INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Nutzung von Indoor-Positionierungsdaten zur mobilen Wegeplanung

Andreas Heindel

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach

Betreuer: Dipl.-Inform. Bernhard Lorenz
Dipl.-Inform. Edgar-Philipp Stoffel

Abgabetermin: 11. März 2006

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Nutzung von Indoor-Positionierungsdaten zur mobilen Wegeplanung

Andreas Heindel

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach

Betreuer: Dipl.-Inform. Bernhard Lorenz
Dipl.-Inform. Edgar-Philipp Stoffel

Abgabetermin: 11. März 2006

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 11. März 2006

. . . . . .

(Unterschrift des Kandidaten)

Zusammenfassung

In dieser Diplomarbeit sollte ein vorhandenes Navigational Assistance System (NAS) für den Einsatz in das Institut für Informatik der Ludwig-Maximilian-Universität München angepasst und an ein Local Positioning System (LPS) angebunden werden, um die von diesem kontinuierlich gelieferten Ortsinformationen zu nutzen. Als Grundlage diente dafür ein an der Universität von Malta entwickeltes NAS, das ursprünglich für das Auffinden von Büchern in einer Bibliothek vorgesehen und für eine zweidimensionale Umgebung konzipiert war. Die Navigationsanweisungen erhielt der mobile Benutzer in Gestalt von segmentierten, natürlichsprachigen Instruktionen, die mit einem Verfahren namens Natural Language Generation erzeugt wurden.

Um diese Anwendung für den beabsichtigten Zweck verwenden zu können, war der Navigationsvorgang stockwerkübergreifend zu realisieren und die Form der Ausgabe durch grafische Symbole zu ergänzen. Ferner musste das System an ein LPS angeschlossen werden, welches in einer parallel laufenden Diplomarbeit entstanden ist. Dafür ist in Kooperation eine entsprechende Schnittstelle festgelegt worden, wobei für die Datenübertragung Bluetooth und als Protokoll der NMEA-Standard Verwendung fand.

Inhaltsverzeichnis

Inhaltsverzeichnis i

Abbildungsverzeichnis iv

1 Einleitung 1

2 Navigational Assistance Systems 3

3 Grundlage der vorliegenden Arbeit 5
- 3.1 Beschreibung des vorhandenen Systems . . . 5
- 3.2 Verwendete Techniken . . . 6
  - 3.2.1 Wegeberechnung und -beschreibung . . . 6
  - 3.2.2 Erzeugung natürlichsprachiger Ausdrücke für die Richtungsanweisungen . . . 7
  - 3.2.3 Ausgabe von Navigationsinstruktionen . . . 7
- 3.3 Systemarchitektur und -komponenten . . . 8
  - 3.3.1 Object Library Component . . . 8
  - 3.3.2 Environment Builder GUI Component . . . 10
  - 3.3.3 Generating Instructions Component . . . 10
  - 3.3.4 Local Positioning Component . . . 13
  - 3.3.5 Simulation GUI Component . . . 13
  - 3.3.6 Communication Component . . . 13
  - 3.3.7 Navigation Assistant Server . . . 13
  - 3.3.8 Mobile Client . . . 14

4 Mögliche Erweiterungen des Systems 15
- 4.1 Unterstützung weiterer Netzwerkschnittstellen . . . 15
- 4.2 Art der Datenspeicherung . . . 15
- 4.3 Erhöhung der Benutzerfreundlichkeit . . . 15
- 4.4 Anpassung an Mehrdimensionalität . . . 15
- 4.5 Dynamische Wegeberechnung . . . 16
- 4.6 Anbindung an ein Positionierungssystem . . . 16

5 Ziele und Lösungsansätze dieser Diplomarbeit 17
- 5.1 Anbinden an ein Ortungssystem und Definieren einer Schnittstelle . . . 18
- 5.2 Navigieren des Benutzers über mehrere Stockwerke hinweg . . . 18
- 5.3 Grafische Darstellung der Wegbeschreibung . . . 19
  - 5.3.1 2D route sketches . . . 19
  - 5.3.2 2D maps . . . 20
  - 5.3.3 3D maps . . . 21
- 5.4 Dynamische Wegberechnung und ggf. Korrektur des Weges . . . 22

6 Verwendete Techniken 23

6.1 Algorithmus von Dijkstra . . . 23
6.2 Empirische Bestimmung der Mali bei vertikalen Kanten . . . 23
6.3 Koordinatentransformation . . . 25
6.4 Bluetooth . . . 25
6.5 IEEE 802.11 . . . 27
6.6 NMEA 0183 . . . 28
6.7 Grafische Ausgabe . . . 29

7 Entwurf des LMUNavAssist **31**
7.1 ObjectLibrary . . . 31
7.2 Environment Builder . . . 32
7.3 Wegeberechnung . . . 33
7.4 Erzeugung und Ausgabe von Navigationsanweisungen . . . 34
7.5 Entwurf der Schnittstelle . . . 35

8 Implementierung **36**
8.1 Object Library . . . 36
8.1.1 EnvironmentCollection . . . 36
8.1.2 ObjectsCollection . . . 37
8.1.3 BaseObject . . . 38
8.1.4 LevelLink . . . 39
8.1.5 LinksCollection . . . 41
8.1.6 XMLClass . . . 42
8.2 EnvironmentBuilder . . . 42
8.3 Wegeberechnung . . . 44
8.3.1 Zelldekomposition . . . 44
8.3.2 Erreichbarkeitsgraph . . . 44
8.3.3 Dijkstra-Algorithmus . . . 44
8.4 Texterzeugung . . . 45
8.4.1 Partielle Textanweisungen . . . 45
8.4.2 Natural Language Generation (NLG) . . . 46
8.5 Navigational Assistance Server . . . 48
8.6 Moblie Client . . . 49
8.6.1 Grafische Anweisungen . . . 49
8.6.2 Anbindung an ein lokales Positionierungssystem . . . 49
8.6.3 Bluetooth . . . 50

9 Evaluierung **51**
9.1 Komponententest . . . 51
9.1.1 Environment Builder und Object Library . . . 51
9.1.2 Wegeberechnung . . . 51
9.1.3 Textanweisungen . . . 52
9.1.4 Client/Server Kommunikation . . . 53
9.1.5 Anbindung an ein externes LPS . . . 54
9.2 Systemtest . . . 54

10 Erkannte Probleme **56**
10.1 Generierung der Richtungsangaben . . . 56
10.1.1 Korrektheit der Ausgabe . . . 56
10.1.2 Auswirkungen auf die aktuelle Arbeit . . . 57
10.2 Bluetooth . . . 58
10.3 Fehler beim Laden der Karte . . . 59
10.4 Ländereinstellung . . . 59

11 Zusammenfassung und Ausblick **60**
11.1 Anbindung an ein externes LPS . . . 60
11.2 Erweiterung für den Einsatz in mehrstöckigen Gebäuden . . . 61
11.3 Grafische Ausgabe . . . 61
11.4 Dynamische Wegefindung . . . 62
11.5 Fazit . . . 62
11.6 Ausblick . . . 63

A Handbuch **65**
A.1 Installation . . . 65
A.2 Environment Builder GUI . . . 65
A.3 Anpassung des Servers . . . 67
A.4 Anpassung des Clients . . . 68

B Codeausschnitte **70**
B.1 Methoden der Klasse Stair . . . 70
B.1.1 setRooms() . . . 70
B.1.2 fillRooms() . . . 72
B.2 Methoden der Klasse SearchCls . . . 73
B.2.1 Auszug aus adjacentToCellsFromDifferentSpace() . . . 73
B.2.2 getRequiredPathUsingDijkstra() . . . 75
B.2.3 dijkstra() . . . 76
B.2.4 getWeight() . . . 77
B.3 Methoden der Klasse TextPlanner . . . 77
B.3.1 finalizePath() . . . 77
B.3.2 getNextSpace() . . . 78
B.3.3 Ausschnitt aus pathProcessing() . . . 79
B.4 Methoden der Klasse GuidingServerCls . . . 80
B.4.1 Aussschnitt aus ClientRead() . . . 80
B.4.2 getChecksum() . . . 81
B.5 Methoden der Klasse NavAssistantCls . . . 81
B.5.1 getDirectionExpression() . . . 81
B.6 Methoden der Klasse assistantClientCls . . . 82
B.6.1 getNMEAPos() . . . 82

Literaturverzeichnis **85**

Abbildungsverzeichnis

2.1 Autonavigationssystem TomTom GO 300 [Tom05] 4

3.1 Quadtree-Verfahren [Miz04] 6
3.2 Allgemeine Systemarchitektur [Miz04] 8
3.3 Überblick über die acht Hauptkomponenten [Miz04] 9
3.4 Objektklassifikation Hierarchie [Miz04] 9
3.5 BaseObject mit seinen Unterklassen [Miz04] 11
3.6 NLG Modul Architektur [Miz04] 12
3.7 MessageCls mit Unterklassen. [Miz04] 12
3.8 Netzwerk Setup [Miz04] 13
3.9 Die mobile Anwendungsschnittstelle [Miz04] 14

5.1 ”2,5”-dimensionale Umgebung 19
5.2 2D route sketch [Kra03] 20
5.3 2D map [Kra03] 20
5.4 3D map [Kra03] 21

6.1 Dijkstra Algorithmus [Ber01] 24
6.2 Piconetz [Hol03] 26
6.3 Scatternetz aus zwei Piconetzen [Hol03] 26
6.4 Netzwerkmodi [Dro04] 27
6.5 Übersicht aller IEEE 802.11-Standards (aus [ELK06]) 28
6.6 Symbol für ”Gehen Sie geradeaus” 30
6.7 Symbol für ”Gehen Sie die Treppe hoch” 30

7.1 Modifiziertes ObjectLibrary Paket 32
7.2 Modifizierter Environment Builder 33

8.1 Neue EnvironmentCollection Klasse 37
8.2 Modifizierte ObjectsCollection 38
8.3 Neues Paket LevelLinks 39
8.4 Neue Klasse Stair 40
8.5 Neue Klasse Lift 41
8.6 Neue Collectionsklasse LinksCollection 42
8.7 Neue DialogBox SetRefPtsFrm 43
8.8 Neue DialogBox SetLevelProperties 43
8.9 Neue DialogBox SetStairWeight 43
8.10 Stichpunktartige Nachricht 47
8.11 NLG Nachricht 47
8.12 Lift Message ”Use the Lift to Floor 4” 49

9.1 Zelldekomposition bei verschieden großen Räumen 52

9.2 Wahl des güstigsten Weges (blau = Lift, braun = Treppe). 53
9.3 Testumgebung mit Positionsdaten (grüne Punkte) 55

10.1 Befehlsanweisung für ”Move to your right” 57
10.2 Beispiel für fehlerhafte Navigationsanweisung 58

A.1 leere Benutzeroberfläche von Environment Builder 66
A.2 die Befehlsleiste ”File”. 66
A.3 die Befehlsleiste ”Draw”. 67
A.4 die Toolbar. 67
A.5 die Objekteigenschaften. 68

Kapitel 1

Einleitung

In der heutigen Zeit ist Mobilität vor allem auch im Bereich der Informationstechnik kaum noch aus dem Alltag wegzudenken und ihre Bedeutung wird in Zukunft aller Voraussicht nach noch weiter steigen. Da Computerhardware in den vergangenen Jahren immer kleiner, billiger und robuster wurde und auf dem Gebiet der drahtlosen Kommunikation große Fortschritte zu verzeichnen waren, hat dies den Weg für mobile Endgeräte wie PDAs oder Handys geebnet. Im Verlauf dieser Entwicklung ist im Bereich des Mobile Computing eine ganze Gruppe neuer Anwendungen entstanden.

Beispielsweise gibt es Anwendungen, die einen mobilen Benutzer bei der Wegefindung in einer unbekannten Umgebung unterstützen können. Solche Navigationssysteme werden in Kraftfahrzeugen schon seit längerem eingesetzt. Personennavigationssysteme waren gerade in letzter Zeit Gegenstand der Forschung [Miu02], [Wah01], [Bau01].

Am *Department of Computer Science and AI* in der Universität von Malta wurde im Juni 2004 von Ms. Doreen Mizzi im Rahmen einer Diplomarbeit ein mobiles Navigations-Assistenzsystem [Miz04] entwickelt, welches Ausgangspunkt der vorliegenden Diplomarbeit ist.

Bei diesem handelt es sich um ein automatisches Wegeberechnungssystem für den Indoor-Bereich, wobei die Navigationsanweisungen dem Benutzer auf einem PDA in Textform präsentiert wurden. Der generierte Text war nicht stichpunktartig, sondern umgangssprachlich mit korrekter Grammatik formuliert. Hierzu kam ein als *Natural Language Generation (NLG)* bezeichnetes Verfahren zur Verwendung.

Da das System nur für den Einsatz auf einer Stockwerkebene konzipiert worden war, konnte der Anwender nicht zwischen den Etagen wechseln. Als Testumgebung fungierte der zweite Stock der Bibliothek der Universität von Malta. Unter anderem wurde die Anwendung für das Auffinden von Büchern entwickelt, deren Kennungszahl vom Benutzer über seine PDA einzugeben ist.

PDAs sind handliche, mobile Endgeräte, die jedoch mit Desktop-PC verglichen nicht so leistungsfähig sind. Um diese möglichst zu entlasten, kam eine Client/Server-Architektur zum Einsatz. Zu den serverseitigen Aufgaben zählten die Bereitstellung von Speicherplatz für den Stockwerkplan, die Berechnung des Weges zum Ziel, die Umsetzung der stichpunktartigen Navigationsanweisungen in natürlichsprachige Ausdrücke und die Segmentierung des erzeugten Textes, so dass dieser auf Seiten des Clients benutzerfreundlich auf dem kleinen Display des PDA ausgegeben werden konnten.

Ausgehend von dem beschriebenen System sollten nun im Rahmen dieser Arbeit neue Funktionalitäten implementiert sowie die Nutzung von Indoor-Positionierungsdaten zur mobilen Wegplanung ermöglicht werden. Der angepeilte Funktionsumfang der überarbeiteten Anwendung umfasste unter anderem die Verbesserung des Navigationsprozesses der Gestalt, dass dieser nun stockwerkübergreifend arbeiten sollte. Durch diese Maßnahme sollte eine wesentliche Einschränkung des zugrunde liegenden Systems beseitigt werden. Als weiteres Ziel wurde gesetzt, die mit Hilfe von NLG erzeugten Texte durch eine grafische Form der Ausgabe zu ergänzen und so die Benutzerfreundlichkeit noch zusätzlich zu steigern. Auch die Tatsa-

che, dass der Weg bisher nur ein einziges Mal, nämlich bei der ersten Anfrage des Anwenders, berechnet wird, kann sich unter Umständen nachteilig auswirken. Aus diesem Grund war ein Wechsel auf ein dynamisches Verhalten wünschenswert. Nicht zuletzt war für die Anbindung an ein Indoor-Lokalierungssystem zu sorgen, welches für die Navigation erforderlichen Positionsdaten liefern sollte.

Die vorliegende Arbeit gliedert sich wie folgt: im nächsten Kapitel soll dem Leser zuerst der Begriff des *Navigational Assistance System (NAS)* näher gebracht werden. Danach erfolgt eine Präsentation des aktuellen Ausgangspunktes, indem eine ausführliche Beschreibung des zu modifizierenden Systems vorgelegt und danach in einem eigenen Kapitel über mögliche Erweiterungen diskutiert wird. In Kapitel 5 werden die Ziele sowie die verwendeten Lösungsansätze dieser Diplomarbeit angesprochen und anschließend die zur Bewältigung der Aufgabenstellung verwendeten Techniken vorgestellt. Der Entwurf und die Architektur des überarbeiteten Systems sind Gegenstand des siebten Kapitels und in einem weiteren wird auf die Implementierung der Anwendung eingegangen. Evaluationsdaten und Testergebnisse sind in Kapitel 9 zu finden. Das darauf folgende gewährt schließlich einen Einblick in bereits bekannte oder im Verlauf dieser Arbeit neu aufgetretene Probleme. Kapitel 11 gibt dem Leser eine Zusammenfassung des Inhalts sowie einen Ausblick auf zukünftige Betätigungsfelder. Zuletzt ist ein Anhang beigefügt, der ein Benutzerhandbuch enthält, welches die Handhabung des Systems erläutert, sowie einige interessante und für das Verständnis wichtige Methoden und Codeabschnitte enthält.

Kapitel 2

Navigational Assistance Systems

Navigation bezeichnet "das *sich Zurechtfinden* in einem geografischen Raum, um einen bestimmten Ort zu erreichen." [Nav05]. Der Navigationsvorgang lässt sich in die drei folgende Teilschritte unterteilen: Positionsbestimmung, Wegeberechnung und den eigentlichen Hinführungsprozess.

Ersteres wird durch ein Positionierungs- oder Lokalisierungssystem realisiert. Solche Anwendungen können lokal oder global arbeiten, man bezeichnet sie entsprechend als *Local Positioning System (LPS)* beziehungsweise *Global Positioning System (GPS)*. Ein globales, in den jungen Jahren der Nautik unverzichtbares Verfahren, war die Benutzung eines Sextanten. Heute verwendet man für die globale Ortung vorwiegend eine satellitengestützte Technologie, die in den USA für militärische Aufgaben zur Zeit des Kalten Krieges entwickelt wurde (*NAVSTAR-GPS*, [GPS05]). Um der Monopolstellung dieses Systems Konkurrenz zu machen, hat die Europäische Union zusammen mit der *European Space Agency (ESA)* und einer Reihe weiterer Staaten, darunter Indien, China, Israel und die Ukraine, das Projekt *Galileo* in Angriff genommen. Dieses Satellitennavigationssystem soll voraussichtlich im Jahr 2010 in Betrieb genommen werden [Gal05].

Der Einsatz von Navigational Assistance Systems begann mit der Entwicklung von *LORAN-C (Long Range Navigation)* im zweiten Weltkrieg, die für die Navigation von Kampfflugzeugen eingesetzt wurden. Mit der Zeit weitete sich der Anwendungsbereich solcher Systeme auch auf die Nautik aus. 1981 kam aus dem Hause Honda das erste Autonavigationssystem auf den Markt, die sich rasch ausbreitete und in unserer heutigen Zeit zur Standardausstattung in gehobenen Klassen zählen. Sogar Fußgänger können sich inzwischen über das Vorhandensein von so genannten *Head Units* und Handgeräten freuen [NAS05]. Ein Beispiel für eine solche kommerzielle Anwendung ist das *TomTom GO 300*, das in PKW eingesetzt wird und in Abbildung 2.1 dargestellt ist.

Die Ortsbestimmung ist Voraussetzung für eine sinnvolle Navigation, gehört aber nicht zu den eigentlichen Pflichten des Navigational Assistance Systems, sondern wird im Normalfall von einem an diesen angebundenen Local Positioning System oder Global Positioning System geleistet. Auf deren Basis kann das NAS anschließend seine beiden Hauptaufgaben, und zwar die Wegeberechnung und den eigentlichen Hinführungsprozess, ausführen.

Für die Bestimmung des Weges vom Startpunkt aus bis hin zum Zielort gibt es einige sinnvolle Ansätze: ein System kann beispielsweise entweder die absolut kürzeste Strecke wählen oder auch einen längeren Weg akzeptieren, wenn dieser weniger komplex oder schneller als ersterer ist. Die von der Entfernung her kürzeste Route muss nämlich nicht zwingend auch die schnellste sein. Ein Autonavigationssystem wird somit bevorzugt die Hauptstraßen verwenden und dabei notfalls Umwege in Kauf nehmen, statt den Benutzer über eine größere Anzahl von Nebenstraßen zu führen, wobei zusätzlich noch die Gefahr bestünde, dass sich dieser verfährt.

Um den Benutzer zur gewünschten Position zu leiten, müssen ihm logischerweise Navigationsanweisungen übermittelt werden. Es gibt auch hier mehrere Ansätze, um dies zu realisieren: so können entweder

Abbildung 2.1: Autonavigationssystem TomTom GO 300 [Tom05]

Textausgabe, Sprachausgabe oder einfach nur schlicht eine graphische Darstellung der Instruktionen Verwendung finden.

Bei ersterer Form können die Navigationsbefehle wahlweise ganze, natürlichsprachige Sätze, wie sie etwa ein menschlicher Kommunikationspartner formulieren würde, oder auch nur stichpunktartig Anweisungen sein. Eine Kombination aus verschiedenen Ausgabeformen ist möglich und manchmal sogar notwendig. Ein Autofahrer z.B. sollte nicht zu stark vom Straßenverkehr abgelenkt und somit hauptsächlich über Sprachausgabe informiert werden, anstatt ständig auf das Display des NAS schauen zu müssen. Auch darf ihn nicht eine Vielzahl von (möglicherweise irrelevanten) Informationen überfordern. Graphische Darstellung gibt es auch in verschiedenen Ausführungen: als zweidimensionale Karte, dreidimensional oder durch einen einfachen Pfeil, der die Richtung anzeigt (*2D route sketches*).

Kapitel 3

Grundlage der vorliegenden Arbeit

3.1 Beschreibung des vorhandenen Systems

Grundlage der vorliegenden Arbeit ist das von Ms. Doreen Mizzi 2004 im Rahmen einer Diplomarbeit mit dem Titel "A Mobile Navigational Assistance System Using Natural Language Generation" an der Universität von Malta entwickelte Navigational Assistant System. In dieser Arbeit [Miz04] wurde ein mobiles NAS speziell für den Indoor-Gebrauch entworfen, wobei der Benutzer mit Hilfe von natürlichsprachigen Textanweisungen zum gewünschten Ziel geführt wird. Dieses konnte im Gegensatz zu üblichen NAS nicht nur ein Raum, sondern auch ein konkretes Objekt, beispielsweise ein Buch, sein. Der Schwerpunkt war also das Auffinden von Büchern und zu diesem Zweck kam das System testweise im zweiten Stock der Bibliothek der Universität von Malta zum Einsatz.

Allerdings war der Navigationsprozess auf ein einziges Stockwerk beschränkt, wodurch das Wechseln zwischen Etagen ausgeschlossen wurde. Auch die verwendeten Umgebungspläne konnten folglich nur ein Stockwerk enthalten. Für die Wegeberechnung wurden die im Stockwerkplan enthaltenen Räume und Korridore in Zellen zerlegt und der kürzeste Weg zum gewünschten Ziel bestimmt, wobei die Mittelpunkte der einzelnen Zellen die Knoten des Pfades darstellen. Aus diesen ließen sich simple Richtungsanweisungen erstellen, die in einem weiteren Prozessschritt zu natürlichsprachigen Navigationsinstruktionen verfeinert wurden. Die Durchführung der Wegeberechnung geschah dabei nur ein einziges Mal, nämlich bei der ersten Anfrage des Benutzers. Folglich können Irrtümer nicht erkannt und verbessert werden, wenn der Benutzer zwischendurch vom vorberechneten Weg abweicht.

Das System basiert auf dem Client/Server-Modell, wobei dem Server folgende Aufgaben zufallen: dauerhaftes Speichern des Stockwerkplans, Wegeberechnung, Generierung und Segmentierung von Navigationsanweisungen. Letzteres war notwendig, da der gesamte Text aus Platzgründen üblicherweise nicht als Ganzes auf dem PDA-Display angezeigt werden konnte. Der verwendete Server war auf *Multithreading* ausgelegt, konnte also mehrere Clients gleichzeitig bedienen. Diese Architektur kam zum Vorzug, um den über weniger Ressourcen verfügenden mobilen Endgerät (PDA oder *Thin Client*) zu entlasten.

Aus finanziellen und zeitlichen Gründen ist auf die Konstruktion eines LPS verzichtet worden. Um trotzdem Funktionstests mit dem NAS durchzuführen zu können, musste ein solches Ortungssystem simuliert werden.

Für die Implementierung des Systems fand die Programmiersprache C# von Microsoft Verwendung, weshalb es nur auf Rechnern lauffähig ist, auf denen das *Microsoft .NET Framework* installiert ist. Ferner wird eine auf dem TCP/IP-Protokoll basierte *IEEE 802.11*-Verbindung oder ersatzweise *Bluetooth* benötigt. In letzterem Fall muss auf dem Rechner der *Microsoft Bluetooth Protocol Stack* vorhanden sein. Die Clientanwendung ist auf einem *Pocket PC* mit dem Betriebssystem *Microsoft Windows CE* zu installieren.

3.2 Verwendete Techniken

In diesem Abschnitt sollen auf einige der Techniken eingegangen werden, die bei dem in [Miz04] vorgestellten System Verwendung fanden. Zu diesen zählen die Berechnung des Weges sowie die Erzeugung und Ausgabe von natürlichsprachigen Navigationsanweisungen.

3.2.1 Wegeberechnung und -beschreibung

Für die Wegeberechnung mussten die Räume und Korridore des Stockwerks in einem ersten Schritt in Zellen unterteilt werden. Dafür wurde das Verfahren der *approximierenden Zelldekomposition* [Ama04] angewandt. Dieses basiert auf eine überlappungsfreie Aufteilung des Raumes in volle, leere oder hybride Zellen. Eine volle Zelle repräsentiert dabei einen unzugänglichen Bereich. Der Benutzer kann logischerweise nur Zellen betreten, die entweder leer oder teilweise gefüllt (*hybrid*) sind, denn nur diese enthalten keine oder wenige Hindernisse.

Die Zellaufteilung wird mit Hilfe des *Quadtree-Algorithmus* (Abbildung 3.1) erzeugt, wobei leere Zellen nicht weiter gesplittet werden. Anschließend werden die vollen Zellen aussortiert und die Mittelpunkte der verbliebenen Zellen durch Kanten verbunden. Dadurch entsteht ein zusammenhängender Graph, nämlich ein *Adjazenzgraph*. Dieser gibt an, welche der leeren oder hybriden Zellen von einer beliebigen anderen aus erreichbar sind und wird in Form einer *Adjazenzliste* abgespeichert.

