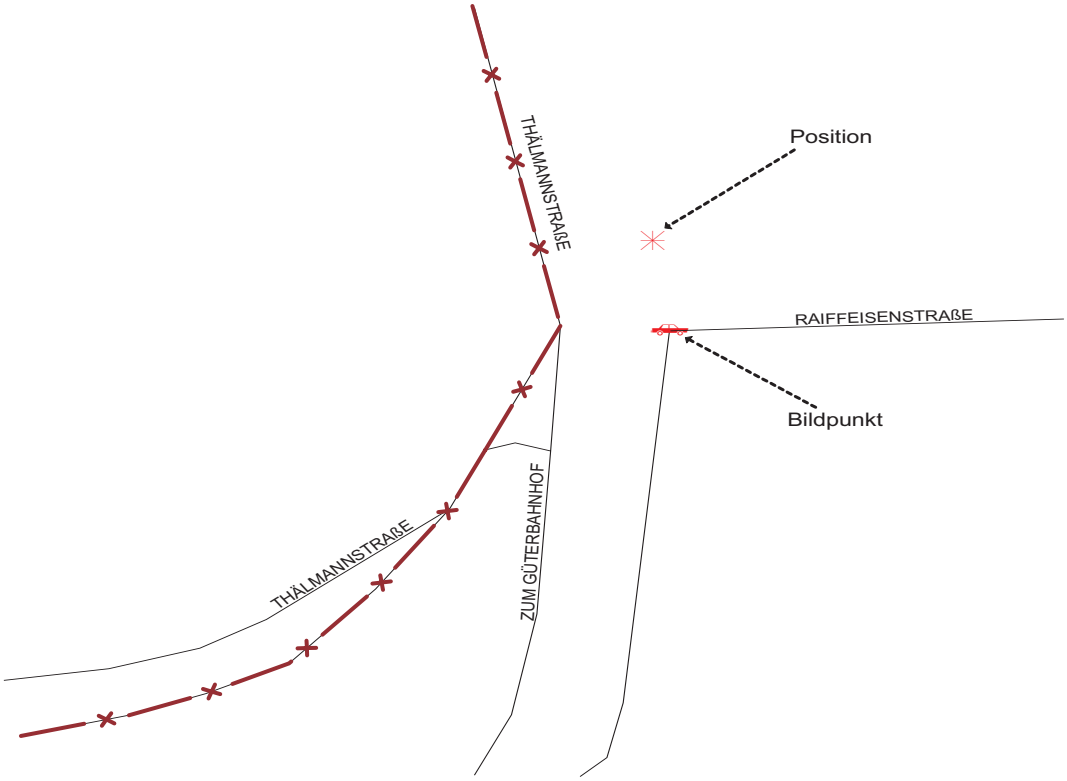


Abbildung 3.8.: einfache Abbildung auf den nächstgelegenen Punkt des Straßennetzes

worden. Er beruht darauf, durch Fällen des Lotes auf benachbarte Abschnitte[1] (alle innerhalb eines bestimmten Radiuses) den Punkt auf dem Straßennetz zu finden, der den geringsten Abstand zur abzubildenden Position hat. Der Fußpunkt des Lot's ist dann ein möglicher Bildpunkt auf dem Straßennetz. Falls dieser Fußpunkt nicht existiert, werden die beiden Netzknoten des Abschnittes als Kandidaten übernommen. Aus allen Kandidaten wird dann derjenige, mit dem geringsten Abstand zum Ursprungsknoten gewählt. Für einen einzelnen Punkt ist dies die sinnvollste Wahl, da keine zusätzlichen Informationen zur Verfügung stehen. Für die Abbildung einer zusammenhängenden Fahrstrecke ist dies allerdings nicht der beste Algorithmus.

Abbildung 3.8 zeigt ein denkbares Problem innerhalb des Ortsbereiches. Die eigentliche Fahrstrecke führt durch die Thälmannstraße. Durch prinzipbedingte Ungenauigkeiten der Datenerfassung liegt die gemessene Position (durch das Kreuz markiert) näher an der Ecke Raiffeisenstraße. Deshalb wird der beschriebene einfache Algorithmus die gezeigte Abbildung (markiert durch das Fahrzeug) wählen. Eine Route durch diesen Punkt würde eine starke Abweichung zur Originalroute aufweisen (wenn sie überhaupt existiert). Es ist bei diesem einfachen Ansatz nicht gewährleistet, daß die Folge der Bildpunkte auch eine zusammenhängende Fahrstrecke bildet. Deshalb sollen im folgenden Heuristiken untersucht

[1] ein Abschnitt ist in diesem Zusammenhang das Teilstück einer Straße zwischen zwei Kreuzungen (Netzknoten).

werden, die diesen Ansatz erweitern.

3.3.3. Berechnung und Abbildung von Fahrstrecken

Im folgenden soll untersucht werden, wie ausgehend von einer Menge von GPS-Koordinaten eine optimale Fahrstrecke rekonstruiert werden kann. Das theoretische Optimalitätskriterium ist dabei die Annäherung an die tatsächlich gefahrene Strecke. Da diese aber nicht bekannt ist, kann dieses Kriterium nicht für eine Bestimmung der Fahrstrecke herangezogen werden. Deshalb müssen andere Kriterien ausgewählt werden. Im allgemeinen besteht eine optimale Route aus einer Kombination zwischen *kürzester Strecke* und *schnellster Strecke*. Dafür werden für einen Abschnitt die mit der Durchschnittsgeschwindigkeit gemittelten Längen als Kostenmaß verwendet. Wenn die Geschwindigkeiten nicht bekannt sind, kann eine Schätzung aufgrund der Straßenklasse erfolgen. Dieses Kostenmaß kann danach noch vom Benutzer angepaßt werden. Ein primitiver Algorithmus zum Finden der optimalen Route für die gegebene Punktmenge ist die erschöpfende Suche. Dabei werden für jeden Punkt alle möglichen Abbildungspunkte auf das Straßennetz untersucht (mögliche Kandidaten werden analog dem Algorithmus für einzelne Positionen aus Abschnitt 3.3.2 bestimmt). Für jede der möglichen Kombinationen wird dann eine optimale Route berechnet und diejenige mit den geringsten Gesamtkosten gewählt. Dieser Algorithmus ist allerdings nicht praktikabel, da er eine exponentielle Laufzeit besitzt. Bei nur 10 Positionen und jeweils 5 möglichen Kandidaten pro Position müßten 5^{10}Routen berechnet werden. Der Berechnungsaufwand ist damit viel zu groß. Es muß also ein Weg gefunden werden, diesen Aufwand zu begrenzen und trotzdem noch eine nahezu optimale Lösung zu erhalten.

Vorher allerdings erst einmal einige Überlegungen dazu, was eine Fahrstrecke in diesem Zusammenhang ist. Durch die Fahrzeugschnittstelle werden zur Zeit nur Positionsdaten mit dem dazugehörigen Zeitpunkt erfaßt. Informationen über den Anfang und das Ende einer Route werden nicht aufgenommen. Da es für ein Fahrzeug sicherlich mehr als eine Fahrstrecke in der Datenbank gibt, muß also ein Weg gefunden werden, die Fahrstrecken mit größtmöglicher Genauigkeit zu separieren. Die einfachste Variante dabei ist, daß der Anwender diese Separierung vornimmt. Er müßte dann immer eine Zeitspanne angeben, die ihn interessiert. Daraufhin wird für diese Zeitspanne die Route berechnet und dargestellt. Nachteilig ist, daß er dann immer genau wissen muß, wann eine Route anfing und wann Sie endete. Auswertungen der Art *Zeige mir die Route von A nach B* sind damit nicht möglich. Dieser Zugang zu den Daten sollte also durch fest bestimmte Routen ergänzt werden. Für die Bestimmung dieser Routen wird die Zeitinformation ausgewertet. Wenn zwei Zeitpunkte nahe genug beieinnander liegen, werden sie einer Fahrstrecke zugeordnet. Da das maximale Intervall für die Positionserfassung bei einer halben Stunde liegt (vgl. Abschnitt 4.3.4), erscheint die Wahl von einer Stunde für dieses Zeitfenster sinnvoll. Damit können erstmal grob verschiedene Routen separiert werden. Als spätere Erweiterung sollte der Zeitverlauf einer Route genau erfaßt werden. Dies könnte eventuell an das Ein- und

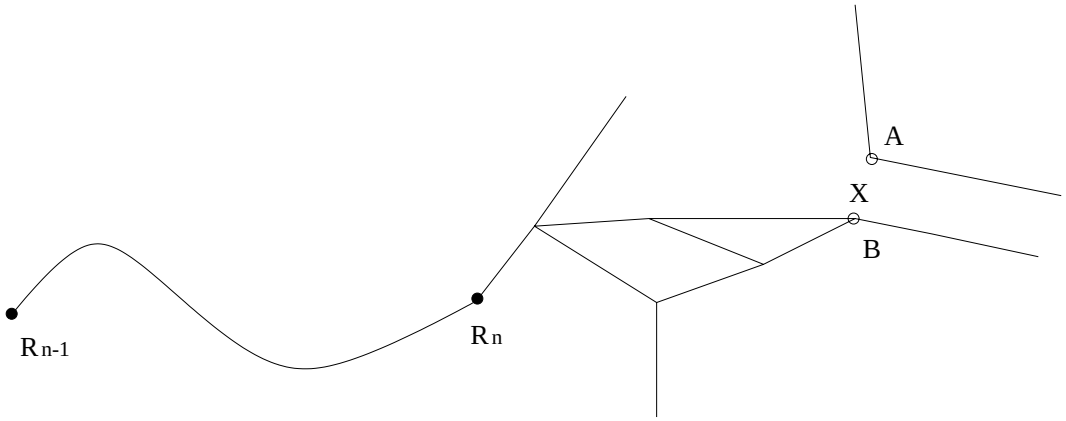


Abbildung 3.9.: Prinzip der konstruktiven Bestimmung der wahrscheinlichsten Fahrstrecke

Ausschalten des Meßgerätes im Fahrzeug gekoppelt werden. Alternativ müßte der Fahrer diese Information übermitteln. Im folgenden wird es jetzt um die Bestimmung des Verlaufs einer bestimmten Fahrstrecke gehen.

Für die jetzt folgenden Betrachtungen ist ein Verständnis des Algorithmus's zum Finden optimaler Fahrstrecken nötig. Dieser soll deshalb jetzt kurz vorgestellt werden, obwohl er nicht im Rahmen dieser Arbeit entwickelt wurde, sondern als bereits vorhandene Komponente integriert wird (vgl. Abschnitt 3.5). Der Algorithmus basiert auf dem bekannten *Algorithmus von Dijkstra* zum Finden kürzester Wege in (nicht negativ) gewichteten Graphen. Für den Einsatz in einem Straßennetz sind verschiedene Anpassungen nötig (z.B. zur Berücksichtigung von Abbiegeverboten). Diese ändern aber nichts am grundlegenden Prinzip. Es werden ausgehend von einem Startpunkt die Längen der kürzesten Wege zu allen anderen Punkten ermittelt. Mit dieser entstehenden Entfernungstabelle kann dann ein kürzester Weg vom Startpunkt zu einem beliebigen, erreichbaren Endpunkt rekonstruiert werden. Der Umstand, daß alle kürzesten Wege die von einem Startpunkt ausgehen ermittelt werden, wird für die folgenden Betrachtungen von Interesse sein.

Da eine erschöpfende Suche nicht praktikabel ist, soll im folgenden versucht werden jeweils nur lokal eine optimale Route zu ermitteln. Dabei sind prinzipiell zwei verschiedene Ansätze denkbar die im folgenden kurz vorgestellt werden.

konstruktiver Ansatz Dieser Ansatz versucht eine gute Annäherung der Route schrittweise aufzubauen. Die Position des Startpunktes wird z.B. über den kürzesten Abstand zum Straßennetz gesetzt. Danach wird jeweils vom aktuellen Knoten ausgehend versucht, den optimalen Nachfolgerknoten zu finden. Dazu werden zu einem Zeitpunkt alle möglichen Nachfolgerknoten betrachtet. Diese werden entsprechend gewichtet und dann der beste ausgewählt. Ein mögliches Wichtungskriterium ist z.B., eine Linearkombination der Entfernung des Punktes von der ermittelten Originalposition (GPS) und der Länge des neuen Teilstückes der Route (vom aktuellen Knoten zu diesem Punkt). Nachdem dieser neue

Knoten bestimmt ist, wird er als aktueller Knoten gesetzt und der Algorithmus fortgesetzt. Abbildung 3.9 zeigt eine Momentaufnahme des Verfahrens. Die ausgefüllten Kreise (Rn-1, Rn) stellen bereits bestimmte Stützpunkte der Route dar. Ausgehend von der letzen festgelegten Position Rn soll jetzt die Position Rn+1 bestimmt werden. Dazu werden alle möglichen Bildpunkte (A, B) für die entsprechende GPS-Koordinate (X) betrachtet. Diese Bildpunkte werden gewichtet und der Sieger wird als neuer Punkt (Rn+1) Teil der Fahrstrecke. Im dargestellten Beispiel würde wohl der Punkt B ausgewählt werden, da die Route von Rn nach A um einiges länger ist (wenn sie überhaupt existiert) als die Route nach B.

Mit diesem Ansatz sind allerdings verschiedene Nachteile verbunden. Zum einen ist es möglich, daß zu einem Zeitpunkt kein neuer Knoten bestimmt werden kann. In diesem Fall muß einen Schritt zurückgegangen werden, und der aktuelle Knoten neu bewertet werden. Dieses *Backtrackingverfahren* führt zu einem rekursiven Algorithmus. Des weiteren ist es schwierig, eine Bewertung zu finden, die immer den optimalen Nachfolgerknoten ermittelt. Der daraus entstehende Algorithmus wird schwer verständlich und erweiterbar sein. Das sich das Problem auch einfacher lösen läßt, zeigt der nächste Ansatz.

inkrementelle Verbesserung Die zweite Variante geht von einer vorhandenen Annäherung der Fahrstrecke an die gegebenen Positionen aus. Danach wird versucht, diese Route durch verschiedene Algorithmen zu optimieren. Eine einfache Anfangsroute kann z.B. durch die Abbildung aller Punkte auf das Straßennetz (vgl. Abschnitt 3.3.2) und Berechnen einer Fahrstrecke durch diese Punkte gefunden werden. Der Gesamtalgorithmus wird dann in mehrere Teilkomponenten zerlegt. Jeder dieser Teile (Strategien) repräsentiert eine einfache Optimierung der aktuellen Fahrstrecke. Jede Strategie kann für sich entwickelt und getestet werden. Durch Kombination dieser einfachen Teile können so komplexe Optimierungen erreicht werden. Der Einbau neuer Verfahren erfolgt ohne Einfluß auf bereits vorhandene Funktionalität. Außerdem kann dabei auf der bereits vorhandenen Annäherung aufgebaut werden (bei der ersten Variante müßte die Route verworfen und völlig neu aufgebaut werden). Gegenüber der direkten Konstruktion verspricht dieser Ansatz also eine bessere Wartbarkeit und Flexibilität der Gesamtlösung. Außerdem entfällt die Rekursion des ersten Ansatzes, da ein Backtracking nicht nötig ist[1]. Allerdings sollte man beide Ansätze nicht unbedingt als völlig gegensätzlich betrachten. Vielmehr ergänzen sie sich gegenseitig. In einem ersten Schritt ist noch keine Route vorhanden. Deshalb muß diese zuerst aufgebaut werden (mit einer Variante des ersten Ansatzes). Danach kann in einer zweiten Phase die konstruierte Route optimiert werden. Die Unterschiede liegen dann darin, wieviel Aufwand in die einzelnen Phasen gesteckt wird. Da der modulare Aufbau der zweiten Variante eine übersichtliche Lösung fördert, soll für diese Arbeit der Hauptaufwand auf diesem Teil liegen. Der erste, konstruktive Abschnitt soll sehr einfach und mit wenig Aufwand umgesetzt

[1] Es kann immer eine zulässige Route ermittelt werden (im Zweifelsfall zumindest die übergebene Ausgangsroute)

werden. Dadurch ergibt sich folgender Ablauf bei der Entwicklung:

1. Konstruktion der anfänglichen Annäherung. Dies erfolgt (im wesentlichen) wie beschrieben mit Hilfe der beschriebenen konstruktiven Bestimmung der Fahrstrecke. Allerdings wird für die Wichtung der Kandidaten nur deren Entfernung von der GPS-Position verwendet, d.h. es wird immer der Punkt mit der geringsten Abweichung ausgewählt (vgl. Abschnitt 3.3.2).

2. Möglichkeit zur Visualisierung einer Fahrstrecke schaffen.

3. Definition und Umsetzung verschiedener Strategien zur Optimierung einer gegebenen Route.

Die Visualisierung ist eine Funktionalität der GIS-Schnittstelle (vgl. Abschnitt 3.2.2). Im folgenden sollen die anderen beiden Punkte näher betrachtet werden.

Bestimmen der anfänglichen Annäherung

Für den gewählten Algorithmus wird eine erste Näherung der Fahrstrecke benötigt. Diese wird wie folgt festgelegt:

1. Jede einzelne Position wird auf die nächste Position im Straßennetz abgebildet (vgl. Abschnitt 3.3.2)

2. Diesen Positionen werden Abschnitte und Stationierungen[1] zugeordnet. Es werden bis zu zwei Netzknoten ausgewählt, die diese Position repräsentieren.

3. Es wird eine Route zwischen den gefundenen Netzknoten berechnet. Falls diese Route nicht existiert, müssen die Netzknoten ohne Verbindung neu gewählt werden.

Der erste Schritt wurde bereits kurz vorgestellt bzw. in der Arbeit von A. Bernklau (vgl. [BERN97]) besprochen. Die dadurch erhaltenen Koordinaten liegen auf dem Straßennetz. Deshalb muß für den zweiten Schritt nur noch der Abschnitt, der direkt an dieser Position verläuft, untersucht werden. Für diesen Abschnitt wird dann der Abstand der Position vom Startknoten ermittelt (Abstand als Fahrstrecke auf dem Abschnitt). Falls diese gefundene Stationierung in der Nähe eines Netzknotens ist (max. Abstand 300m) wird dieser Netzknoten als Zwischenstation für den Routingalgorithmus benutzt. Falls die Stationierung mitten im Abschnitt liegt, müssen beide Netzknoten (und damit der Abschnitt selber) Teil der Route sein. Dafür muß zuerst die Durchfahrtrichtung bestimmt werden. Bei Einbahnstraßen ist diese automatisch vorgegeben. Ansonsten wird die Fahrtrichtung gewählt, die die Strecke von der Vorgängerzwischenstation über diesen Abschnitt zur Nachfolgerzwischenstation

[1] Mit *Stationierung* ist der Abstand der Position vom Anfangsnetzknoten des Abschnittes gemeint (nicht der direkte Abstand sondern der tatsächliche Abstand auf dem Straßenverlauf).

Abbildung 3.10.: Beispielstrecke für die erste Annäherung

minimiert. Nachdem alle Zwischenstationen ermittelt sind, kann eine optimale Fahrstrecke zwischen diesen Netzknoten ermittelt werden. Falls keine zusammenhängende Route existiert, müssen einige Netzknoten neu abgebildet werden.

Abbildung 3.10 zeigt ein Beispiel für eine so erhaltene Route. Dafür ist exemplarisch die in Abbildung 3.8 gezeigte Situation dargestellt. Wie dort aufgeführt, hat der Algorithmus zur Bestimmung der Abbildung auf dem Straßennetz in dieser Situation versagt. Es ist eine Position auf dem falschen Abschnitt bestimmt worden. Demzufolge ist auch die ermittelte Route nicht optimal (durch die gestrichelten Abschnitte dargestellt). Es gibt eine unnötige Fahrstrecke durch die Ruhrstraße - Rathenaustraße - Raiffeisenstraße und zurück. Trotzdem bleibt festzustellen, daß diese Abweichung nur lokaler Natur ist. Die Annäherung kehrt danach wieder zu der eigentlichen Fahrstrecke zurück. Für eine globale Approximation ist dieser Algorithmus also durchaus geeignet. Trotzdem sollen jetzt noch verschiedene Strategien besprochen werden, mit denen die auftretenden Abweichungen eliminiert werden. Im Ziel soll eine möglichst direkte Verbindung ermittelt werden, die keine unnötigen Schleifen und Abweichungen enthält.