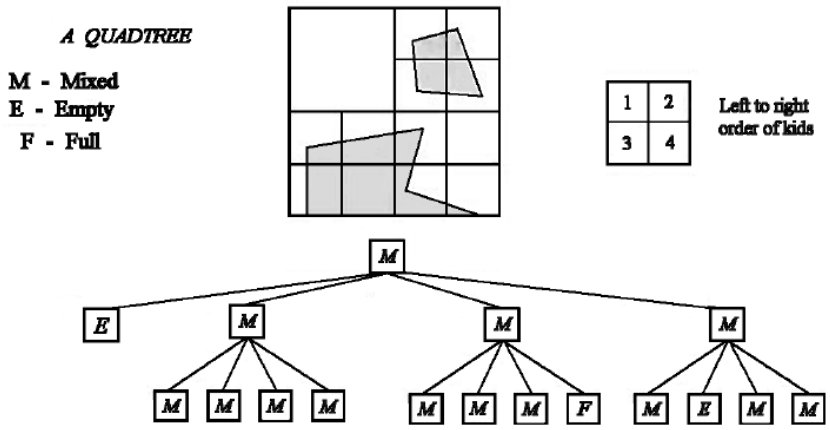

Abbildung 3.1: Quadtree-Verfahren [Miz04]

Das System verwendet den *A*-Algorithmus* für die effiziente Pfadberechnung. Dabei handelt es sich um einen *Shortest-Path*-Algorithmus, also um den kürzesten Pfad zwischen einem Startknoten und einem beliebigen Knoten in einem kantengewichteten Graphen zu finden (vergleiche auch: [ASt05], [Les05]). Es wird also stets der kürzeste, aber nicht unbedingt der einfachste Weg zum Ziel berechnet, außer es ist eine Metrik für die Komplexität gegeben.

Der A*-Algorithmus arbeitet wie folgt: nehmen wir an, jeder Knoten repräsentiert eine der oben erwähnten Zellen. Zu Beginn werden zwei Listen *OpenList* und *ClosedList* erzeugt und der Start- und Zielpunkt ermittelt. Startpunkt S wird in *OpenList* gespeichert, der Liste der zu betrachtenden Knoten. Im nächsten Schritt werden aus der Adjazenzliste alle möglichen Nachbarknoten von S ausgewählt und mit einem Verweis auf seinen Vorgängerknoten in *OpenList* eingefügt. Als nächster wird derjenige benachbarte Knoten ausgesucht, der die kleinste Kosten K besitzt. K ergibt sich aus der Summe der minimalen Wegekosten K_{min} vom Startpunkt S aus und einen durch eine Heuristikfunktion berechneten Schätzwert H. Letzterer ist die untere Schranke für die Luftlinie vom aktuellen Knoten zum Ziel. Für die Berechnung der Kosten benötigt

man die Gewichte der einzelnen Kanten der Adjazenzliste, die sich aus der *euklidischen Distanz* zwischen den beiden Eckpunkten einer Kante ergeben. Sollten zwei oder mehrere äquivalente Knoten existieren, wird das Zufallsprinzip angewendet. Der besuchte Knoten wird aus *OpenList* entfernt und in *ClosedList* kopiert, der Liste der nicht mehr zu betrachtenden Knoten. Wenn der Zielknoten in der *OpenList* auftritt, haben wir den kürzesten Weg gefunden, den wir durch umgekehrte Reihenfolge der Knoten ausdrücken können.

Aus diesem Pfad wird anschließend eine Folge von stichpunktartigen Richtungsanweisungen wie beispielsweise "Reach Door" oder "turn to left" generiert. Um die Richtungsänderungen zu bestimmen, werden zwei hypothetische Geraden erzeugt, wobei erstere den Mittelpunkt des alten Raumes und den der zuletzt passierten Tür schneidet, die zweite das Zentrum der vorhergehenden und der folgenden Tür. Aus dem Winkel, mit dem sich die beiden Geraden schneiden, ergibt sich die als nächstes einzuschlagende Richtung. Unabhängig vom exakten Verlauf des berechneten Weges werden die Richtungsänderungen also immer von Tür zu Tür bestimmt.

3.2.2 Erzeugung natürlichsprachiger Ausdrücke für die Richtungsanweisungen

Für die Ausgabe der Navigationsanweisungen war in der originalen Arbeit eigens ein Algorithmus entwickelt worden, der eine Kombination aus *templatebasierender* Vorgehensweise und Grammatiken für die Generierung natürlicher Sprache (NLG) darstellte [Miz04]. Dies war notwendig, um einen Kompromiss zwischen Laufzeitkomplexität der Berechnung und Qualität der natürlichsprachigen Ausdrücke zu erhalten.

Unter templatebasierender Vorgehensweise versteht man in diesem Zusammenhang das Einfügen von passenden Wörtern in einer vorgegebenen Textvorlage. Das Ergebnis hat aber keine wirklich natürliche Struktur mehr, weil bei diesem Verfahren immer derselbe, notwendigerweise relativ simple, Sprachaufbau verwendet wird. Flexible und qualitativ hochwertige Texte können nicht mit solchen Algorithmen erzeugt werden.

Dieser Nachteil ist ein Grund, warum NLG-Techniken immer populärer werden. [Rei97] beschreibt NLG als Teilgebiet der künstlichen Intelligenz und berechenbaren Sprachen, welche sich mit dem Aufbau von Computersystemen beschäftigt, die aus einer zugrunde liegenden nichtsprachlichen Informationsdarstellung verständliche Texte in englischer oder anderer natürlicher Sprache erzeugen können.

Eine der vielen Möglichkeiten, ein NLG System zu erzeugen, ist die *corpusbasierende* Vorgehensweise. Dieser Ansatz findet auch in [Miz04] Verwendung. Dabei wird in der initiierenden Phase ein so genannter *corpus* erzeugt, der sich durch eine Menge von Eingabedaten und den jeweils zugehörigen, von Menschen geschriebenen Ausgabedaten bestimmt. Aus der Analyse dieses corpus erhält man Regeln, die dann im NLG System für die Erzeugung der Ausgabe verwendet werden.

NLG generiert bessere natürlichsprachige Texte als das templatebasierte Verfahren, hat aber dafür ein schlechteres Laufzeitverhalten. Deswegen wurde in [Miz04] ein Kompromiss aus beiden Methoden bevorzugt.

Eine genauere Betrachtung dieses Themas würde den Rahmen der vorliegenden Arbeit sprengen. Weiterführende Literatur ist in [McD87], [Rei97] zu finden.

3.2.3 Ausgabe von Navigationsinstruktionen

Um den Benutzer und die begrenzte Darstellungsmöglichkeit eines PDA nicht durch einen zu langen Text zu überfordern, kam außerdem ein Verfahren zur Segmentierung der Menge von Navigationsbefehlen zum Einsatz. Dabei wird der gesamte Text in mehrere Abschnitte geteilt und sukzessiv ausgegeben. Genaue Details zusammen mit dem verwendeten Algorithmus können aus [Miz04] entnommen werden.

3.3 Systemarchitektur und -komponenten

Ziel in [Miz04] war es, ein Navigationssystem für den Einsatz innerhalb eines Stockwerkes zu entwickeln, das einen Benutzer mit Hilfe von verbalen Anweisungen in einer unbekannten Umgebung zu seinem Ziel leitet.

Dabei mussten zwei Perspektiven berücksichtigt werden, zum einen die Sicht des Systemadministrators und zum anderen die des mobilen Benutzers.

Der Systemadministrator ist für die Erzeugung eines zweidimensionalen Plans aus Vogelperspektive und für die Eingabe der Details von besonderen Objekten der Umgebung zuständig. Er muss ferner entscheiden, welche Netzwerkschnittstelle das System benutzen soll, wobei ihm TCP/IP-basierende 802.11b- oder Bluetoothvernetzung zur Wahl stehen. Ob ein externes LPS serverseitig oder clientseitig angebunden wird, ist ein weiterer Aspekt, den der Administrator festlegen muss.

Dem mobilen Benutzer soll wiederum ermöglicht werden, sein gewünschtes Ziel zu spezifizieren, welches entweder ein Raum oder ein konkretes Objekt wie ein Buch sein kann. Außerdem ist er in der Lage, dem NAS seinen momentanen Startpunkt mitzuteilen, indem er ein auffälliges Objekt in seiner Blickrichtung angibt.

Abbildung 3.2 gibt die grundlegende Architektur für das in [Miz04] entwickelte System an.

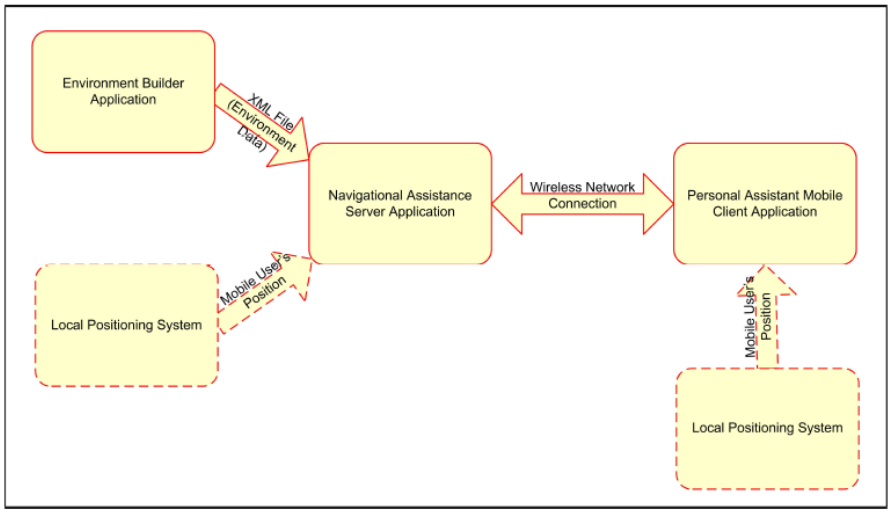

Abbildung 3.2: Allgemeine Systemarchitektur [Miz04]

Wie bereits in Abschnitt 3.1 schon erwähnt wurde das System als eine Client/Server-Anwendung konzipiert. Um die Prozessorlast des mobilen Endgeräts zu minimieren, erfolgen die Berechnung der Ortsposition und auch die Abspeicherung der 2D-Pläne auf dem Server. Das komplette NAS besteht aus den drei Untersystemen *Navigational Assistance Server Application*, *Personal Assistant Mobile Client Application* und *Environment Builder Application* (vgl. Abb. 3.2), die sich wiederum aus acht Hauptkomponenten (Abbildung 3.3) zusammensetzen, welche im Folgenden beschrieben werden sollen.

3.3.1 Object Library Component

Diese Klassenbibliothek enthält alle notwendigen Klassen, aus denen sich der vom NAS verwendete Umgebungsplan zusammensetzt. Es gibt davon drei Formen zur Darstellung: Rechteck, Kreis und Linie. Alle Objekte lassen sich mit Hilfe dieser drei Formen zeichnen. Die aus den Umrissen resultierenden minimal

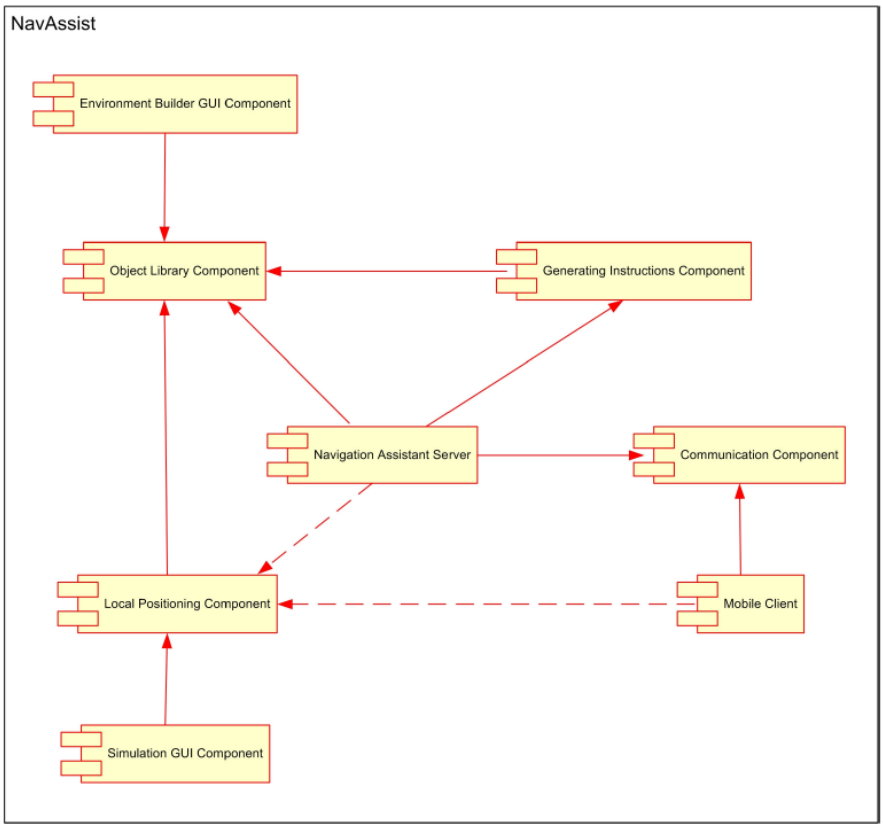

Abbildung 3.3: Überblick über die acht Hauptkomponenten [Miz04]

umgebenen Rechtecke werden zusammen mit anderen relevanten Informationen gespeichert. Der mobile Benutzer selbst wird als beweglicher Punkt repräsentiert. Die Klassenhierarchie dieser Datenstruktur ist in Abb. 3.4 dargestellt.

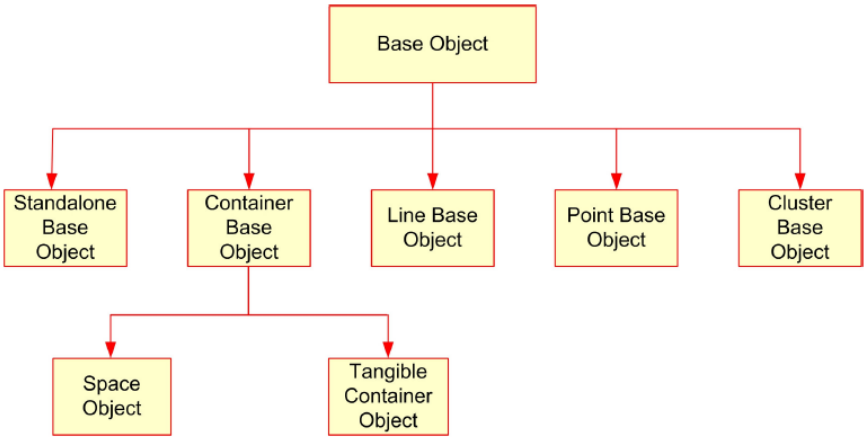

Abbildung 3.4: Objektklassifikation Hierarchie [Miz04]

Alle Daten werden in einer Datei im XML-Format abgelegt. Das *Object Library Component* beinhaltet auch Methoden zum Laden und Speichern dieses Datenformats.

Die Komponente enthält neun Pakete, nämlich *BaseObjects*, *Collections*, *General*, *Lines*, *SpaceDir*, *StandaloneObjectsDir*, *TangibleContainerDir*, *TypeConverters* und *XML*.

In *BaseObjects* sind folgende Klassen definiert: *BaseObject*, *ClusterObject*, *ContainerBaseObject* und *PointCls*. Dabei ist *BaseObject* die höchste Oberklasse der Objekthierarchie und die anderen sind seine direkten Unterklassen. *PointCls* wird benötigt, um die gegenwärtige Benutzerposition zu repräsentieren. Ein Objekt vom Typ *ContainerBaseObject* kann wieder selbst Objekte enthalten. Hier wären beispielsweise ein Raum oder ein Tisch zu nennen, die logischerweise Kindelemente besitzen können. Des Weiteren findet man eine separate Klasse *FrontFaceOfObj* in diesem Paket, die eine Frontansicht der Objekte speichert.

Das Paket *Lines* enthält eine einzige Klasse *Line*, welche verwendet wird, um Öffnungen von Raumobjekten, wie zum Beispiel Türen, darzustellen. Hierfür verfügt die Klasse über zwei Attribute, welche das links- und rechtsseitige *Space*-Objekt angeben. Bei der Erzeugung des Erreichbarkeitsgraphs wird diese Information benutzt, um zu erkennen, wo die Möglichkeit besteht, von einem Raum zum nächsten zu gelangen.

General besteht ebenfalls nur aus einer Klasse, *PropertyCls*, für die Speicherung von Namen und Werten benutzerdefinierter Eigenschaften.

Collections enthält drei verschiedene Collections von Objekten: je eine Sammlung von *BaseObjects* für das Speichern der kompletten Umgebungskarte, eine von *Lines*, sowie auch eine von *Properties* für das Ablegen von benutzerdefinierten Eigenschaften eines bestimmten Objekts.

Die Klassen im Paket *SpaceDir* sind Unterklassen von *ContainerBaseObject*, sie repräsentieren z.B. Räume und Korridore. Zu ihr gehören eine Oberklasse *Space* mit drei Unterklassen: *Stairs* (Treppen), *SquareSpace* und *EllipseSpace* (quadratische bzw. kreisförmige Räume).

Ebenfalls Unterklassen von *ContainerBaseObject* sind die Klassen aus dem Paket *TangibleContainerDir*, sie enthalten jedoch konkrete Objekte wie Tisch und Stuhl. In diesem Paket kommt die Klasse *TangibleContainer* mit seinen beiden Unterklassen *QuadraticContainerObject* und *CircularContainerObject* vor.

Objekte, die selbst kein anderes Objekt enthalten können (z.B. ein Buch oder Laptop), werden aus den Klassen des Pakets *StandaloneObjectsDir* erzeugt. In dieser ist *StandaloneObject* mit ihrer Unterklasse *QuadraticStandaloneObject* implementiert.

Das neunte und letzte Paket *XML* besteht aus der Klasse *XMLClass*, welche für das Laden und Speichern des 2D-Plans und der Objekteigenschaften zuständig ist. Zur Verarbeitung der XML-Daten wurden die .NET Klassen *XmlTextWriter* und *XmlTextReader* verwendet.

3.3.2 Environment Builder GUI Component

Diese Komponente bietet dem Systemadministrator eine graphische Oberfläche, mit deren Hilfe er die Objekte der Umgebung verändern kann. Hauptaufgaben dieses Moduls sind die Erstellung eines passenden 2D-Plans des relevanten Stockwerks aus der Vogelperspektive sowie von Frontansichten und zugehörigen Beschreibungen von besonderen Objekten. Gespeichert werden die Umgebungsdaten im XML-Format. Weitere Informationen zu dieser Komponente können der Originalarbeit [Miz04] entnommen werden.

3.3.3 Generating Instructions Component

In dieser Komponente erfolgen die Wegeberechnung und deren Textausgabe. Dies geschieht sequentiell in vier Schritten, wie Abbildung 3.6 entnommen werden kann.

Die Hauptaufgabe des *Path Planner* ist, einen geeigneten Weg hin zum gewünschten Ziel zu finden. Das genaue Verfahren wurde in Abschnitt 3.2.1 bereits erklärt. Ausgeführt wird diese Aufgabe von den Klassen im Paket *RoutePlanner*, nämlich von *CellDecomp*, *CellCls* und *SearchCls*. Erstere zerlegt jeden Raum oder

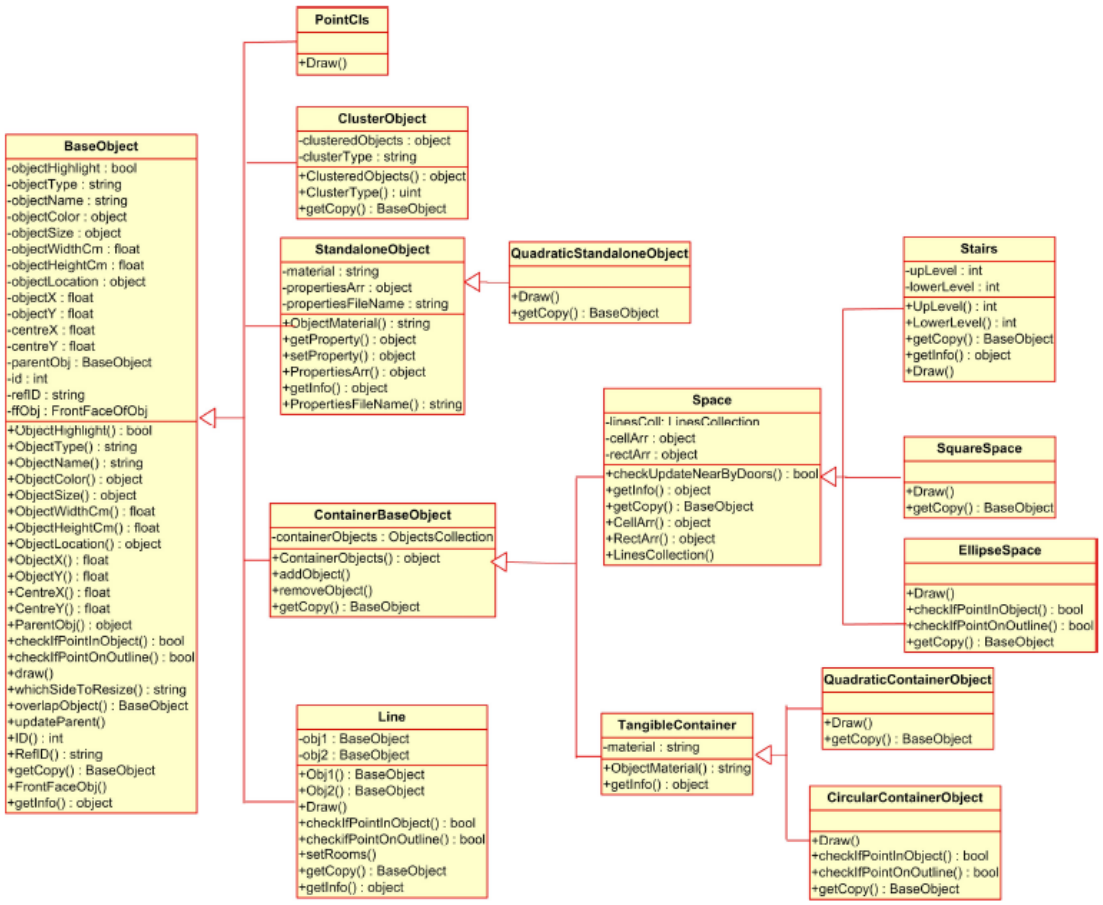

Abbildung 3.5: BaseObject mit seinen Unterklassen [Miz04]

Korridor mit Hilfe der approximierenden Zelldekomposition in Zellen, die durch Objekte des Typs *CellCls* repräsentiert werden. In der letztgenannten Klasse wird zuerst der zusammenhängende Adjazenzgraph erzeugt, dessen Kanten durch Objekte der Klasse *PathClass* gebildet werden. Anschließend wird aus diesem Graph unter Verwendung des *A*-Algorithmus* der kürzeste Weg vom Startpunkt bis zum Ziel berechnet und als Folge von zusammenhängenden Punkten (*PtUsedInPathCls*) im zweidimensionalen Raum mit den jeweils zugehörenden Raumstrukturen ausgegeben.

In der zweiten Stufe setzt der *Text Planner* den berechneten Pfad in eine stichpunktartige Wegebeschreibung um, die sich aus eine Serie von Nachrichten zusammensetzt. Das gleichnamige Paket besteht aus folgenden Klassen: *ClusteringCls*, *MessageCls*, *PrepositionClass*, *ReferingExprCls*, *SegmentationCls*, *SweepClass* und *TextPlannerCls*. Letztere ist das Herzstück dieses Paketes, welches Nachrichten vom Typ *MessageCls* generiert. Diese besitzt zahlreiche Unterklassen, die die Art der Nachricht angibt, wie zum Beispiel *AlongMsg* und *AcrossMsg* (Abbildung 3.7).

Diese Nachrichtenserie wird anschließend im *Segmentation Process* in mehrere handliche Textsegmente unterteilt. Verantwortlich dafür ist die Klasse *SegmentationCls*, die ebenfalls im vorher erwähnten Paket enthalten ist.