Abbildung 3.11.: die Fahrstrecke nachdem Schleifen eliminiert sind

verschiedene Optimierungsstrategien

Im folgenden sollen verschiedene Strategien vorgeschlagen werden, mit denen eine Annäherung an die tatsächliche Fahrstrecke optimiert werden kann. Alle Ansätze folgen dabei dem Grundsatz, daß eine möglichst direkte Route ohne Schleifen und Umwege gefunden werden soll. Des weiteren wird die Benutzung besser ausgebauter Straßen bevorzugt (dies wird durch die benutzte Routingkomponente unterstützt, vgl. Abschnitt 3.5).

kurze Schleifen Diese Strategie eliminiert kurze Schleifen in der Fahrstrecke. Solche Schleifen können dort entstehen, wo zwei berechnete Teilstrecken aneinander grenzen. Es gibt am Ende der ersten Teilroute eine Strecke, die zu einem Knoten abseits der optimalen Fahrstrecke führt. Am Anfang der nächsten Route gibt es dann einen Teil, der von diesem falschen Knotenpunkt wieder zur Route zurückführt (in Abbildung 3.10 trifft dies z.B. für die Teilstrecke Rathenaustraße - Raiffeisenstraße und zurück zu). Wenn diese Schleife kurz genug ist (c.a. 500 ... 700 m), wird davon ausgegangen, daß sie nur durch Meßungenauigkeiten entstanden ist. Deshalb kann sie eliminiert werden. Falls das Fahrzeug wirklich diese Schleife gefahren ist, kann dies für den hier betrachteten Einsatz ignoriert werden. Für eine Flottenüberwachung sind diese Ungenauigkeiten sehr klein (gegenüber der Länge der Gesamtstrecke von mehreren hundert Kilometern). Zur Erhöhung der Flexibilität könnte

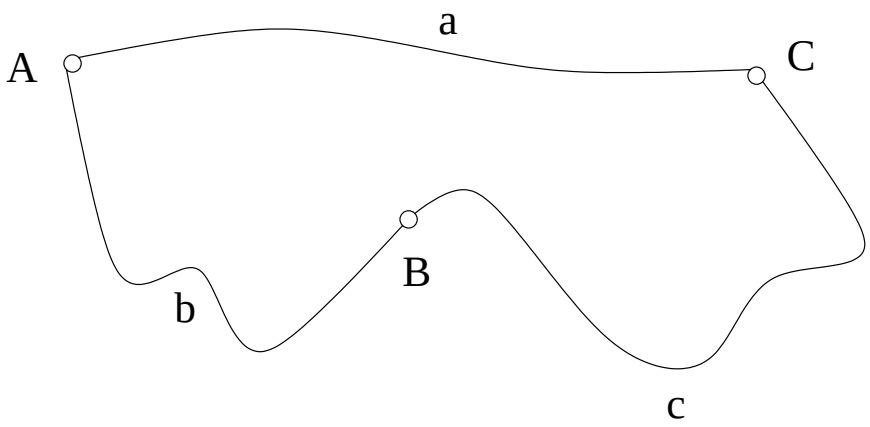


Abbildung 3.12.: Prinzip der Eliminierung zu langer Teilstrecken

darüber hinaus die Möglichkeit eingebaut werden, daß der Benutzer selber bestimmt, wie lang eine zu entfernende Schleife maximal sein darf.

Abbildung 3.11 zeigt das Beispielszenario, nach Anwendung des Algorithmus' zur Eliminierung von Schleifen. Die unnötige Fahrt durch die Rathenau- und Raiffeisenstraße ist aus der Route entfernt worden. Die Qualität der Route hat sich dadurch verbessert. Allerdings wird die Fahrstrecke noch in Richtung zu dem falsch bestimmten Zwischenpunkt abgelenkt. Anstatt einer direkten Strecke erhält man eine mit vielen Abbiegeaktionen. Im folgenden soll versucht werden, dies zu verbessern.

zu lange Teilstrecken Mit dieser Strategie sollen grob falsch bestimmte Zwischenpunkte eliminiert werden. Dafür werden jeweils drei zusammenhängende Teilstationen betrachtet. Wenn die direkte Verbindung zwischen erster und letzter Position viel kürzer ist, als der Weg über die Zwischenstation, wird die Auswahl dieser mittleren Station neu untersucht. Dafür werden jeweils alle möglichen Wege durch die drei Zwischenstationen berechnet und der kürzeste ausgewählt. Abbildung 3.12 illustriert diesen Ansatz. Es werden immer drei Stützpunkte der aktuellen Route untersucht. Das bisherige Teilstück der Route ist von A über B nach C. Als Test dient die Länge der optimale Route von A nach C (a). Wenn gilt a << b+c (b ist die Länge der Route von A nach B, c von B nach c), dann wird davon ausgegangen, daß der Zwischenpunkt B nicht optimal ist. Deshalb wird dieser Punkt neu bewertet. Es werden alle denkbaren Alternativen zu B untersucht, ob eine Verbesserung der Strecke erreicht werden kann. B wird dann durch die beste verfügbare Alternative ersetzt.

Durch den Längentest soll erreicht werden, daß der recht hohe Aufwand nur betrieben wird, wenn der Verdacht besteht, daß die Strecke über B stark abseits der tatsächlichen Fahrstrecke liegt. Der Aufwand kann dabei reduziert werden, wenn man ausnutzt, daß der verwendete Algorithmus immer alle optimalen Wege von einem Startknoten aus findet. Der allerdings trotzdem recht große Aufwand lohnt sich nur, wenn die Fahrstrecke sehr stark von der Vorgabe abweicht. Diese Strategie sollte also nur greifen, wenn die ermittelte Strecke

viel kostenintensiver als die direkte Verbindung ist. Das ist in dem hier betrachteten Beispiel nicht der Fall. Deshalb hat diese Strategie keine Auswirkungen auf die in Abbildung 3.11 dargestellte Fahrstrecke.

andere Strategien Neben den beiden vorgestellten Strategien sind noch weitere denkbar. Der Vorteil des gewählten Verfahrens ist, daß es einfach erweitert werden kann. Die verschiedensten Strategien lassen sich einbauen. Dadurch kann gut reagiert werden, wenn sich bei Testläufen grobe Fehlbestimmungen der Routen ergeben. Es ist z.B. denkbar, neben den analytischen Straßennetzdaten die geographischen auszuwerten. Das Ziel könnte dann z.B. sein, möglichst glatte Routen mit minimalen Abbiegeaktionen zu erhalten. Dies soll aber als mögliche Erweiterung betrachtet werden. Für die hier entwickelte Grundarchitektur einer Flottenmanagementanwendung reichen die beschriebenen Strategien aus, um recht gute Annäherungen für die meisten Routen zu ermitteln.

Außerdem sind die bisherigen Betrachtungen von einem *worst-case* Szenario ausgegangen. Innerhalb eines Ortsbereiches gibt es sehr viele verschiedene Kreuzungen und Abschnitte innerhalb eines kleinen Radiuses. Oftmals gibt es zu einer Straße parallel verlaufende Straßen mit nur wenigen hundert Metern Abstand. In einem solchen Umfeld gibt es natürlich Probleme mit der ungenauen Positionserfassung. Für eine Flottenmanagementanwendung wird es aber weniger um genaue Rekonstruktion von Stadtdurchfahrten gehen. Vielmehr ist das Ziel die Approximation einer langen Fahrstrecke außerhalb des Ortsbereiches. In diesem Einsatzgebiet gibt es weniger Mehrdeutigkeiten bei der Positionsabbildung. Dadurch muß die ermittelte Fahrstrecke auch weniger stark nachbearbeitet werden. Falls es zu Abweichungen innerhalb einer Ortsdurchfahrt kommt, ist dies für die Gesamtfahrstrecke weniger wichtig. Wichtig ist nur, welche Orte das Fahrzeug durchquert hat sowie der ungefähre Verlauf der Ortsdurchfahrt. Außerhalb des Ortsbereiches gibt es meistens nur eine Strecke die in Frage kommt und damit eine eindeutige Fahrstrecke.

3.4. Bereitstellung der Basisdienstkomponenten

Nachdem jetzt Lösungsvorschläge für die wichtigsten anwendungsspezifischen Probleme gefunden sind, wird es im folgenden darum gehen, die Basisdienste umzusetzen. Im Ergebnis ist dann die Anwendungsschicht vollständig umgesetzt. Zuerst sollen dabei einige allgemeine Probleme zu dem verwendeten Komponentenmodell dargestellt werden.

Unter den untersuchten Komponentenmodellen unterstützt eigentlich nur CORBA die Anforderungen an die Anwendung. Es ist damit möglich programmiersprachenunabhängige, objektorientierte und plattformübergreifende Komponenten zu entwickeln. Deshalb soll diese Architektur für die Flottenmanagementanwendung benutzt werden. Nachdem diese strategische Entscheidung getroffen ist, werden im folgenden einige Probleme in diesem Zusammenhang besprochen. Es soll erreicht werden, daß die Komponenten CORBA

Technologie benutzen, ohne fest damit verbunden zu sein. Insbesondere soll eine Lösung gefunden werden, die einen Einsatz der Geschäftslogik auch ohne CORBA ermöglicht (z.B. für Einzelplatzanwendungen).

Normalerweise folgt die Entwicklung eines verteilten Systems auf Grundlage von CORBA folgendem Ablauf:

1. Definition der öffentlichen Schnittstelle, Schreiben der IDL-Datei
2. Übersetzen der IDL Definition in die zu benutzende Programmiersprache, d.h. ein Werkzeug generiert aus einer IDL Datei die nötigen Basisklassen für die gewählte Sprache
3. Ableiten eigener Klassen von den erzeugten Basisklassen. Diese eigenen Klassen implementieren die Geschäftslogik.
4. Übersetzen (und ggf. Linken) der generierten und eigenen Klassen
5. Schreiben und Übersetzen des Klienten
6. Starten des Servers und Test des Klienten

Das Problem an diesem Ablauf steckt in Schritt 3. Alle Geschäftsobjekte müssen von generierten CORBA Klassen abgeleitet werden. Dadurch ist die Geschäftslogik sehr eng mit der zugrundeliegenden Infrastruktur verbunden. Eine Benutzung ohne Verwendung von CORBA ist nicht möglich. Des weiteren paßt es nicht unbedingt zum Entwicklungsprozeß, wenn die CORBA Schnittstelle als erstes definiert wird. Normalerweise sollte eine allgemeingültige Klassenarchitektur entwickelt werden (Stichpunkt: OOAD), die dann mit einer geeigneten Infrastruktur veröffentlicht werden kann. Aus diesen Gründen erscheint der folgende Ablauf besser geeignet:

1. OO-Analyse & Design, Umsetzen der Klassenarchitektur
2. Übersetzen und Testen der Geschäftslogik
3. Festlegen der zu veröffentlichenden Komponenten, Definieren der öffentlichen Schnittstelle (IDL)
4. Übersetzen der IDL
5. Implementierung der CORBA Klassen. Die eigentliche Abarbeitung wird an die Klassen aus dem 1. Schritt delegiert
6. Übersetzen der CORBA Klassen
7. Schreiben und Übersetzen des Klienten

8. Starten des Servers und Test des Klienten

In diesem Ablauf besteht eine klare Trennung zwischen Entwicklung der Geschäftslogik (Schritt 1+2) und Umsetzung mit einer bestimmten Infrastruktur (Schritt 3-8). Dieser Ansatz eignet sich auch besser für die Aufteilung innerhalb eines Teams. Die Anwendungsentwickler können die eigentliche Geschäftslogik getrennt entwickeln und testen. Die Bereitstellung mittels eines speziellen Komponentenmodells kann dann durch Entwickler, die sich damit gut auskennen, durchgeführt werden. Im folgenden sollen einige Probleme die sich bei der Umsetzung dieses Ansatzes ergeben besprochen werden.

3.4.1. allgemeine Probleme

3.4.1.1. Zugriff auf die Implementierung

Als Parameter und Rückgabewerte für Methoden der CORBA Schnittstelle sind nur folgende Typen erlaubt:

1. atomare Standardtypen (Ganzzahlen, Zeichenketten, Fließkommazahlen, ...)
2. Referenzen auf CORBA Objekte
3. zusammengesetzte Typen. Die Elemente dieser zusammengesetzten Typen müssen gültige CORBA Parameter sein

Eine benutzerdefinierte Klasse als Parameter einer Methode wird also immer als Referenz auf ihre CORBA Hüllklasse übergeben. Die Klasse an die die Abarbeitung delegiert wird kann diese Referenz aber nicht weiterverarbeiten. Anstatt des CORBA Hüllobjektes erwartet sie eine Referenz des gekapselten Delegatorobjekt's. Es muß also ein Mechanismus gefunden werden, der aus einer CORBA Referenz die Referenz auf das jeweilige Implementierungsobjekt liefert. Um diese Frage zu klären ist ein Verständnis des Zusammenhangs zwischen Objektimplementierung und dessen öffentlicher Schnittstelle Voraussetzung. Deshalb zuerst eine genauere Beschreibung der Bereitstellung eines Objektes.

Ein Entwickler setzt verschiedene Objekte um und möchte diese dann über einen ORB anderen Klienten verfügbar machen. Zur Anbindung von Objekten an den ORB dient ein Objekt Adapter. Seit den Anfängen der CORBA gibt es den *Basic Object Adaptor - BOA*. Um ein CORBA Objekt zu veröffentlichen wird eine Instanz seines Dienstobjektes[1] erzeugt und über den BOA dem ORB bekanntgegeben. Dieser erzeugt dann eine Objektreferenz. Mit Hilfe dieser Referenz kann dann ein Klient eine Instanz eines Schnittstellenobjektes erhalten. Über diese Schnittstelle können dann die Dienste des Servants benutzt werden. Dabei kann die Referenz auch zur Generierung von Schnittstellenobjekten auf anderen Systemen benutzt werden.

[1] auf englisch *Servant*, es ist versteckt innerhalb des Servers, Anfragen des Klienten werden durch den ORB an dieses Objekt weitergeleitet

Der BOA hat allerdings noch einige Nachteile. Der größte Nachteil ist, daß die BOA Schnittstelle nicht genau spezifiziert ist. Dadurch bestehen zwischen den Umsetzungen verschiedener Hersteller kleine Unterschiede. Ein einfacher Austausch der Serverplattform ist meist nicht möglich. Insbesondere spezielle Möglichkeiten, wie die hier benötigte Zuordnung zwischen Schnittstellenreferenz und Servantreferenz, sind herstellerabhängig. Diese Probleme wurden von der OMG erkannt und ab der CORBA Version 2.2 gelöst. Dafür wurde ein neuer Objekt Adapter, der *Portable Object Adapter - POA*, spezifiziert. Der Aufbau eines POA ist genau festgelegt. Die Schnittstellen sind Bestandteil des Standards. Dadurch kann die Umsetzung von CORBA Objekten allgemeingültig erfolgen. Die Umstellung auf einen anderen ORB ist problemlos möglich. Des weiteren ist die API auch umfangreich genug um auftretende Probleme zu lösen. So stellt die POA-Methode *object_to_servant* auch einen Weg bereit, zu einer CORBA Referenz die Referenz des umsetzenden Dienstobjektes zu erhalten. Dadurch steht einer strikten Trennung der Geschäftsobjekte von der CORBA Infrastruktur nichts mehr im Wege.

3.4.2. Umsetzung verschiedener Basisdienste

Bisher wurden verschiedene Bausteine der Flottenmanagementanwendung entwickelt. Es gibt für jede Komponente der Datenschicht eine entsprechende Schnittstelle zum Zugriff. Teil jeder Schnittstelle sind verschiedene Funktionspakete. Diese Pakete stellen anwendungsspezifische Komponenten dar, die jeweils nur auf die Daten ihrer Schnittstelle zugreifen. Die Datenbankpakete dienen zum Lesen und Schreiben der Flottendaten. Die GIS-Pakete dienen zur Darstellung und Manipulation des Kartenmaterials und der Erzeugung thematischer Karten. Durch die Fahrzeugschnittstelle wird die Kommunikation mit der Fahrzeugflotte unterstützt. Eine Anwendung soll aber keinen direkten Zugriff auf diese Schnittstellen haben. Was jetzt also noch fehlt, ist eine übergeordnete Schnittstelle die deren Funktionalität vereint. Diese stellt komplexe Dienste für die verschiedenen Anwendungen bereit. Im allgemeinen werden dabei die Daten und Dienstleistungen mehrerer Datenschnittstellen benutzt und zusammengeführt. Diese Funktionalität wird in den Basisdiensten zusammengefaßt, die in Abschnitt 2.2.4 entworfen wurden. Im folgenden werden jetzt die konkret umgesetzten Basisdienste beschrieben.

3.4.2.1. Initialisierung (Init)

Der Basisdienst zur Initialisierung dient zur Bereitstellung der Schnittstellen zur Datenschicht. Er ist dafür zuständig, daß die jeweiligen Schnittstellen initialisiert werden. Darüber hinaus bietet er allen anderen Diensten Zugriff auf diese Schnittstellen.

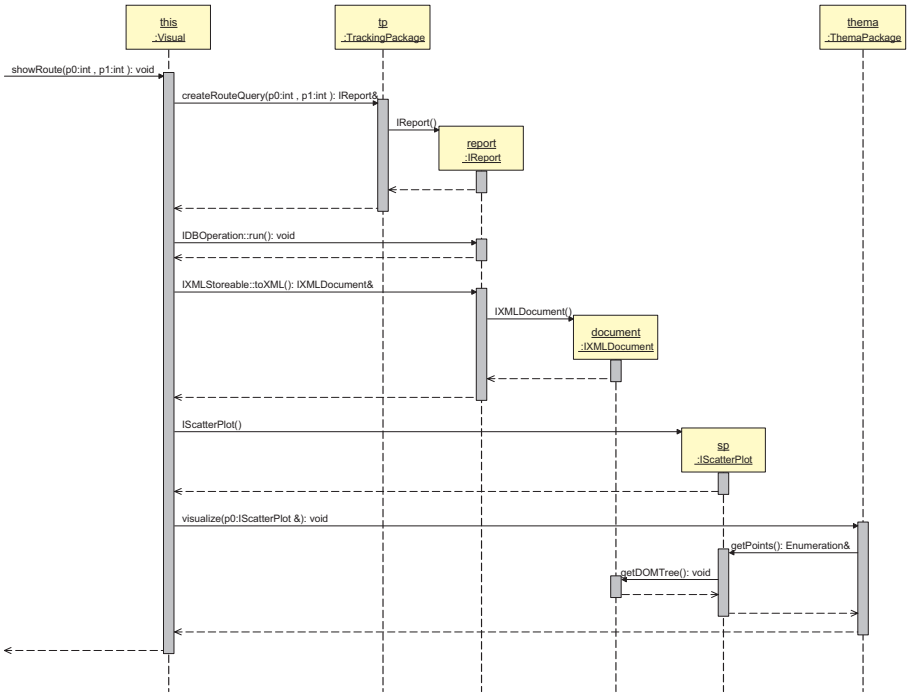


Abbildung 3.13.: ein Sequenzdiagramm für die Operation zur Visualisierung von Fahrstrecken

3.4.2.2. Visualisierung des Straßennetzes und thematische Karten (Visual)

Bei der Visualisierung gibt es zwei verschiedene Komplexe. Zum einen müssen die vorhandenen Straßendaten angezeigt werden. Dabei ist auch eine Unterstützung zur Referenzierung von geographischen Objekten nötig. Eine Suche und Darstellung von Städten, Straßen und wichtigen Gebäuden ist Teil der nötigen Funktionalität. Für diese Operationen werden meistens nur die Dienste der GIS-Schnittstelle benötigt. Allerdings könnte eine Datenbankunterstützung für die Suche von Objekten günstig sein. Für die zweite Funktionsgruppe ist die Datenbankschnittstelle unabdingbar. Es geht hierbei darum, die Fahrzeugdaten auszulesen und in einer thematischen Karte darzustellen.

Abbildung 3.13 zeigt die Zusammenarbeit zwischen Datenbank- und GIS-schnittstelle am Beispiel der Fahrstreckenvisualisierung. Zuerst wird das Tracking-Paket der Datenbankschnittstelle (vgl. Klassendiagramme in Anhang A.3) dazu benutzt, einen Report zur Abfrage einer Fahrstrecke zu erzeugen (*createRouteQuery*). Nachdem dieser Report ausgeführt ist (*run*) kann das Ergebnis als XML-Dokument angefordert werden (*toXML*). Für die Visualisierung der Daten muß eine *IScatterplot* Instanz übergeben werden. Diese Schnittstelle ist eine Anwendung des Adapter Entwurfsmusters. Dadurch können verschiedene Datenquellen für die Visualisierung benutzt werden. In der Anwendung hier, wird das XML-Dokument eingesetzt. Die Umsetzung der Methode *getPoints* holt die Baumstruktur des

Methode	Beschreibung
getRenderFileURL	liefert die URL für das letzte, erzeugte Kartenabbild
beginTransaction	Start einer Kartenmanipulation
commitTransaction	aktuelle Änderungen übernehmen, neues Abbild bereitstellen
rollBackTransaction	letzte Änderungen verwerfen
zoom	Verändern der Größe des aktuellen Kartenfensters
pan	Verschieben des aktuellen Kartenfensters
showCar	Visualisierung der letzten Position eines Fahrzeuges
showRoute	Visualisierung einer Fahrstrecke

Tabelle 3.4.: Funktionalität des Basisdienstes zur Visualisierung

XML-Dokuments[1] und erzeugt daraus eine Aufzählung der Stützpunkte der Fahrstrecke. Diese können dann vom *Thema* Paket der Kartenschnittstelle angezeigt werden.