In der letzten Stufe (*Linguistic Realisation*) werden aus den stichpunktartigen Richtungsanweisungen mit Hilfe von NLG (vgl. 3.2.2) natürlichsprachige Instruktionen erzeugt. Dabei wird von zwei Textdateien Gebrauch gemacht, nämlich *rules.txt*, die die Regeln für die Grammatik enthält und *lexicon.txt*, in denen die Singular- und Pluralform von bestimmten Substantiven gespeichert sind. Das zugehörige Paket umfasst

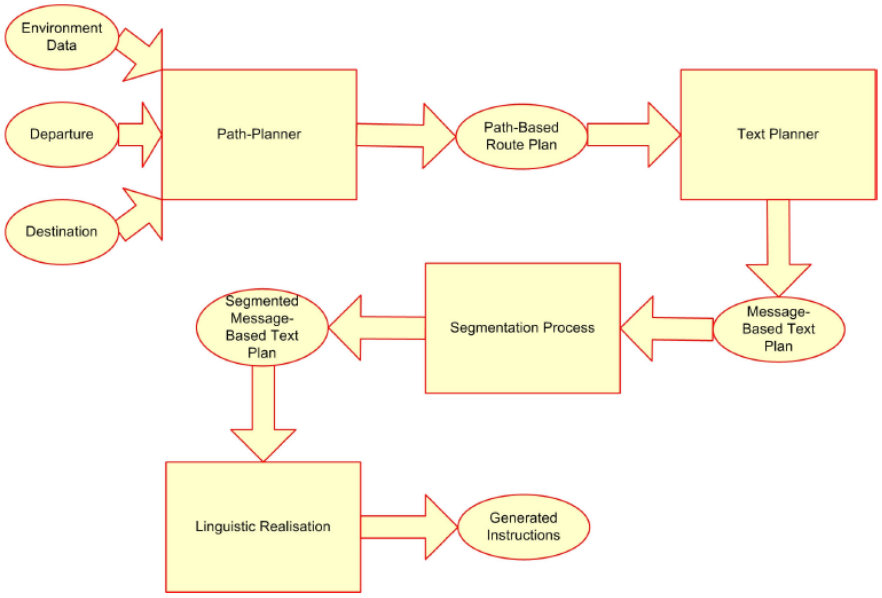

Abbildung 3.6: NLG Modul Architektur [Miz04]

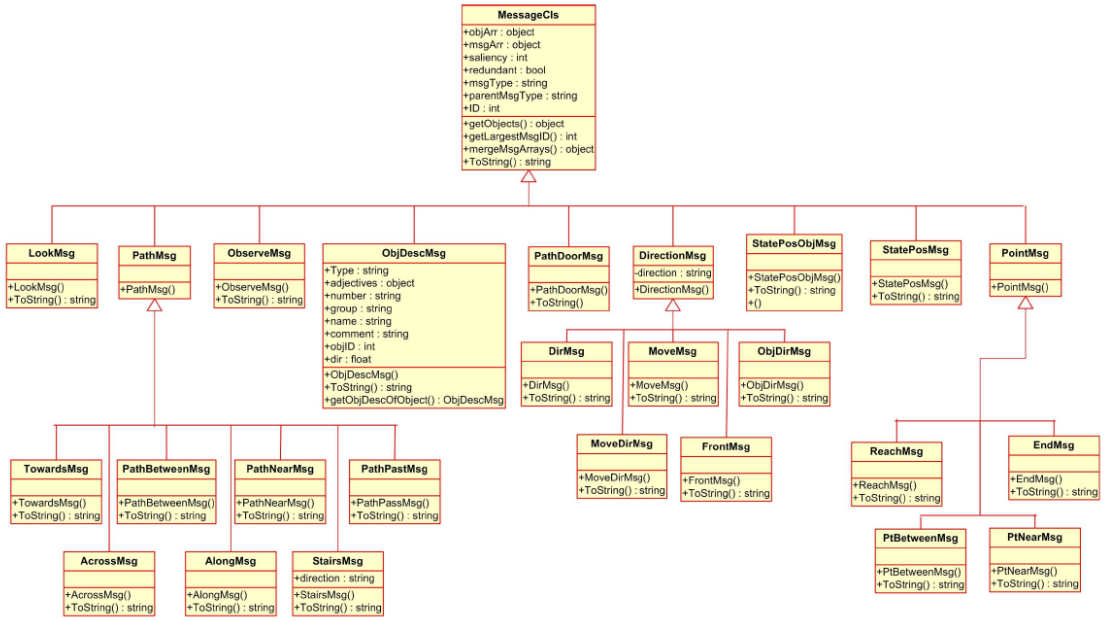

Abbildung 3.7: MessageCls mit Unterklassen. [Miz04]

die Klassen *LexicaliseCls*, *LexiconCls* und *RuleCls*.

Außerdem existiert ein viertes Paket *General*, dessen Mitglieder in allen vier Stufen benötigt werden. Dabei handelt es sich um die beiden bereits erwähnten Klassen *PathClass* und *PtUsedInPathCls*, sowie um *SortingClasses* und *GeometryClass*, die für die Berechnung der einzuschlagenden Richtung zuständig ist.

3.3.4 Local Positioning Component

Diese Komponente übermittelt dem NAS die aktuelle Ortsposition. In der originalen Arbeit wurde sie aus bereits erwähnten Gründen nur simuliert. Im realen Einsatz ist sie jedoch durch ein externes LPS zu ersetzen, das je nach Konfiguration wahlweise an die *Navigational Assistance Server Application* oder die *Personal Assistant Mobile Client Application* angeschlossen werden kann.

3.3.5 Simulation GUI Component

Diese extra für das Projekt in Malta entwickelte Komponente stellt eine graphische Benutzeroberfläche für die Bedienung des simulierten LPS bereit. Dabei bekommt der Benutzer eine Karte angezeigt, auf der er mit Hilfe von Richtungsbefehlen, die über die Tastatur eingegeben werden müssen, eine sich durch das Gebäude bewegende Person nachbilden kann.

3.3.6 Communication Component

Hier sind alle notwendigen Klassen enthalten, die für 802.11b- oder Bluetoothverbindungen benötigt werden. Man entschied sich in der Uni von Malta für die zusätzliche Implementierung von Bluetooth-Funktionen, weil dort kein WLAN vorhanden war. Da die Kommunikation auf dem TCP/IP-Protokoll basiert, kann jedoch selbstverständlich auch ein IEEE 802.11-Funknetz verwendet werden (Abb. 3.8).

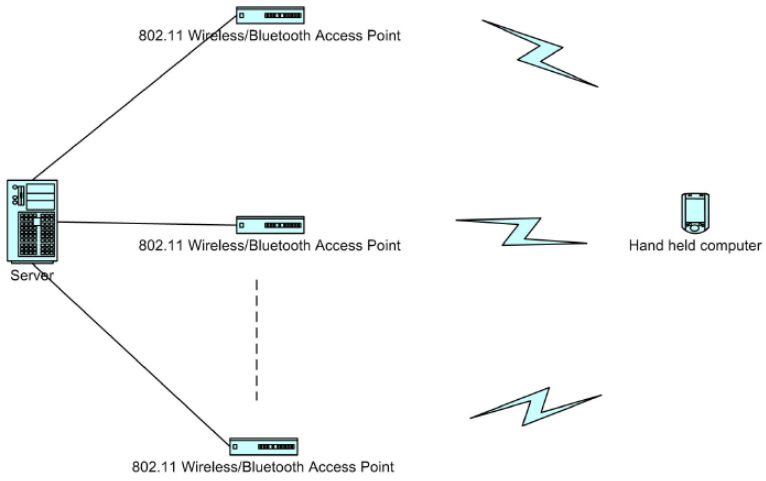

Abbildung 3.8: Netzwerk Setup [Miz04]

3.3.7 Navigation Assistant Server

Der *Navigation Assistant Server* ist das Herzstück der Anwendung, er steuert alle für die Navigation benötigten Komponenten. Dabei ist die Klasse *NavServerFrm* für die Anzeige einer einfachen Benutzeroberfläche zuständig, *GuidingServerCls* für die Kommunikation mit dem Client und *NASAssistantCls* für das Ausführen der Wegeberechnung. Ersterer ist außerdem verantwortlich für das Laden der richtigen Grammatikregeln für die spezifischen Umgebungen, damit eine natürlich klingende Textausgabe erzeugt werden kann. Der Server wird mit Hilfe einer vom Administrator erstellten Datei im XML-Format konfiguriert, welche die eigene IP-Adresse und die Portnummer sowie die Dateinamen des 2D-Plans, der Grammatikregeln und des Wörterbuchs enthält.

3.3.8 Mobile Client

Diese Komponente stellt dem Benutzer eine grafische Oberfläche zur Verfügung, durch die er auf seinem mobilen Endgerät mit dem NAS interagieren kann. Der Anwender wird aufgefordert, den Namen eines Zielraumes oder - im Falle eines Buches - die entsprechende ID einzugeben und gegebenenfalls auch ein auffälliges Objekt zu benennen, um dem NAS dadurch indirekt seine Startposition mitzuteilen. Auf dem Display des PDA werden die Navigationsanweisungen in Textform ausgegeben. Auch der *Mobile Client* erwartet eine XML-Konfigurationsdatei. Diese muss die IP-Adresse und Portnummer des Servers sowie eine Vorgabe enthalten, ob der Text segmentiert werden und ob der Benutzer seinen Startpunkt manuell eingeben soll. Die Clientanwendung besteht aus zwei Klassen, wobei eine, *AssistClientCls*, für die Kommunikation mit dem Server zuständig ist und die andere, *PersAssistFrm*, die grafische Benutzeroberfläche bereitstellt. Diese ist in Abbildung 3.9 zu sehen.

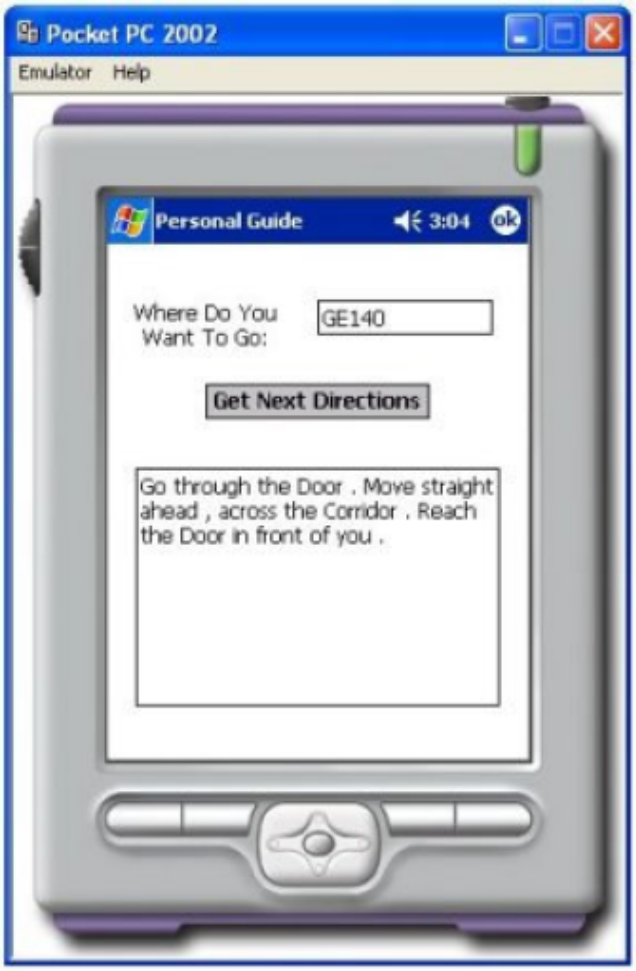

Abbildung 3.9: Die mobile Anwendungsschnittstelle [Miz04]

Kapitel 4

Mögliche Erweiterungen des Systems

Das in [Miz04] beschriebene NAS wurde in der Praxis erfolgreich getestet, bietet aber in manchen Punkten noch Spielraum für sinnvolle Erweiterungen. Entsprechende Vorschläge wurden vollständig der Originalarbeit entnommen und sollen im Folgenden kurz besprochen werden.

4.1 Unterstützung weiterer Netzwerkschnittstellen

Wie bereits in Abschnitt 3.3.6 erwähnt, unterstützt das NAS nur TCP/IP-basierende Verbindungen mit den IEEE 802.11- und Bluetooth-Standards. Wünschenswert wäre die Unterstützung weiterer drahtloser Schnittstellen wie beispielsweise Infrarot. In diesem Fall könnte das NAS auf verschiedene Netzwerkinfrastrukturen zurückgreifen. Somit würde die Anwendung weiter an Flexibilität gewinnen und sein Einsatzbereich erweitert werden.

4.2 Art der Datenspeicherung

Da das System ursprünglich nur für ein Stockwerk konzipiert wurde, war es bisher ausreichend, den 2D-Umgebungsplan in einer einzigen XML-Datei abzulegen. Dies führt dazu, dass der Server stets nur ein Gebäude kennen kann und dessen Plan in der Konfigurationsdatei fest vorgegeben werden muss. Komfortabler und aus Effizienzgründen besser wäre es, die Gebäudekarte in einer konventionellen Datenbank (z.B. SQL) abzuspeichern, aus der sie bei Bedarf abgerufen werden können.

4.3 Erhöhung der Benutzerfreundlichkeit

Die Rückfrage der aktuellen Position des Anwenders könnte besser in Form eines Benutzerdialoges realisiert werden. Auch durch eine Navigation mit Hilfe von gesprochenen Anweisungen ließe sich eine weitere Steigerung des Benutzerkomforts erzielen. Noch benutzerfreundlicher wäre jedoch wohl eine graphische Repräsentation der Wegbeschreibung.

4.4 Anpassung an Mehrdimensionalität

Der Verwendbarkeit des NAS in seiner Originalform sind gewisse Grenzen gesetzt, weil es konzeptionell nur für ein einziges Stockwerk ausgelegt ist. Es wäre daher wünschenswert, es an eine mehrdimensiona-

le Umgebung anzupassen. Hierzu muss sowohl der Gebäudeplan mehrere Etagen enthalten als auch der Navigationsprozess stockwerkübergreifend erfolgen. Die einzelnen Ebenen müssen dabei geeignet miteinander verknüpft werden. Als solche Verknüpfungselemente fungieren dabei Treppen oder Aufzüge, die die Übergänge zwischen den einzelnen Stockwerken darstellen. Durch das Verbinden der einzelnen 2D-Karten erhält man praktisch einen ”2,5”-dimensionalen Umgebungsplan des ganzen Gebäudes. In die Berechnung des kürzesten Weges zum gewünschten Ziel sollten am besten auch die Kosten (Zeitbedarf für die Benutzung einer Treppe beziehungsweise eines Aufzugs) in geeigneter Form einfließen.

4.5 Dynamische Wegeberechnung

Zu Beginn berechnet das NAS ein einziges Mal den einzuschlagenden Weg und erzeugt anschließend die kompletten Navigationsanweisungen. Der Nachteil bei diesem Ansatz ist, dass das System nicht erkennt, wenn der Benutzer versehentlich oder aus einem bestimmten Grund von der beschriebenen Route abweicht und diesen logischerweise auch nicht warnen oder korrigieren kann. Daher wäre es sinnvoll, ein dynamisches Verhalten zu realisieren, bei der die Bewegung der zu navigierenden Person durchgehend nachverfolgt und gegebenenfalls - unter gleichzeitiger Benachrichtigung des Benutzers - entsprechend bereinigt wird.

4.6 Anbindung an ein Positionierungssystem

Die im vorhergehenden Unterabschnitt besprochene Modifizierung setzt zwingend voraus, dass der genaue Standort des Anwenders zu jedem Zeitpunkt bekannt ist. Hierfür muss auf die von einem Lokalisierungssystem gelieferten Positionsdaten zurückgegriffen werden können. In der originalen Arbeit wurde aus Zeitgründen auf die Entwicklung eines eigenen Positionierungssystems verzichtet und ein solches nur simuliert. Hier ließe sich eine Anbindung an ein externes LPS realisieren, wobei die Schnittstelle kompatibel zu aktuellen Standards sein sollte, um auch beispielsweise einen handelsüblichen GPS-Empfänger für den möglichen Einsatz im Outdoor-Bereich anschließen zu können.

Kapitel 5

Ziele und Lösungsansätze dieser Diplomarbeit

Im Rahmen dieser Diplomarbeit war es meine Aufgabe, das von [Miz04] entwickelte System in verschiedenen Punkten zu modifizieren beziehungsweise zu erweitern und dabei einige der im vorhergehenden Kapitel aufgelisteten Unzulänglichkeiten zu korrigieren. Außerdem sollte das vorhandene NAS in die Infrastruktur des *Institut für Informatik der LMU München* geeignet eingegliedert und gleichzeitig eine Anbindung an ein Indoor-LPS geschaffen werden, um die von diesem gelieferten Positionsdaten für die Navigation zu nützen. Dieses LPS wurde in einer parallel laufender Diplomarbeit [Ric06] entwickelt. Bei der Art der Anbindung war darauf zu achten, dass sie kompatibel zu aktuellen Standards ausfiel, um gegebenenfalls handelsübliche GPS-Empfänger verwenden und das System somit später möglicherweise auch im Outdoor-Bereich einsetzen zu können.

Weiterhin war ein wesentlicher Nachteil des bestehenden Systems zu beseitigen, nämlich die Beschränkung auf ein einzelnes Stockwerk. Dies ist für ein NAS, das in größeren Gebäuden eingesetzt werden soll, nicht akzeptabel und mindert den Wert eines solchen Systems wesentlich. Um dieses Ziel zu erfüllen, waren jedoch tiefgehende Veränderungen unvermeidbar.

Schließlich sollte das System ein dynamisches Verhalten aufweisen, das heißt, den Weg zum Ziel nicht nur ein einziges Mal beim Start bestimmen, sondern diese Berechnung bei Bedarf selbständig erneut ausführen. Hierfür ist es notwendig, dass der aktuelle Standort des Benutzers in regelmäßigen Zeitabständen abgefragt und zu welchem Zweck auf das angebundene LPS zurückgegriffen wird.

Ferner ergab sich der Wunsch, die bisherige Ausgabe der Anweisungen in Textform durch einen grafischen Ansatz zu ergänzen und zu erweitern. Um dabei einen möglichst großen Teil des vorhandenen Codes wiederverwenden zu können, sollte in einem ersten Schritt die Textausgabe geeignet angepasst werden, so dass der Anwender auch in mehrstöckigen Gebäuden eine korrekte Folge von Anweisungen präsentiert bekommt. Anschließend kann man diese in einem weiteren, zweiten Schritt in grafische Symbole umsetzten, um sie so auf dem Display des mobilen Clients anzuzeigen.

Wie es sich im späteren Verlauf dieser Diplomarbeit jedoch gezeigt hat, führte dieser verwendete Ansatz bei der Umsetzung von Text in bildliche Symbole zu einem nicht ganz befriedigenden Ergebnis und hatte nachteilige Auswirkungen auf die Art und Weise, wie die grafische Ausgabe realisiert werden musste. Auch ließ sich dadurch kein einwandfreies dynamisches Verhalten mehr erreichen. Auf diese Problematik und den daraus folgenden Konsequenzen wird in Kapitel 10 noch detailliert eingegangen.

5.1 Anbinden an ein Ortungssystem und Definieren einer Schnittstelle

Um die von einem Ortungssystem gelieferten Indoor-Positionierungsdaten zur mobilen Routenplanung nutzbar zu machen, musste ein Weg gefunden werden, um ein solches LPS an das vorhandene System anzubinden. Ferner war es erforderlich, einen geeigneten Standard für die zu verwendende Schnittstelle zu finden. Bei alledem mussten zukünftige Erweiterungsoptionen offen bleiben. Speziell sollte die Anwendung ja möglicherweise auch einmal im Outdoor-Bereich einsetzbar sein. Daher sollten zum Erreichen des Ziels möglichst gängige Industriestandards Verwendung finden. Insbesondere war die Kompatibilität zu auf dem Markt verfügbaren GPS-Empfängern einzuhalten, um von diesem bewährten Positionierungssystem Ortsdaten abfragen zu können.

In [Ric06] wurde am Institut für Informatik der Ludwig Maximilian Universität München ein Lokalisierungssystem für Personen in Gebäuden entwickelt, von welchem die gewünschten Daten bezogen werden sollten. Die Entwicklung der zu benutzenden Schnittstelle geschah in Kooperation mit dem Autor der vorstehend erwähnten Arbeit.

Bei der Art der Anbindung fiel die Wahl auf den Bluetooth-Standard, der auch in handelsüblichen GPS-Geräten verwendet wird. Die Positionierungsdaten selbst werden als ASCII-Strings im NMEA 0183 Format übertragen. Auch bei Letzterem handelt es sich um einen im Bereich der Ortungs- und Navigationssysteme viel genutztes Protokoll. Sowohl auf den Bluetooth- als auch auf den NMEA-Standard wird in Kapitel 6 noch ausführlich eingegangen werden.

5.2 Navigieren des Benutzers über mehrere Stockwerke hinweg

Da das vorhandene System so erweitert werden sollte, dass es den Benutzer über mehrere Stockwerke eines Gebäudes hinweg leiten kann, musste ein geeigneter Lösungsansatz gefunden werden, um dies zu realisieren.

Ein einzelnes Stockwerk kann man sich als zweidimensionale Ebene vorstellen. Wenn wir die Treppen und Aufzüge als Übergänge zu den benachbarten Etagen mit einbeziehen, erhalten wir eine ”2,5”-dimensionale Umgebung. Abbildung 5.1 gibt dem Leser eine Vorstellung von einem solchen Raum. Liegen Start- und Zielpunkt auf verschiedenen Ebenen, muss man die Route in mehreren Schritten berechnen. In diesem Fall wird zuerst der Weg zu einer Treppe bzw. Aufzug gesucht, von wo aus im Optimalfall dann in das Zielstockwerk gewechselt werden kann. Vom Lift beziehungsweise Treppenausgang der korrekten Etage aus ist dann letztendlich nur noch der Weg bis zum erwarteten Zielpunkt zu bestimmen.

Unter Umständen wird die Zieletage nicht direkt über eine einzige Treppe oder Aufzug erreicht. In solchen Fällen sind auch die dazwischen liegenden Stockwerke bei der Wegefindung geeignet zu berücksichtigen.

Abgesehen von den notwendigen Modifizierungen des Umgebungsplanes war also auch ein geeigneter Algorithmus zur Bestimmung des kürzesten Pfades zu finden. Der bisher eingesetzte A*-Algorithmus eignet sich aufgrund seiner Heuristik nur für die Wegeberechnung im zweidimensionalen Raum [Miu02]. Die bei diesem Verfahren verwendete Schätzfunktion bestimmt die Luftlinie vom zuletzt besuchten Knoten bis zum Zielpunkt und klammert alle Kandidaten aus, die über dem erhaltenen Wert liegen. Dadurch verkleinert sich der Suchraum und die Berechnung kann dadurch schneller durchgeführt werden. Während das bei zweidimensionalen Karten beziehungsweise im Freien keine Probleme aufwirft, bewegt sich ein Mensch in einem mehrdimensionalen Gebäude üblicherweise nicht in direkter Luftlinie auf sein Ziel zu. Der A*-Algorithmus leistet in einer solchen Umgebung aus diesem Grund oftmals nicht das Gewünschte, weshalb er in dieser Arbeit durch den *Dijkstra*-Algorithmus ersetzt wurde. Dieser wird in Kapitel 6 genauer beschrieben.

Zusätzlich sollte die neue Wegeplanung auch den notwendigen Zeitbedarf in Betracht ziehen. Hier musste berücksichtigt werden, dass die Benutzung einer Treppe oder eines Aufzuges unterschiedliche Kosten nach

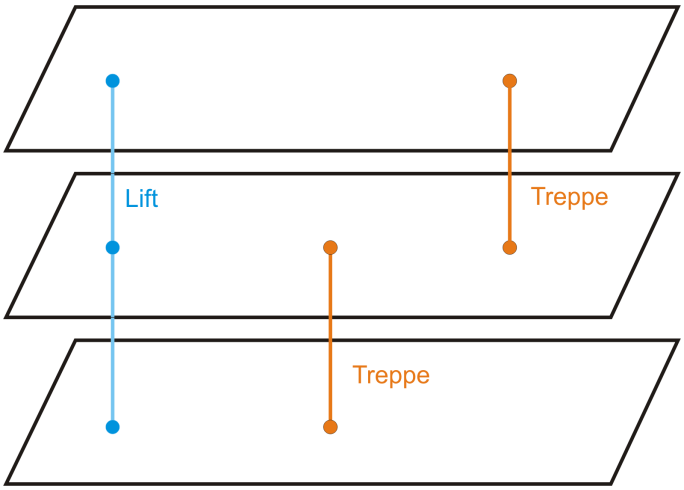

Abbildung 5.1: ”2,5”-dimensionale Umgebung

sich ziehen, wobei ”unter Kosten” der notwendige Zeitbedarf zu verstehen ist. Mangels Erfahrungswerten waren diese empirisch zu bestimmen.

5.3 Grafische Darstellung der Wegbeschreibung

Im Rahmen dieser Diplomarbeit sollte außerdem die ursprüngliche Textausgabe durch eine grafische Darstellung mit aussagekräftigen Symbolen ersetzt werden.

Bei der Implementierung einer solchen Ausgabe auf einem mobilen Endgerät wie beispielsweise ein Handheld oder PDA sind zwei Arten von Ressourcen zu berücksichtigen: technische und kognitive. Zur ersteren zählen Parameter wie Rechenleistung, Geschwindigkeit, Bandbreite oder Bildschirmgröße. Diese beeinflussen, wie schnell und in welcher Qualität die Anweisungen berechnet und dargestellt werden können. Kognitive Ressourcen bezeichnen die momentane Aufnahmefähigkeit des Benutzers, die von Faktoren wie Zeitdruck oder Ablenkungen durch Ereignisse seines Umfelds abhängen und die nicht überbeansprucht werden darf. Das bedeutet zum Beispiel, dass einfache Wege den komplizierten vorzuziehen sind, auch wenn sie über eine längere Strecke führen. Das heißt weiterhin, dass die angebotenen Informationen leicht erfassbar sein müssen und den Benutzer nicht vom Geschehen um ihn herum ablenken oder gar überfordern dürfen. Weiterführende Literatur zu diesem komplexen Thema kann [Kra03] entnommen werden.

Vor allem in Gebäuden, wo im Vergleich zu einer Outdoor-Umgebung nur wenige auffällige Orientierungspunkte existieren, ist eine grafische Ausgabe aus oben erläuterten Gründen von Vorteil. Navigationsanweisungen in Textform sind unter solchen Umständen oft relativ schwer nachvollziehbar.

Gängige Möglichkeiten, Navigationsanweisungen grafisch darzustellen, sind *2D route sketches*, *2D* oder *3D maps* [Kra03].

5.3.1 2D route sketches

Instruktionen in Form von 2D route sketches sind am leichtesten verständlich. Ein schlichter Pfeil teilt dem Benutzer den einzuschlagenden Weg mit. Optional kann diese zusätzliche Informationen besitzen, wie in Abbildung 5.2 zu sehen ist. Jedoch resultiert aus dieser Anzeigeart das Problem, dass zu Beginn des Navigationsprozesses die Blickrichtung des Anwenders bekannt sein und der richtige Weg durch zusätzliche Maßnahmen, wie etwa das Hinweisen auf ein bestimmtes Objekt, verdeutlicht werden muss. Ferner kann der hohe Abstraktionsgrad bei 2D route sketches zum Verlust von Ortsinformationen führen, die bei

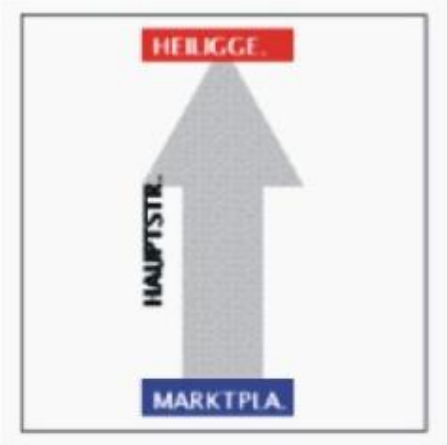

Abbildung 5.2: 2D route sketch [Kra03]

der Navigation möglicherweise hilfreich gewesen wären. Diese Form der Wegbeschreibung wird häufig in Navigationssystemen für Personenkraftfahrzeuge verwendet [Kra03].

5.3.2 2D maps

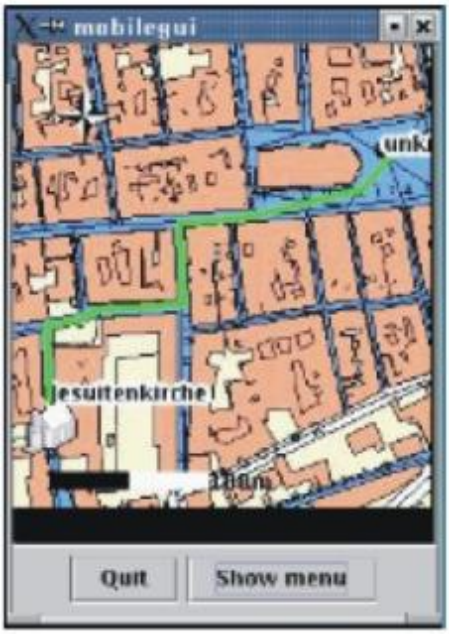

Abbildung 5.3: 2D map [Kra03]

Bei 2D maps werden die Navigationsanweisungen zusammen mit einer Karte der Umgebung aus der Vogelperspektive zur Verfügung gestellt. Im Vergleich zu den 2D route sketches muss hierbei die Blickrichtung des Benutzers nicht bekannt sein, den einzuschlagenden Weg kann man anhand des Umgebungsplans nachvollziehen. Bei der Verwendung einer 2D map ist es wichtig, dass die in ihr enthaltenen geografischen Informationen vom Anwender leicht erfassbar sind. Das heißt, die Karte muss mit zusätzlichen Daten versehen werden, damit er weiß, wo er sich befindet und welche Gebäude den auf dem Plan abgebildeten

entsprechen. Weitere Faktoren, die man bei der Anzeige einer Karte zu berücksichtigen hat, sind die Größe des Ausschnitts und ihre Ausrichtung. Erstere darf weder zu groß noch zu klein gewählt sein, da der Anwender sonst entweder zu wenige Informationen bekommt oder von einem zu großen Kartenausschnitt verwirrt wird. Letztere sollte möglichst in Lauf- oder Fahrrichtung des Benutzers liegen, um zu vermeiden, dass er das Gerät ständig drehen muss. Zuletzt ist noch von Bedeutung, wie viele Details, wie beispielsweise Straßennamen oder Raumnummern, die 2D map enthält. Dieser höhere Genauigkeitsgrad im Vergleich zu 2D route sketches wird allerdings mit einem zusätzlichem Ressourcenverbrauch erkauft.

5.3.3 3D maps

Abbildung 5.4: 3D map [Kra03]

Eine andere Art zur Ausgabe von grafischen Instruktionen ist die 3D map, wie in Abbildung 5.4 vorgestellt. Dabei wird dem Benutzer eine dreidimensionale Ansicht der Umgebung geboten, anhand der er schnell seinen einzuschlagenden Weg erkennen kann. Man bezeichnet diese Form auch als *pseudo realistic instructions* [Kra03]. Hinsichtlich Detailgrad und Informationsgehalt gelten bei 3D maps im Wesentlichen dieselben Gesichtspunkte wie im vorhergehenden Abschnitt beschrieben. Großer Vorteil dieser Darstellungsart ist hierbei, dass auch Anweisungen in der dritten Dimension klar ausgegeben werden können. Auch hier muss aber ein Kompromiss zwischen Abstrahierung und der Menge der gebotenen Informationen gefunden werden.

Diese Ausgabeform hat sich bei Umfragen großer Beliebtheit erfreut, wenn das NAS für den City-Bereich eingesetzt wurde. Die Möglichkeit, Gebäude mit hohem Wiedererkennungswert als dreidimensionale Objekte darzustellen, erleichtert den Navigationsprozess sehr. Im Gebäudeinneren können 3D maps aber ihre wahren Stärken nicht vollständig zur Geltung bringen und es ist ferner für die Anzeige und Speicherung des Umgebungsplanes eine zusätzliche Ressourcenbelastung in Kauf zu nehmen. Im Fall mobiler und handlicher Geräte stellt das jedoch oft keine Alternative dar, da dies einen höheren Energie- und Speicherverbrauch zur Folge hat und die Berechnung der Navigationsinstruktionen längere Zeit in Anspruch nimmt.

5.4 Dynamische Wegberechnung und ggf. Korrektur des Weges

Das von [Miz04] entwickelte NAS berechnet den Weg nur einmal zu Beginn der Navigationsaufgabe, zeigt also ein statisches Verhalten. Dieser Ansatz könnte sich als problematisch erweisen, wenn der User versehentlich einen anderen Weg nimmt oder der Startpunkt falsch ausgegeben wird. Es sollte daher eine dynamische Berechnung realisiert werden, bei der in bestimmten Zeitabständen der Ort des mobilen Endgerätes festgestellt und mit der erwarteten Position verglichen wird.

Stimmt die Position des Users mit dem erwarteten Ort nicht überein, wird angenommen, dass der Benutzer vom richtigen Weg abgewichen ist. Der Server sollte in einer solchen Situation selbständig einen alternativen Weg zum Ziel berechnen und ihn dem Client übermitteln. Dabei muss in Betracht gezogen werden, dass auch die vom LPS ermittelte Position nicht zwingend hundertprozentig korrekt ist, so dass eine gewisse Fehlertoleranz einzukalkulieren ist.

Im Hinblick auf die eingeschränkte Rechenleistung mobiler Geräte im Vergleich zu Desktop PCs darf die Benutzerposition nicht unnötig oft abgefragt und mit dem erwarteten Wert verglichen werden. Diese Ortsaktualisierung muss aber andererseits auch ausreichend häufig geschehen, um Navigationsfehler rechtzeitig korrigieren zu können. Es wäre also ein Kompromiss zwischen beiden Anforderungen zu finden gewesen.

Aufgrund der in Kapitel 10 dargelegten Probleme mit dem *Text Planner*-Modul und der von ihm generierten Ausgabe war eine saubere Realisierung des dynamischen Verhaltens beim vorliegenden System nicht durchführbar. Es konnte zwar die Anwendung so angepasst werden, dass die im ersten Abschnitt aufgeführte dynamische Wegeberechnung wie beabsichtigt ausgeführt worden ist, aber die erhaltene Ausgabe war in der erhaltenen Form nur beschränkt verwertbar.

Kapitel 6

Verwendete Techniken

6.1 Algorithmus von Dijkstra

Der Holländer *Edsger Wybe Dijkstra* [Dij05] erfand 1959 einen *Greedy*-Algorithmus, der für jeden Knoten in einem kantengewichteten Graphen den kürzesten Weg zum Startknoten berechnet. Bei diesem handelt es sich um einen *single-source-shortest-path*-Algorithmus, der weitgehend dem A*-Algorithmus entspricht, aber keine Heuristik verwendet. Der Algorithmus [Ber01] funktioniert wie folgt:

Sei v_{Start} der Startknoten eines gewichteten, zusammenhängenden Graphen. Betrachte alle Nachbarknoten von v_{Start} und notiere die jeweiligen Kosten für den Weg zum Startknoten. Im Folgenden wähle immer den billigsten, noch nicht besuchten Knoten und teste, ob es von ihm aus einen günstigeren Weg zu einem anderen Knoten gibt als bisher der Fall war. Aktualisiere die Kosten für den kürzesten Weg zum Startknoten und wiederhole dies so lange, bis alle Knoten besucht wurden. Am Ende erhalten wir die günstigsten Wege von v_{Start} zu jedem anderen Knoten. Bei dem im Rahmen dieser Arbeit entwickelten Systems wird der Algorithmus beendet, wenn der gewünschte Zielknoten v_{Ziel} erreicht worden ist.

Formal ist der Algorithmus in Abbildung 6.1 aufgelistet.

Im vorliegenden Fall berechnet sich das Gewicht einer Kante des Graphen aus der euklidischen Distanz ihrer Eckpunkte. Aufgrund der verminderten Laufgeschwindigkeit bei Treppen und die zu berücksichtigende Wartezeit bei den Aufzügen mussten den durch sie repräsentierten vertikalen Kanten spezielle Mali zugewiesen werden. Diese sind, wie im nächsten Abschnitt beschrieben, empirisch zu bestimmen.

6.2 Empirische Bestimmung der Mali bei vertikalen Kanten

Die Kantengewichte für Treppen und Aufzüge waren mit einem geeigneten Malus zu versehen. Das Malus für Treppen fand sich, indem die typische Laufgeschwindigkeit eines Anwenders bei der Benutzung einer solchen empirisch bestimmt worden war. Dazu war zuerst die Höhe zweier übereinander liegender Stockwerke zu vermessen. Durch Stoppen der Zeit, die eine Person für das Hinauf- beziehungsweise Hinunterlaufen der Treppe benötigt, konnten empirische Werte für den zu verwendeten Malus erhalten werden.

Der Mittelwert aus 25 solcher Messungen bei verschiedenen Personen betrug bei einer Treppenhöhe von 3,17 Metern 13,78 Sekunden. Daraus lässt sich eine ungefähre Laufgeschwindigkeit von 0,23 m/s (0,83 km/h) in der Vertikalen ableiten. In Relation zur durchschnittlichen Fußgängergeschwindigkeit auf horizontaler Ebene, die laut [Spe05] zwischen 3,6 und 5,4 km/h beträgt, war somit in diesem Fall die Benutzung einer Treppe 4,35 bis 6,52-mal aufwändiger. Daher wurde im Rahmen dieser Arbeit ein Malus von 5 für

Dijkstra-Routine:

1	$v_{Start}, v_{Ziel} \in V$	Start-, Zielknoten gegeben
2	$w_{i,j} \in \mathbb{R}^+$	Kantenlängen,
3	$v.Adj \subset V$	Nachfolger gegeben
4	$v.dist \in \mathbb{R}^+$	Distanzen,
5	$v.pre \in V$	Vorgänger gesucht
6	$v, v_K \in V$	
7	$v.dist, d \in \mathbb{R}^+$	Deklaration
8	$B \subset V$	
9	foreach $v \in V$ do	Initialisierung
10	$v.pre \leftarrow NULL$	kürzester Vorgänger leer
11	end foreach	
12	$v_{start}.pre \leftarrow v_{start}$	Startknoten initialisieren
13	$v_{start}.dist \leftarrow 0$	
14	$B \leftarrow \{v_{start}\}$	Front initialisieren
15	while $B \neq \emptyset$ do	
16	$v_K \leftarrow$ wähle $v \in B \wedge v.dist = min$	Knoten aus Front wählen
17	$B \leftarrow B \setminus \{v_K\}$	und entfernen
18	if $v_K = v_{Ziel}$ do end	Ziel erreicht?
19	foreach $v \in v_K.Adj$ do	Nachfolger untersuchen
20	$d \leftarrow v_K.dist + w_{v_K,v}$	Neue Distanz berechnen
21	if $v.pre = NULL$ do	Nachfolger neu?
22	$v.pre \leftarrow v_K$	Nachfolger initialisieren
23	$v.dist \leftarrow d$	
24	$B \leftarrow B \cup \{v\}$	und in Front aufnehmen
25	else do	Nachfolger bereits in Front
26	if $d < v.dist$ do	neuer Weg kürzer?
27	$B \leftarrow B \setminus \{v\}$	
28	$v.pre \leftarrow v_K$	Nachfolger reinitialisieren
29	$v.dist \leftarrow d$	
30	$B \leftarrow B \cup \{v\}$	und in Front umbewerten
31	end if	
32	end if	
33	end foreach	
34	end while	

Abbildung 6.1: Dijkstra Algorithmus [Ber01]

vertikale Kanten, welche eine Treppe repräsentiert, gewählt, um außerdem die normalerweise etwas langsamere Laufgeschwindigkeit des Benutzers zu berücksichtigen, wenn dieser einen PDA zu Navigationszwecken in der Hand hält. Jedoch ist mit Vorsicht vorzugehen, wenn man die Absicht hat, diese Werte auf andere Gebäude zu übertragen. Denn je nach Architektur eines Gebäudes erhält man dafür andere Werte, die gegebenenfalls analog zu dem soeben beschriebenen Verfahren bestimmt werden müssen.

Im Gegensatz dazu gestaltet sich eine empirische Bestimmung der Kantengewichte bei Aufzügen schwieriger. Die Kosten für die Benutzung eines Lifts hängen nämlich stark von der Anzahl der Stockwerke ab, die von ihm überbrückt werden. Während beispielsweise im Normalfall der Weg über die Treppe schneller ist, wenn man nur von dem ersten in den zweiten Stock will, ist der Aufzug klar im Vorteil, wenn das Ziel etwa auf der dreißigsten Etage liegt. In ersterem Fall bedeutet alleine schon die notwendige Wartezeit einen deutlichen Vorsprung für die Treppe. Als weitere Schwierigkeit bei der Festlegung der Gewichte für einen Lift kommen hinzu, dass sich weder die Warte- noch Fahrzeit allgemeingültig bestimmen lassen. Diese sind nicht nur von der Fortbewegungsgeschwindigkeit abhängig, sondern auch wie viele Stockwerke zurückgelegt werden müssen und wie oft und wie lange während der Fahrt angehalten wird, damit Leute ein- und aussteigen können.

Das Gebäude, in dem das vorliegende System getestet worden war, verfügte lediglich über einen einzigen Aufzug, der sich über drei Etagen erstreckte. Dies erlaubte nur eine relativ grobe Bestimmung des Malus, für den ein Wert von 10 gefunden wurde und sich bei Tests auch als durchwegs realistisch und brauchbar erwiesen hat.

6.3 Koordinatentransformation

Da die aktuelle Benutzerposition von einem LPS im NMEA Datenformat übergeben wird, müssen die darin enthaltenen Längen- und Breitengrade zuerst in das für die Wegeberechnung verwendete Pixel-Koordinatensystem transformiert werden. Diesen Vorgang bezeichnet man als *Koordinatentransformation* [Moe04].

Hierfür sind zwei Referenzpunkte notwendig, deren Koordinaten sowohl im Quell- als auch im Zielsystem bekannt sind. Mit Hilfe dieser beiden Punkte kann man dann jeden beliebigen anderen Punkt von dem geographischen in das Pixel-Koordinatensystem transformieren und umgekehrt.

Für die Koordinatentransformation wird folgende Formel [Moe04] verwendet:

Seien A und E die Referenzpunkte. Sei P_i der zu transformierende Punkt, dessen Koordinaten dem Quellsystem bekannt sind. Dann gilt für die Koordinaten (x_i,y_i) dieses Punktes im Zielsystem:

$$x_i = x_A + \alpha \cdot (\zeta_i - \zeta_A) - \sigma \cdot (\eta_i - \eta_A)$$

$$y_i = y_A + \sigma \cdot (\zeta_i - \zeta_A) + \alpha \cdot (\eta_i - \eta_A)$$

Dabei repräsentieren ζ die x-Koordinate und η die y-Koordinate des Quellensystems. Des Weiteren gilt

$$\alpha = \frac{\triangle\eta \cdot \triangle y + \triangle\zeta \cdot \triangle x}{s^2}$$

und

$$\sigma = \frac{\triangle\zeta \cdot \triangle y - \triangle\eta \cdot \triangle x}{s^2}$$

wobei

$$\triangle\zeta = \zeta_E - \zeta_A$$

$$\triangle\eta = \eta_E - \eta_A$$

$$\triangle x = x_E - x_A$$

$$\triangle y = y_E - y_A$$

die Abstände zwischen den (x,y)-Koordinaten von A und E im alten und neuen Koordinatensystems bezeichnen. Ferner gilt für s^2

$$s^2 = \triangle\zeta^2 + \triangle\eta^2$$

6.4 Bluetooth

Die universelle Kurzstrecken-*Ad-hoc*-Funkschnittstelle Bluetooth [Hol03], nach dem Wikingerkönig Blåtand, auf Deutsch "Blauzahn", benannt, geht auf eine Entwicklung des skandinavischen Handyherstellers *Ericsson* im Jahre 1994 zurück, als man nach einer einfachen, ressourcensparender und kostengünstigen Möglichkeit suchte, die lästigen Kabel zwischen Mobiltelefonen und Zusatzgeräten zu ersetzen. 1998

wurde von einigen weiteren Hardwareherstellern schließlich das Konsortium *Bluetooth Special Interest Group (SIG)* ins Leben gerufen, zu der heute mehr als 2000 Einrichtungen angehören.

Im Vergleich zu damals verbreiteten drahtlosen Übertragungsverfahren wie Infrarot besitzt Bluetooth den Vorteil, dass eine Sichtlinie zwischen den Komponenten keine zwingende Vorauszusetzung ist. Über ein global verfügbares, lizenzfreies 2,4-GHz-*Industrial Scientific and Medical, (ISM)*-Frequenzband können in einem sogenannten *Piconetz* (Abb. 6.2), das je nach Stärke der Sendeleistung eine maximale Reichweite von etwa 10, 25 oder 100 Metern besitzt, bis zu 255 Geräte angeschlossen werden, wobei acht (ein *Master* und bis zu sieben *Slaves*) gleichzeitig aktiv sein dürfen.

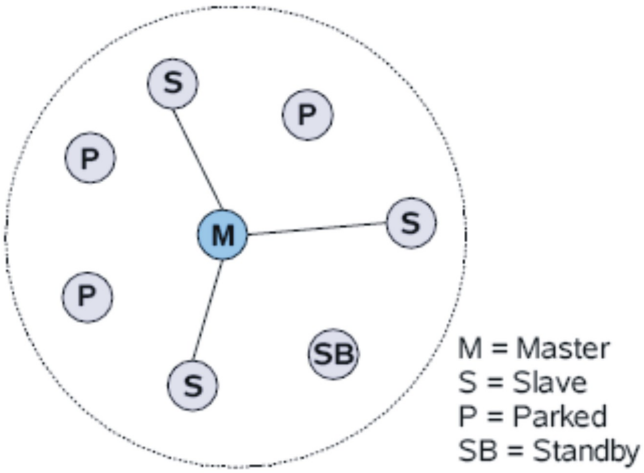

Abbildung 6.2: Piconetz [Hol03]

Die geringe Anzahl der Kommunikationspartnern ergibt sich aus der lediglich drei Bit langen Adresse, die für den Medienzugriff verwendet wird. Dafür können bis zu zehn Pikonetze zu sogenannten *Scatternetzen* aneinandergekoppelt werden, indem ein Slave Mitglied von zwei Pikonetzen wird. Abbildung 6.3 soll dies veranschaulichen, wobei "geparkte" (P, parked) Geräte passiv sind und nicht an der Kommunikation teilnehmen, sondern auf das Freiwerden anderer warten müssen.

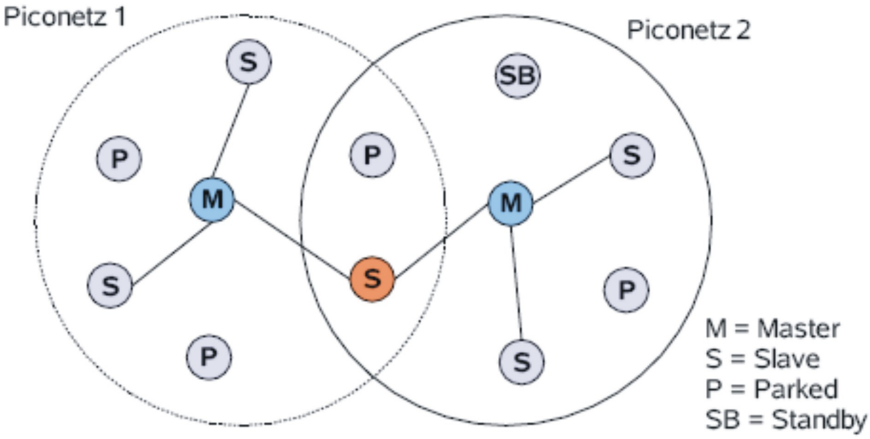

Abbildung 6.3: Scatternetz aus zwei Piconetzen [Hol03]

Die Anmeldung von neuen Geräten läuft dabei ganz schlicht ab. Eine Profilkonfiguration ist nicht notwendig, sie wird *Ad hoc* ausgeführt. Ein weiterer Vorteil dieser Funkschnittstelle ist, dass sie von dem

Frequency Hopping Spread Spectrum (FHSS) [Gm99] Gebrauch macht. Dabei werden 79 Kanäle im Bereich zwischen 2,402 und 2,480 GHz genutzt, von der jeder 1 MHz breit ist. Der Wechsel zwischen den Trägerfrequenzen findet bis zu 1600 mal pro Sekunde nach dem Zufallsprinzip statt, was zu einer hohen Robustheit gegenüber Störer und Abhörer führt. Bei der neuesten Version 2.0 sind Dank *Enhanced Data Rate (EDR)* Übertragungsgeschwindigkeiten bis zu 2,2 Mbit/s [Lis06] möglich, wobei für die Zukunft noch weitere Steigerungen erwartet werden. Auch der Marktanteil Bluetooth-fähiger Geräte wie Handys und PDAs ist in den letzten Jahren kontinuierlich gestiegen und illustriert die zunehmende Bedeutung dieses Übertragungsstandards im mobilen Alltag.

6.5 IEEE 802.11

Das *Institute of Electrical and Electronics Engineers (IEEE)* verabschiedete 1997 nach einer siebenjährigen Entwicklungsphase den Funkstandard 802.11 ([ELK06] [Dro04]), als nach einem drahtlosen Äquivalent zum *Ethernet* gesucht wurde und trug somit zur Popularität des *WLAN (Wireless Local Area Networks)* bei. Ebenso wie Bluetooth nutzt der Standard das lizenzfreie ISM-Frequenzband von 2400 bis 2483,5 MHz. Er orientiert sich an den zwei untersten Schichten des *OSI-Referenzmodells*, nämlich der *Bitübertragung* und *Sicherung* [Gie03].

Ursprünglich nutzte er neben Funk mit dem *Frequency Hopping Spread Spectrum (FHSS)* beziehungsweise *Direct Sequence Spread Spectrum (DSSS)* Verfahren auch Infrarot, was sich jedoch wegen der erforderlichen Sichtlinie als unbrauchbar erwiesen hat. Ein IEEE 802.11 Netzwerk kann dabei sowohl *Ad hoc*- als auch *Infrastrukturbasierend* aufgebaut sein, wobei letzteres die Existenz von mindestens einer Basisstation (*Access Point (AP)*) voraussetzt.

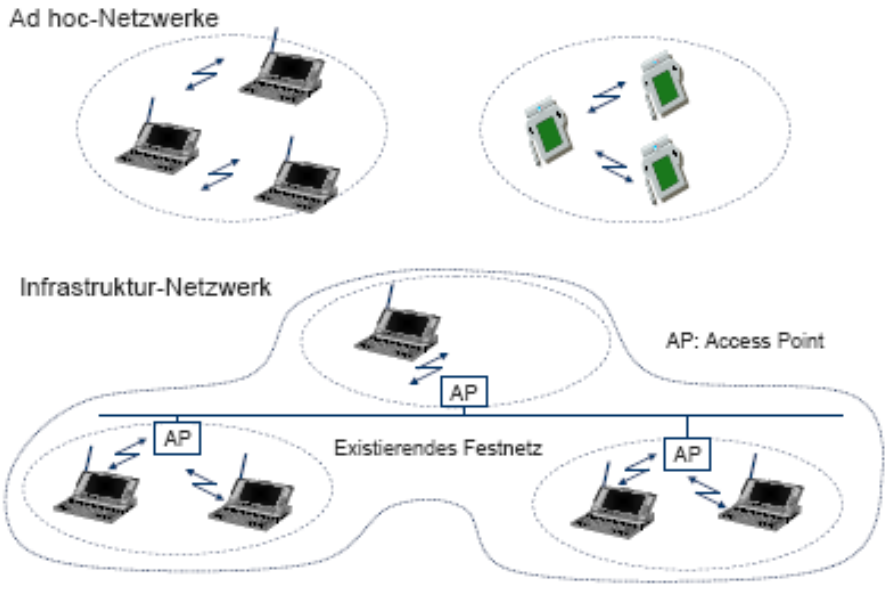

Abbildung 6.4: Netzwerkmodi [Dro04]

Nach anfänglichen Sendeleistungen von bis zu 2 Mbit/s waren zwei Jahre später durch die Einführung von 802.11a und 802.11b Übertragungsgeschwindigkeiten bis zu 54 beziehungsweise 11 Mbit/s möglich. Ersteres verwendet dabei ein neues Modulationsverfahren *Orthogonal Frequency Divison Multiplexing (OFDM)* und nutzt das 5 GHz-Band, was zwar im Gegensatz zum vielbenutzten ISM-Frequenzbereich relativ störungsfrei, jedoch in Deutschland auf eine Indoor-Nutzung beschränkt ist und dazu auch noch eine geringe Reichweite hat. Im Laufe der Zeit hat sich der Standard praktisch zu einem ”Buchstabenwett-

bewerb" entwickelt. Der bisher vorwiegend eingesetzte Standard 802.11b wurde vor einiger Zeit durch die im Juli 2003 entwickelte 802.11g-Norm abgelöst, welche jetzt auf dem ISM-Kanal Dank des OFDM-Verfahrens auch mit einer Geschwindigkeit von 54 Mbit/s Daten austauschen kann, eine größere Reichweite besitzt und trotzdem abwärtskompatibel zu seinem Vorgänger .11b ist. Einen Überblick über die einzelnen IEEE 802.11-Standards kann man der Abb. 6.5 entnehmen.