Die weiteren Operationen des Basisdienstes zur Visualisierung folgen einem ähnlichen Ablauf. Welche Operationen unterstützt werden, ist aus Tabelle 3.4 und den Klassendiagrammen in Anhang A ersichtlich.

3.4.2.3. Funktionen rund um das Straßennetz (StreetMgmt)

Neben diesen einfachen Darstellungsoperationen ist auch eine analytische Verarbeitung der Daten nötig. Ein wichtiger Teil betrifft dabei verschiedene Operationen über dem Straßennetz. Ein wichtiger Komplex dabei ist die Abbildung eingegangener Positionsdaten auf das Straßennetz. Dabei werden nicht nur einzelne Positionen betrachtet. Vielmehr ist eine Abbildung einer kompletten Fahrstrecke auf das Netz zu ermöglichen. Die Probleme und Lösungen in diesem Umfeld werden in Abschnitt 3.3.2 und 3.3.3 dargestellt. Diese Algorithmen werden teilweise durch das StreetMgmt Paket bereitgestellt. Es enthält u.a. die Funktionalität zur Abbildung von Positionsdaten auf das Straßennetz.

Ein zweiter wichtiger Komplex bei der analytischen Auswertung des Straßennetzes betrifft das Finden optimaler Fahrstrecken. Bei einer Menge von gegebenen Stützpunkten soll dabei eine optimale Fahrstrecke durch diese Punkte gefunden werden. Um die verwendeten Heuristiken möglichst einfach zu halten, soll zunächst eine Fahrstrecke in der gegebenen Reihenfolge gefunden werden. Als spätere Erweiterung wäre eine Komponente denkbar, die vorher versucht eine möglichst optimale Reihenfolge zu finden. Es wird zwar davon ausgegangen, daß die Lösung dieses sogenannte *Problem des Handlungsreisenden* (*Travelling Salesman Problem,TSP*) exponentielle Laufzeit erfordert. Allerdings gibt es Heuristiken, die recht gute Annäherungen in geringerer Zeit liefern. Dies soll aber in der ersten Umsetzung vernachlässigt werden. Eine genauere Beschreibung des Problems der Fahrstreckenbestimmung und möglicher Algorithmen erfolgte in Abschnitt 3.3.3. Da für deren

[1] Die Struktur des Baumes wird vom WWW-Konsortium im Rahmen der Document Object Model - DOM Spezifikation festgelegt (vgl. [DOM])

Methodenname	Beschreibung
getReportCars	liefert einen Report, der alle verfügbaren Fahrzeuge auflistet
getReportRoutesForCar	zählt alle Fahrstrecken für ein Fahrzeug auf
getReportRoute	liefert die Details zu einer konkreten Fahrstrecke

Tabelle 3.5.: angebotene Leistungen des Basisdienstes für Reporte

Lösung eine bereits vorhandene Komponente, die auf einem anderen Komponentenmodell basiert, verwendet werden soll, wird die Integration in die Gesamtarchitektur in Abschnitt 3.5 behandelt.

3.4.2.4. Reporte und statistische Auswertungen (Report)

Dieser Basisdienst hat gewisse Ähnlichkeiten mit dem Basisdienst zur Visualisierung. Es geht ebenfalls um eine Aufbereitung und Darstellung der Daten. Dies soll allerdings nicht in Form einer geographischen Anzeige erfolgen. Vielmehr sollen die Daten in Tabellenform dargestellt werden. Diese können dann direkt dargestellt und ausgedruckt werden. Die Funktionalität dieses Basisdienstes wird im wesentlichen durch die Datenbankschnittstelle zur Verfügung gestellt. Die Daten werden dadurch ausgelesen und als XML Dokumente zurückgegeben. Diese XML Dokumente müssen danach nur noch an den Klienten weitergegeben werden. Die Reportkomponente ist also nur eine sehr dünne Schicht, um den Klienten einen Zugriff auf einen Teil der Datenbankschnittstellen Funktionalität zu geben. Welche Reporte angeboten werden ist aus Tabelle 3.5 bzw. dem Klassendiagramm aus Anhang A.4 ersichtlich.

3.4.3. Umsetzung der öffentlichen Schnittstellen

Damit die jetzt umgesetzten Basisdienste von verschiedenen Klienten verwendet werden können, müssen sie veröffentlicht werden. Dies bedeutet im Zusammenhang mit der hier betrachteten drei Ebenen Architektur, daß die Basisdienste auf einem CORBA basierenden Anwendungsserver bereitgestellt werden. Die prinzipielle Vorgehensweise ist bereits am Anfang dieses Abschnitts behandelt worden. Im folgenden soll diese Vorgehensweise in der Praxis dargestellt werden. Die beiden ersten Punkte des vorgeschlagenen Ablaufs (OO-Analyse & Design, Umsetzen und Testen der Komponentenlogik) sind bereits umgesetzt. Durch diese Basisdienste ist auch schon festgelegt, welche Komponenten bereitzustellen sind. Für jede dieser Komponenten müssen im folgenden CORBA Hüllobjekte erstellt werden. Dadurch können die Basisdienste dann auf einem CORBA ORB bereitgestellt werden.

In einem ersten Schritt müssen dafür geeignete Schnittstellen umgesetzt werden. Diese ergeben sich aus den bereits entwickelten Basisdienst Schnittstellen. Allerdings müssen ggf. Anpassungen bei den Parametern erfolgen, da wie bereits erwähnt nicht alle Typen zulässig sind. Einfache Grundtypen (Zahlen, Zeichenketten, Wahrheitswerte, ...) können

direkt übernommen werden. Falls andere Basisdienste als Parameter vorkommen, müssen an deren Stelle Referenzen auf die entsprechenden Hüllobjekte eingefügt werden. In einigen seltenen Fällen sind auch andere Objekttypen möglich. Für diese müssen dann ebenfalls entsprechende Hüllobjekte bereitgestellt werden. Da allerdings von Anfang an eine verteilte Anwendung berücksichtigt wurde, gibt es nur wenige Klassen für die dies nötig ist. Dies betrifft die Rückgabewerte im Zusammenhang mit der Erzeugung von Reporten.

Die eigentliche Umsetzung der Basisdienst Schnittstellen ist jetzt recht einfach und geradlinig. Im folgenden soll die Umsetzung anhand der Report Komponente dargestellt werden. Diese Klasse definiert die folgenden drei Methoden:

- getReportCars(): IReport&
- getReportRoutesForCar(carId: int): IReport&
- getReportRoute(carId: int, routeId: int): IReport&

Als Parameter kommen nur Grundtypen (Ganzzahlen) vor. Diese können beibehalten werden. Allerdings liefern alle Methoden eine Referenz auf eine eigene Klasse (IReport) zurück. Dieser Rückgabewert muß angepaßt werden. Dazu wird zuerst eine CORBA Hüllklasse für *IReport* umgesetzt (*ReportWrapper*). Die CORBA Komponente für den Report Basisdienst liefert dann jeweils eine Referenz auf eine ReportWrapper Instanz zurück. Mit dieser können alle Operationen durchgeführt werden, die auch mit einer IReport Instanz möglich sind. Alle Methodenaufrufe werden nur weitergeleitet. Die Schnittstelle für die Basisdienst Reportkomponente (*TrackingReportsWrapper*) hat nach der Übersetzung die folgenden Methoden:

- getReportCars(): ReportWrapper&
- getReportRoutesForCar(carId: int): ReportWrapper&
- getReportRoute(carId: int, routeId: int): ReportWrapper&

Die Methodenimplementierung ist dabei trivial. Der Aufruf wird an die jeweilige Methode der gekapselten *Report* Instanz weitergeleitet (Delegationsmechanismus). Der zurückgegebene Report wird in einer *ReportWrapper* Instanz verpackt und an den Aufrufer durchgereicht.

Das Klassendiagramm aus Anhang A.4.1 zeigt alle entstandenen Komponenten. Im Zentrum ist eine besondere Komponenten *TrackingFactory*. Diese stellt eine Klassenfabrik zur Verfügung. Damit können Instanzen für alle Basisdienste erzeugt werden. Für die meisten Aufrufe muß dabei eine bereits erzeugte Referenz auf eine Initialisierungskomponente mit angegeben werden. Die Schnittstellen der einzelnen Basisdienstkomponenten sind direkt aus den bereits entworfenen Basisdienstklassen abgeleitet. Es sind nur die nötigen

Anpassungen bei den komplexen Typen erfolgt. Allerdings besteht nicht der Zwang, alle Methoden auch in der CORBA Hülle bereitzustellen. Vielmehr wurden nur die nötigsten durchgereicht. Insbesondere bei der *IReport* Hüllkomponente (*ReportWrapper*) sind einige der Methoden, die nur zur Initialisierung dienen, nicht mehr enthalten. Diese können dann nur innerhalb des Anwendungsservers verwendet werden.

3.5. Anbindung von bereits vorhandenen Komponenten und anderer Komponentenmodelle

Die neu zu entwickelnden Komponenten bauen auf der CORBA Spezifikation auf. Allerdings kann es während der Entwicklung nötig werden, auch andere Komponenten in die Gesamtarchitektur einzubinden. Diese benutzen u.U. auch andere Komponentenmodelle und Entwicklungsumgebungen. Wie dies möglich ist, soll im folgenden dargestellt werden. Dabei wird davon ausgegangen, daß eine Komponente zur Navigation und Ermittlung optimaler Fahrstrecken zur Verfügung steht. Diese Komponente soll auf Microsofts COM Architektur aufbauen. Es sollen Möglichkeiten untersucht werden, wie diese Komponente von den anderen benutzt werden kann. Dabei soll es dann keinen Unterschied geben, ob eine COM oder CORBA Komponente eingebunden wird.

Die einfachste Möglichkeit ist die Entwicklung einer CORBA Hüllkomponente. Dafür benötigt man eine Entwicklungsumgebung, die sowohl COM als auch CORBA Komponenten unterstützt. Für dieses Beispiel soll dabei Microsoft Visual C++ 6.0 zum Einsatz kommen. Die COM Unterstützung ist bei diesem Produkt naturgemäß sehr gut. Da viele ORB's auch für C++ zur Verfügung stehen, ist auch eine CORBA Anbindung möglich. Für diese Anwendung wird dafür der Visibroker ORB von Visigenic benutzt. Dieser steht sowohl für C++ als auch für Java zur Verfügung. Durch Einsatz des genau festgelegten Übertragungsprotokolls (IIOP), können Produkte verschiedener Hersteller auf beiden Seiten eingesetzt werden. Um die umzusetzende Beispielanwendung möglichst einfach zu halten, wird dies allerdings nicht ausgenutzt.

Für die Flottenmanagementanwendung ist die Ermittlung optimaler Fahrstrecken eine wichtige Komponente. Sie wird z.B. bei der Rekonstruktion gefahrener Strecken oder bei der Planung von Aufträgen benötigt. Da die Entwicklung einer solchen Komponente eine recht umfangreiche Aufgabe ist, soll sie nicht extra entwickelt werden. Vielmehr soll eine bereits vorhandene Komponente eingebunden werden. Da diese Komponente allerdings die COM Technologie benutzt, muß eine Anbindung an CORBA Komponenten ermöglicht werden. Dazu soll der beschriebene Weg verwendet werden. Es gibt zwar auch verschiedene kommerzielle Produkte die sich der Thematik COM-CORBA Brücken widmen, allerdings ist deren Einsatz hier nicht unbedingt erforderlich, da nur eine einzige nicht CORBA Komponente eingebunden wird. Es ist hierbei kostengünstiger, die bereits vorhandenen Werkzeuge zu verwenden.

Methode	Beschreibung
readIt	lädt die Netzstruktur aus der übergebenen Datei
writeIt	speichert die aktuelle Netzstruktur unter dem übergebenen Dateinamen
addNode	fügt einen Netzknoten zum aktuellen Straßennetz hinzu
addEdge	fügt einen Abschnitt hinzu
calculateRoutes	berechnet eine optimale Fahrstrecke
getNodePre	geht zum nächsten Knoten der berechneten Strecke
getNowAt	liefert den aktuellen Netzknoten auf der berechneten Strecke
getEdgeId	liefert die Kante die benutzt wurde, um zum aktuellen Knoten zu gelangen

Tabelle 3.6.: Kernfunktionalität der Dijkstra Schnittstelle

Die COM Komponente zur Navigation muß auf einem Rechner installiert werden. Diese Komponente stellt verschiedene Schnittstellen zur Verfügung. Für die Kernfunktionalität wird allerdings nur eine dieser Schnittstellen benötigt. Diese Schnittstelle (*Dijkstra*) bietet verschiedene Methoden an. Tabelle 3.6 zeigt den minimal nötigen Teil der Schnittstelle. Damit ist sowohl der Aufbau des Straßennetzes als auch die Berechnung von Routen möglich. Damit reicht diese Funktionalität für einfache Anwendungen aus. Sie soll deshalb jetzt den CORBA Komponenten zur Verfügung gestellt werden. Dazu erwies sich folgender Ablauf als sinnvoll:

1. Erstellen der CORBA Hüllkomponente. Diese stellt die nötige Infrastruktur bereit, ohne allerdings die exportierten Methoden zu implementieren.

2. Schreiben eines einfachen Klienten zum Test, ob Zugriff auf diese Komponente möglich.

3. Nachdem der CORBA Teil funktioniert, hinzufügen der COM Funktionalität. Die CORBA Komponente erhält dazu ein privates Dijkstra COM Schnittstellenobjekt. Umsetzung der öffentlichen CORBA Methoden indem die Abarbeitung an das Schnittstellenobjekt delegiert werden.

4. Testen der Funktionalität

Nachdem die Komponente getestet ist, kann sie jetzt für die Anwendungsentwicklung verwendet werden. Da sie als normale CORBA Komponente ansprechbar ist, gliedert sie sich gut in die Gesamtarchitektur ein. Es können die vorhandenen Produkte und Hilfsklassen verwendet werden.

Der größte Nachteil der beschriebenen Lösung besteht darin, daß jetzt ein Aufruf zwei entfernte Methodenaufrufe nach sich zieht. Ein Methodenaufruf muß zuerst durch den ORB übermittelt werden. Danach wird der Aufruf an die COM Instanz weitergeleitet. In diesem

Schritt ist wiederum ein entfernter Methodenaufruf nötig. Allerdings erfolgt zumindest der zweite Aufruf zwischen zwei Prozessen auf ein und demselben Rechner. Da der COM Server zusätzlich als in-process Server umgesetzt ist (also im selben Prozeß wie der Klient läuft), hält sich der zusätzliche Ressourcenbedarf in Grenzen.

Der große Vorteil dieser Lösung ist, daß er allgemeingültig ist. Er kann zur Anbindung der verschiedensten Komponenten benutzt werden. Einzige Voraussetzung ist eine Programmierumgebung, die beide zu verbindenden Komponentenmodelle unterstützt. Selbst die Anbindung nicht verteilter Komponenten (z.B. reine C Bibliotheken ohne Extra Komponentenmodell) ist möglich.

4. Zusammenbau der Flottenmanagementanwendung

In den vorangegangenen Kapiteln ist die Geschäftslogik einer Flottenmanagementanwendung entwickelt und umgesetzt worden. Diese steht als eine Sammlung von wiederverwendbaren Komponenten auf einem Anwendungsserver zur Verfügung. Was jetzt noch fehlt, ist ein Klient der diese Komponenten zur Bereitstellung der Anwendung verwendet. Dieser Klient stellt die gesamte Bedienoberfläche zur Verfügung. Die eigentliche Verarbeitung der Daten erfolgt durch die entwickelten Geschäftsprozesse. Zur Umsetzung der Oberfläche können verschiedene Ansätze benutzt werden. Die Klienten können sowohl traditionelle ausführbare Programme auf einem Rechner, als auch durch einen Browser dargestellte Webseiten sein (vergleiche Abbildung 4.1). Die eigentlichen Firmendaten sind dabei versteckt. Ein Zugriff von außen kann nur über die Dienste des Anwendungsservers erfolgen. Eine Firewall dient dazu, den Zugriff zu beschränken. Sie muß so konfiguriert werden, daß der Zugriff auf den Anwendungsserver über IIOP möglich ist. Wenn von außen nur Internetklienten zugelassen sind, kann die Firewall so konfiguriert werden, daß IIOP Zugriffe nur vom Webserver möglich sind. Neben diesen grundlegenden Schutzmaßnahmen sollte auch der Anwendungsserver eine Benutzerauthentifikation unterstützen. Beim zusätzlichen Einsatz sicherer Verbindungen (z.B. SSL) können die Firmendaten wirkungsvoll geschützt werden, obwohl sie weltweit verfügbar sind.

Die hier umzusetzende Beispielanwendung wird für die Darstellung den Weg über einen Webserver gehen. Jeder authorisierte Benutzer soll weltweit in der Lage sein, auf die Anwendung zuzugreifen. Dabei sollen keine besonderen Installationen auf Seiten des Klienten erforderlich sein. Die verwendete Hard- und Software soll irrelevant sein. Einzige Voraussetzung ist ein Internetzugang und ein handelsübliches WWW-Zugriffsprogramm. Im folgenden werden einige der wichtigsten Technologien zur Umsetzung dieses Zieles vorgestellt. Danach werden die für diesen Einsatzfall besten Technologien ausgewählt und die Anwendung zusammengestellt. Zum Abschluß wird dann die benutzte Infrastruktur (sowohl Hard- als auch Software) kurz vorgestellt.

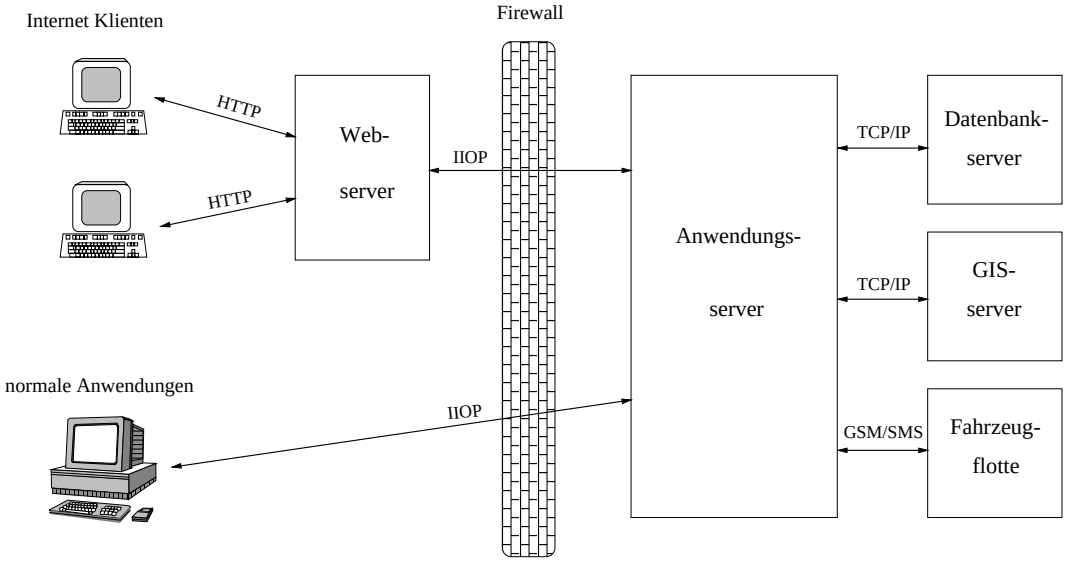


Abbildung 4.1.: physische Aufteilung der Anwendung und mögliche Klienten

4.1. Technologien für die Anwendungsoberfläche

4.1.1. Hypertext Markup Language - HTML

HTML ist eine standardisierte Seitenbeschreibungssprache. Sie ist eine Anwendung der, in der ISO-Norm 8879 festgeschriebenen, *Standard Generalized Markup Language - SGML*. SGML ist eine sogenannte Metasprache. Mit ihr lassen sich beliebige Seitenbeschreibungssprachen definieren. Eine dieser speziellen Anwendung, die auf die Darstellung von Inhalten im World Wide Web (WWW) spezialisiert ist, ist HTML. Neben diesen beiden Sprachen gibt es noch die etwas neuere Entwicklung der *Extensible Markup Language - XML*. Diese ist eine vereinfachte Form der SGML, die auf die Übertragung von Daten im WWW spezialisiert ist. Sie ist also weniger für die Erstellung der Benutzerschnittstelle als vielmehr für die Übermittlung von Daten von Interesse. Deshalb wird XML im Zusammanhang mit der Datenbankschnittstelle besprochen (siehe Abschnitt 3.2.3).

Das Protokoll zur Übertragung von Daten (z.B. HTML Seiten) im WWW ist das *Hypertext Transfer Protokoll - HTTP* (siehe [RFC2068]). Die HTML Seiten liegen auf einem Webserver. Dieser wartet auf eintreffende HTTP-Kommandos. Das *GET* Kommando dient z.B. dazu, Dateien vom Server anzufordern. Im Fall einer HTML Datei muß der Klient diese jetzt untersuchen und dann darstellen. Eine HTML Datei kann Verweise auf andere Dateien enthalten. Diese werden dann entweder in das aktuelle Dokument eingebunden (z.B. bei Bilddaten) oder können durch Anklicken des Benutzers aktiviert werden (sogenannte *Hyperlinks*). Durch die *Hyperlinks* werden mehrere HTML Dokumente miteinander verknüpft. Der Benutzer kann so von einem Dokument zu einem anderen wechseln.

Zur Anzeige und Navigation zwischen HTML Seiten existieren spezielle Anwendun-

Vorteile	Nachteile
schnell, da kein Parsen auf Seiten des Servers nötig	beschränkte Interaktion mit Benutzer
universell und auf vielen Plattformen verfügbar	keine Einbindung externer Daten (z.B. aus einer Datenbank) möglich
klientenseitige Zwischenspeicher sind möglich, da die Daten nur in unregelmäßigen Abständen geändert werden	feste Struktur und Aussehen, keine Anpassung während einer Sitzung
	genaue Darstellung ist vom Zugriffsprogramm abhängig

Tabelle 4.1.: Vor- und Nachteile von statischen HTML Seiten

gen. Diese sogenannten *Browser* unterscheiden sich darin, welche HTML Syntaxelemente (sogenannte *tags*) sie interpretieren. Die offizielle Spezifikation der Sprache wird vom *World Wide Web Consortium* - W3C entwickelt. Darüber hinaus hat aber jeder Hersteller eigene Elemente umgesetzt. Diese werden dann allerdings auch nur von diesem Anzeigeprogramm erkannt. Der allgemeinste Standard, den eigentlich alle Browser verstehen, ist HTML in der Version 2.0 (siehe [RFC1866]). Eine gute Einführung dazu bietet das Buch von Rainer Maurer ([MAU96]). Weiterführende Versionen des HTML-Standards (Version 3.2 und Version 4.0) führen neue Möglichkeiten ein. Diese betreffen z.B. die Möglichkeit, einfache Anweisungen in HTML Seiten einzubinden. Diese werden dann auf Seiten des Klienten interpretiert und ausgeführt. Die Möglichkeiten dieses *Klienten Seitigen Skripting* werden im folgenden Abschnitt behandelt.

Zuvor sollen allerdings zunächst einige Vor- und Nachteile erwähnt werden, die sich aus dem Einsatz von HTML ergeben (Tabelle 4.1). Die wesentlichste Einschränkung ist, daß die Dateien statisch auf dem Server abgelegt sind. Jeder Aufrufer bekommt genau diese Dateien. Sie können nicht während einer Sitzung angepaßt werden. Ein großer Vorteil besteht darin, daß es damit möglich ist Dokumente zu schreiben, die auf vielen verschiedenen Plattformen darstellbar sind. Allerdings hängt die Darstellung vom benutzten Zugriffsprogramm ab. Es gibt kaum Möglichkeiten, daß eigentliche Layout direkt zu bestimmen. Vielmehr gibt man nur Hinweise, wie eine Seite aussehen soll. Die eigentliche Interpretation erfolgt dann durch das Zugriffsprogramm des Klienten.

HTML eignet sich damit gut zur Vermittlung statischer Inhalte. Ähnlich wie bei Zeitungen und Büchern sieht ein Anwender jeweils nur die Sichten, die für ihn bereitgestellt werden. Die Zusammenstellung der Daten kann nicht an individuelle Bedürfnisse angepaßt werden. Allerdings ist HTML gut zur Definition von Dokumenten geeignet, die auf vielen Plattformen darstellbar sind. Wenn man es zur Präsentation einer komplexen Anwendung verwenden will, müssen Wege gefunden werden, HTML Seiten individuell anzupassen. Einige dabei denkbaren Ansätze sollen im folgenden kurz vorgestellt werden.

Vorteile	Nachteile
Darstellung auf Seiten des Klienten beeinflussbar	teilweiser Verlust von Plattformunabhängigkeit, da nicht standardisiert
schnell, da kein Parsen auf Seiten des Servers	teilweise unübersichtlich, da eingeschränkte Möglichkeiten und schlecht formulierte Syntax der Skriptsprachen
verringerte Netzbelastung, da Interaktionen und Validierungen teilweise auf Seiten des Klienten behandelt werden können	kein Zugriff auf externe Datenquellen (z.B. Datenbanken) möglich

Tabelle 4.2.: Vor- und Nachteile von klientenseitigen Skriptsprachen

4.1.2. klientenseitige Skriptsprachen und Java Applets

Ein großer Nachteil von HTML Seiten ist, daß deren Darstellung statisch ist. Es gibt keine Möglichkeit, Elemente nachträglich (d.h. während das Dokument dargestellt wird) zu ändern. Dieses Problem kann durch Skriptsprachen auf Seiten des Klienten (*Klienten Seitiges Skripting*) gelöst werden. Damit besteht die Möglichkeit, Befehle einer Skriptsprache (z.B. JavaSkript, VisualBasicSkript oder PerlSkript) in ein HTML Dokument einzubinden. Diese werden dann auf Seiten des Klienten ausgewertet und ausgeführt. Es ist damit möglich, auf verschiedene Ereignisse zu reagieren (z.B. Mausklicks oder -bewegungen) oder auch kontinuierlich Funktionen auszuführen (z.B. für Animationen). Um das Aussehen einer Seite zu ändern, müssen deren Elemente für die Skriptsprache zugänglich sein. Dafür haben die verschiedenen Browserhersteller ihre eigene Klassenstruktur entwickelt. Eine HTML Seite mit eingebauten Skriptanweisungen kann dann auf diese Klassen zugreifen und dadurch die Darstellung der Seite ändern. Es ist z.B. möglich, die Position einzelner Elemente zu verändern oder deren Darstellung anzupassen. Im wesentlichen können alle Attribute eines Tags verändert werden.

Der Nachteil dabei ist, daß jeder Hersteller sein eigenes Dokumentenmodell verwendet. Dadurch ist es schwer möglich, Seiten zu erstellen die für mehrere Browser zugänglich sind. Dieses Problem hat auch das WWW Konsortium erkannt und deshalb das *Document Object Model - DOM*[1] entwickelt. Dadurch wird die Klassenstruktur zum Zugriff auf die Elemente einer HTML Seite festgelegt. Leider haben noch nicht alle Hersteller dieses vollständig in ihre Produkte integriert. Deshalb muß man beim Einsatz von klientenseitigen Skriptsprachen vorsichtig sein. Man gewinnt dadurch Flexibilität bei der Darstellung von HTML Seiten. Ein großer Teil der Plattformunabhängigkeit kann allerdings verloren gehen (vgl. Tabelle 4.2).

Ein anderer Ansatz zur Darstellung interaktiver Inhalte ist die Einbindung von Java Applets. Ein Applet ist eine spezielle Oberflächenkomponente. Diese wird in die HTML Seite eingebunden. Der Browser lädt die dazu gehörenden Binärdateien vom Server und startet

[1]Das DOM dient allgemein zum Zugriff auf Daten in SGML Notation. Es ist damit auch ein wichtiger Ansatz für den Zugriff auf XML Daten.

Vorteile	Nachteile
umfangreiche Möglichkeiten der Oberflächengestaltung	abhängig von der Plattform des Klienten
vielfältige Anbindung an Datenquellen	u.U. recht lange Ladezeiten der Programmdateien vom Server, d.h nur kurze Programme unter Benutzung von Standardbibliotheken sinnvoll
alle Möglichkeiten einer 3GL (Java) verwendbar	voller Zugriff auf den Anwendungsserver muß für den Klienten möglich sein (mit allen eventuell damit verbundenen Sicherheitsproblemen)
gute Wartbarkeit, da Programmkode an einer zentralen Stelle (auf dem Server)	

Tabelle 4.3.: Vor- und Nachteile von klientenseitigem Java

die Ausführung. Die Darstellung fügt sich direkt in die umgebende HTML Datei ein. Ein Vorteil von Java auf Seiten des Klienten ist das hohe Sicherheitskonzept der Sprache. Die Ausführung aller Befehle wird überwacht. Wenn ein Applet versucht, verbotene Anweisungen auszuführen (z.B. Zugriff auf Dateien des Klienten) wird eine Ausnahmesituation erzeugt, d.h. die Befehlsabarbeitung wird an dieser Stelle unterbrochen. Ein Benutzer hat die Möglichkeit, diese Beschränkungen für vertrauenswürdige Applets aufzuheben. Dadurch können alle Möglichkeiten der Sprache Java zur Anwendungsentwicklung eingesetzt werden. In früheren Java Versionen (1.0.x bzw. 1.1.x) kommt hierbei das sogenannte Sandboxmodell zum Einsatz. Ein Applet kann entweder innerhalb des Sandkastens ablaufen (nur die eingeschränkten Rechte) oder außerhalb (alle Rechte, erst ab Version 1.1.x). Dieser Alles oder Nichts Ansatz wird in der neuesten Version (Java 2) durch ein Domänenkonzept ersetzt. Eine Klasse (also z.B. auch ein Applet) gehört in diesem Modell zu einer Sicherheitsdomäne. Die Rechte lassen sich für jede Domäne einzeln festlegen. Dadurch wird eine bessere Steuerung der Rechte eines Applets ermöglicht.

Außer den durch das Sicherheitskonzept gegebenen Einschränkungen kann ein Applet auf alle Leistungen der Sprache Java zugreifen. Ein Problem in diesem Zusammenhang ist, daß die Laufzeitumgebung vom verwendeten Browser abhängt. Die Hersteller haben dabei erst vor kurzem die Umstellung auf die Version 1.1.x durchgeführt. Die neueren Möglichkeiten der Version 1.2 (Java 2) werden derzeit von keinem Zugriffsprogramm unterstützt. Es gibt zwar Ansätze, auch in bestehenden Browsern die jeweils aktuelle Version zur Verfügung zu stellen (z.B. das Java-Plugin der Firma Sun), es läßt sich aber sagen, daß die Benutzung der aktuellsten Möglichkeiten die unterstützten Plattformen stark einschränkt. Es bestehen die selben Probleme wie bei anderen klientenseitigen Ansätzen. In Umgebungen in denen die benutzten Zugriffsprogramme beeinflußbar sind (z.B. im firmeneigenen Intranet), können diese Technologien u.U. eingesetzt werden. Wenn es aber um die Anbindung möglichst vieler Benutzer geht, deren Plattform nicht vorhersehbar ist, sollte man

eher sparsam damit umgehen. Des weiteren bestehen Probleme, wenn Applets direkt auf die Dienste des Anwendungsservers zugreifen sollen. Da sie auf Seiten des Klienten laufen, muß für diesen ein Zugriff möglich sein (durch entsprechende Konfiguration der Firmen Firewall). Damit hat der Klient dann Zugriff auf den Anwendungsservers, den er eventuell auch mit nicht authorisierten Programmen ausnutzen kann. Außerdem ist es einem Applet in der Standardkonfiguration nicht erlaubt, TCP/IP Verbindungen zu anderen Rechnern herzustellen (außer zu dem, von dem es geladen wurde). Dies kann zu einem erhöhten Konfigurationsaufwand für den Klienten führen, wenn Applikations- und Webserver nicht identisch sind.

4.1.3. Servererweiterungen

Da dynamische Inhalte auf Seiten des Klienten die unterstützten Plattformen einschränkt, soll dort nur reines HTML eingesetzt werden. Da andererseits eine dynamische Generierung der Seiten aber unabdingbar ist (z.B. für Datenbankzugriffe), müssen Wege für dynamische Inhalte auf dem Server gefunden werden. Dieser sollte nicht mehr ein reiner Dateiserver sein. Vielmehr soll er die zu übertragenden Seiten für jede Anfrage neu erzeugen und individuell anpassen. Ein möglicher Ansatz dabei ist eine Erweiterung des Webservers. Dazu haben sich verschiedene herstellerspezifische Schnittstellen herausgebildet.

Für den *Microsoft Internet Information Server* (IIS) gibt es die Möglichkeit der Einbindung von Erweiterungen durch die *Internet Server API - ISAPI*. Solche Erweiterungen werden in Form einer dynamischen Bibliothek (DLL) konzipiert und müssen die Internet Server Schnittstelle unterstützen. Danach können sie in die Abarbeitung von HTTP Anfragen eingebunden werden. Dies geschieht in Form sogenannter ISAPI Filter. Ein Filter hat die Möglichkeit, in die Bearbeitung einer Anfrage einzugreifen. Dabei kann der Entwickler eines Filters festlegen, zu welchen Zeitpunkten der Filter aktiv werden soll. Dadurch kann z.B. eine Umleitung von Anfragen mit bestimmten Adressen oder auch eine Nachbearbeitung der zu übertragenden Dokumente realisiert werden. Eine spezielle Art von ISAPI Filtern dient dazu, HTML Seiten vollständig dynamisch zu erzeugen. In diesem Fall gibt es keine statischen Quelldateien auf dem Webserver. Die entsprechenden Anfrageergebnisse werden vollständig durch den Filter erzeugt (z.B. als Ergebnis von Datenbankabfragen). Eine kurze Einführung in die ISAPI-Filter Programmierung ist z.B. im Microsoft System Journal erschienen (siehe [MSJ98]).

Andere Webserver haben ähnliche Möglichkeiten zur Erweiterung. Allerdings sind alle darauf basierenden Lösungen an das jeweilige Produkt gebunden. Eine Umstellung auf andere Produkte ist nahezu unmöglich bzw. nur mit großem Aufwand realisierbar. Einen Ausweg aus diesem Problem bietet die Servlet API der Java Architektur. Die Servlet API ist eine von der Firma Sun entwickelte API zur Programmierung von Webservererweiterungen in Java. Gegenüber anderen Lösungen hat dieser (eigentlich auch proprietäre) Ansatz den Vorteil, daß er als eine reine Spezifikation ohne direkten Produktbezug entwickelt wird.

Vorteile	Nachteile
dynamische Seiten unabhängig von den Fähigkeiten des Klienten	teilweise abhängig vom verwendeten Webserver
keine Beschränkungen bei der Einbindung externer Daten	größere Belastung des Servers durch neue Aufgaben
Anwendungslogik auf dem Server versteckt, da nur dynamisch erzeugte Dateien zum Klienten gelangen (kein direkter Zugriff auf sensitive Firmenressourcen möglich)	mind. zwei verschiedene Programmierumgebungen
	jede Aktion erfordert erneuten Datenaustausch zwischen Klient und Server (Umgehung eventueller Caching Strategien, da Dateien immer neu erzeugt)

Tabelle 4.4.: Vor- und Nachteile von Servererweiterungen

Dadurch besteht mittlerweile die Möglichkeit der Einbindung in viele Webserver. So ist z.B. auch ein Einsatz im Zusammenhang mit dem IIS realisierbar. Die Aktivierung eines Servlets kann entweder direkt beim Starten des Webservers erfolgen oder an den Zugriff auf bestimmte URL's gebunden sein. Dadurch kann z.B. ein Servlet geschrieben werden, welches immer bei Dateien mit einer bestimmten Endung aktiv wird (siehe *Server Seitiges Skripting*). Den aktuellen Stand der Entwicklung der Servlet-API kann man auf Sun's Java Webseite unter [SERVLET] nachlesen.

Tabelle 4.4 zeigt einige Vor- und Nachteile im Zusammenhang mit dem Einsatz dieser Techniken. Der große Vorteil ist die Möglichkeit der dynamischen Erzeugung von HTML Dateien. Dadurch können recht umfangreiche Lösungen erzeugt werden, die mit praktisch jedem Zugriffsprogramm zusammenarbeiten können. Die Anbindung beliebiger externer Daten (Datenbank, Netzwerkverbindung, Dateien, ...) ist möglich. Diese große Flexibilität bringt allerdings auch einige Nachteile mit sich. Der gravierendste Nachteil ist, daß jede Anfrage auf dem Server abgearbeitet wird. Dies bedeutet eine stärkere Belastung an diesem zentralen Punkt (insbesondere bei vielen gleichzeitigen Anfragen wird schnell die Belastungsgrenze erreicht). Dadurch wird die Skalierbarkeit dieser Lösung eingeschränkt. Durch bessere Strategien beim Webserverzugriff (z.B. Verteilung der Anfragen auf mehrere physikalisch verschiedene Rechner) lassen sich diese Probleme allerdings lösen. Kritischer in diesem Zusammenhang ist, daß jede Änderung einer Seite einen erneuten Datenaustausch zwischen Server und Klient bedingt. Dadurch entsteht eine größere Netzbelastung als z.B. beim Einsatz von klientenseitigen Ansätzen. Außerdem werden eventuelle Caching[1] Strategien umgangen, da die Seiten immer neu aufgebaut werden. Bei der Entwicklung fällt auf, daß es mehrere verschiedene Programmierumgebungen gibt. Statische (HTML) und

[1] Beim Caching werden Daten aus einer langsamen Datenquelle nach dem ersten Zugriff in einem schnelleren Zwischenspeicher gehalten. Dadurch werden spätere Zugriffe beschleunigt. Bei Webseiten bedeutet das, daß sie lokal gespeichert werden. Dies ist allerdings nur möglich, wenn die Dateien auf dem Server sich nicht verändern.

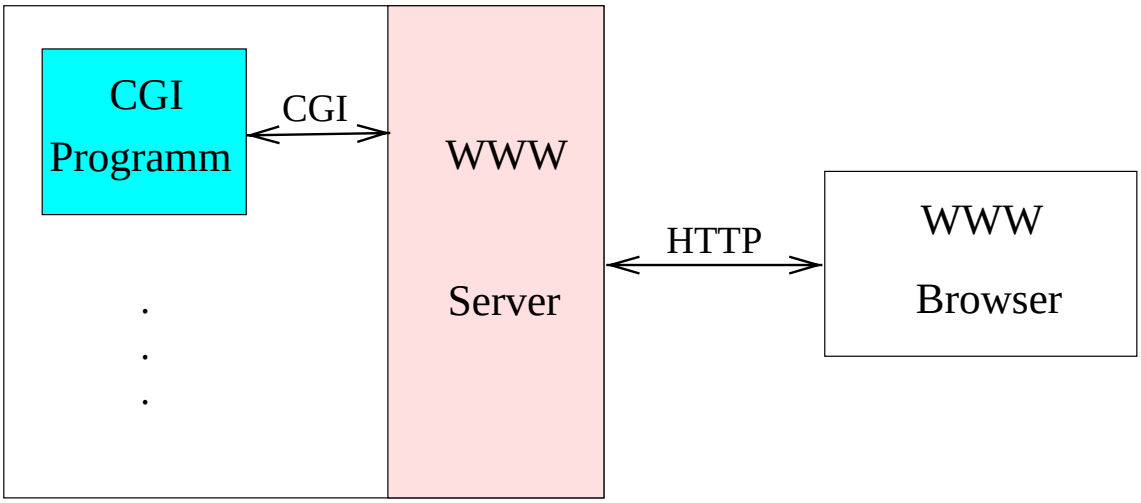


Abbildung 4.2.: prinzipielle Struktur einer CGI Anwendung

dynamische (Filter, Servlets) Inhalte sind voneinander getrennt und können nicht als eine Datei bearbeitet werden. Dadurch wird die Gesamtlösung schwerer verständlich und in Folge schlechter wartbar. Insbesondere dieser Nachteil soll durch Einsatz der im folgenden beschriebenen Skriptsprachen aufgehoben werden.

4.1.4. serverseitiges Skripting

Die im folgenden zu besprechenden Technologien dienen der Ausführung von Skripten und Programmen auf dem Server. Im Gegensatz zu den Servererweiterungen müssen diese nicht innerhalb des Servers installiert werden, sondern können direkt durch eine URL referenziert werden. Die ersten zu beschreibenden Technologien sind dabei historisch gesehen die älteren. Sie erlauben teilweise schon recht komplexe Anwendungen. Allerdings gibt es Probleme bei der Umsetzung großer Firmenanwendungen. Für dieses Einsatzgebiet sind dann die letzten beiden Skriptsprachen besser geeignet.