Standard	Beschreibung
802.11	Protokoll und Übertragungsverfahren für drahtlose Netze, 1997 zunächst nur für 2 MBit/s bei 2,4 GHz definiert.
802.11a	WLAN mit bis zu 54 MBit/s im 5 GHz Bereich, 12 nicht-überlappende Kanäle, Modulation: Orthogonal Frequency Division Multiplexing (OFDM).
802.11b	WLAN mit bis zu 11 MBit/s im 2,4GHz Bereich, 3 nicht-überlappende Kanäle.
802.11b+	WLAN mit bis zu 22 MBit/s im 2,4 GHz Bereich, Modulation: PBCC, Hardware basiert meist auf TI-ACX100 Chipset.
802.11c	Wireless Bridging zwischen Access Points. Spezifiziert das MAC-Layer-Bridging gemäß IEEE 802.1d.
802.11d	Beinhaltet länderspezifische Anpassungen an die jeweilige regulatorischen und gesetzlichen Bestimmungen, wie etwa die Wahl der Funkkanäle. Wurde zuerst für den US-Markt entwickelt. Es wurden mit dieser Erweiterung regionale Besonderheiten, z. B. auch der Frequenzbereich berücksichtigt.
802.11e	Erweitert WLAN um QoS (Quality of Service) - Priorisierung von Datenpaketen, z. B. für Multimedia-Anwendungen und Streaming.
802.11f	Regelt die Interoperabilität zwischen Basisstationen. Ermöglicht Roaming zwischen Access Points verschiedener Hersteller.
802.11g	54-Mbit/s-WLAN im 2,4-GHz-Band, Modulation OFDM.
802.11h	Ergänzungen zum 802.11a für Europa: DFS (Dynamic Frequency Selection) und TPC (Transmit Power Control).
802.11i	Verbesserung der Verschlüsselung: AES, 802.1x (Ergänzend/Aufbauend auf WEP und WPA).
802.11j	Japanische Variante von 802.11a für den Bereich 4,9 GHz - 5 GHz.
802.11k	Stellt Informationen über Funk- und Netzwerkaktivitäten zu Verfügung. Bessere Messung/Auswertung/Verwaltung der Funkparameter (z. B. Signalstärke). Soll unter anderem ortsbezogene Dienste ermöglichen (Location Based Services).
802.11m	Zusammenfassung früherer Ergänzungen, Bereinigung von Fehlern aus vorausgegangenen Spezifikationen (Maintenance).
802.11n	Geplante Erweiterung für ein zukünftiges, schnelleres WLAN mit 108 Mbit/s bis 320 MBit/s.
802.11o	Soll die Priorisierung von Sprache im WLAN gegenüber dem Datenverkehr definieren.
802.11p	Drahtloser Funkzugriff von Fahrzeugen aus.
802.11q	Unterstützt Virtual LANs (VLAN).
802.11r	Spezifiziert das Fast Roaming beim Wechsel zwischen Access Points. Interessant im Zusammenhang mit VoIP, um Gesprächsunterbrechungen zu vermeiden.
802.11s	Regelt den Aufbau von Wireless Mesh Networks.
802.11t	Wireless Performance Prediction (WPP), legt unter anderem Testverfahren fest.
802.11u	Behandelt das Zusammenspiel mit anderen nicht 802-konformen Netzen, wie etwa den zellularen Handy-Netzen.
802.11v	Wireless-Network-Management.

Abbildung 6.5: Übersicht aller IEEE 802.11-Standards (aus [ELK06])

6.6 NMEA 0183

Dieser Standard dient der Übermittlung von Positionsdaten von einem LPS zu einer Anwendung wie beispielsweise einem Personennavigationssystem. Er wurde in seiner aktuellen Form im Jahr 1983 von der amerikanischen Gesellschaft für Marineelektronik (*National Marine Electronics Association, NMEA*) ver-

abschiedet [Nme06].

NMEA Daten sind ASCII-Strings, die mit einem Dollarzeichen beginnen und mit einem *CR/LF* enden. Sie enthalten am Anfang eine Kennung (zum Beispiel GPRMC) und am Ende gewöhnlich eine Hexadezimalzahl als Prüfsumme. Es existieren mehrere solche Datensätze, wobei jedes NMEA-kompatible Gerät zumindest den *RMC (Recommended Minimum Specific)* beherrschen muss. Dessen Aufbau ist wie folgt [Nme06]:

$GPRMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,xxxx,x.x,a*hh, wobei

Satzfeld	**Bedeutung**
$GPRMC	Satzkennung
hhmmss.ss	Zeit (UTC)
A	Status (A=OK, V=Warnung)
llll.ll	Breite
a	N(orth) oder S(outh)
yyyyy.yy	Länge
a	E(ast) oder W(est)
x.x	Geschwindigkeit
x.x	Richtung in Grad
ddmmyy	Datum
x.x	Magnetische Abweichung
a	Richtung der magnetischen Abweichung: E(ast) oder W(est)
hh	Prüfsumme

Tabelle 6.1: Aufbau des GPMRC Datensatz

Beim vorliegenden System wurde aber in erster Linie der GGA-Datensatz benötigt, da dieser neben dem Längen- und Breitengrad auch die Höhe über dem Meeresspiegel beinhaltet. Die Verarbeitung dieser Datensätze wird in Kapitel 8 erläutert.

6.7 Grafische Ausgabe

Wie in Abschnitt 5.3 erläutert stehen drei verschiedene Optionen zur Verfügung, um Navigationsanweisungen grafisch auszugeben. Dabei handelt es sich um 2D sketches, 2D maps und 3D maps. In dieser Arbeit wurde aus folgenden Gründen eine Darstellung mit Hilfe von 2D sketches gewählt: zum Einen bieten diese eine klare und leicht verständliche Möglichkeit, Richtungsangaben darzustellen, ohne die bestehenden Ressourcen eines PDA übermäßig zu beanspruchen, wie es etwa bei 2D und 3D maps der Fall ist. Zum Anderen wären die dafür notwendigen Änderungen am originalen System so tiefgehend und umfassend gewesen, dass dieses teilweise praktisch von Grund auf neu konzipiert und implementiert werden musste. Dies war nicht Ziel dieser Diplomarbeit und auch im vorgegebenen Zeitrahmen nicht zu schaffen.

Verwendet wurden daher einfache grafische Symbole, wie sie auch in Kfz-Navigationssystemen zum Einsatz kommen und die für den Benutzer leicht verständlich sowie auf dem kleinen Display des PDA gut zu erkennen sind. Beispiele sind auf den Abbildungen 6.6 und 6.7. Da es nicht möglich war, den Navigationstext des Systems aus [Miz04] stets 1:1 in grafische Symbole umzusetzen, musste eine selektive Auswahl getroffen werden, wobei darauf zu achten war, dass keine wichtige Informationen verloren gingen. Die Anwendung enthält daher Navigationssymbole mit folgenden Bedeutungen: ”geradeaus”, ”rechts”, ”links”, ”gehe durch eine Türe”, ”benutze eine Treppe nach oben”, ”benutze eine Treppe nach unten”, ”fahre mit dem Lift hinauf zum Stockwerk X”, ”fahre mit dem Lift hinunter zum Stockwerk X”, ”Ziel erreicht”.

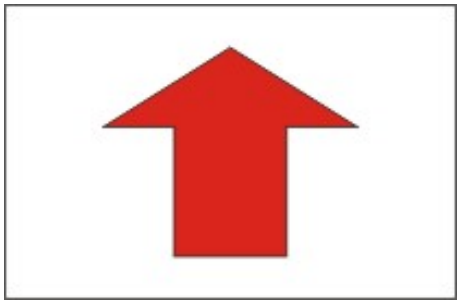

Abbildung 6.6: Symbol für ”Gehen Sie geradeaus”

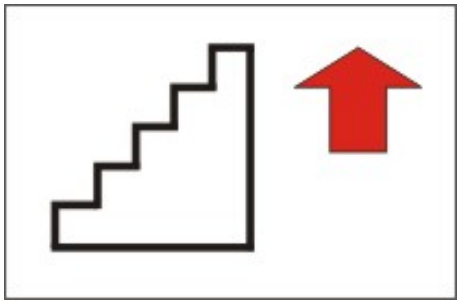

Abbildung 6.7: Symbol für ”Gehen Sie die Treppe hoch”

Kapitel 7

Entwurf des LMUNavAssist

Ein Ziel der vorliegenden Diplomarbeit war die Erweiterung des in [Miz04] vorgestellten Systems, so dass der Navigationsprozess mehrdimensional, also über mehrere Stockwerke hinweg, erfolgen konnte. Weiterhin sollte es von Textausgabe auf grafische Instruktionen umgestellt werden. Erwünscht wurde außerdem eine dynamische Berechnung des Weges sowie natürlich in erster Linie die Anbindung an ein Positionierungssystem. Um all diese Anforderungen zu erfüllen, war die Anwendung in mehreren Punkten zu modifizieren.

7.1 ObjectLibrary

Der vom System verwendete Stockwerksplan setzt sich aus den Klassen zusammen, die im Paket *ObjectLibrary* enthalten sind. Diese Komponente wurde im Abschnitt 3.3.1 bereits ausführlich beschrieben. Vereinfacht dargestellt besteht der Umgebungsplan aus einer Sammlung von *BaseObjects* (*ObjectsCollection*) und wird als Datei im XML-Format abgespeichert. Räume und Korridore werden als *BaseObjects*, genauer gesagt als Objekte vom Typ *Space*, repräsentiert. Sie können ihrerseits weitere Objekte enthalten wie beispielsweise Tische oder auch leer sein. Die Klasse *Line* (Türen) dient dazu, zwei verschiedene Räume miteinander zu verbinden.

Nachteilig bei dieser Implementierung war, dass das NAS in seiner ursprünglichen Form den Benutzer nur auf einer einzigen Stockwerkebene navigieren konnte, weil der Umgebungsplan konzeptionell nur aus einer Etage bestand. Außerdem gab es keine geeigneten Klassen, um zwei Stockwerke miteinander zu verknüpfen, was einen Ebenenwechsel unmöglich machte.

Um mit einen mehrstöckigen Umgebungsplan arbeiten zu können, musste im Paket *Collections* zuerst eine neue Klasse *EnvironmentCollection* implementiert werden, die sich am Vorbild von *ObjectsCollection* orientiert. Sie enthält eine Sammlung von *ObjectsCollection*-Objekte, also die einzelnen Stockwerkpläne.

Anschließend war der Wechsel zwischen den einzelnen Stockwerken zu ermöglichen. Zu diesem Zweck wurden die Klassen *Stair* und *Lift* als Verbindungselemente hinzugefügt. Wie die Klasse *Line* zwei verschiedene Objekte vom Typ *Space* derselben *ObjectsCollection* verbindet, sollten diese beiden neuen Klassen mindestens zwei verschiedene Spaces aus verschiedenen ObjectsCollections kombinieren. Die neue Klasse *Lift* arbeitet nach demselben Prinzip wie *Stair*, jedoch mit dem Unterschied, dass sie nicht nur zwei, sondern mehrere Stockwerke verbinden kann.

Die ursprüngliche Klasse *Line* wurde in *Door* umbenannt und zusammen mit *Stair* und *Lift* in dem neuen Paket *LevelLinks* abgelegt. Alle drei Klassen besitzen eine gemeinsame Oberklasse namens *LevelLink*.

Ebenfalls neu ist die Collectionsklasse *LinksCollection* im Paket *Collections*. Jedes *Space*-Objekt eines *Environments* enthält eine solche *LinksCollection*. In ihr sind alle Verbindungselemente, also Türen, Treppen

und Aufzüge, aufgelistet, die zu diesem Raum gehören. Diese Klasse spielt eine wichtige Rolle bei der Berechnung der Adjazenzliste und bei der Wegefindung. Das ursprüngliche System besaß an ihrer Stelle eine Klasse *LinesCollection*, die jedoch nur Türen (*Line*) berücksichtigt hat und aus diesem Grund nicht weiter verwendet werden konnte.

Abgesehen von diesen grundlegenden Änderungen waren auch bei den restlichen Klassen einige Anpassungen notwendig, die im folgenden Kapitel an passender Stelle besprochen werden. Das modifizierte *ObjectLibrary*-Paket ist in Abbildung 7.1 zu sehen, wobei die neuen Klassen eine grüne Farbe haben.

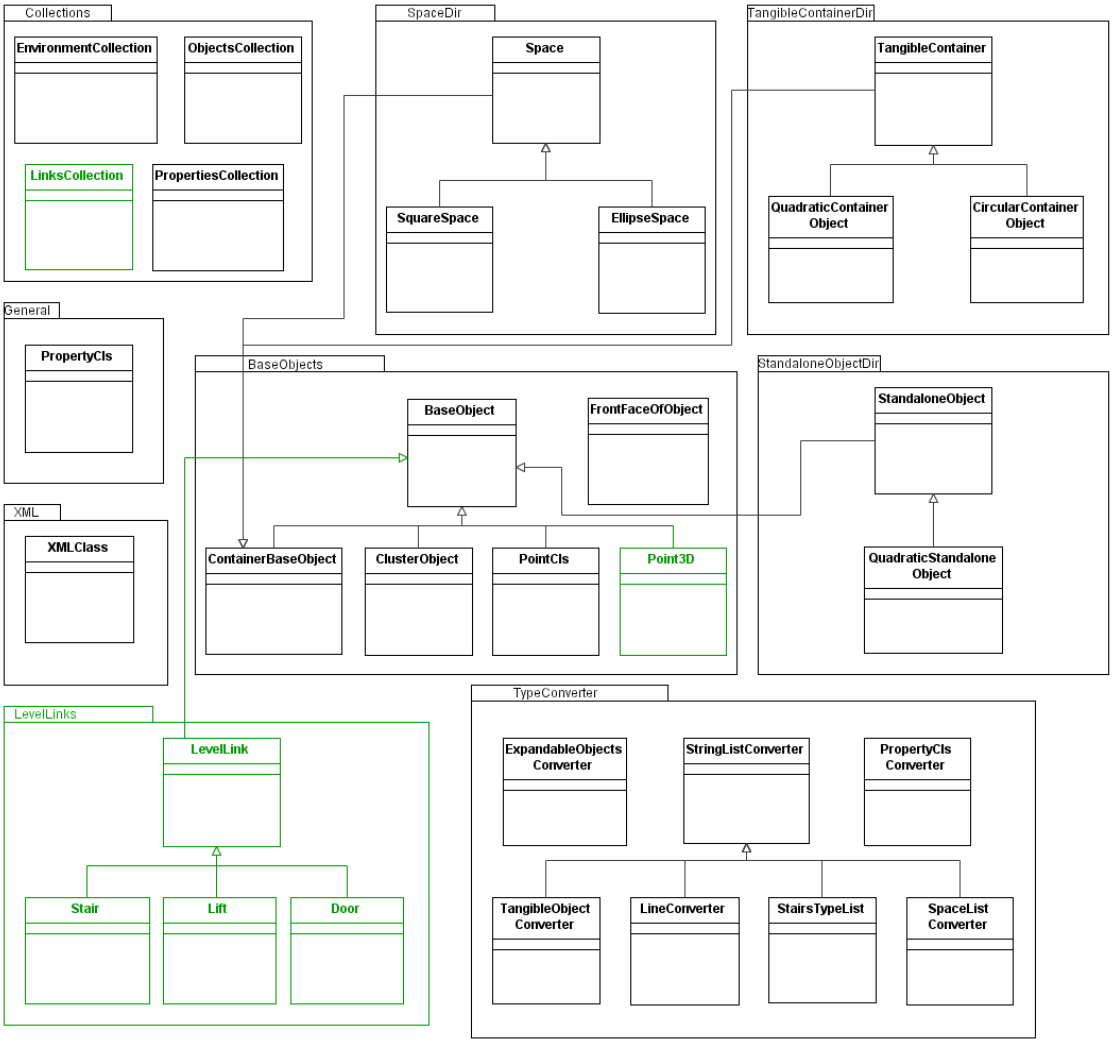

Abbildung 7.1: Modifiziertes ObjectLibrary Paket

7.2 Environment Builder

Diese Komponente dient zum Zeichnen und Anzeigen des Gebäudeplans. Nachdem die Datenstruktur des Umgebungsplans, wie im vorhergehenden Abschnitt besprochen, verändert wurde, waren auch bei der *Environment Builder* Anwendung einige Modifikationen erforderlich.

Der erste Schritt dabei war, die *Graphical User Interface (GUI)* an die neue Datenstruktur anzupassen, um die jeweiligen Objekte auch erwartungsgemäß anzuzeigen. Zusätzlich waren einige neue Funktionen zu

implementieren, die dem Systemadministrator die Möglichkeit geben, neue Ebenen erstellen und zwischen ihnen wechseln zu können. Zu diesem Zweck erhielt die Menüleiste eine ComboBox, in der alle vorhandenen Stockwerke aufgelistet sind, so dass der Anwender beim Zeichnen des Gebäudeplans zwischen diesen hin und her wechseln kann. Des Weiteren wurde eine DialogBox entworfen, um dem Administrator die Möglichkeit zu bieten, jeder Etage eine Bezeichnung zuzuweisen und dessen Höhe über dem Meeresspiegel einzugeben. Der Wert für letzteres lässt sich beispielsweise mit Hilfe eines GPS-Empfängers bestimmen. Die Gewichte für Treppen und Aufzüge sowie die beiden Referenzpunkte für die Koordinatentransformation lassen sich ebenfalls mit Hilfe von dieser GUI setzen. Auch hier kann man ein GPS-Gerät einsetzen, um die Längen- und Breitengrade dieser zwei Punkte zu erhalten. Sinnvolle Gewichte für die vertikalen Verbindungselemente können mit Hilfe der in Kapitel 6 erläuterten gefunden werden. Vorgehensweise Der komplette, modifizierte *Environment Builder* ist in Abbildung 7.2 zu sehen.

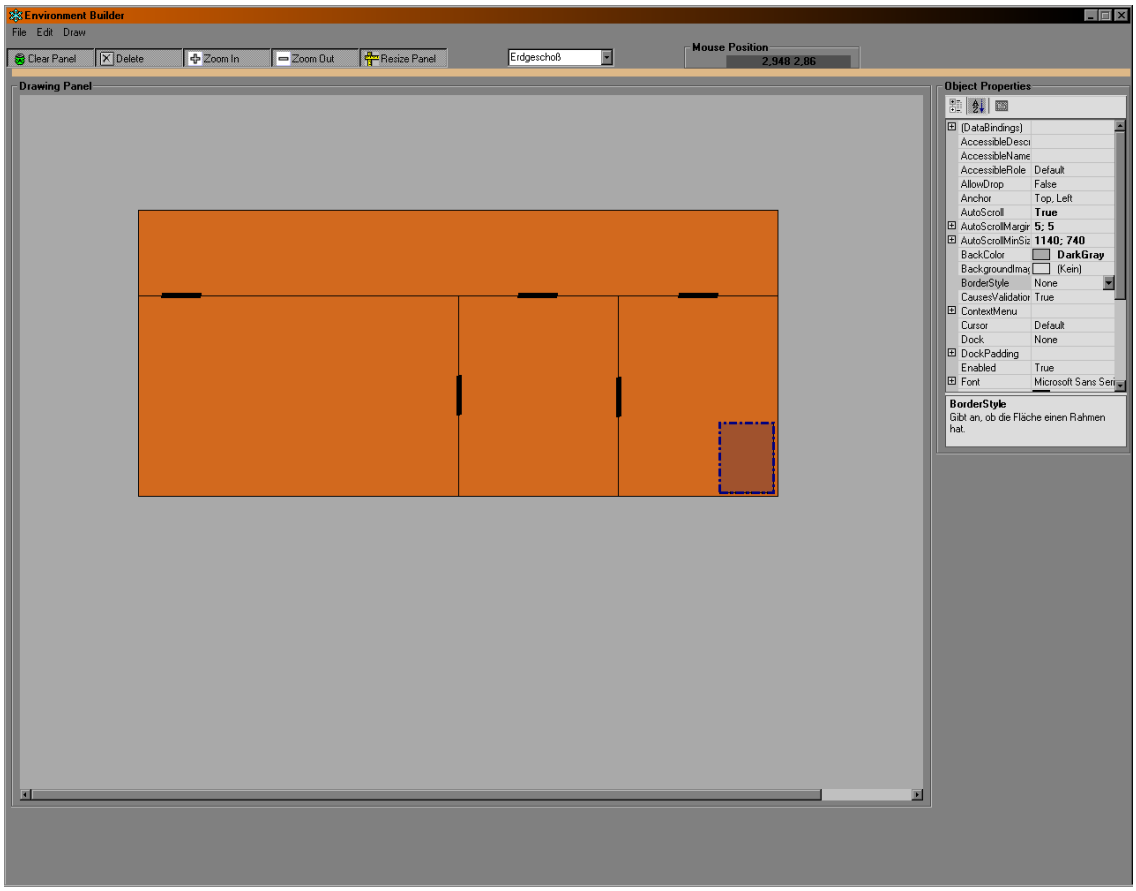

Abbildung 7.2: Modifizierter Environment Builder

7.3 Wegeberechnung

Für die Wegeberechnung sind die beiden Klassen *CellDecomp* und *SearchCls* verantwortlich. *CellDecomp* zerlegt jeden Raum mit Hilfe der *Approximate Cell Decomposition* Technik (vgl. Abschnitt 3.2.1) in verschiedene Typen von Zellen: leere, teilweise gefüllte (hybride) oder volle. *SearchCls* erzeugt danach einen Erreichbarkeitsgraph und berechnet den kürzesten Weg vom Start- bis zum Zielpunkt unter Verwendung des A*-Algorithmus.

Es hat sich beim Testen mit eigenen Plänen gezeigt, dass die Navigationsanweisungen ziemlich ungenau oder sogar falsch sind, wenn die Umgebung großflächige und leere Räume enthält. Der Grund dafür ist,

dass *CellDecomp* leere Räume überhaupt nicht dekomponiert. Da das System als mögliche Benutzerpositionen nur die Mittelpunkte von Zellen akzeptiert, führen große Zellen zu einer starken Abweichung vom tatsächlichen Benutzerstandort und damit zu einer schlechten Navigationsleistung. Dies war in der originalen Arbeit vermutlich kein Problem und dürfte wahrscheinlich auch nicht aufgefallen sein, weil das System zum Auffinden von Büchern in einer Bibliothek verwendet wurde und in einer solchen Testumgebung keine großen, leeren Räume existieren. Da bei der Anpassung an das Institut für Informatik der LMU München jedoch dieser Effekt häufig auftrat, war dies nicht akzeptabel. Um hier Abhilfe zu schaffen, wurden Zellen solange gesplittet, bis ihre Seitenlänge kleiner als zwei Meter war.

Als Ergebnis liefert das obige Verfahren einen Erreichbarkeitsgraph, in dem alle zusammenhängenden Kanten enthalten sind. Eine Kante verbindet die Mittelpunkte zweier Zellen, genau dann wenn sie benachbart und voneinander aus erreichbar sind. Im Gegensatz zum alten System ist der Erreichbarkeitsgraph nun nicht länger flach, sondern mehrdimensional.

Wie in [Miu02] festgestellt, ist der ursprünglich verwendete A*-Algorithmus aufgrund seiner Heuristik für den Einsatz im mehrdimensionalen Raum nicht optimal. Daher wurde er in der vorliegenden Arbeit durch den im Abschnitt 6.1 beschriebenen Algorithmus von Dijkstra ersetzt.

7.4 Erzeugung und Ausgabe von Navigationsanweisungen

Bei der Umstellung auf eine grafische Ausgabe der Benutzerinstruktionen fanden die Module des originalen Systems, in leicht modifizierter Form Verwendung, eine Neuentwicklung war aus Zeitgründen nicht möglich. Wie in Abschnitt 3.3.3 beschrieben, werden die Textanweisungen im *Generating Instructions Component* in mehreren Schritten erzeugt.

Zuerst generiert der *Text Planner* stichpunktartige Richtungsangaben wie beispielsweise ”turn to right”. Um dieses Modul weiterverwenden zu können, war ein ergänzender Code für die Erkennung notwendig, ob der Weg nach oben oder unten führt und dabei eine Treppe oder ein Lift benutzt wird.

Während der *Segmentation Process* (Vergleich Abbildung 3.6) nicht mehr nötig war und unangetastet blieb, wurde die nächste Stufe, *Linguistic Realisation*, ebenfalls an die neuen Navigationsinstruktionen (d.h. Benutzung von Lift und Treppe) angepasst. Obwohl nicht eigentlich Gegenstand dieser Arbeit, geschahen diese Modifikationen, um testen zu können, ob die Wegbeschreibung auch in diesem Fall noch korrekt ist. Es hat sich dabei gezeigt, dass es bei der Umsetzung des Pfades in Richtungsangaben in einzelnen Fällen in komplex konstruierten Gebäuden zu einer Verwechslung von rechts und links kommt. Dieses Problem ist bereits in der originalen Arbeit vorhanden und darauf zurückzuführen, dass der als nächstes einzuschlagende Weg, wie in Abschnitt 3.2.1 erwähnt und Kapitel 10 nochmals ausführlich erläutert, von Tür zu Tür berechnet wird.