Abbildung 4.2 zeigt den prinzipiellen Ablauf beim Einsatz des *Common Gateway Interface - CGI*. Dabei kann der Klient durch Auswahl spezieller Adressen auf dem Webserver eine dort bereitgestellte ausführbare Datei starten. Der Webserver übergibt dabei die Parameter vom Klienten an das CGI Programm. Das Ergebnis wird dann als Dokument an den Webserver übergeben die dieser an den Klienten weiterleitet (zumeist erfolgt die Ausgabe in Form von HTML Daten). Für eine Einführung in die Möglichkeiten und den Einsatz von CGI empfiehlt sich das Buch von Rainer Maurer ([MAU96]). Die Möglichkeiten dieser Schnittstelle entsprechen im wesentlichen den der bereits beschriebenen Servererweiterungen. Allerdings besteht der Nachteil, daß für jeden Klientenzugriff diese Datei erneut gestartet werden muß (bei Servererweiterungen wird sie im allgemeinen nach dem ersten Zugriff im Speicher gehalten). Einige neuere Ansätze beseitigen diesen Nachteil. Insge-

samt merkt man dieser Technologie ihr Alter und den Ursprung im Unix Umfeld aber an. CGI Anwendungen sind teilweise schwerer zu verstehen und zu warten als bei anderen der hier vorzustellenden Technologien.

Der Vorteil ist, daß man eine große Flexibilität bei der Auswahl der benutzten Programmierumgebung erhält. Bei Einsatz einer interpretierten Sprache (z.B. PERL) entfällt die Programmübersetzung (die bei Servererweiterungen nötig ist). Dadurch können HTML- und Skriptdokumente in der selben Programmierumgebung (im einfachsten Fall ein normaler Texteditor) erstellt und getestet werden. Allerdings sind statische und dynamische Anteile immer noch auf verschiedene Dateien verstreut.

Diesen Nachteil sollen die folgenden Technologien lösen. Einer der ersten Versuche, dynamische Inhalte auf dem Server direkt in die HTML Dateien einzubetten, ist die *Server Side Include - SSI* Technik. Wie der Name schon andeutet, geht es im wesentlichen um Einbettung neuer Inhalte in den ansonsten statischen HTML Teil. Dies geschieht mit Hilfe von speziellen Kommandos, die in einem SGML-Kommentar eingebettet sind. Ein solcher Befehl folgt immer der Form *<!–#command par_1=”val_1” par_2=”val_2” ... par_n=”val_n”–>*. Um z.B. das aktuelle Datum in ein Dokument einzubinden genügt die Anweisung *<!–#echo var=”DATE_LOCAL”–>*. Neben diesem einfachen Beispiel gibt es noch weitere Möglichkeiten. Es ist z.B. möglich, die Ausgabe einer ausführbaren Datei oder eine Textdatei in eine HTML Seite einzubinden. Allerdings ist diese Technologie kein wirklicher Standard. Insbesondere ist nicht genau festgelegt, welche Anweisungen unterstützt werden. Es gibt zwar eine kleine Basis von Standardbefehlen, die alle SSI-fähigen Webserver anbieten. Insgesamt sind allerdings die Möglichkeiten eher begrenzt. Zwar wird eine Erweiterung um eigene Befehle von einigen Anbietern ermöglicht. Allerdings verliert man dadurch die Vorteile von serverseitigen Skripten (getrennte Entwicklungsumgebungen und Übersetzen sind für diese Erweiterungen zumeist nötig). Darüber hinaus ist eine solche Lösung auch nur schwer auf andere Webserver und Plattformen übertragbar. Sowohl das Common Gatway Interface als auch die Server Side Include Techniken sind erste Versuche, dynamische Inhalte auf Seiten des Servers zu ermöglichen. Sie entsprechen aber nicht mehr den heutigen Anforderungen an eine schnelle Entwicklung von leicht wartbaren und sicheren Firmenanwendungen. Deshalb haben sich neuere Möglichkeiten entwickelt, die diesen Anforderungen besser gerecht werden. Diese bauen auf den Erfahrungen der ersten Ansätze auf. Auch Sie ermöglichen das dynamische Erzeugen von HTML Seiten durch eingebettete Befehle. Allerdings ist die Mächtigkeit dieser Technologien größer. Sie stellen komplette Skriptsprachen und Anbindungen an Komponentenmodelle zur Verfügung, so daß damit sehr komplexe Anwendungen umgesetzt werden können.

Die erste der hier vorzustellenden neueren serverseitigen Skriptsprachen sind die sogenannten *Active Server Pages - ASP*. Diese sind von Microsoft im Zusammenhang mit ihrem Webserver[1] eingeführt worden. Wie bei SSI sind ASP normale HTML Dateien mit einge-

[1] Internet Information Server - IIS

bundenen dynamischen Inhalten. Die dynamischen Anteile werden dabei durch <% %> von den statischen getrennt. Innerhalb der Trennzeichen können komplette Skriptsprachen verwendet werden. Dabei ist die zu benutzende Sprache nicht festgelegt. Vielmehr können beliebige Skriptsprachen installiert und dann verwendet werden[1]. Normalerweise stehen dabei Visual Basic Skript und Javaskript zur Verfügung. Daneben werden aber auch andere Skriptsprachen (z.B. PERL) von Drittanbietern zur Verfügung gestellt. Die folgende ASP Datei benutzt Visual Basic Skript um Daten aus einer relationalen Datenbank abzufragen:

```
1.   <HTML>
2.     <HEAD>
3.       <TITLE>ein einfaches ASP Beispiel</TITLE>
4.     </HEAD>
5.     <BODY>
6.       <%
7.         Set conn=Server.CreateObject("ADODB.Connection")
8.         conn.open "tracking", "tracking", "tracking"
9.         Set rs=Server.CreateObject("ADODB.Recordset")
10.        rs.Open "Select * From SYSADM.GRUPPE", conn, 3, 3
11.      %>
12.      <TABLE>
13.        <TR>
14.          <TD>Gruppe</TD>
15.          <TD>Firma</TD>
16.          <TD>Beschreibung</TD>
17.          <TD>Telefon</TD>
18.        </TR>
19.      <%
20.        On Error Resume Next
21.        rs.MoveFirst
22.        do while Not rs.eof
23.      %>
24.        <TR>
25.          <TD><%Server.HTMLEncode(rs.Fields("Gruppe").Value)%></TD>
26.          <TD><%Server.HTMLEncode(rs.Fields("Firma").Value)%></TD>
27.          <TD><%Server.HTMLEncode(rs.Fields("Beschreibung").Value)%></TD>
28.          <TD><%Server.HTMLEncode(rs.Fields("Telefon").Value)%></TD>
29.        </TR>
30.      <%
```

[1] dies wird durch Benutzung des *Windows Scripting Host's* erreicht

```
31.         rs.MoveNext
32.         loop
33.       %>
34.       </TABLE>
35.     </BODY>
36.  </HTML>
```

Das Beispiel zeigt, wie effektiv ASP zur dynamischen Erzeugung von HTML Seiten eingesetzt werden können. Dabei kann die Entwicklung innerhalb der selben Umgebung erfolgen. Der Zusammenhang zwischen statischen und dynamischen Anteilen ist gut erkennbar. Zur Umsetzung von komplexeren Operationen sind Skriptsprachen weniger geeignet. Deshalb besteht hier die Möglichkeit Komponenten einzubeziehen. Wie bei einer Microsofttechnologie zu erwarten, basieren diese Komponenten auf dem COM bzw. DCOM. Komponenten anderer Technologien müssen über entsprechende Brücken bzw. COM-Hüllobjekte eingebunden werden (vgl. Abschnitt 3.5). Im dargestellten Beispiel werden Komponenten zum Datenbankzugriff instanziiert (Zeile 7 und 9). Diese werden zur Ausführung einer einfachen SQL-Abfrage (Zeile 10) benutzt. Mit dem Ergebnis der Abfrage wird dann eine HTML-Tabelle aufgebaut (Zeile 12 bis 34).

Mit Active Server Pages lassen sich also beliebige, HTML basierte Benutzeroberflächen bereitstellen. Die eigentliche Anwendungslogik kann dabei in Komponenten ausgelagert werden. Diese können dann recht einfach eingebunden werden. Der größte Nachteil besteht darin, daß ASP Dateien jedesmal untersucht und ausgeführt werden müssen, wenn ein Klient sie anfordert. Gegenüber dem reinen Dateiserveraufgaben herkömmlicher Webserver entsteht so eine große zusätzliche Belastung. Entsprechende Strategien um diesen Nachteil abzumildern (z.B. Einsatz von serverseitigen Zwischenspeicher und Lastverteilung) sind also sehr wichtig. Dies ist allerdings Sache des Serverherstellers. Ein Anwender muß für ein gegebenes Projekt abschätzen, ob die Leistung bei der zu erwartenden Anzahl von Benutzern ausreicht oder nicht. Für die hier zu entwickelnde Flottenmanagementanwendung ist dies gegeben, da die Anwendung kein Massenprodukt ist. Es werden relativ wenig gleichzeitige Benutzer erwartet. Des weiteren besteht natürlich auch hier die Möglichkeit, die Last auf mehrere Webserver zu verteilen. Dadurch können serverseitige Skriptsprachen auch für Szenarien mit vielen gleichzeitigen Anwendern eingesetzt werden.

Nicht nur von der Bezeichnung vergleichbar ist die *Java Server Page - JSP* Technologie von Sun (siehe z.B. [JSP]). Das folgende Beispiel verdeutlicht die großen Ähnlichkeiten:

```
1.    <HTML>
2.      <HEAD>
3.        <TITLE>ein einfaches JDBC Beispiel</TITLE>
```

```
4.      </HEAD>
5.      <BODY>
6.        <%page language="java"%>
7.        <%page import="java.sql.*"%>
8.        <%
9.          DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
10.         Connection con=DriverManager.getConnection(
11.               "jdbc:oracle:thin:@geo_server:1521:tracking,
12.               "tracking", "tracking");
13.         ResultSet rs=con.createStatement().executeQuery(
14.               "Select * From SYSADM.GRUPPE");
15.       %>
16.       <TABLE>
17.         <TR>
18.           <TD>Gruppe</TD>
19.           <TD>Firma</TD>
20.           <TD>Beschreibung</TD>
21.           <TD>Telefon</TD>
22.         </TR>
23.       <%
24.         while (rs.next()) {
25.       %>
26.         <TR>
27.           <TD><%out.write(""+rs.getInt("Gruppe"));%></TD>
28.           <TD><%out.write(""+rs.getInt("Firma"));%></TD>
29.           <TD><%out.write(""+rs.getString("Beschreibung"));%></TD>
30.           <TD><%out.write(""+rs.getString("Telefon"));%></TD>
31.         </TR>
32.       <%
33.         }
34.       %>
35.       </TABLE>
36.     </BODY>
37.   </HTML>
```

Das Ergebnis entspricht dem ASP Beispiel. Sowohl Active Server Pages als auch Java Server Pages bieten vergleichbare Leistungen. Der Unterschied besteht in den verfügbaren Skriptsprachen. ASP bauen auf dem Windows Skripting Host auf. Dadurch können

Vorteile	Nachteile
dynamische Seiten unabhängig von den Fähigkeiten des Klienten	Server muß Datein vor Übertragung untersuchen und abarbeiten (langsam)
Anbindung an Komponentenmodelle ermöglicht komplexe Anwendungen	
Anwendungslogik auf dem Server versteckt, da nur dynamisch erzeugte Dateien zum Klienten gelangen (kein direkter Zugriff auf sensitive Firmenressourcen möglich)	

Tabelle 4.5.: Vor- und Nachteile von serverseitigen Skriptsprachen

verschiedene Skriptsprachen zum Einsatz kommen. Java Server Pages sind an Java gebunden. Allerdings ist das auch ein großer Vorteil für Java Programmierer. Die innerhalb von Skriptbereichen verwendbare Sprache ist kein Javaskript Dialekt sondern reines Java. Das bedeutet nicht nur gleiche Syntax sondern auch die Möglichkeit der Einbindung von allen Java Bibliotheken. Java Komponenten (Beans) können sehr einfach verwendet werden.

Die Entscheidung zwischen ASP und JSP kann nicht allgemeingültig getroffen werden. Vielmehr müssen die Randbedingungen des jeweiligen Projektes berücksichtigt werden. Wenn die meisten Komponenten das COM benutzen, sind ASP eine natürliche Wahl. Insgesamt passen ASP gut mit anderen Microsoft Technologien zusammen. JSP hingegen sind eine gute Wahl, wenn ein Großteil des Projekts sowieso in Java geschrieben wird. Des weiteren bieten JSP auch die Möglichkeit der direkten Einbindung von verschiedenen Komponenten (basierend auf CORBA, RMI, EJB, ...) und stehen auf vielen Plattformen zur Verfügung.

Interessant ist eine Einordnung der serverseitigen Skriptsprachen im Vergleich zu Servererweiterungen. Normalerweise wird eine größere Anwendung beide Technologien benutzen. Als Entscheidungshilfe, welcher Teil der Anwendung mit welchen Verfahren umgesetzt wird, müssen folgende Punkte berücksichtigt werden:

1. Verhältnis der statischen Anteile zu den dynamischen
2. Häufigkeit nötiger Aktualisierungen
3. mögliche Einsatzfelder (eher begrenzt oder vielfältig einsetzbar)

Ein Beispiel zum Einsatz von Servererweiterungen sind die serverseitigen Skriptsprachen selber. Diese, auf den ersten Blick vielleicht seltsame Aussage, bezieht sich auf die Art der Einbindung von Skriptsprachen in den Webserver. Unterstützung für JSP wird meistens als ein Servlet bereitgestellt (für ASP analog als ISAPI Erweiterung). Auf diese Weise kann

diese Komponente mit allen Servlet fähigen Webservern benutzt werden. Eine JSP Komponente hat keinerlei statische Anteile. Vielmehr wird die gesamte Ausgabe dynamisch aufgrund der Eingabedatei (der .JSP Datei) erzeugt. Außerdem kann eine solche Komponente einmal entwickelt und dann an vielen Stellen eingesetzt werden. Deshalb wird man sie als Servererweiterung umsetzen. Dahingegen würde eine Komponente, die zur Anmeldung eines Benutzers dient, eher als serverseitiges Skript umgesetzt werden. Der Einsatz ist relativ begrenzt und der Anteil statischer Elemente (HTML Formular) überwiegt.

4.1.4.1. Ausblick

Der Einsatz neuerer Server Skriptsprachen hat für den Entwickler den Vorteil einer homogenen Entwicklungsumgebung für statische und dynamische Anteile. Er pflegt jeweils die selben Dateien. Allerdings werden innerhalb dieser Dateien die Skriptanteile anders behandelt als die statischen HTML Fragmente. Die dynamischen Teile werden durch spezielle Kennzeichen (z.B. <% %>) vom Rest getrennt. Innerhalb dieser Kennzeichen sind die Skripte als reiner Text, ohne weitere Struktur abgelegt. Dies bedingt, daß diese Teile bei der automatischen Auswertung oder Generierung gesondert behandelt werden müssen. Besser wäre es, wenn auch die Skriptanteile in einer SGML/XML Notation formuliert wären. Dadurch gäbe es auch bei der automatischen Auswertung keine Unterschiede zwischen dynamischen und statischen Anteilen. Entsprechende Ansätze (basierend auf XML) sind bereits in Entwicklung (siehe z.B. [MIVA]) und werden sicherlich mittel- bis langfristig die hier vorgestellten Technologien ersetzen.

4.2. Entwurf und Umsetzung der Benutzeroberfläche

Die bisher vorgestellten Technologien ermöglichen jeweils die Erstellung von webbasierten Benutzeroberflächen. Es ist allerdings nicht möglich einen dieser Ansätze als besten auszuwählen. Vielmehr sollte ein Entwickler alle Möglichkeiten kennen und daraus die für sein Projekt geeigneten auswählen. Eine Internetanwendung besteht aus einer sinnvollen Kombination verschiedener Techniken. Zur Einbindung der darzustellenden Daten und Erzeugung der HTML Seiten eignen sich Server Seitige Skriptsprachen. Die dadurch erzeugten Seiten können dann Java Applets oder Skriptsprachen zur verbesserten Darstellung und Validierung von Benutzereingaben enthalten. Der Prozeß der Entwicklung einer Internetanwendung besteht also in dem Finden der richtigen Kombination der vorgestellten Techniken. Die Auswahl erfolgt dabei nach folgenden Kriterien:

1. erwartete Ausstattung der Klienten
2. verfügbare Bandbreite
3. notwendige Anbindung an vorhandene Unternehmensdaten

4. einfache und übersichtliche Bedienbarkeit

Bei dem Entwurf der Anwendung sollte man also unbedingt immer den Nutzer im Vordergrund sehen. Wenn durch den Einsatz von Klienten Seitigen Technologien die Zahl der möglichen Benutzer stark eingeschränkt wird, sollte deren Einsatz noch mal überdacht werden. Allerdings hängt dies stark von dem jeweiligen Einsatzgebiet ab (z.B. kann eine Beschränkung auf nur eine Klientenplattform in einem Intranet durchaus praktikabel sein). Die verfügbare Bandbreite ist ebenfalls sehr wichtig. Die schönste Benutzeroberfläche wird vom Benutzer nicht akzeptiert, wenn die Darstellung zu lange dauert. Für den Einsatz in einem Unternehmen ist es wichtig, daß bereits vorhandene Datenquellen mit einbezogen werden können. Schließlich ist eine einfache Bedienung Voraussetzung für die Akzeptanz der Anwendung durch den Benutzer. Die Steuerungsmöglichkeit muß sich auch nach dem zu erwartenden Kenntnisstand der Benutzer richten. Eventuell ist die Entwicklung mehrerer Möglichkeiten sinnvoll (z.B. ein einfacher, menügesteuerter Zugang für Einsteiger und ein direkter Zugriff für Profis). Für eine Einführung zur Entwicklung von Benutzeroberflächen ist z.B. das Buch [MAY92] empfehlenswert.

Alle diese Kriterien müssen bei der Erstellung einer Internetanwendung berücksichtigt werden. Die hier zu entwerfende Flottenmanagementanwendung soll leicht zu bedienen sein. Dies erfordert eine einfache, klar strukturierte Benutzeroberfläche. Außerdem muß eine solche Anwendung schnell auf Benutzeranfragen reagieren. Die Benutzerakzeptanz sinkt, wenn der Aufbau der Oberfläche zu lange dauert. Aus diesem Grund soll auf Seiten des Klienten möglichst nur HTML benutzt werden. Falls für eine einfache Bedienung nötig, können dann noch klientenseitige Skriptsprachen und Java Applets verwendet werden. Allerdings sollte insbesondere der Einsatz von Applets (wenn möglich) vermieden werden. Dies liegt einerseits an der längeren Ladezeit (durch Starten der Virtuellen Maschine, Laden der Klassendateien, ...). Andererseits ist eine nahtlose Einbindung in HTML Seiten schwierig. Die Koordinierung der benutzten Farben, Schriftarten und Seitenhintergründe erfordert einigen Aufwand. Deshalb soll der Hauptteil der Anwendung aus dynamischen Anteilen auf Seiten des Servers (Servererweiterungen und Serverskriptsprachen) und statischen auf Seiten des Klienten (HTML) bestehen. Da die zugrundeliegenden Anwendungskomponenten Java benutzen, bieten sich auch andere Java basierende Produkte an. Deshalb werden vorwiegend Servlets und JSP Dateien serverseitig Verwendung finden.

Zur Demonstration hier eine typische Datei des Projektes:

```
1.  <%@ page language="java" %>
2.
3.  <HTML>
4.  <HEAD>
5.      <TITLE>TT-Tracking : verfügbaren Fahrstrecken</TITLE>
6.      <LINK REL=stylesheet HREF="/styles/tracking.css" TYPE="text/css">
```

```
7.  </HEAD>
8.
9.  <BODY>
10.
11.  <%@ include file="../snippets/tracking/appglobals.jsp" %>
12.
13.  <%
14.     String sDta=null;
15.     int carNr=new Integer(request.getParameter("value")).intValue();
16.     IReport rep=report.getReportRoutesForCar(carNr);
17.     rep.run();
18.     IXMLDocument repdoc=rep.toXML();
19.     repdoc.transform(sBaseUrl+"/xsl/table_with_select.xsl");
20.     sDta=repdoc.toString();
21.  %>
22.
23.  <%
24.    String sTitleText="gefahrene Strecken für Auto "+carNr;
25.    String sAction=sBaseUrl+"/tracking/"+"appcentral.jsp";
26.    String sTarget="data";
27.  %>
28.
29.  <%@ include file="../snippets/header.jsp" %>
30.
31.  <h2>alle Routen für das Auto Nr. <%out.write(""+carNr);%></h2>
32.
33.  <FORM action="<%out.write(sAction);%>"
           target="<%out.write(sTarget);%>"
           methode="POST">

34.  <%
35.     out.write(sDta);
36.  %>
37.  <BR>
38.  <INPUT TYPE="HIDDEN" NAME="command" VALUE="showRoute"></INPUT>
39.  <INPUT TYPE="SUBMIT" VALUE="Anzeigen"></INPUT>
40.
41.  </FORM>
42.
```

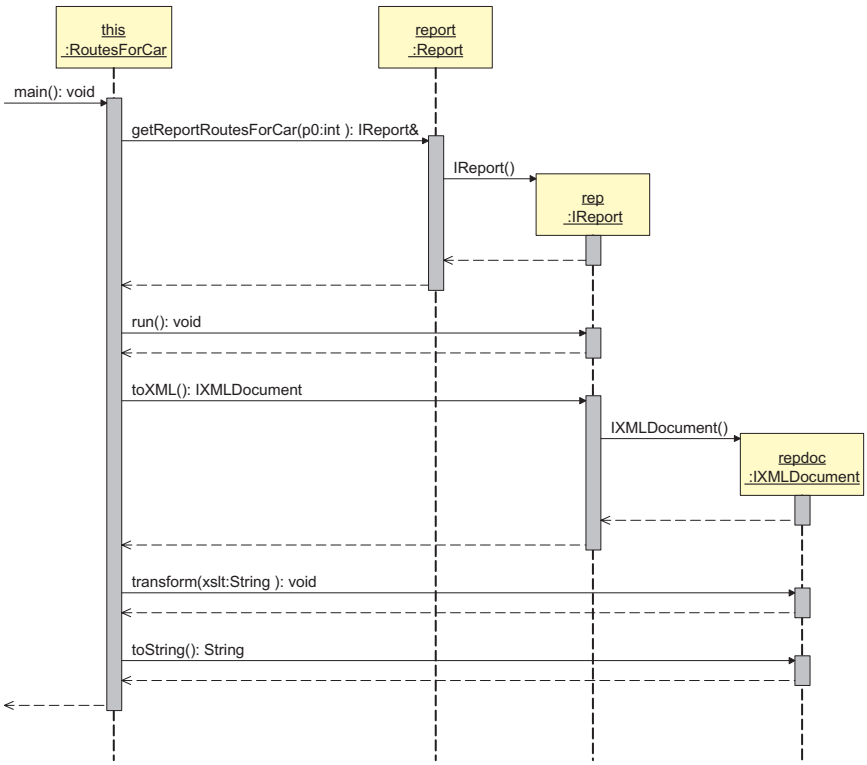


Abbildung 4.3.: dynamischer Ablauf der dargestellten JSP Datei

```
43.  <%@ include file="../snippets/footer.jsp" %>
44.  </BODY>
45.  </HTML>
```

Diese Datei dient dazu, die verfügbaren Fahrstrecken für ein Auto aufzulisten. Aus dieser Auswahl kann der Benutzer dann eine Strecke wählen. Danach wird die Seite zur Anzeige der Fahrstrecke aufgerufen . Zuerst wird ein typischer HTML Vorspann erzeugt (Zeile 1-10). In Zeile 11 wird eine Datei eingebunden, die benutzte globale Variablen (sogenannte *session beans*) einbindet. Diese speichern den gesamten Status der Anwendung (da HTTP ein Sitzungsloses Protokoll ist). Zwischen Zeile 13 bis 21 wird die eigentliche Arbeit ausgeführt (Der im folgenden dargestellte dynamische Anteil ist auch in Abbildung 4.3 dargestellt). Zuerst wird die Fahrzeugnummer aus den übergebenen Parametern extrahiert. Danach wird das Report Bean *report* (eine Anwendung des Report Basisdienstes) dazu benutzt, einen Report zur Auflistung der Fahrstrecken für dieses Fahrzeug zu erzeugen (Zeile 16). Nachdem dieser Report ausgeführt ist, wird in Zeile 18 das Ergebnis als XML Dokument abgerufen. Dieses Dokument wird in Zeile 19 mit einer XSLT Datei transformiert. Das XML Dokument wird dadurch in ein gültiges HTML Fragment transformiert. Dieses Fragment stellt die verfügbaren Fahrstrecken in einer Tabelle dar. Des weiteren besteht die Möglichkeit, eine Zeile auszuwählen. Damit dieses Fragment in die Datei eingebunden

werden kann, wird es in Zeile 20 in einer Zeichenkettenvariable abgelegt. Die restlichen Zeilen dienen dazu, das Ergebnis darzustellen. Zwischen Zeile 33 und 41 wird das eigentliche Formular dargestellt. Dieses enthält neben der erzeugten Tabelle (Zeile 35) auch einen Schalter, mit dem die aktuelle Auswahl angezeigt werden kann (Zeile 39). Zeile 43 bis 45 schließen die erzeugte HTML Datei ab. Die Ausgabe auf Seiten des Klienten sieht dann z.B. so aus:

```
<HTML>
<HEAD>
    <TITLE>TT-Tracking : verfügbaren Fahrstrecken</TITLE>
    <LINK REL=stylesheet HREF="/styles/tracking.css" TYPE="text/css">
</HEAD>

<BODY>

<center>gefahrene Strecken für Auto 1</center>
<img src="/images/mapxtreme.gif" width="100%" height="3">

<h2>alle Routen für das Auto Nr. 1</h2>

<FORM action="http://192.168.1.10:80/tracking/appcentral.jsp"
      target="data"
      methode="POST">

<TABLE border="1" cellspacing="1" cellpadding="2">
   <TR>
      <TH>.</TH>
      <TH>Anfang</TH>
      <TH>Ende</TH>
      <TH>Pos.Daten</TH>
   </TR>
   <TR>
      <TD>
         <INPUT TYPE="RADIO" NAME="key" VALUE="ROUTE=1&#38;GRUPPE=1"/>
      </TD>
      <TD>1999-08-02 14:48:52.0</TD>
      <TD>1999-08-02 15:18:59.0</TD>
      <TD>7</TD>
   </TR>
   <TR>
```

```
      <TD>
         <INPUT TYPE="RADIO" NAME="key" VALUE="ROUTE=6&#38;GRUPPE=1"/>
      </TD>
      <TD>1999-08-02 17:00:00.0</TD>
      <TD>1999-08-02 17:00:00.0</TD>
      <TD>1</TD>
   </TR>
</TABLE>

<BR>
<INPUT TYPE="HIDDEN" NAME="command" VALUE="showRoute"></INPUT>
<INPUT TYPE="SUBMIT" VALUE="Anzeigen"></INPUT>

</FORM>

<hr>
<a href="/index.jsp"><img src="/icons/navbar/home.gif"></a>
<br>
(c)1999 Technotrend Systemtechnik GmbH
<br>
<br>
<img src="/images/banner.gif" height=20></img>
<img src="/images/mapxtreme.gif" height=20></img>

</BODY>
</HTML>
```

Die gesamte Anwendungslogik ist versteckt. Der Klient kann nicht mehr erkennen, wie diese Datei erzeugt wurde.

Der dargestellte Ablauf ist typisch für die gesamte Anwendung. Innerhalb der JSP Dateien werden nur die für die Repräsentation nötigsten Aktionen durchgeführt. Alle aufwendigeren Algorithmen werden als Java Bean Komponenten eingebunden. Diese wiederum bieten eine Schnittstelle zu den entwickelten Basisdiensten.

4.3. Verwendete Infrastruktur und Produkte

Neben den vorgestellten Verfahren und Technologien sind natürlich auch die zur Verfügung stehenden Produkte zu deren Umsetzung wichtig. Im Rahmen dieser Arbeit wurden verschiedene Produkte untersucht. Die Ergebnisse sollen im folgenden kurz vorgestellt werden.

4.3.1. Entwicklungsumgebungen

Die grundlegenden Werkzeuge für die Entwicklung von Java Komponenten werden von der Firma Sun kostenlos verteilt. Dieses *Java Developement Kit - JDK* wird mit allen nötigen Werkzeugen und einer Virtuellen Maschine ausgeliefert. Allerdings kann die Produktivität recht eingeschränkt sein, da keinerlei graphische Benutzerführung vorhanden ist. Je nach Einsatzgebiet können hier spezialisierte Entwicklungsumgebungen helfen. Insbesondere die Erstellung von Oberflächen wird dadurch beschleunigt. Aber auch bei der Komponentenentwicklung kann die Entwicklungszeit mit entsprechender Werkzeugunterstützung verringert werden. Die Schnittstellen können visuell umgesetzt werden. Das Werkzeug kann dabei die nötigen Programm- und Schnittstellendateien weitestgehend automatisch pflegen. Der Entwickler kann jetzt darauf aufbauend die eigentliche Funktionalität umsetzen. Dadurch können auch Anwendungsentwickler, die nur wenig Erfahrung mit der jeweiligen Komponententechnologie haben, diese verwenden. Die Hauptaufgabe des Komponentenentwicklers ist dann die Analyse des Anwendungsumfeldes und das Design der Schnittstellen. Einige Werkzeuge unterstützen dabei sowohl die Entwurfsphase (meistens basierend auf der UML Notation) als auch die eigentliche Umsetzung.

Stellvertretend für andere Entwicklungsumgebungen wurde die *JBuilder Client-Server Edition* von Inprise getestet. Dieses Werkzeug ist für eine Entwicklung verteilter Anwendung prädestiniert. Es wird zusammen mit einem CORBA kompatiblen ORB ausgeliefert (Visibroker). Die Integration des ORB in die Entwicklungsumgebung ist gut gelöst. Die Entwicklung erfolgt größtenteils menügeführt. Der Entwickler wird von Routineaufgaben befreit und kann sich um die eigentliche Geschäftslogik kümmern. Dadurch kann die eigentliche Umsetzung verteilter Anwendungen recht schnell erfolgen.

4.3.2. Produkte und Bibliotheken für die mittlere und Datenschicht

4.3.2.1. ORB

Der Object Request Broker ist der zentrale Bestandteil der gesamten Anwendung. Er bestimmt maßgeblich die Geschwindigkeit, Portabilität und Flexibilität der Gesamtlösung. Deshalb muß eine sorgfältige Auswahl getroffen werden. Dies ist um so schwerer, da es mittlerweile recht viele Anbieter gibt. Dabei lassen sich die Produkt grob in zwei Lager einteilen - die kommerziellen und die frei verfügbaren. Die kommerziellen Produkte haben den Vorteil einer besseren Unterstützung durch den Hersteller. Zusätzlich bieten sie oftmals eine bessere Unterstützung für Geschäftsanwendungen (z.B. verschlüsselte Verbindungen mittels SSL[1] oder Anbindung an andere Komponentenmodelle). Der Vorteil der frei verfügbaren Lösungen ist deren größere Dynamik. Sie haben oftmals eine bessere Unterstützung neuester Technologien. So ist eine Unterstützung des POA bei vielen kommerziellen Anbietern erst in der Entwicklung während schon einige frei verfügbare Umsetzungen existieren

[1] Secure Socket Layer, ein Protokoll zum Aufbau einer sicheren TCP-IP Verbindung

(Dies liegt sicherlich auch an der besseren Qualitätskontrolle und umfangreicher Testphasen kommerzieller Anbieter).

Im Rahmen dieser Arbeit wurden verschiedene Produkte untersucht. Stellvertretend sollen hier kurz zwei Produkte vorgestellt werden. Für das kommerzielle Lager ist dies der Visibroker ORB von Visigenic (Version 3.4 für Java, vgl. [VISI]). Dieser stellt neben einem CORBA 2.0 kompatiblen ORB auch verschiedene CORBA Services zur Verfügung. Dies betrifft z.B. Dienste zur Lokalisation von Objektinstanzen (Namingservice), für Ereignisse (Eventservice) und Transaktionen (Integrated Transactionservice). Als Erweiterung stehen außerdem Komponenten zur Datensicherheit zur Verfügung. Dies betrifft z.B. eine SSL Komponente und den sogenannten *Visibroker Gatekeeper*. Der Gatekeeper ermöglicht, IIOP Anfragen über HTTP zu übertragen (HTTP Tunneling). Dadurch wird der Einsatz von CORBA zwischen Firewall geschützten Netzen vereinfacht. Visibroker ist einer der wenigen ORB's, die eine recht gute Werkzeugunterstützung bieten. Einerseits gibt es bereits mitgelieferte Werkzeuge zur Verwaltung der verschiedenen Komponenten. Des weiteren ist eine gute Einbindung in Inprise Entwicklungsumgebungen (JBuilder, Delphi, ...) gegeben. Dadurch wird die visuelle Entwicklung verteilter Komponenten ermöglicht. Deshalb lohnt sich diese Ausgabe für eine professionelle Entwicklung.

Für kleinere Projekte sind diese Leistungen allerdings nicht immer nötig. In diesem Fall reicht auch eine der frei verfügbaren Lösungen aus. Diese bieten meistens nur den eigentlichen ORB mit nur wenigen zusätzlichen Diensten oder Werkzeugen. Allerdings besteht u.U. der Vorteil, daß der ORB die aktuelle Version der CORBA Spezifikation berücksichtigt. Die Entwicklung ist gegenüber kommerziellen Produkten dynamischer. Als Beispiel dient dafür z.B. der *JavaOrb* der *Distributed Object Group* (vgl. [JORB]). Dieses Produkt ist frei verwendbar für kommerzielle und nicht kommerzielle Anwendungen. In der getesteten Version 1.2.5 unterstützt der ORB den CORBA Standard in der Version 2.2. Damit enthält er u.a. auch bereits einen POA. Darüber hinaus werden Naming-, Event- und Transaktionsdienste angeboten. Durch diese Leistungsmerkmale ist er von der technischen Seite gut für komplexe Anwendungen geeignet. Allerdings gibt es kaum Unterstützung bei der Entwicklung. Außer den nötigsten Kommandozeilentools gibt es keine Werkzeuge. Eine visuelle Entwicklung ist damit nicht ohne weiteres möglich. Des weiteren fehlen auch einige der für Geschäftsanwendungen wichtigen Dienste. Insbesondere eine SSL Komponente ist eigentlich für professionelle Anwendungen unumgänglich.

Als Fazit gilt also, daß die freien Lösungen gut für kleinere Anwendungen und Prototypen geeignet sind. Wegen der besseren Integration in Entwicklungsumgebungen und zusätzlicher Dienste für Geschäftsanwendungen sollte man aber bei professionellem Einsatz kommerzielle Produkte einsetzen.

4.3.2.2. Datenbank

Für eine Flottenmanagementanwendung fallen verschiedenste Datenarten an. Diese müssen persistent abgelegt werden. Im Rahmen dieser Arbeit wird dafür eine relationale Datenbank benutzt (im Detail: Oracle 8i). Für eine objektorientierte Anwendung würde eigentlich eine objektorientierte Datenbank vorrangig in Frage kommen. Damit kann ein Programm dann mit den selben Objekten arbeiten, die auch in der Datenbank abgelegt werden (vgl. z.B. [POET]). Allerdings ist hierbei auch das Anwendungsumfeld zu berücksichtigen. In der Umgebung, in der die Anwendung eingesetzt werden soll, werden bereits relationale Datenbanken eingesetzt. Um hier eine völlige Umstrukturierung zu vermeiden, baut auch diese Anwendung darauf auf. Außerdem haben relationale Datenbanken den Vorteil, daß sie schon länger am Markt sind. Diese Produkte sind oftmals bereits seit Jahrzehnten in der Entwicklung. Dadurch sind sie kaum zu schlagen in Punkten wie Sicherheit, Schnelligkeit und Skalierbarkeit. Die sich aus der Verwendung einer relationalen Datenbank ergebenden Anforderungen für eine OO-Entwicklung werden im Abschnitt über die Datenbankschnittstelle behandelt (Abschnitt 3.2.3). Im folgenden sollen einige der interessanteren Möglichkeiten und Gründe besprochen werden, warum gerade Oracle 8i eingesetzt wird.

Für die hier umzusetzende Beispielanwendung reicht im Prinzip eine beliebige relationale Datenbank aus. Allerdings ist Oracle gerade dabei, ihre Datenbank in eine *Alles in Einem* Lösung für Internetanwendungen zu verwandeln. Zentraler Punkt dabei ist ein virtueller Java Prozessor im Kern der Datenbank. Dadurch können Javakomponenten direkt in der Datenbank ablaufen. Dies bringt mehrere Vorteile mit sich. Zum einen beschleunigt sich dadurch der Zugriff auf die Daten. Zum anderen stehen die selben Mechanismen für die Prozesse zur Verfügung, die auch für die Daten da sind. Zugriffsbeschränkungen und Transaktionen lassen sich so in einer einheitlichen Art und Weise festlegen. Als Komponentenmodell enthält die Datenbank einen ORB, aufbauend auf dem bereits erwähnten Visibroker. Darauf aufbauend wird dann als zusätzliche, dünne Schicht eine Enterprise Java Beans Anwendungsschnittstelle angeboten.

Neben dieser grundlegenden Architektur sollen auch weitere interessante Leistungen integriert werden. Dies betrifft zum Beispiel die Einbindung der Extensible Markup Language. Dadurch ist ein einfacher Datenaustausch möglich. Sowohl das Erzeugen als auch das Einlesen von XML-Dateien wird unterstützt. Mittels eines eigenen Dateisystems (*Internet File System - iFS*) kann dann die Datenbank wie ein normales Dateisystem angesprochen werden. XML Dateien können einfach in der Datenbank abgelegt und wieder ausgelesen werden.

Die Zielrichtung von Oracle 8i ist also, innerhalb der Datenbank einen kompletten Anwendungsserver bereitzustellen. Wenn man jetzt noch einen Webserver in der virtuellen Maschine installiert, bietet Oracle 8i die Infrastruktur für alle drei Ebenen der drei Schichten Architektur. Insbesondere für kleinere bis mittelgroße Projekte ist dies ein interessanter Weg. Dadurch können die Kosten für Wartung und Integration der Anwendung reduziert

werden. Allerdings wird voraussichtlich erst zum Ende dieses Jahres eine Version auf den Markt kommen, die diese hohen Erwartungen erfüllen kann. Durch konsequenten Einsatz von serverseitigen Java in Zusammenhang mit einem Plattformunabhängigen Komponentenmodell kann allerdings bei Bedarf eine Integration der Flottenmanagementanwendung in Oracle recht schnell erfolgen. Dies erscheint insbesondere deshalb recht interessant, da der GIS Anbieter (Mapinfo) plant, seinen GIS-Server ebenfalls in diese Umgebung einzubetten.

Trotz dieser Vorteile die Oracle im Moment noch gegenüber anderen Mitbewerbern hat, ist ein Einsatz anderer Datenbanken möglich. Ein Hauptziel der Entwicklung besteht darin, eine offene und zukunftsichere Lösung zu erreichen. Dies gilt natürlich auch für die Datenbank. Der Einsatz einer anderen Datenbank ist problemlos möglich, wenn ein JDBC Treiber verfügbar ist. Dies gilt für die meisten Datenbanken. Wenn der Hersteller selber keinen bereitstellt (z.B. bei Microsoft Sql Server) gibt es oft Drittanbieter, die diese Lücke füllen. Außerdem würde die entwickelte Architektur auch die Anbindung über nicht JDBC Schnittstellen ermöglichen (allerdings steigt der nötige Aufwand). Zusammen mit den in Abschnitt 3.5 vorgestellten Möglichkeiten lassen sich dadurch prinzipiell alle relationalen Datenbanken mit einer Programmierschnittstelle einbinden.

4.3.2.3. GIS

Ein geographisches Informationssystem dient der graphischen Darstellung und Manipulation von ortsbezogenen Daten. Für eine Flottenmanagementanwendung liegt der Schwerpunkt bei der Visualisierung. Eine Darstellung des vorhandenen Kartenmaterials ist Grundvoraussetzung. Darüber hinaus müssen auch thematische Karten, also die Markierung von Kartenteilen nach bestimmten Kriterien, möglich sein. Dies muß möglich sein, ohne große Programmbibliotheken auf Seiten des Klienten einzusetzen. Deshalb müssen die Daten auf dem Server vorverarbeitet werden. Für größtmögliche Portabilität sollte auf dem Server eine Bilddatei erzeugt werden, die dann z.B. mittels HTTP auf den Klienten übertragen wird. Diesen Weg geht auch die hier benutzte Bibliothek - *MapXtreme für Java* der Firma Mapinfo. Dieses vollständig in Java umgesetzte System arbeitet in einer Klienten Server Architektur. Es gibt einen Server, der die eigentlichen Berechnungen und Darstellungen übernimmt. Ein Klient kann diese Dienste über eine TCP/IP Verbindung in Anspruch nehmen. Das Erzeugen von thematischen Karten wird unterstützt. Des weiteren ist es möglich, zeitweise neue Objekte einzufügen. Dadurch können z.B. die letzten bekannten Positionen der Fahrzeuge auf der Karte dargestellt werden. Durch eine reine Java Lösung ist ein Einsatz nicht auf eine bestimmte Plattform begrenzt. Eine Einbindung in die unterschiedlichsten Umfelder ist denkbar. Eine Partnerschaft zwischen Oracle und Mapinfo sieht z.B. vor, daß der Server direkt in der virtuellen Maschine der Datenbank läuft. Insbesondere beim gleichzeitigen Einsatz von Oracle als Spatialware-Plattform[1], kann dadurch eine gu-

[1] Normalerweise werden geographische Daten in eigenen Dateien abgelegt. Bei Einsatz eines Spatialware-Servers kann dies entfallen. Die Daten werden dann direkt in der Datenbank abgelegt und von dort visualisiert.

te Integration geographischer und nicht geographischer Daten erreicht werden. Dadurch verschwinden die Unterschiede zwischen diesen Datenarten. Geographische Daten können genauso abgefragt und durchsucht werden wie alle anderen Daten.