Für die Umsetzung in grafische Anweisungen war dieses Modul aber ebenso nicht notwendig und konnte damit entfallen. Stattdessen wurde der *Text Planner* entsprechend angepasst, so dass er die Navigationsinstruktionen sofort in Form von Steuerbefehlen für den Client generiert. Zuletzt musste nur noch der Client so modifiziert werden, dass er die zugehörigen grafischen Symbole auf dem Display des PDA anzeigen kann.

Dabei wurde ein zweites Problem offensichtlich: beim Systemstart ist die Blickrichtung des Benutzers nicht bekannt, so dass das System nicht wissen kann, mit welcher Richtungsangabe es beginnen soll. Dasselbe passiert unter Umständen auch beim Wechsel von Stockwerken. So führen beispielsweise bogenförmig verlaufende Treppen zu einer Drehung des Anwenders, die für das System unbemerkt bleibt und somit zu einem Verlust der korrekten Sichtperspektive führt. Auch im Falle eines Aufzugs mit zwei Ein- und Ausgängen kann dieses Problem auftreten.

7.5 Entwurf der Schnittstelle

Da in der Arbeit von [Miz04] ein Lokalisierungssystem nur simuliert wurde, sollten hier die Grundlagen geschaffen werden, um ein externes LPS anbinden zu können. Ein solches System für Indoor-Umgebungen wurde in einer parallel laufenden Diplomarbeit [Ric06] entwickelt. Da es erwünscht war, das NAS zusammen mit einem handelsüblichen GPS-Empfänger oder einem vergleichbaren Ortungssystem einsetzen zu können, kamen für die Anbindung nur Bluetooth und als Datenformat NMEA 0183 in Frage. Dieses Format wurde bereits in Abschnitt 6.6 beschrieben.

Im Gegensatz zum originalen System ist die Anbindung ausschließlich auf Clientseite realisiert. Der Grund für diese Entscheidung war, dass das LPS sich sowieso in unmittelbarer Nähe des mobilen Client befinden muss und aufgrund der beschränkten Reichweite von Bluetooth-Verbindungen ein direkter Anschluss an den stationären Server wenig sinnvoll war.

Dem Client fällt somit die Aufgabe zu, die aktuelle Position des Benutzers in regelmäßigen Abständen beim LPS abzufragen und das Ergebnis an den Server zu übermitteln. Dieser kann aus den NMEA-Strings anschließend die benötigten Daten, nämlich Längengrad, Breitengrad und Höhe extrahieren. Bevor diese für die Berechnung verwendet werden, müssen sie allerdings noch in Pixelkoordinaten umgewandelt werden. Dafür wird das in Abschnitt 6.3 beschriebene Verfahren der Koordinatentransformation eingesetzt.

Kapitel 8

Implementierung

Für die Implementierung der Anwendung kam, wie in der originalen Arbeit auch, das *Microsoft .NET Framework* mit der Programmiersprache C# zur Verwendung. Als passende Entwicklungsumgebung diente *Microsoft Visual Studio .NET 2003* und für die grafischen Komponenten konnte auf die *GDI+* Bibliothek von Windows zurückgegriffen werden. Die Implementierung der Bluetooth-Anbindung schließlich erfolgte mit C++, da C# keinen direkten Zugriff auf die entsprechenden *APIs* und Systemcalls gestattet. Zusätzlich wurde für die Bluetooth-Funktionen noch das *Microsoft Platform SDK* benötigt, das unter [MSP06] zum freien Download verfügbar ist. Einen Überblick über die *Microsoft .NET* Technologie gibt [Dot06].

8.1 Object Library

Bei dieser Komponente wurde die ursprüngliche Implementierung in drei Punkten modifiziert. Erstens musste eine Collectionsklasse hinzugefügt werden, die alle Ebenen des Gebäudeplanes aufnehmen kann. Zweite Aufgabe war es, alle Objekte des Umgebungsplans mit einem Attribut für die Höhe zu versehen, um die Datenstruktur an einen dreidimensionalen Raum anzupassen. Und zuletzt sind zusätzliche Verbindungselemente zwischen den einzelnen Stockwerken notwendig gewesen. Diese grundlegenden Veränderungen wurden bereits in Abschnitt 7.1 dahergelegt. An dieser Stelle soll noch auf die Details der Implementierung eingegangen werden.

8.1.1 EnvironmentCollection

Der neue mehrstöckige Umgebungsplan wird in der Collectionsklasse *EnvironmentCollection*, zu sehen in Abbildung 8.1, gespeichert. Diese wurde nach dem Vorbild von *ObjectsCollection* entworfen und konsequenterweise im selben Paket (*Collections*) abgelegt.

Die ersten vier Deklarationen enthalten die geografischen Koordinaten der beiden Referenzpunkte, wobei jeweils die x-Achse den Breitengrad und die y-Achse den Längengrad entspricht. Die Koordinaten im *Greenwich-Format* (zum Beispiel 48^o08'2400N für den Längengrad von München-Schwabing) werden dabei als Zahl vom Typ *double* dargestellt, und zwar in der Form 4808,2400. Negative Werte implizieren hier südliche Längengrade beziehungsweise westliche Breitengrade. *pixelPt1* und *pixelPt2* repräsentieren die Position der Referenzpunkte auf der gezeichneten Karte, gemessen in Pixel. Die Koordinaten dieser beiden Punkte sind notwendig, um eine Ortsangabe im Greenwich-System in das bei der Wegberechnung verwendete Pixelformat zu transformieren (vergleiche Abschnitt 6.3).

Damit die für die Bestimmung des kürzesten Pfades benötigten Kantengewichte der *Lift-* und *Stair-*Elemente zusammen mit der Karte gespeichert können, musste in *EnvironmentCollection* noch zwei zusätz-

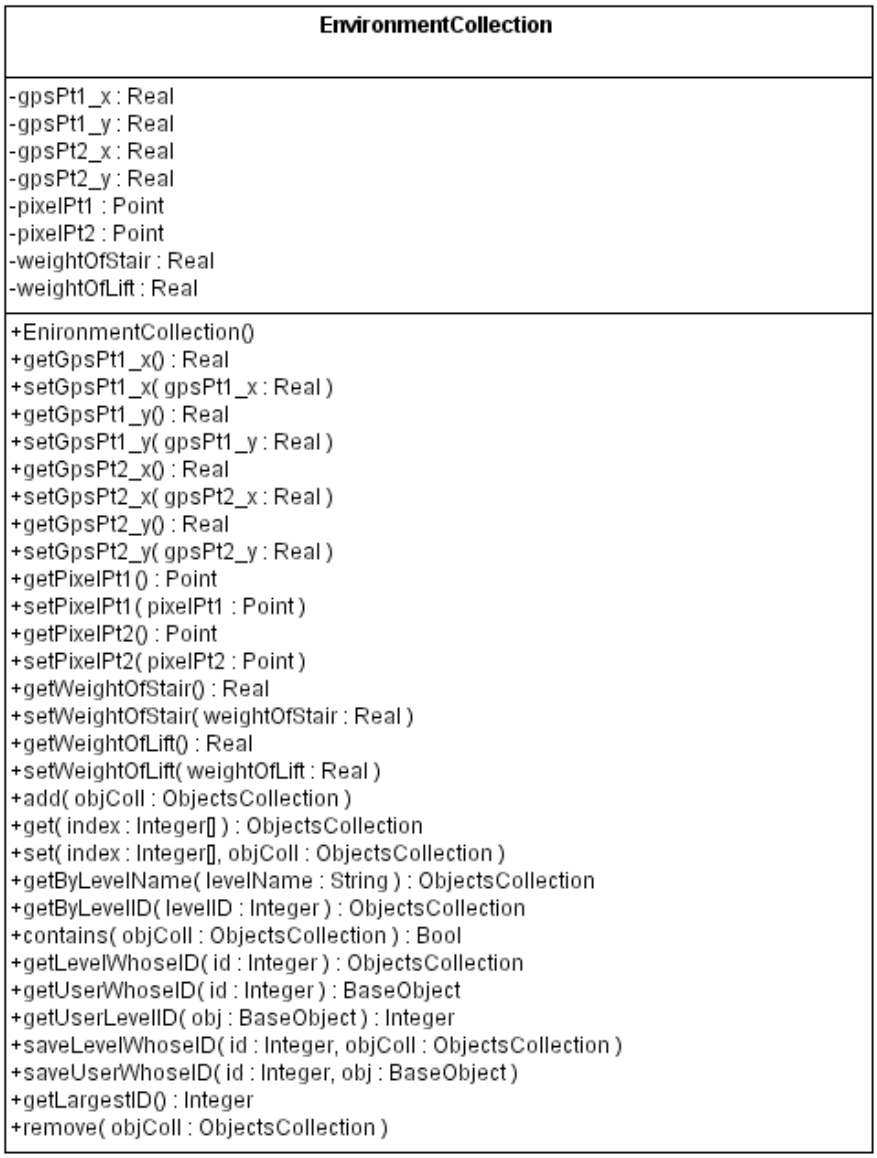

Abbildung 8.1: Neue EnvironmentCollection Klasse

liche Attribute eingefügt werden, nämlich *weightOfLift* und *weightOfStair*. In beiden Fällen fand der Typ *double* Verwendung, da diese zusätzliche Genauigkeit bei der Berechnung benötigt wurde.

8.1.2 ObjectsCollection

Die in der *EnvironmentCollection* enthaltenen einzelnen Stockwerke sind selbst wieder eine Collectionsklasse, nämlich die bereits bekannte *ObjectsCollection*. Diese besteht aus einer Sammlung der grundlegenden Basisobjekte (*BaseObjects*), welche die Räume und deren Inhalt (Stühle, Tische, Bücher etc.) bezeichnen. Da nun im Datenraum mehr als eine *ObjectsCollection* vorhanden sein kann, mussten diese unterscheidbar gemacht werden.

Die Deklarationen wurden deshalb um *levelId* und *levelName* erweitert, welche für den jeweiligen Stock-

werkidentifikator beziehungsweise für den Namen der Etage stehen. Ferner gibt das boolesche Attribut *saved* an, ob das aktuelle Level seit seiner letzten Speicherung verändert worden ist. Die überarbeitete *ObjectsCollection* wird in Abbildung 8.2 dargestellt.

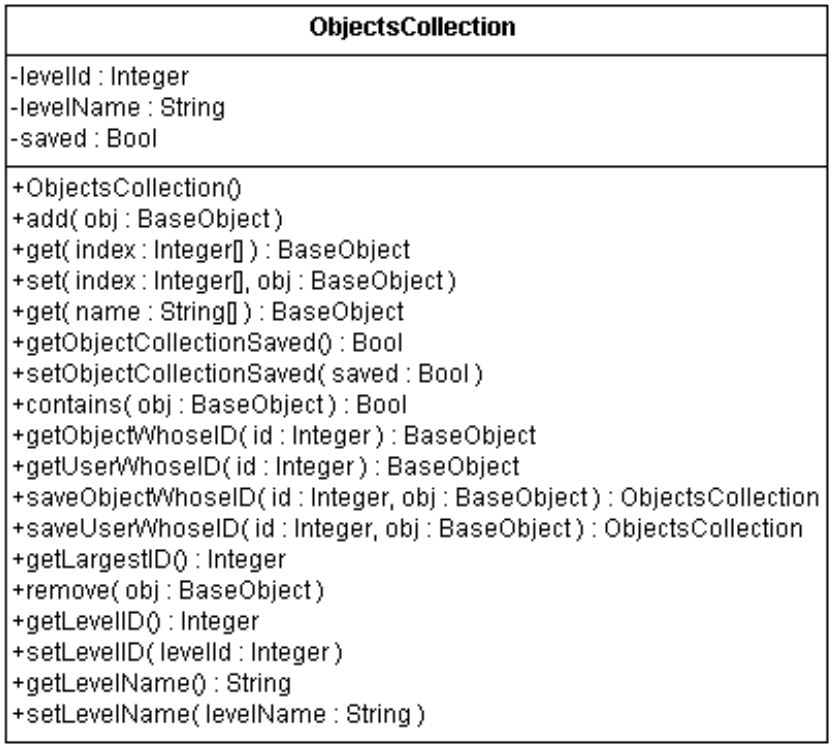

Abbildung 8.2: Modifizierte ObjectsCollection

8.1.3 BaseObject

Bei der Frage nach der Unterbringung des Attributs für die z-Achse fiel die Wahl auf die Klasse *BaseObject*, da diese bereits die x- und y-Koordinaten enthielt. Da *BaseObject* die Oberklasse der Datenstruktur ist, erben ihre Unterklassen folglich auch dieses neue Attribut. Als mögliche Alternative für das Einbinden der z-Koordinate hätte sich noch *ObjectsCollection* angeboten. Im Hinblick auf eine bessere zukünftige Erweiterbarkeit wurde jedoch die erste Lösung bevorzugt.

Die überarbeitete Klasse *BaseObject* enthält somit folgende neue Attribute:

- objectZ (Typ *double*): gibt die Höhe über den Meeresspiegel an.
- levelId (Typ *int*): gibt den Stockwerkidentifikator an.

Als Datentyp für das Attribut, das den Wert für die dritte Dimension bezeichnet, musste statt *float double* verwendet werden, weil die entsprechende C#-Methode beim Speichern im XML-Format sonst zu viele Nachkommastellen weggeschnitten hätte. Ferner wurde das Attribut *levelId* deklariert, welches aussagt, zu welchem Stockwerk das betreffende Objekt gehört.

8.1.4 LevelLink

Um bei der Wegberechnung von Raum zu Raum gehen zu können, mussten die entsprechenden *Space*-Objekte geeignet miteinander verknüpft werden. Dazu dienen drei Klassen, nämlich *Lift*, *Stair* und *Door*, die den realen Bauelementen Aufzug, Treppe beziehungsweise Türe entsprechen. Erstere beiden verbinden zwei oder mehr Räume, die sich auf verschiedenen Etagen (d.h. unterschiedlichen *ObjectsCollections*) befinden. Im Gegensatz dazu müssen sich im Falle einer Türe die beiden *Space*-Objekte, die miteinander kombiniert werden sollen, natürlich auf derselben Ebene liegen.

Alle drei erwähnten Klassen haben eine Reihe von Eigenschaften gemeinsam und die Implementierung dieser Methoden und Attribute fand in der Oberklasse *LevelLink* statt. Diese wurde zusammen mit seinen drei Unterklassen im neuen Paket *LevelLinks* untergebracht. Abbildung 8.3 stellt dieses Paket dar.

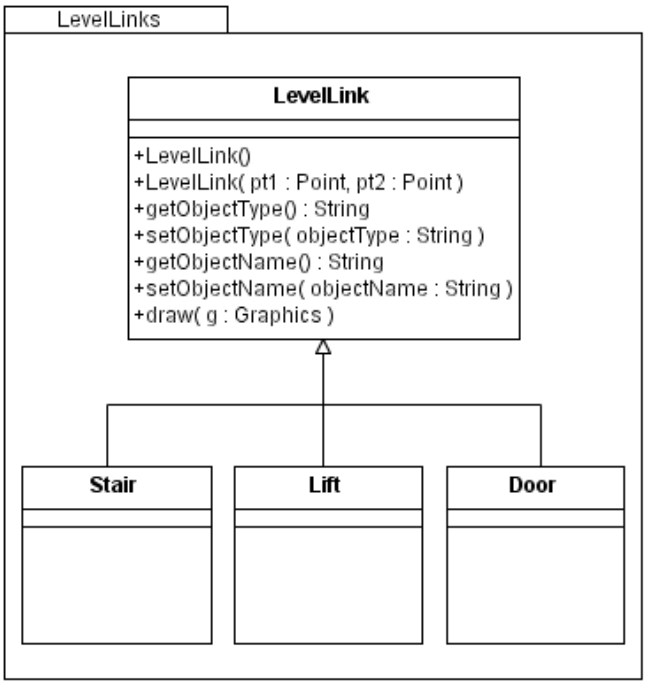

Abbildung 8.3: Neues Paket LevelLinks

Stair

Bei *Stair* handelt es sich um einen der beiden neuen Klassen, die zu dem Zweck implementiert wurden, Raumobjekte aus unterschiedlichen Etagen *(ObjectCollections)* miteinander zu verbinden und damit eine Wegberechnung über die Stockwerkgrenzen hinaus zu ermöglichen. Abbildung 8.4 zeigt diese Klasse.

Wie auch auf dem UML-Diagramm ersichtlich ist, besitzt *Stair* sechs Deklarationen:

- lowerRoom (Typ *Space*): Zeiger auf den darunter liegenden Raum.
- upperRoom (Typ *Space*): Zeiger auf den darüber liegenden Raum.
- lowerRoomId (Typ *int*): der Identifikator des darunter liegenden Raumes.
- upperRoomId (Typ *int*): der Identifikator des darüber liegenden Raumes.
- lowerLevelId (Typ *int*): der Identifikator der darunter liegenden Etage.

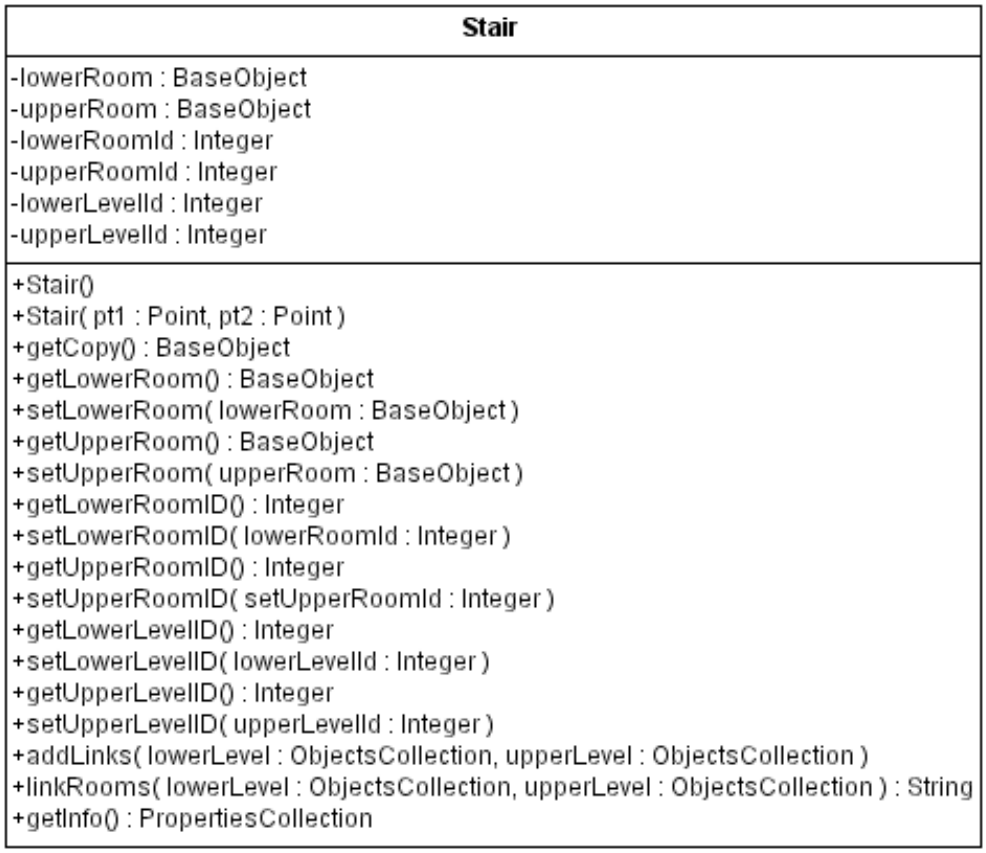

Abbildung 8.4: Neue Klasse Stair

- upperLevelId (Typ *int*): der Identifikator der darüber liegenden Etage.

Ferner sollen noch zwei Methoden dieser Klasse kurz besprochen werden: *setRooms()* und *fillRooms()* (vergleiche Anhang B.1). Aufgabe der ersten Operation ist es, festzustellen, welche zwei *Space*-Objekte des Gebäudes von dieser Treppe geschnitten werden und den eingangs aufgelisteten Attributen anschließend die korrekten Werte zuzuweisen.

Dabei wird zuerst geprüft, welcher Raum den Mittelpunkt dieser Treppe enthält, indem man alle Objekte der *EnvironmentCollection*, zu der die Treppe gehört, nacheinander durchläuft. Findet sich ein solcher Raum, wird der Zeiger entsprechend auf diesen gesetzt und *lowerRoomId* sowie *lowerLevelId* erhalten die zugehörigen Werte. Zuvor ist allerdings noch zu prüfen, ob diesen Attributen bereits in der Vergangenheit etwas zugewiesen wurde. Ein solcher Fall kann beispielsweise eintreten, wenn man die Treppe nachträglich verschoben hat. Dann muss aus der *LinksCollection* des vorhergehenden *Space*-Objekts der alte Verweis auf die Treppe gelöscht und anschließend der aktuell gefundene Raum eingetragen werden. Das Auffinden des entsprechenden oberen Raumes läuft genau analog ab. Schneidet die Treppe keinen Raum, bekommt sie stattdessen einen *Nullpointer* eingetragen und der Identifikator erhält den Wert *-255*. Die Methode *setRooms()* wird jedes Mal aufgerufen, wenn der Benutzer eine Treppe zeichnet oder den Gebäudeplan abspeichert.

Für den umgekehrten Fall, nämlich beim Laden der Umgebungskarte, wird die Operation *fillRooms()* benötigt. Da *Pointer* auf Objekte nicht statisch sind und somit nicht abgespeichert werden können, hat man die ursprüngliche Verzeigerung der Datenstruktur nach dem Laden der Datei wieder zu rekonstruieren. *fillRooms()* sucht also diejenigen Objekte, welche die entsprechenden Level- und Raumidentifikatoren besitzen und trägt sie bei *lowerRoom* und *upperRoom* ein.

Lift

Die zweite Verbindungsklasse mit dem Namen *Lift* repräsentiert, wie der Bezeichner schon sagt, einen Aufzug und dient dazu, nicht nur zwei, sondern mehrere Räume aus unterschiedlichen Stockwerken zu verknüpfen. Die Zeiger auf die einzelnen *Space*-Objekte werden in einer *ObjectsCollection* mit dem Namen *roomColl* abgelegt. Weiterhin existieren zwei ArrayLists, *roomIdArr* und *levelIdArr*, welche die zugehörigen Identifikatoren der Räume und Ebenen beinhalten. Das letzte Attribut, *maxLevelId*, gibt an, auf welcher Etage der Aufzug endet. Initialisiert wird dieser Parameter mit dem Integerwert ”999”, somit muss der Administrator ihn nur in dem Fall verändern, wenn der Lift nicht bis zum obersten Stockwerk geht. Die unterste Haltestelle des Aufzugs braucht hingegen nicht angegeben zu werden. Diese ergibt sich nämlich automatisch aus dem Level, auf dem der Lift im *EnvironmentBuilder* gezeichnet wurde.

Wie bei der Klasse *Stair* sind auch in *Lift* die beiden Operationen *setRooms()* und *fillRooms()* enthalten. Ihre Aufgabe und Funktionsweise ist dieselbe wie die im vorhergehenden Abschnitt beschrieben, allerdings arbeiten sie auf mehr als nur zwei Etagen. Abbildung 8.5 zeigt das zugehörige Klassendiagramm.

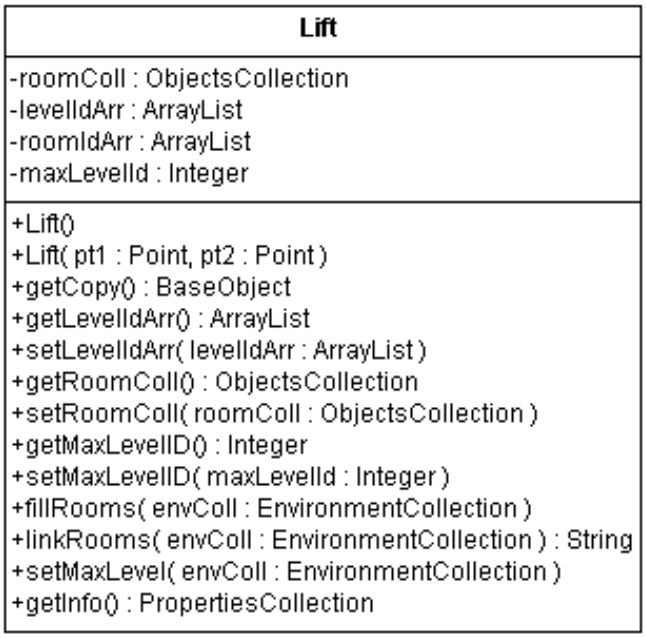

Abbildung 8.5: Neue Klasse Lift

Door

Diese leicht modifizierte Version der ursprünglichen Klasse *Line* verbindet ebenfalls zwei *Space*-Objekte, die aber - im Gegensatz zu den vorher besprochenen Klassen - auf derselben Ebene liegen müssen. Sie repräsentiert entweder eine Türe oder eine Öffnung (”opening”) zwischen zwei Räumen.

8.1.5 LinksCollection

Für das überarbeitete System wurde außerdem eine weitere Collectionsklasse (*LinksCollection*) für die Aufnahme von *LevelLink*-Objekte benötigt (Abb. 8.6). Diese ersetzt die alte *LinesCollection* und enthält nur eine Liste mit *Door*-, *Stair*- und *Lift*-Elementen. Wie in Abbildung 7.1 zu sehen war, wurde diese Klasse dem Paket *Collections* zugewiesen.

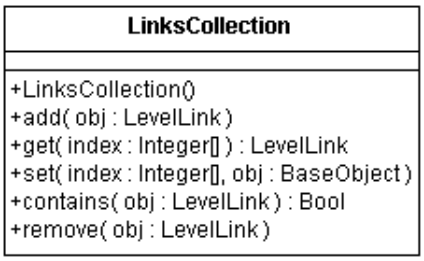

Abbildung 8.6: Neue Collectionsklasse LinksCollection

8.1.6 XMLClass

Um den modifizierten, mehrstöckigen Gebäudeplan speichern und laden zu können, war es notwendig, die *XMLClass* entsprechend anzupassen. Im Einzelnen musste zusätzlicher Code für die Verarbeitung der neuen Klassen (im Wesentlichen *Stair* und *Lift*) implementiert werden. Ferner war es selbstverständlich erforderlich, dass diese Klasse eine komplette *EnvironmentCollection* als Datei im XML-Format abspeichern und später auch wieder lesen kann.