4.3.2.4. GSM

Zur Kommunikation mit der Fahrzeugflotte wird ein GSM Dienst benutzt. Die Kommunikation erfolgt dabei entweder über e-Mail oder direkt über ein Modem (vgl. Abschnitt 2.2.1). Die dafür nötigen Bibliotheken gehören zum Standardumfang der Java 2EE von Sun. Für den Zugriff auf einen Mailserver dient die Java Mail API. Für Zugriff auf ein Modem kann die Java Communications API benutzt werden. Beide Bibliotheken werden auch mit den wichtigsten Treibern ausgeliefert (z.B. für POP/IMAP oder Modemzugriff). Die Entwicklung von Anwendungen zur elektronischen Kommunikation kann so ohne Einbindung von Fremdbibliotheken erfolgen.

4.3.3. Klientenschicht

4.3.3.1. Webserver

Die ersten Webserver waren reine Dateiserver. Ein Klient konnte so Inhalte abrufen, die auf dem Server abgelegt sind. Dies reicht für eine Flottenmanagementanwendung nicht aus. Der Klient muß Inhalte interaktiv erstellen und aus einer Datenbank abrufen können. Dafür muß der Webserver dynamische Inhalte unterstützen. Für die Flottenmanagementanwendung sollen dabei die Server Seitigen Java Technologien benutzt werden (vgl. Abschnitt 4.1). Deshalb muß der verwendete Webserver die Einbeziehung der *Servlet* und *Java Server Pages* Technologien ermöglichen. Dabei bestehen prinzipiell zwei Möglichkeiten. Einerseits kann ein Webserver durch externe Erweiterungen diese Fähigkeiten erhalten. Des weiteren haben viele neuere Webserver schon eine eingebaute Unterstützung für diese Technologien.

Für das hier zu umzusetzende Projekt soll der *Java Webserver* von Sun in der Version 2.0 Verwendung finden. Da dieser von der selben Firma die auch Java entwickelt herausgegeben wird, ist eine gute Unterstützung der aktuellen Spezifikationen gewährleistet. Eine Installation und Wartung wird durch diese Integration vereinfacht. Außerdem bietet dieser Server eine graphische Benutzeroberfläche mit der die Konfiguration erheblich vereinfacht wird. Ein Nachteil ist allerdings, daß er kostenpflichtig ist. Insbesondere gegenüber kostenlos erhältlichen Angeboten wie dem *Internet Information Server* von Microsoft ist dies ein Nachteil. Allerdings würde die Konfiguration des IIS zum Einsatz von Servlets und JSP erheblichen Aufwand bedeuten. Außerdem wird diese Unterstützung nur durch Fremdanbieter ermöglicht, so daß es bei einem professionellen Einsatz zu Problemen bei der Integration und Stabilität des Gesamtsystems kommen kann. Dieser Mehraufwand bei

der Wartung und Konfiguration ist der Grund dafür, daß eine Komplettlösung wie der Java Webserver doch empfehlenswert ist.

Durch den Einsatz von genau spezifizierten Technologien wird der Austausch der Serversoftware erleichtert. Dies ist insbesondere deshalb wichtig, da nicht sicher ist, daß Sun dieses Produkt weiterführt. In diesem Fall kann man dann ohne größere Umstellungen das Projekt auf einer anderen Plattform bereitstellen.

4.3.3.2. Klientenzugriff

Ein Hauptziel der Entwicklung besteht darin, einem Benutzer die Möglichkeit zu geben die Anwendung von vielen verschiedenen Plattformen zu benutzen. Demzufolge sind die klientenseitigen Anforderungen recht allgemein. Prinzipiell kann jeder Internetfähige Arbeitsplatz benutzt werden. Allerdings sollte die Anwendung sich auf die Unterstützung der gängigsten Browser (den Internet Explorer von Microsoft und den Navigator von AOL/Netscape) konzentrieren. Eine optimale Präsentation der Anwendung auf diesen beiden Plattformen ist wichtig, da deren Marktanteil zusammen bei über 90% liegt (vgl. z.B. [BSTAT]). Insbesondere die Testphase sollte darauf ausgelegt werden. Dies gilt insbesondere deshalb, da bei einer Flottenmanagementanwendung oftmals die möglichen Benutzer bekannt sind. In einem solchen Umfeld ist auch eine Beeinflussung der benutzten Zugriffsprogramme eine Option. Die Benutzung anderer Browser sollte trotzdem möglich sein, allerdings ohne extra getestet zu werden. Dadurch kann ein unverhältnismäßig großer Aufwand bei der Entwicklung und Qualitätssicherung vermieden werden.

4.3.4. Fahrzeughardware

Im Zentrum der Lösung innerhalb eines Fahrzeuges steht das GSM-Modul Falcom A1 (vgl. [WI99]). Über eine lokale Schnittstelle wird daran ein GPS Empfänger angeschlossen (speziell der Garmin GPS 35). Innerhalb des GSM-Moduls existieren sowohl Programm- (Flash ROM) als auch Datenspeicher (RAM). Dadurch kann die gesamte Datenerfassung durch dieses Modul gesteuert werden. Falls zu einem Zeitpunkt mehr Daten empfangen werden, als gesendet werden können, besteht die Möglichkeit der Zwischenspeicherung. Die installierte Software dient nicht nur zur direkten Übertragung der Daten. Vielmehr sind auch Strategien umgesetzt, die die Übertragungskosten minimieren. Dafür werden nicht alle Positionsdaten übermittelt. Es gibt ein Intervall in dem Positionen übertragen werden. Dieses kann zwischen Einer Minute bis einer halben Stunde gewählt werden. Außerdem wird versucht, entstehende Fehler einer Positionserfassung zu minimieren. Dafür wird nicht eine Position übertragen, sondern der Mittelwert eines Zeitfensters. Bei einer angenommenen Gleichverteilung der Fehler kann dadurch die Qualität verbessert werden.

4.3.5. Serverplattform

Neben der Hardware die innerhalb eines Fahrzeuges Verwendung findet, ist auch die Serverhardware von Bedeutung. Allerdings ist dieser Punkt relativ unkritisch. Ein Hauptziel der Entwicklung bestand darin, eine Plattformunabhängige und portable Lösung zu erhalten. Demzufolge wurde auf Technologien und Produkten aufgebaut, die für viele verschiedene Rechner und Betriebssysteme zur Verfügung stehen. Für die ersten Tests reicht bereits ein PC mit einem Standardbetriebssystem aus. Auf diesem Rechner kann sowohl der Webserver, Anwendungsserver, Datenbankserver, GIS-Server als auch der Flottenserver vereint werden. Für eine spätere kommerzielle Nutzung sollte aber zumindest der Webserver auf einen eigenen Rechner ausgelagert werden. Dies ergibt sich aus dem kurz vorgestellten Zugriffsszenario aus Abbildung 4.1. Nur der Webserver soll einem möglichen Angreifer zugänglich sein. Alle anderen Server, und damit die sensitiven Firmendaten, sollen hinter einer Firewall versteckt sein. Die Sicherheit kann weiter erhöht werden, wenn der Anwendungsserver ebenfalls separiert wird (da dieser von außerhalb der Firewall erreichbar sein muß). Eine darüber hinausgehende Aufteilung der Anwendung auf verschiedene Rechner wird durch die benutzten Technologien und Werkzeuge unterstützt. Damit kann recht gut eine Anpassung an den tatsächlichen Ressourcenbedarf erfolgen.

5. Zusammenfassung

Was ist das Ergebnis dieser Arbeit? Der wohl größte Teil der Arbeit beschäftigt sich mit der Erstellung einer Internetanwendung. Dabei wird der gesamte Entwicklungsprozeß dargestellt. Zuerst werden die allgemeinen Anforderungen an die Anwendung festgelegt. Für die Flottenmanagementanwendung ist besonders wichtig, daß eine gut erweiterbare und flexible Lösung entsteht. Im nächsten Schritt muß jetzt eine Anwendungsarchitektur gewählt werden, welche die Umsetzung der Anforderungen unterstützt. Dabei zeigt sich, daß eine drei Schichten Architektur dafür geeignet ist. Der größte Vorteil dieses Ansatzes ist, daß vorhandene Ressourcen effektiver auf mehrere Klienten verteilt werden können. Gegenüber dem reinen Klienten Server Ansatz ist es einfacher, daß Klienten sich die Ressourcen teilen. Für größere Anwendungen kann eine Lastverteilung eingeführt werden. Dies wird dadurch unterstützt, daß die Anwendung aus einer Sammlung von Komponenten aufgebaut ist. Dadurch wird eine spätere Erweiterung und Aufteilung auf mehrere Rechner gut unterstützt. Im Ergebnis erhält man also eine gut an Nutzerwünsche anpaßbare Anwendung.

Durch eine durchgängig objektorientierte Umsetzung kann eine gute Aufteilung der Anwendung erreicht werden. Eine Kapselung und Bündelung der Daten und der dazugehörigen Operationen in Klassen fördert die Wartbarkeit der Gesamtlösung. Diese Herangehensweise entspricht eher den natürlichen Begriffen von einer Umwelt (bestehend aus Objekten und Aktionen). Ausgehend von diesen grundsätzlichen Entscheidungen wird eine objektorientierte Analyse zur Definition der Komponenten und der Beziehungen zwischen ihnen durchgeführt. Die detaillierte Anwendungsarchitektur wird dann mit den Mitteln des objektorientierten Designs entwickelt. Soweit möglich werden dabei bestehende Lösungen für ein Problem, die sogenannten Entwurfsmuster, berücksichtigt.

Zur Anbindung der verschiedenen Datendienste werden spezielle, an die Anwendung angepaßte Schnittstellen umgesetzt. Diese Schnittstellen kapseln die Funktionalität des Datenaustauschs zwischen Anwendungs- und Datenschicht. Dadurch erreicht man eine Entkopplung der Geschäftslogik von der eigentlichen Datenschicht. Die Geschäftsobjekte agieren über ihren Daten. In welcher Infrastruktur diese Daten abgelegt werden, ist nicht erkennbar und relevant. Dies wird durch den Einsatz einer allgemeingültigen Datenbeschreibungssprache unterstützt. Durch Einsatz der Extensible Markup Language - XML können Daten zwischen verschiedenen Komponenten ausgetauscht werden. Dabei besteht der Vorteil, daß die genau festgelegte Form der Daten eine allgemeingültige Verarbeitung

ermöglichen. Es stehen bereits verschiedenen Bibliotheken zur Verarbeitung solcher Daten zu Verfügung. Des weiteren kann durch XML die Integration von Komponenten verschiedener Hersteller erleichtert werden.

XML ist aber nicht nur für die Anwendungsschicht von Bedeutung. Die Möglichkeit, XML für die verschiedensten Einsatzgebiete zu verwenden, gilt auch für die Präsentationsschicht. Dies betrifft im Moment eine XML konforme Version für HTML (XHTML). Als zukünftige Erweiterung sicherlich sinnvoll ist eine XML Notation für Skriptsprachen. Dadurch erhält man eine konforme Beschreibung für statische und dynamische Anteile einer HTML Datei mit aktiven Inhalten. Auf Seiten des Servers und des Klienten können die verschiedenen Teile einer Seite dann mit den selben Werkzeugen bearbeitet werden. Dadurch kann sich die Erstellung und Verarbeitung von Benutzeroberflächen für das World Wide Web vereinfachen.

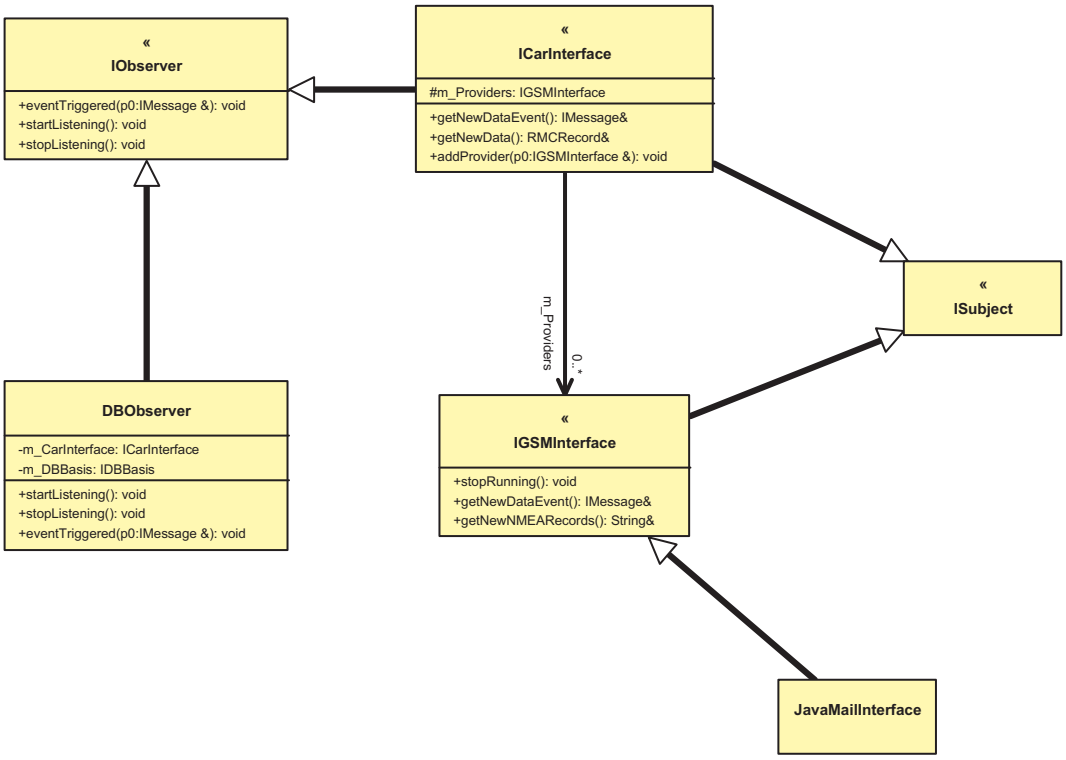
Die Einbindung mobiler Datenquellen in diese Architektur ist recht einfach. Durch den komponentenbasierten Aufbau können verschiedene Komponenten integriert werden, ohne die Funktionsfähigkeit der anderen Anwendungsteile zu beeinflussen. Eine Komponente zur Kommunikation über einen GSM Dienst läßt sich genauso einfach einbauen. Durch die Verwendung von Java als Entwicklungsumgebung ist eine gute Unterstützung für Kommunikationsprotokolle vorhanden. Die Programmierschnittstellen für Modems oder elektronische Postdienste werden direkt durch Sun spezifiziert. Im Rahmen der Java Version für Geschäftsanwendungen (Java 2 Enterprise Edition) werden diese Schnittstellen fester Bestandteil der Sprache. Dadurch kann die Entwicklungszeit für solche Anwendung verringert werden. Desweiteren ist dadurch auch eine Portierung auf andere (Java) Plattformen möglich, ohne sich um die Verfügbarkeit bestimmter Bibliotheken kümmern zu müssen.

Die entwickelte Anwendungsarchitektur kann sehr gut als Basis für zukünftige Anwendungen in diesem Umfeld eingesetzt werden. Eine Wertbeständigkeit der entwickelten Geschäftslogik wird durch den komponentenbasierten und objektorientierten Entwurf gefördert. Durch die Trennung der Geschäftskomponenten von der Darstellung gilt dies auch, wenn neue Technologien oder neue Anforderungen zu Änderungen bei der Klientenschicht führen. CORBA als zugrundeliegendes Komponentenmodell ermöglicht die Verwendung der umgesetzten Geschäftsobjekte in verschiedenen Szenarien. Der Einsatz verschiedener Programmiersprachen und Oberflächen wird durch den produktneutralen Charakter der CORBA Spezifikation ermöglicht. Dadurch kann die Anwendungslogik auch außerhalb einer webbasierten Anwendung wiederverwendet werden. Die Anpassung an neue Plattformen ist gut möglich.

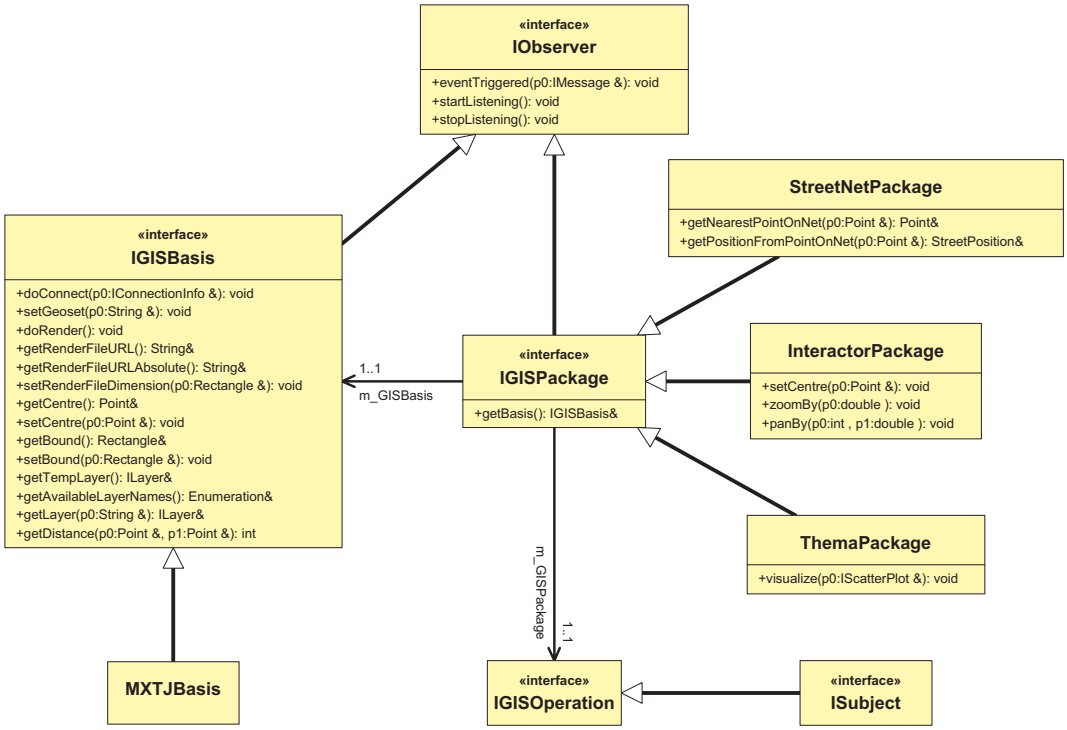
A. Klassenstruktur

Im folgenden ist die komplette Klassenstruktur des entwickelten Systems aufgeführt. Diese dient zur Dokumentation der Gesamtlösung. Deshalb handelt es sich im Gegensatz zu den Diagrammen im Text nicht um direkt entworfene Modelle. Vielmehr sind sie aus dem tatsächlich umgesetzten Programmkode erzeugt worden. Dies erfolgte mit Unterstützung der Objekt Technologie Werkbank - OTW der Firma OWiS ([OTW]). Allerdings stimmt diese konkrete Umsetzung recht gut mit dem entworfenen System überein. Eventuelle Unterschiede sind entweder rein kosmetischer Natur (z.B. andere Bezeichner) oder auf Java als benutzte Sprache zurückzuführen (z.B. Trennung von Verhaltensdefinition und -umsetzung durch separate Schnittstellenklassen).