In ihrer ursprünglichen Form hat *XMLClass* nur den Umgang mit *ObjectsCollection* beherrscht. Das Implementieren der notwendigen, bereits erwähnten Anpassungen, gestaltete sich glücklicherweise nicht übermäßig aufwändig und soll daher an dieser Stelle nicht im Detail besprochen werden. Zu erwähnen wäre jedoch noch, dass bei dieser Gelegenheit auch ein Bug beseitigt wurde, der zum Absturz des Systems geführt hat, wenn eine Türe keinen Kontakt zu einem zweiten Raum hatte (vergleiche Kapitel 10).

8.2 EnvironmentBuilder

Die Anwendung *EnvBuilder* enthält die Klassen *GISForm*, *ObjectsFrm*, *PanelCtrl*, *PanelDimFrm*, *propertiesFrm*, *xmlTaggingCls*, *xmlTaggingFrm*, wobei der größte Teil seiner Funktionalität in der Klasse *GISForm* implementiert ist. Folglich war auch nur diese von Veränderungen betroffen.

Nach der Implementierung der bereits in Kapitel 7 aufgeführten Modifikationen enthält die Klasse *GISForm* nun die folgenden neuen Attribute:

- count (Typ *int*): der Index des obersten Levels, startet bei 0.
- envColl (Typ *EnvironmentCollection*): enthält den kompletten Gebäudeplan.
- refPt1 (Typ *boolean*): gibt an, ob der erste Referenzpunkt gesetzt werden soll.
- refPt2 (Typ *boolean*): gibt an, ob der zweite Referenzpunkt gesetzt werden soll.

Neu hinzugekommen sind somit *count*, *envColl*, *refPt1* und *refPt2*. Erstere gibt die Nummer des obersten Levels an, wobei berücksichtigt werden muss, dass die Zählung bei 0 beginnt. In *envColl* ist der aktuelle Umgebungsplan abgelegt, während *refPt1* und *refPt2* in dem Moment den Wert *true* erhalten, wenn das System vom Anwender die Eingabe beider Referenzpunkte erwartet.

Zum Setzen dieser Punkte war eine neue Klasse (*SetRefPtsFrm*) hinzuzufügen, die eine Dialogbox (Abb. 8.7) auf dem Bildschirm erzeugt, in welche der Anwender deren Koordinaten in Form von Längen- und Breitengrad eingeben kann. Erzeugt wird diese Klasse durch die Methoden *setRefPt1MItem_Click()* und *setRefPt2MItem_Click()* in *GISForm*, wobei gleichzeitig auch *refPt1* beziehungsweise *refPt2* auf *true* gesetzt werden.

Ebenfalls dort wurde die Methode *setLevelPropMItem_Click()* eingebettet, welche ein Dialogfenster erzeugt, das zum Eintragen der einzelnen Stockwerkbezeichnungen sowie deren Höhe über dem Meeres-

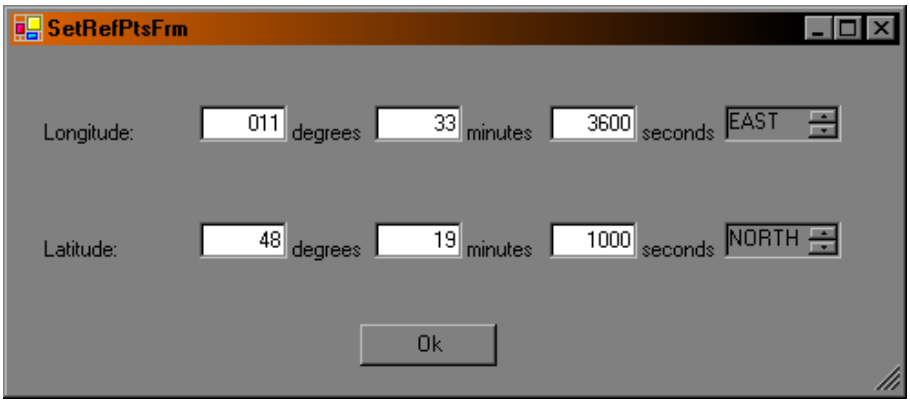

Abbildung 8.7: Neue DialogBox SetRefPtsFrm

spiegel dient. Bereitgestellt wird dieses Fenster durch die Klasse *SetLevelProperties*, welche in Abbildung 8.8 zu sehen ist.

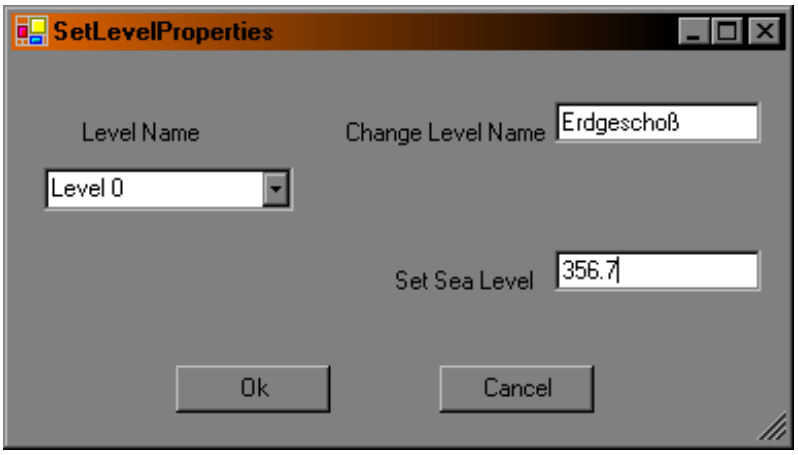

Abbildung 8.8: Neue DialogBox SetLevelProperties

Auch die Kantengewichte, die bei der Berechnung des kürzesten Pfades für Treppen und Aufzüge Verwendung finden, kann der Anwender im *EnvironmentBuilder* eingeben. Dies ermöglichen zwei neue Klassen *SetStairWeight* und *SetLiftWeight*, die ihn durch eine entsprechende Dialogbox (Abb. 8.9) dazu auffordern.

Abbildung 8.9: Neue DialogBox SetStairWeight

8.3 Wegeberechnung

8.3.1 Zelldekomposition

Das erste Ziel, welches bei der Anpassung der Wegberechnung an die neuen Anforderungen erreicht werden musste, war die feinere Zerlegung der großen Räume in einzelne Zellen. Wie bereits im vorherigen Kapitel erwähnt, wurden während der Zelldekomposition leere Räume unabhängig von ihrer Größe nicht weiter gesplittet. Um nun auch in solchen Fällen einen Split zu erzwingen, war in der Klasse *cellDecomp* die Bedingung

```
if (cellClass2.Flag.Equals("mixed"))
```

wie folgt abzuändern:

```
if (cellClass2.Flag.Equals("mixed") || cellClass2.Flag.Equals("empty"))
```

Dies führte zu einem rekursiven Split von leeren (*empty*) Zellen bis zu einer Mindestgröße von zwei Metern. Hierfür sind in der Methode *decomposeCell()* folgende Anweisungen verantwortlich:

```
if (cellClass.ObjectWidthCm > 2)
```

sowie

```
if (cellClass.ObjectHeightCm > 2)
```

8.3.2 Erreichbarkeitsgraph

Anschließend musste in dieser Klasse noch die für die Berechnung der Adjazenzliste verantwortliche Methode *adjacentToCellsFromDifferentSpace()* (vgl. Anhang B.2) so angepasst werden, dass auch Zellen, die zu verschiedenen Stockwerken gehören, Berücksichtigung finden, falls zwischen ihnen eine Verbindung in Form einer Treppe oder eines Lifts existiert. Diese Methode betrachtet alle *LevelLink*-Objekte, die in der *EnvironmentCollection* enthalten sind. Sie sucht die *Space*-Objekte, die von dem aktuell bearbeiteten *LevelLink* verbunden werden und durchläuft in zwei oder mehr Schleifen alle zum Raum gehörenden Zellen (*CellCls*). Während diesen Durchlaufs identifiziert sie nun diejenigen *CellCls* aus beiden (oder im Falle eines Aufzugs mehreren) Räumen, welche dieses *LevelLink*-Objekt enthalten und somit wirklich benachbart sind. Diese sind dann in die Adjazenzliste einzufügen.

Zusätzlich waren alle Kanten des Erreichbarkeitsgraphen mit einem Gewicht zu versehen. Repräsentiert werden diese Kanten durch Objekte vom Typ *PathClass*. Somit war es notwendig, in dieser Klasse ein neues Attribut für die Aufnahme des Kantengewichts hinzuzufügen. Dieser Parameter (*weight*) erhält während der Generierung des Graphen in der Methode *computeEdges()* korrekte Werte, wobei an dieser Stelle auch die passende Mali für Treppe und Lift zu berücksichtigen sind.

8.3.3 Dijkstra-Algorithmus

Die modifizierte Wegeberechnung wurde in der Klasse *SearchCls* implementiert und zwar in der Methode *getRequiredPathUsingDijkstra()* (siehe Anhang B.2). Sie benötigt fünf Parameter als Input: den Startpunkt *pt1*, den Zielpunkt *pt2*, den Gebäudeplan *envColl*, den Erreichbarkeitsgraph *edgeArr* sowie die Anzahl der Zellen, die im Umgebungsplan enthalten sind (*noOfPoints*). Letztere wird in der Methode *ComputeEdges()* in derselben Klasse wie der Algorithmus berechnet. Da dieser Wert jedoch nicht in der Klasse selbst permanent gespeichert werden kann, wird er beim Aufruf von *getRequiredPathUsingDijkstra()* als Parameter übergeben.

In der originalen Arbeit wurde auf diesen Kunstgriff verzichtet und stattdessen die Variable *noOfPoints* mit dem festen Wert 500 initialisiert. In diesem Fall war das jedoch nicht praktikabel, da sich dieser Wert aufgrund der feineren Zelldekomposition und der zusätzlichen Stockwerke bei Weitem als nicht ausreichend erwies und andererseits die Wahl einer sehr hohen Zahl zu Laufzeiteinbüssen geführt hätte.

Der Anfangs- und Endpunkt muss nun für die weitere Berechnung zuerst auf den Mittelpunkt der Zelle, in der er sich befindet, abgebildet werden, was mit der Methode *getRequiredPts()* geschieht. Diese bestimmt das Zentren der Start- und Endzelle und gibt beide in einer ArrayList aus.

Anschließend wird mit der Methode *dijkstra()* eine Menge von Vorgängerknoten berechnet. Mit dieser Ausgabe lässt sich nun der kürzeste Weg bestimmen. Dieser ist eine Folge von Punkten (*PtUsedInPathCls*), die in der ArrayList *finalPath* eingetragen werden. Der unten aufgelistete Codeabschnitt ist für das Zuweisen eines Wertes an den Parameters *finalPath* zuständig:

```
this.pathFromPred(predArr, (PtUsedInPathCls)ptsArr[0], (PtUsedInPathCls)
ptsArr[1], finalPath);
```

Die Methode *dijkstra()* (Anhang B.2) erwartet drei Eingabeparameter, nämlich den Mittelpunkt der Zelle des Startpunkts (*startPt*) beziehungsweise des Zielpunkts (*destPt*) und den Erreichbarkeitsgraph als ArrayList (*edgeArr*).

Für die Bestimmung des kürzesten Weges sind vier Listen notwendig: *nodeQueue*, *predecessor*, *color* und *distance*. *nodeQueue* nimmt diejenigen Knoten des Erreichbarkeitsgraphen auf, welche als nächstes besucht werden. *predecessor*, *color* und *distance* haben die Länge *noOfPoints* + 1. Erstere enthält die Vorgänger der betreffenden Knoten, die zweite die Farbe, wobei *white* noch nicht, *grey* aktuell und *black* bereits besuchte Punkte bezeichnen. Die letzte speichert die zurückgelegte Entfernung vom Startpunkt bis zum momentan betrachteten Knoten.

Sobald die korrekte Initialisierung der Listen abgeschlossen ist, wird beginnend vom Startknoten jeder Nachfolgerknoten besucht und die Distanz berechnet, bis der Zielpunkt erreicht wurde oder es keine weiteren Nachfolger mehr gibt. Im letzteren Fall existiert kein Pfad zum Ziel und der Algorithmus liefert kein Ergebnis.

Bei der Bestimmung der Distanzen wird die Methode *getWeight()* verwendet, die ebenfalls in Anhang B.2 zu sehen ist. Diese Funktion sucht diejenige Kante im Erreichbarkeitsgraph *edgeArr*, welche die beiden eingegebenen Punkte *pt1* und *pt2* als Eckpunkte besitzt und gibt deren Kantengewicht aus.

8.4 Texterzeugung

Der Prozess der Texterzeugung findet in zwei Schritten statt. Zuerst werden simple, stichpunktartige, partielle Navigationsanweisungen generiert, aus denen in einem zweiten Schritt natürlichsprachige Sätze entstehen. Die erste Stufe läuft im sogenannten *Text Planner* ab. Für die zweite ist das *Linguistic Realisation*-Modul zuständig. Die in den verantwortlichen Komponenten implementierten Änderungen sollen im Folgenden erörtert werden.

8.4.1 Partielle Textanweisungen

Um partielle Textanweisungen zur Benutzung einer Treppe oder eines Liftes erzeugen zu können, war *MessageCls* um zwei neue Unterklassen, nämlich *StairsMsg* und *LiftMsg* zu erweitern. Dabei enthält die erste Klasse als Attribute einen Verweis auf das zugehörende *Stair*-Objekt sowie die einzuschlagende Richtung als String. Beim Zugriff auf ihre *ToString()*-Methode liefert sie ”Go up/down the Stairs” aus. Analog dazu enthält *LiftMsg* noch das zusätzliche Attribut *nextSpace*, welches den Raum bezeichnet, wo der Benutzer den Aufzug wieder verlassen soll. Ihre *ToString()*-Methode gibt diese Informationen als ”Use the Lift up/down to Floor X” aus. X steht in diesem Zusammenhang für die *levelId* des *Space*-Objekts.

Weiterhin war die Klasse *TextPlanner* so zu modifizieren, dass *StairMsg* und *LiftMsg* erkannt und an richtiger Stelle verwendet wurden. Hierzu sind zwei neue Methoden dazu gekommen: *getNextSpace()* und *finalizePath()*.

Letztere, in Anhang B.3 gezeigt, war notwendig, da ein Lift über mehrere Stockwerke auch entsprechend viele einzelne Kanten im berechneten Pfad erzeugt hat, aber diese als Einheit zu betrachten sind und nicht mehrere gleich lautende Anweisungen erzeugen dürfen. Ohne diese Methode hätte ein Aufzug, der beispielsweise über fünf Etagen geht, viermal die Ausgabe "Use the Lift to Floor X" erzeugt. Es war daher zwingend erforderlich, die redundanten Kanten aus dem Graph auszuschneiden. Dies geschieht, indem diese Funktion den kompletten Pfad vom Anfang bis zum Ende durchläuft und nach einer Folge von Kanten sucht, die zu einem Lift gehören. Diese werden dann jeweils zu einer einzigen Kante zusammengefasst und der resultierende Weg wird ausgegeben.

Die zweite Funktion *getNextSpace()*, ebenfalls in Anhang B.3 zu sehen, dient, wie der Name schon vermuten lässt, zum Ermitteln des im Navigationsprozess bei einem Stockwerkwechsel als nächstes zu besuchenden Raumes. Dies konnte durch Betrachten des aktuellen Verbindungselementes erreicht werden. Handelt es sich um eine Treppe, so wird das entsprechende *Space*-Objekt ausgegeben, wo der Endpunkt der von der Treppe gebildeten Kante liegt. Anderenfalls verfolgt die Methode die Kanten solange, bis sie wieder in die Horizontale gehen, denn in diesem Fall muss es sich bei dem *LevelLink* um einen Lift handeln. Geliefert wird hier wiederum derjenige Raum, in dem der Anfangspunkt der waagrechten Kante liegt.

Mit Hilfe dieser beiden Methoden konnte nun eine passende Folge von Navigationsbefehlen (*MessageCls*) generiert werden. Um die Stockwerksübergänge zu berücksichtigen und die neuen *StairMsg* und *LiftMsg* einzubinden, war in der Methode *pathProcessing()* noch der im Anhang B.3 abgelegte Codeabschnitt hinzufügen.

An dieser Stelle waren zwei Dinge zu prüfen. Erstens, ob es sich bei dem Etagenübergang um eine Treppe oder einen Aufzug handelt und zweitens, ob der Anwender sich dort nach oben oder nach unten zu bewegen hat. Mit Hilfe dieser Information und mit Kenntnis des als nächsten zu besuchenden Raumes, was mit zuvor beschriebener Methode herausfinden lässt, war *pathProcessing()* jetzt in der Lage, die korrekten *MessageCls* zu erzeugen und in die Folge von Navigationsanweisungen einzubinden.

Nach Durchführung dieser Änderungen und Erweiterungen liefert das *Text Planner*-Protokoll nun die erwünschten Folgen von Textbefehlen, die auch Etagenwechsel durch Benutzung einer Treppe oder Fahren eines Lifts richtig berücksichtigt. Abbildung 8.10 zeigt ein Beispiel für die neuen Instruktionen und den Erfolg dieser Maßnahmen.

8.4.2 Natural Language Generation (NLG)

Die Anpassung des letzten Schrittes beim Prozess der Erzeugung von Textinstruktionen war im Rahmen dieser Arbeit optional, wurde aber im Sinne der Vollständigkeit durchgeführt. Diese Änderungen ließen erfreulicherweise sogar relativ einfach realisieren. Lediglich in die Klasse *LexicaliseCls* musste folgender Codeabschnitt eingefügt werden:

```
if (msg.GetType().Equals(typeof(LiftMsg)))
{
  liftMsg liftMsg = (LiftMsg) msg;
  msgSem.msgValues.Add(liftMsg.direction);
  msgSem.msgValuesLbl.Add("direction");
}
```

Dieser erzeugt für der Fall, dass eine *LiftMsg* auftritt, den zugehörigen natürlichsprachigen Ausdruck. Der entsprechende Code für *StairsMsg* war bereits vorhanden, weil auch im originalen System bereits ein Objekttyp *Stair* implementiert worden ist, jedoch nur als Platzhalter und ohne tatsächlichen Einfluss auf den Navigationsprozess.

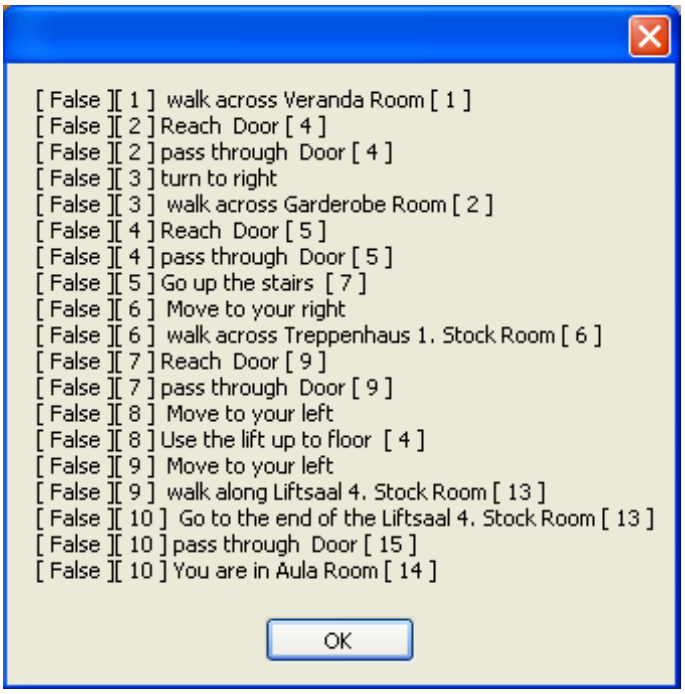

Abbildung 8.10: Stichpunktartige Nachricht

Die syntaktische Regeln, mit deren Hilfe die natürlichsprachigen Instruktionen erzeugt werden, stehen in der Datei *rules.txt*, welche zusammen mit dem Gebäudeplan im Verzeichnis *Data* des *NavAssistServer* abgelegt ist. In dieser war das Einfügen zusätzlicher Regeln notwendig, nämlich:

```
VP|LiftMsg(OBJ) = V|MovVerb2() P|[direction] P|"to" NP|ObjDescMsg(OBJ)
V|MovVerb2() = V|"use_the_lift"
```

Dabei waren die Grammatikregeln um eine neue, so genannte *Verb Phrase (VP)* [Leu97], [Col94] zu erweitern, damit das ebenfalls neue Konstrukt ”use the lift” bei der Erzeugung des Satzes wie gewünscht angehängt wird.

Nach Durchführen dieser Neuerungen erhalten wir am Schluss nun die gewünschte Textausgabe mit grammatikalisch und syntaktisch weitestgehend korrekter Ausdrucksweise, die jetzt den Anwender auch dann zu seinem beabsichtigten Ziel führen kann, wenn dieses auf einem anderen Stockwerk liegt. Abbildung 8.11 zeigt die partiellen Anweisungen aus dem vorhergehenden Bild nach der Umwandlung in natürlichsprachige Instruktionen.

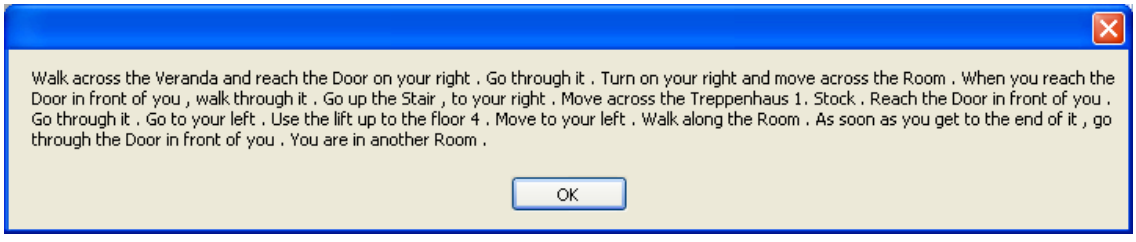

Abbildung 8.11: NLG Nachricht

8.5 Navigational Assistance Server

Die *Navigational Assistance Server*-Anwendung besteht aus drei Klassen: *NASServerFrm*, *GuidingServerCls* und *NavAssistantCls*. Bei allen dreien waren Anpassungen an die neue Struktur des Gebäudeplans notwendig. Hierzu musste die bisherige *ObjectsCollection* durch eine *EnvironmentCollection* ersetzt werden. Da die Benutzerposition und der gewünschte Zielpunkt nun im dreidimensionalen Raum liegt, wurden bei den zugehörigen Attributen die alten Datentypen *Point* beziehungsweise *PointF*, wo es jeweils notwendig war, durch die selbstdefinierte Klasse *Point3D* ausgetauscht.

Abgesehen von diesen allgemeinen Modifikationen waren noch einige spezielle Anpassungen vorzunehmen, auf die im Folgenden genauer eingegangen werden soll. Da der Client seinen Standort in Form eines NMEA-Strings an den Server weiterleitet und dieser dort für die Wegeberechnung benötigt wird, ist serverseitig zusätzlicher Code eingebaut worden, der diese Daten aufnimmt und verarbeitet. Dies geschieht in der Klasse *GuidingServerCls*, und zwar genau gesagt in der Methode *ClientRead()* (Anhang B.4).

Der zusätzlich eingefügte Code erkennt zuerst, ob die vom Client gesendete Botschaft einen NMEA-Datensatz enthält, indem geprüft wird, ob das Präfix ”NMEA” vorhanden ist. Trifft dies zu, wird der String an den Kommastellen aufgesplittet, der Längen- und Breitengrad sowie die Höhe extrahiert und in Gleitkommazahlen konvertiert. Für die Wegberechnung sind allerdings die zugehörenden Pixelkoordinaten gesucht. Diese erhalten wir durch Aufruf der Methode *transform()*, mit deren Hilfe geschieht die Transformation in das Zielkoordinatensystem. Das Ergebnis wird anschließend in das Attribut *state.navAssist.userPos*, welches die aktuelle Position des Benutzers enthält, hineingeschrieben und steht nun für die weitere Wegefindung zur Verfügung.

In derselben Klasse kam auch die neu erstellte Methode *getChecksum()* zum Einsatz. Diese dient zum Bestimmen der NMEA-Prüfsumme und garantiert somit, dass die Daten korrekt übertragen worden sind. Die entsprechende Berechnung erfolgt durch bitweises XOR der einzelnen Zeichen zwischen ”$” und ”*” und verwendet den unter [Per06] frei im Web verfügbaren Code (siehe Anhang B.4).

In einem letzten Schritt waren die partiellen Textanweisungen in spezielle Steuerbefehle umzuwandeln, die dem Client mitteilen, welche grafische Symbole er auszugeben hat. Die entsprechende Methode *getDirectionExpression()* wurde in der Klasse *NavAssistantCls* untergebracht. Diese Operation durchsucht die vom *TextPlanner* erzeugten Anweisungen nach bestimmten Schlüsselworten und generiert daraus für den Client verständliche Kommandos. Wie anhand des Codes in Anhang B.5 unschwer zu erkennen, finden für die Steuerung der Clientanwendung die folgenden Befehle Verwendung:

- STRAIGHT
- DOOR
- RIGHT
- LEFT
- STAIRSUP
- STAIRSDOWN
- LIFTUP
- LIFTDOWN
- SMILE

Das letzte Kommando *SMILE* teilt dem Benutzer mit, dass er an seinem gewünschten Zielort angekommen ist.