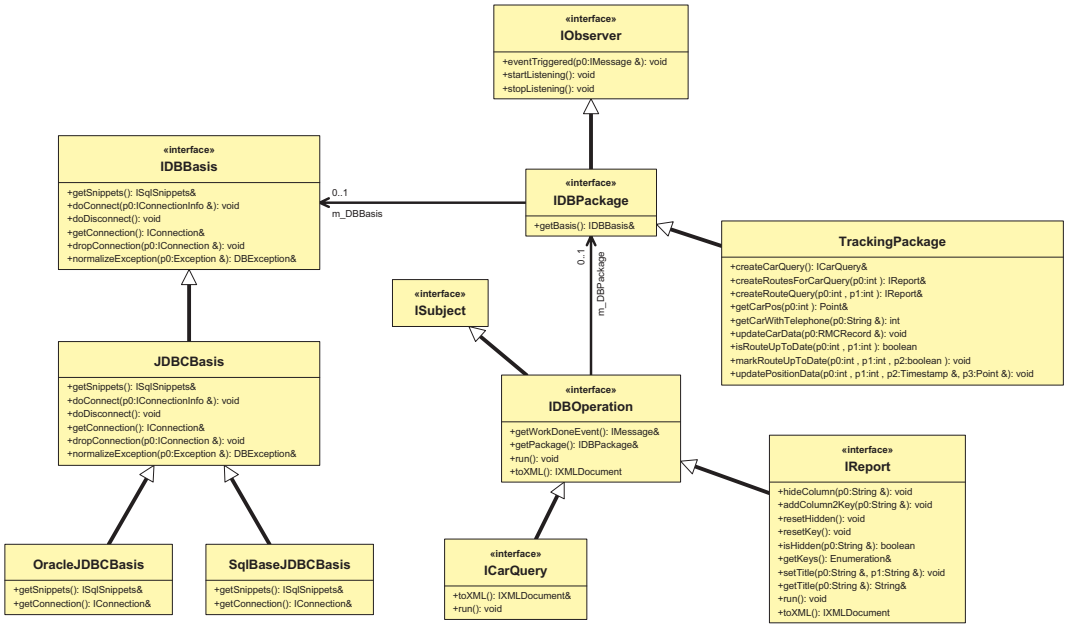
A.1. Fahrzeugschnittstelle



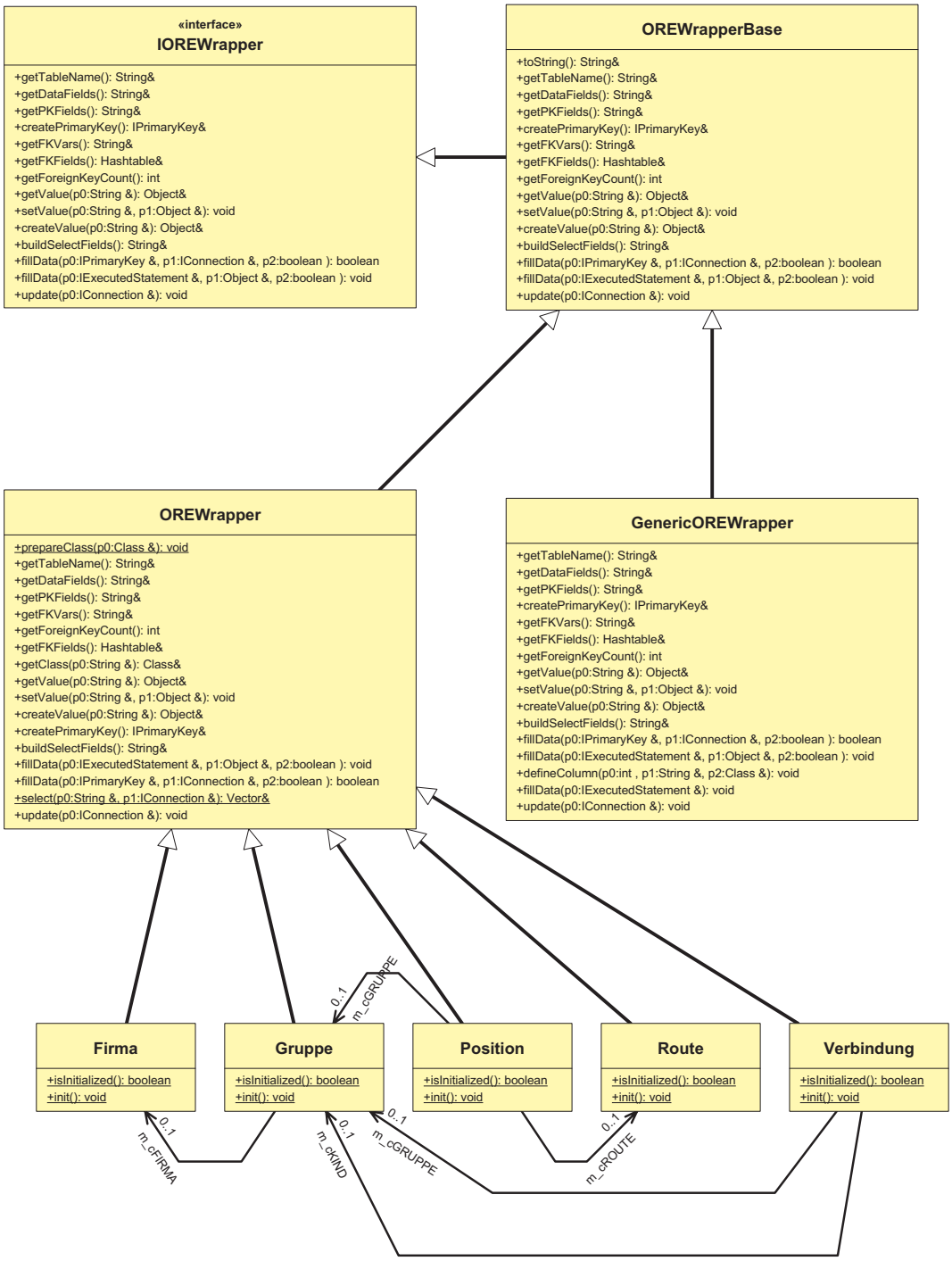
A.2. GIS-Schnittstelle



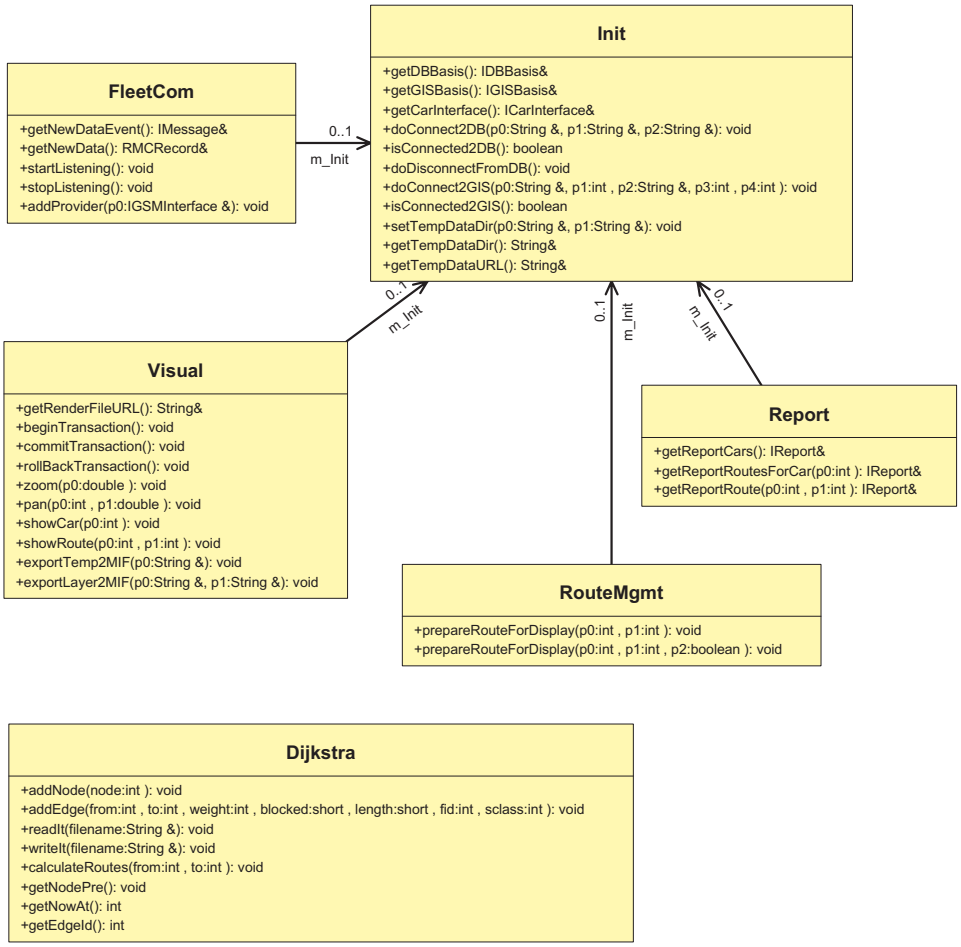
A.3. Datenbankschnittstelle



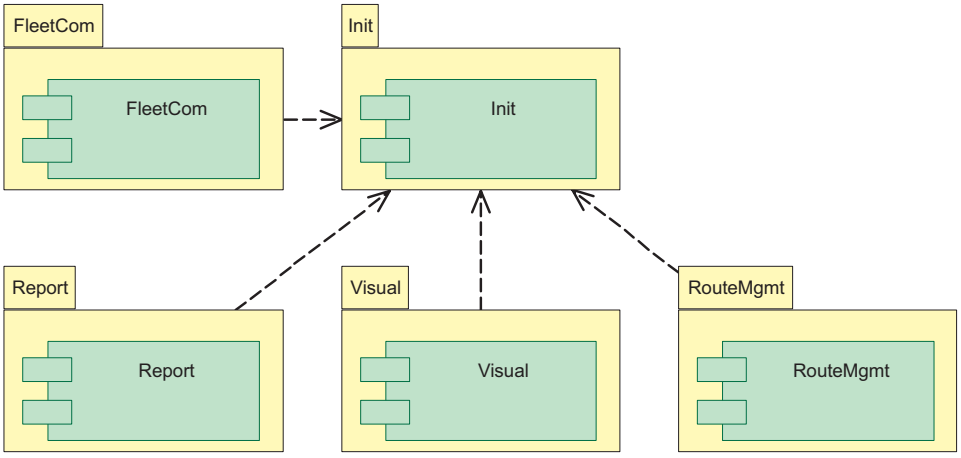
A.3.1. Objektrelationale Schicht

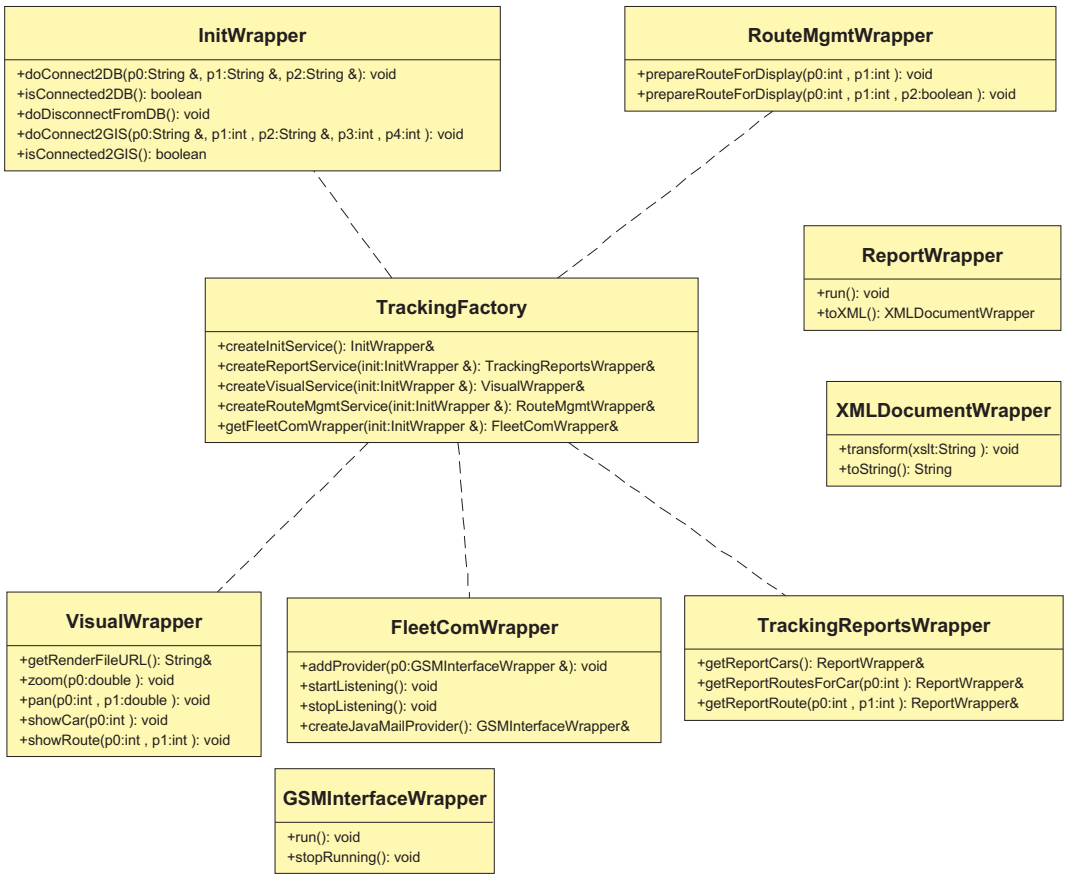


A.4. Basisdienste



A.4.1. Basisdienste - CORBA Komponenten





CORBA Hüllkomponenten für die Basisdienste. Komponenten werden über die Klassenfabrik (TrackingFactory) erzeugt. Die Komponente zur Kommunikation mit der Fahrzeugflotte (FleetComWrapper) wird dabei nur beim ersten Aufruf erzeugt. Alle anderen Aufrufe liefern die selbe Referenz zurück (Stichpunkt: Singleton).

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[TIF96] “Transport in Figures”, Bericht der Europäischen Kommision zur Verkehrsbelastung in Europa, http://europa.eu.int/en/comm/dg07/tif/index.html

[FLTMGMT] Def. Flottenmanagement, Telekom Glossar, Interest Verlag, http://www.interest.de/online/tkglossar/Flottenmanagement

[COMER98] D. Comer, Computernetzwerke und Internet, Prentice Hall 1998

[MEYER97] B. Meyer, Object-Oriented Software Construction, Prentice Hall 1997

[BOOCH94] G. Booch, Object-Oriented Modeling and Design with Applications, Addison Wesley 1994

[OOMD] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall 1991

[JAC94] I. Jacobson et al., Object-Oriented Software Engineering: A Use-Case Driven Approach, Addison Wesley 1994

[OMG97] Object Management Group, UML Notation Guide Version 1.1, ad/97-08-05

[OMG98] Object Management Group, The Common Object Request Broker : Architecture and Specification, Revision 2.3, formal/98-12-01

[SCO] J. Scourias, Overview of the Global System for Mobile Communication, University of Waterloo, aus Internet

[GPRS] S. Lamprecht, GPRS: High Speed Networking mit dem Handy, Internet Professionell 04/99, Seite 93

[GOF] E. Gamma, R. Helm, R. Johnson und J. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley 1995

[BROCK95] K. Brockschmidt, Inside OLE Second Edition, Microsoft Press 1995

[OMG99] Object Management Group, Corba Components, orbos/99-07-01

[OSF98] Open Software Foundation, DCE Today: An Indispensible Guide to DCE, Prentice Hall 1998

[MIPP] Microsoft PressPass, ActiveX Stakeholders Choose The Open Group To Drive Direction of ActiveX in the Future, Microsoft Presserklärung vom 3. Oktober 1996

[JAVA2EE] Sun Microsystems Inc., Java 2 Enterprise Edition Spezifikation, http://www.javasoft.com/products/java2ee

[OMG98/2] Object Management Group, CORBAservices: Common Object Services Specification, formal/98-12-09

[OOSPEK] G. Wanner, K.-D. Jäger, Wiederverwendung von Softwarekomponenten durch entkoppelte Schnittstellen, Objektspektrum 1/99, Seite 30ff

[GPS] B. Hoffmann-Wellenhof, G. Kienast, H. Lichtenegger, GPS in der Praxis, Springer Verlag Berlin/Heidelberg 1994

[BERN97] A. Bernklau, Erfassung und Verarbeitung von geographischen und Betriebsdaten von Fahrzeugen zur Straßenunterhaltung und -instandsetzung, Diplomarbeit Technische Universität Ilmenau, Reg. nr: 121-97D-05, 1997

[JDBC] Sun Microsystems Inc., Java Database Connection API, http://www.java.sun.com/products/jdbc

[SQL99] J. R. Groff, P. N. Weinberg, SQL: The Complete Reference, Osborne Publishing 1999

[XML10] World Wide Web Konsortium, Extensible Markup Language 1.0 vom 10.02.1998, http://www.w3.org/TR/1998/REC-xml-19980210

[XSLT] World Wide Web Konsortium, Extensible Stylesheet Transformation, Arbeitsversion vom 13.08.1999, http://www.w3.org/TR/WD-xslt

[DOM] World Wide Web Konsortium, Document Object Model, Level 1, http://www.w3.org

[GDF30] Geographic Data Files, Version 3.0, First Edition, ISO/TR 14825:1996(E)

[MSJ96] M. Blaszcak, Die neuen ISAPI Klassen der MFC, Microsoft System Journal 5/96, Seite 86ff

[MSJ98] L. Braginski, M. Powell, Die höhere Schule des Internet Information Servers, Microsoft System Journal 5/98, Seite 110ff

[SERVLET] Sun Microsystems Inc., Servlet API Dokumentation, http://www.java.sun.com/products/servlet

[JSP] Sun Microsystems Inc., Java Server Pages Spezifikation 1.0, http://www.java.sun.com/products/jsp

[MIVA] MIVA Corporation, MIVA Script Language, aus Internet

[NMEA0183] National Marine Equiper Association, NMEA0183 Standard, http://www.nmea.org

[RFC2068] Network Working Group, RFC 2068: Hypertext Transfer Protocol - HTTP 1.1, Internet Standard

[RFC1866] Network Working Group, RFC 1866: Hypertext Markup Language - HTML 2.0, Internet Standard

[MAU96] R. Maurer, HTML und CGI Programmierung, dpunkt Verlag 1996

[MAY92] D.J. Mayhew, Principles and Guidelines in Software User Interface Design, Prentice Hall 1992

[VISI] Visigenic, Distributed Object Computing in the Internet Age, Visigenic White Paper 1997

[JORB] Distributed Object Group, JavaOrb, http://www.multimania.com/dogweb

[POET] Poet, objektorientierte Datenbank, http://www.poet.de

[BSTAT] WebSide Story, StatMarket - accurate internet statistics and user trends in real time, http://www.statmarket.com

[WI99] M. Winkler, Tracker Techdoc, interne Dokumentation Firma Technotrend 1999

[OTW] OWiS Ilmenau, Objekttechnologie Werkbank, http://www.otw.de

Thesen

- Es gibt bereits mehrere Lösungen für Flottenmanagementanwendungen. Allerdings sind diese Systeme oft auf eine spezielle Umgebung angewiesen. In dieser Arbeit wird eine Architektur dargestellt, die als Grundlage für ein offenes, gut erweiterbares und an verschiedene Plattformen anpaßbares System dient.
- Gegenüber der klassischen zwei Schichten Architektur kann die Einführung einer dritten (mittleren) Schicht zu einer besseren Ausnutzung vorhandener Ressourcen führen. Es ist hierbei effizienter möglich, daß mehrere Klienten auf die selben Dienste (Datenbanken, GIS-Systeme, ...) zugreifen.
- Die Aufteilung der Anwendungslogik auf verschiedene Komponenten kann zu einem besseren Gesamtsystem führen. Dadurch erhöht sich die Flexibilität der Anwendung, da die Komponenten auf verschiedene Rechner verteilt werden können.
- Eine komponentenbasierte Entwicklung verbessert auch die Wartbarkeit der Lösung, da jede Komponente für sich entwickelt und gewartet werden kann.
- Um eine zu enge Kopplung einzelner Komponenten zu verhindern, sollte auf die direkte Kommunikation zwischen den Komponenten verzichtet werden. Komponenten sollten nur Daten erzeugen und verbrauchen.
- Die zwischen Komponenten ausgetauschten Datenpakete sollten ein einheitliches Format haben. Durch den Einsatz von XML können auch verschiedene Fremdbibliotheken integriert werden.
- Die gewählte drei Schichten Architektur ist an viele Klienten anpaßbar. Durch die Trennung der Klientenschicht von der Anwendungslogik können verschiedene Benutzeroberflächen angeboten werden.
- Durch die neueren serverseitigen Skriptsprachen können die Komponenten direkt in die entsprechenden Dateien eingebunden werden. Statische und dynamische Anteile werden in den selben Dateien gepflegt.
- Mobile Geräte können in die Gesamtarchitektur gut eingebunden werden. Durch Einsatz eines gut verfügbaren und verbreiteten Systems (GSM) können dabei weitestgehend Standarddienste verwendet werden. Serverseitig erfolgt die Einbindung über eine spezielle Komponente (Fahrzeugschnittstelle). Andere Teile der Anwendung bleiben davon unberührt.
- Die Verwendung von Java erleichtert die Einbindung von verschiedensten Kommunikationsdiensten. Bibliotheken für elektronische Postdienste und Modem zu Modem Kommunikation gehören zum Standardumfang der Java 2 Enterprise Edition.

Ilmenau, den 11. Oktober 1999 Heiko Hüter

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen und Hilfsmitteln wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ilmenau, den 11. Oktober 1999 Heiko Hüter