8.6 Moblie Client

8.6.1 Grafische Anweisungen

Clientseitig müssen die empfangenden Befehle nun in grafische Anweisungen für den Benutzer umgesetzt werden. Zu diesem Zweck enthält der Client eine Reihe von aussagekräftigen Bildern (vergleiche Sektion 6.7), die im Verzeichnis *Data/Images* gespeichert sind. Die mittlere Größe der einzelnen Bitmaps beträgt lediglich 9 KB und belastet somit auch PDAs und mobile Kleingeräte nicht übermäßig. Um diese auf dem Display anzeigen zu können, wurde die ursprüngliche *TextBox* durch eine *PictureBox* ersetzt und diese wiederum mit einem kleinen Textfeld für die Angabe des Stockwerks, auf dem der Anwender einen Lift verlassen soll, versehen (Abbildung 8.12). Mit diesen Veränderungen war die Umstellung auf eine grafische Ausgabe schließlich abgeschlossen.

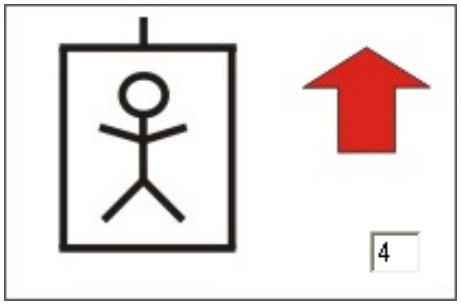

Abbildung 8.12: Lift Message ”Use the Lift to Floor 4”

8.6.2 Anbindung an ein lokales Positionierungssystem

Nun war also für eine Anbindung des Clients an ein Positionierungssystem zu sorgen. Im Einzelnen sind clientseitig dazu folgende Schritte auszuführen:

1. Verbindung zum LPS herstellen
2. Positionsdaten von diesem holen
3. Daten an den Server weiterleiten

Um das zu realisieren, wurde die Methode *getNMEAPos()* (Anhang B.6) in der Klasse *assistantClientCls* implementiert. *getNMEAPos()* verbindet sich über Bluetooth mit dem LPS und liest anschließend solange den Ausgabestrom des Positionierungsgerätes, bis es den benötigten GPGGA-Datensatz erhält oder bis die Zeit für den Timeout abgelaufen ist. Konnte kein geeignetes LPS gefunden werden, so gibt der Client eine entsprechende Fehlermeldung aus und die Anwendung wird beendet. Anderenfalls ist der von dieser Funktion gelieferte Datensatz noch in das von der Anwendung verwendete Kommunikationsprotokoll zu konvertieren, das heißt, mit einem passenden Präfix und einem ”#” am Ende zu versehen. Dies geschieht durch den Aufruf von *sendNMEA()* in der Klasse *protocolHandlingCls*.

Die Methode *sendNMEAPos()* aus der Klasse *assistantClientCls* schließlich sendet diese Positionsdaten an den Server und gibt die von ihm erhaltene Antwort aus:

```
//send the current user pos to the server
public string sendNMEAPos(string nmea)
{
  tcpcls.sendToServer(nmea, serverSck);
  string msg = tcpcls.recDataFromServer(serverSck);
  return msg;
}
```

8.6.3 Bluetooth

Wie bereits in Kapitel 7 erörtert, sollte für die Anbindung des externen LPS eine Bluetooth-Verbindung Verwendung finden. Die entsprechende Logistik ist beim originalen System im Paket *bluetoothDllNet* bereits vorhanden. Die ursprüngliche Erwartung, die existierende Bibliothek in ihrer Originalform weiterverwenden zu können, hat sich leider nicht erfüllt. Vielmehr trat hierbei eine Reihe von Problemen auf, die zu beheben längere Zeit in Anspruch nahm und auf die im Kapitel 10 noch näher eingegangen wird. An dieser Stelle soll nur Erwähnung finden, dass eine teilweise Neuimplementierung der Bluetooth-Bibliothek notwendig war. Die C#-Umgebung gestattet keinen direkten Zugriff auf die entsprechenden APIs, so dass die Bibliothek - wie auch im Original - in *C++* geschrieben und als *Dynamic Link Library (DLL)* eingebunden werden musste.

Mit der modifizierten DLL ließ sich anschließend erfolgreich eine Bluetooth-Kommunikation herstellen.

Kapitel 9

Evaluierung

Die Evaluation des Systems geschah in zwei Phasen. In der ersten wurden die einzelnen Komponenten nach der jeweiligen Fertigstellung ausführlichen Tests unterzogen und auf Korrektheit und Fehlerfreiheit überprüft. Nach dem Abschluss dieser Untersuchungen war in einer zweiten Phase noch das System als Ganzes zu validieren. Die Vorgehensweise bei diesen Tests sollen in diesem Kapitel näher erläutert werden.

9.1 Komponententest

9.1.1 Environment Builder und Object Library

Beide Module, *Environment Builder* und *Object Library*, sind zusammen getestet worden. Bei Ersterem war sicherzustellen, dass die neuen Funktionalitäten eine fehlerfreie Implementation nach sich zogen und das Gewünschte leisten. Augenmerk wurde hier vor allem auf das korrekte Erstellen neuer Stockwerke und auf das Zeichnen der neuen Verbindungselemente Treppen beziehungsweise Aufzüge gelegt. Ferner war zu testen, ob das Umschalten zwischen den Etagen reibungslos abläuft und ob die Attribute für die Gebäudekarte, also die Referenzpunkte, die Stockwerkbezeichner, die einzelnen Höhen und die Gewichte für *Stair*- und *Lift*-Objekte, richtig gesetzt werden können.

Während eingehende Tests mit dem *Environment Builder* durchgeführt wurden, ließ sich gleichzeitig die Korrektheit der zugrunde liegenden Datenstruktur (*ObjectLibrary*) zeigen. Kritischer Punkt hierbei war, den Attributen der neu dazugekommenen Klassen *Lift* und *Stair* die richtigen Werte zuzuweisen. Bei dieser Operation haben sich in ersten Versuchen auch tatsächlich einige Unregelmäßigkeiten ergeben. Die Ursache für diese Fehler konnte jedoch in allen Fällen schnell lokalisiert und beseitigt werden.

Nach Überarbeitung der Klasse *XMLClass* war noch zu prüfen, ob das Laden und Speichern der Gebäudekarten ohne Probleme abläuft. Bei dieser Gelegenheit wurde ein bereits im originalen System vorhandener Bug gefunden, der unter Umständen einen kompletten Absturz der Anwendung zur Folge hatte. Auf die Natur dieses Fehlers und das Lösungsverfahren wird im folgenden Kapitel noch näher eingegangen.

9.1.2 Wegeberechnung

Bei der Evaluierung der für die Wegeberechnung zuständigen Komponenten ergaben sich drei Schwerpunkte: die Zelldekomposition, der neuimplementierte Dijkstra-Algorithmus sowie die Koordinatentransformation.

Bei ersterer musste sichergestellt werden, dass diese nun die gewünschte feinere Zerlegung liefert. Dazu wurde eine abstrakte Gebäudekarte mit teilweise großen, leeren Räumen erzeugt und die vom Algorithmus

berechnete Zerlegung der *Space*-Objekte durch Einzeichnen der erzeugten Zellen in die Karte visualisiert. Wie in Abbildung 9.1 gezeigt, werden nun auch große, leere Flächen wie erwartet dekomponiert und somit ließ sich das gesetzte Ziel erreichen.

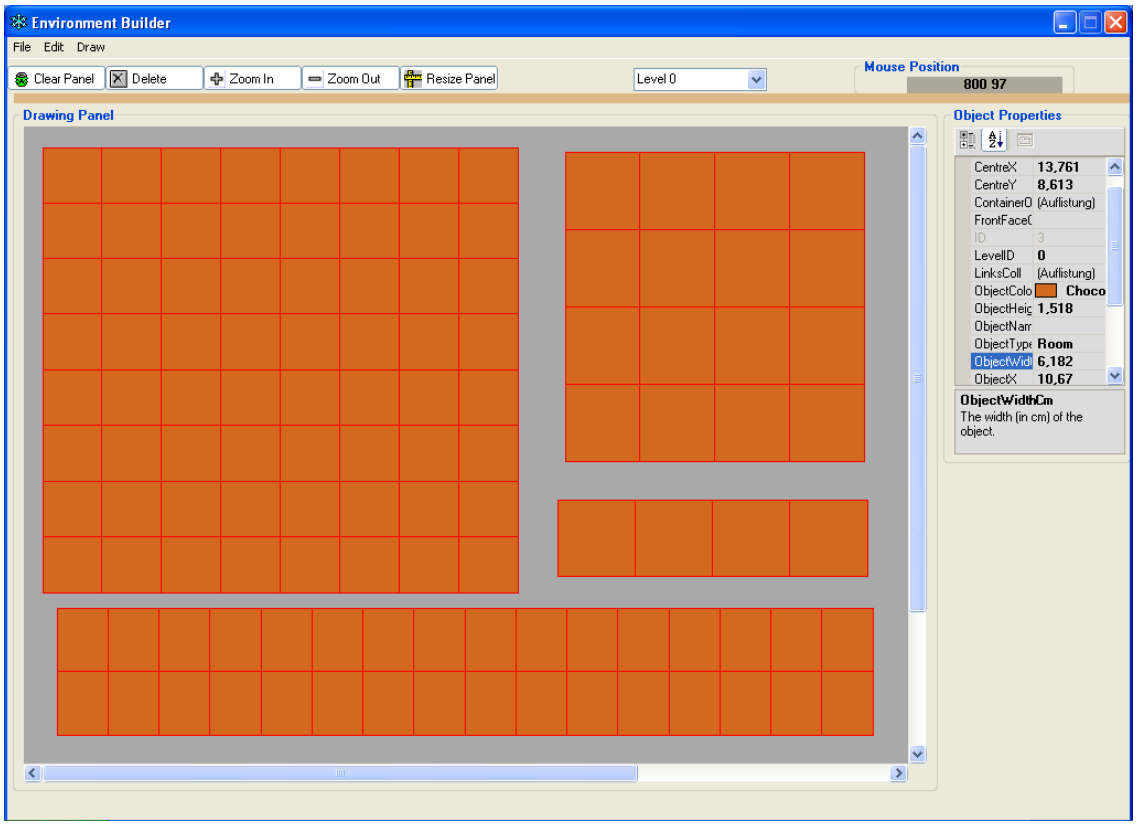

Abbildung 9.1: Zelldekomposition bei verschieden großen Räumen

Die Implementierung des Algorithmus von Dijkstra ist analog zum vorher beschriebenen Ansatz untersucht worden, indem der berechnete Weg ebenfalls durch Einzeichnen in die Karte sichtbar gemacht wurde. Mit Hilfe verschiedener Umgebungspläne ließ sich nun testen, ob dieser in allen Fällen den erwarteten kürzesten Weg ausgibt. Dabei fanden die meisten Versuche in einer mehrdimensionalen Umgebung statt, weil auf die korrekte, stockwerkübergreifende Navigation besonderer Wert zu legen war. Da beim Dijkstra-Algorithmus die Kantengewichte für die Wegeberechnung ausschlaggebend sind, musste nachgewiesen werden, dass letztere auch entsprechend Anwendung finden. Um also zu prüfen, ob bei Vorhandensein von zwei alternativen, gleich langen Wegen zum Ziel, wobei der eine über eine Reihe von Treppen führt und der andere die Benutzung eines Aufzuges erzwingt, war eine spezielle Karte zu erstellen, die diese Gegebenheit widerspiegelt. Wie in Abbildung 9.2 zu sehen, wird tatsächlich der schnellere (günstigere) Weg über den Aufzug gewählt. Als Testumgebung kam dabei eine Karte mit 5 Stockwerken zum Einsatz, Startpunkt war der Erdgeschoss, das Ziel lag im 5. Stock, die Kosten für Treppe beziehungsweise Lift betrugen 5 und 10.

Zuletzt blieb nur noch die Korrektheit der Koordinatentransformation zu untersuchen. Dies geschah mit einer simplen Vorgehensweise, bei der die Quellkoordinaten eines selbst definierten Punktes eingegeben und die erhaltene Ausgabe mit der erwarteten Position im Zielkoordinatensystem verglichen wurde.

9.1.3 Textanweisungen

Aus dem berechneten Weg zum Ziel werden als nächstes Textanweisungen generiert. Die dafür verantwortlichen Module mussten nach ihrer Überarbeitung auf Richtigkeit getestet werden, wobei speziell darauf

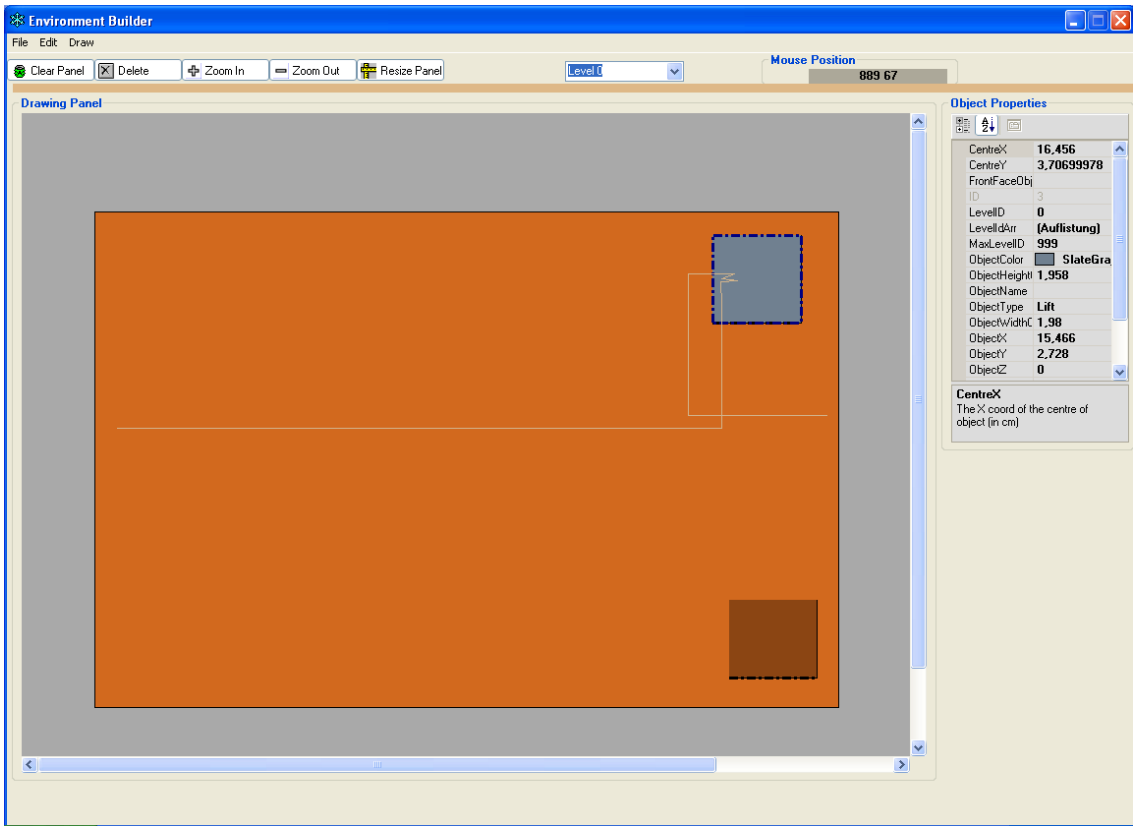

Abbildung 9.2: Wahl des güstigsten Weges (blau = Lift, braun = Treppe).

zu achten war, ob die neuen Verbindungselemente zwischen den Stockwerken passend Berücksichtigung finden. Bei der Evaluierung dieser Komponente war ein Zurückgreifen auf die zuvor verwendeten Pläne möglich. Die Tests erfolgten durch simple Ausgabe der erzeugten Textanweisungen auf dem Bildschirm, die dann problemlos mit dem berechneten Pfad verglichen und auf Stimmigkeit geprüft werden konnten. Im Verlauf dieses Komponententests trat zu Tage, dass bei der Umsetzung der Route in Richtungsbefehle in besonderen Situationen eine Verwechslung von rechts und links auftreten kann. Auf diesen Umstand wird im Kapitel 10 noch detaillierter eingegangen.

9.1.4 Client/Server Kommunikation

Die TCP/IP-basierende Verbindung zwischen dem Navigationsserver und dem für die Ausgabe der Anweisungen verantwortliche Client wurde über die *localhost*-Schnittstelle getestet. Auf eine Überprüfung der Funktionalität über eine WLAN-Verbindung konnte an dieser Stelle verzichtet werden, da an dem für die Verbindung zuständige Modul keine Veränderungen durchgeführt worden sind. Wie auch nicht anders erwartet, traten dabei auch keine Probleme auf.

Serveranwendung

Auf Serverseite musste man sicherstellen, dass dieser die NMEA-Daten, die ihm über den Client weitergeleitet werden, richtig erkennt und die benötigten Informationen (Längengrad, Breitengrad und Höhe über dem Meeresspiegel) daraus sauber extrahiert. Bei diesen Tests kam noch ein weiterer Fehler im Hinblick auf die Koordinatentransformation zum Vorschein, nämlich dass bei den Referenzpunkten die Greenwich-Koordinaten nicht im richtigen Format in die Karte eingetragen wurden. Dort war das Komma bereits nach

der Gradzahl gesetzt, während es im NMEA-Datensatz nach den Minuten stand. Die Folge davon war, dass bei der Transformation die falschen Pixelkoordinaten heraus kamen.

Eine weitere Aufgabe des Serverprozesses ist die Erzeugung von Steuerbefehlen für den mobilen Client aus den vom *GenerationInstructions*-Komponente gelieferten Textanweisungen. Bei diesem Vorgang traten keine Fehler auf.

Clientanwendung

Diese ist verantwortlich für die Weiterleitung der von einem Indoor-LPS stammenden Positionierungsdaten an den Server und außerdem für die Umsetzung der Steuerbefehle, die der Server sendet, in grafische Symbole, die auf dem Display des PDAs ausgegeben werden. Erstere Funktion wurde durch die Übertragung von realen und selbst definierten NMEA-Daten überprüft. Um letzteres zu validieren, waren dann nur die vom Server abgeschickten Steuerbefehle mit der Ausgabe auf dem Client zu vergleichen. Bei beiden traten weder Fehler noch sonstige Auffälligkeiten auf.

9.1.5 Anbindung an ein externes LPS

Bevor es zum Versuch kam, ein externes LPS anzubinden, sollte zuerst das Bluetooth-Modul des originalen Systems in unveränderter Form auf ihre Funktionstüchtigkeit überprüft werden. Unerwarteterweise traten bereits hier Probleme auf, über deren Ursache und Behebung im folgenden Kapitel zu sprechen sein wird.

Nachdem die Anwendung mit einer lauffähigen Bluetooth-Komponente ausgestattet wurde, erfolgten alle weiteren Tests direkt mit der späteren Gegenstelle [Ric06]. Im Hinblick auf die Anwendung des LPS sind hier drei Dinge sicherzustellen gewesen:

- problemloser Verbindungsaufbau
- fehlerfreie Übertragung
- korrekte Weiterleitung des NMEA-Strings

Dass die ersten beiden Eigenschaften erfüllt waren, konnte, wie bereits erwähnt, durch Funktionstests mit dem späteren Zielsystem nachgewiesen werden. Da außerdem die Evaluierung von Client- und Serveranwendung parallel ablief, war es ohne Schwierigkeiten möglich, auch die letzte Anforderung zu validieren, indem die vom Client erhaltene Nachricht auf dem Server ausgegeben wurde.

9.2 Systemtest

Nach dem Abschluss der vorstehend beschriebene Komponententest, war selbstverständlich noch das System als Ganzes zu testen. Dies ist zusammen mit dem in [Ric06] entwickelten externen LPS durchgeführt worden, um sicherzustellen, dass die Anbindung wie gewünscht funktioniert und die Positionsdaten nutzbar gemacht werden können. Anschließend wurden diese Tests nochmals mit synthetischen NMEA-Strings wiederholt, um auch wirklich Aussagen über die Leistungsfähigkeit der im Rahmen dieser Diplomarbeit realisierten Anwendung treffen zu können, denn reale Daten enthalten immer einen unvermeidlichen Ungenauigkeitsfaktor.

Als Testumgebung diente eine abstrahierte Karte eines Kellerabschnitts des Instituts für Informatik der LMU München (Abb. 9.3). Die eingezeichneten Punkte illustrieren den angenommenen Weg des Benutzers, wobei für jede Position ein ”Pseudo” NMEA-String per Knopfdruck vom Client an den Server geschickt wurde. Diese NMEA-Daten waren im Einzelnen:

- $GPGGA,191410,120.0,N,83.0,E,1,04,4.4,0.0,M,0.0,M,,*45
- $GPGGA,191410,334.0,N,111.0,E,1,04,4.4,0.0,M,0.0,M,,*45

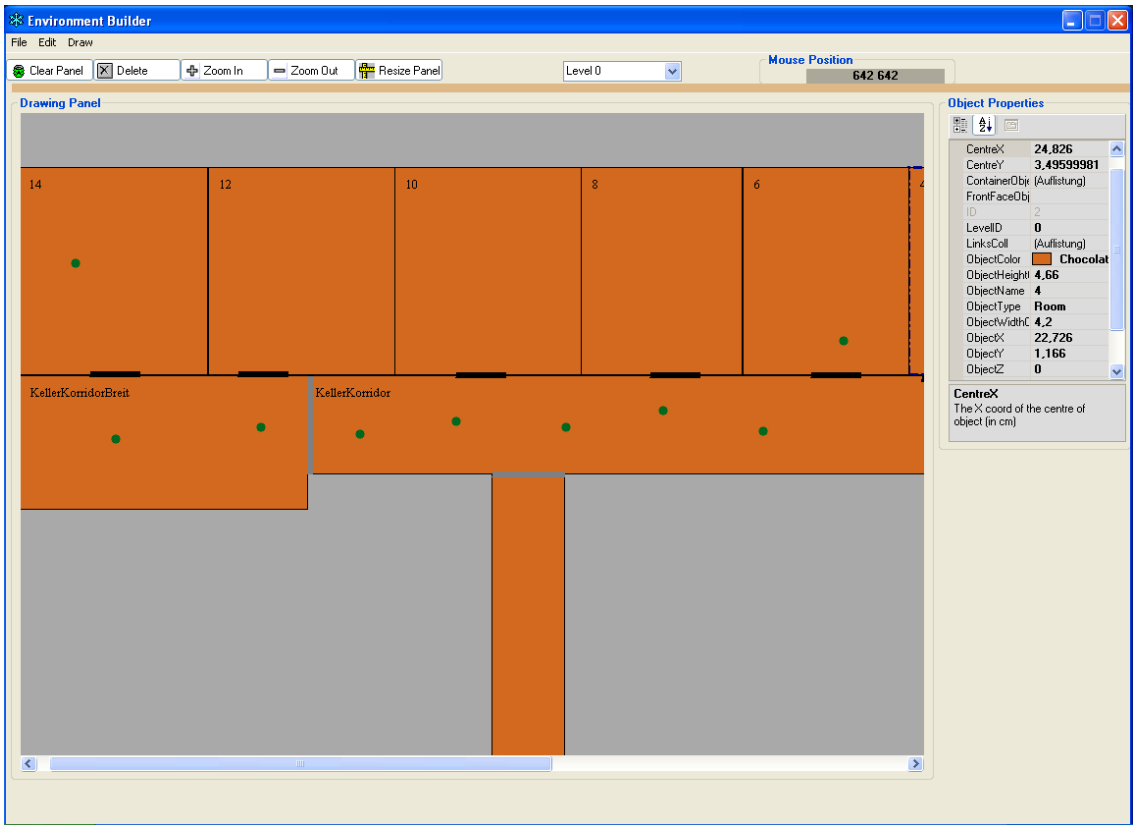

Abbildung 9.3: Testumgebung mit Positionsdaten (grüne Punkte)

- $GPGGA,191410,325.0,N,253.0,E,1,04,4.4,0.0,M,0.0,M,,*45
- $GPGGA,191410,328.0,N,354.0,E,1,04,4.4,0.0,M,0.0,M,,*45
- $GPGGA,191410,313.0,N,478.0,E,1,04,4.4,0.0,M,0.0,M,,*45
- $GPGGA,191410,316.0,N,637.0,E,1,04,4.4,0.0,M,0.0,M,,*45
- $GPGGA,191410,310.0,N,764.0,E,1,04,4.4,0.0,M,0.0,M,,*45
- $GPGGA,191410,318.0,N,860.0,E,1,04,4.4,0.0,M,0.0,M,,*45
- $GPGGA,191410,237.0,N,971.0,E,1,04,4.4,0.0,M,0.0,M,,*45

Wie unschwer zu erkennen, repräsentieren die Einträge für Längen- und Breitengrad in den verwendeten Strings die Koordinaten in Pixel, so dass auf dem Server keine Transformation in ein anderes Koordinatensystem notwendig war und somit ein weiterer Ungenauigkeitsfaktor durch mögliche Rundungsfehler vermieden wurde. Neben dem statischen Verhalten, bei dem von den gesendeten Positionen nur die erste tatsächlich Beachtung fand, ist im Anschluss auch noch kurz ein annähernd dynamisches Verhalten untersucht worden. Bei letzterem fand jeder einzelne Punkt Berücksichtigung und führte stets zu einer Neuberechnung des Weges. Die dabei erhaltenen Ergebnisse waren allerdings, im Gegensatz zur statischen Version, nicht zufrieden stellend. Die Ursache dafür ist in der Art und Weise zu suchen, wie aus dem berechneten Weg Richtungsanweisungen gewonnen werden. Auch darauf wird im folgenden Kapitel noch einzugehen sein.

Kapitel 10

Erkannte Probleme

Im vorliegenden Kapitel sollen abschließend noch einige Probleme, die im Verlauf dieser Arbeit aufgetreten sind, einer näheren Betrachtung unterzogen werden. Hierbei wird gleichzeitig auch, soweit möglich, der verwendete Lösungsansatz vorgestellt. An dieser Stelle sei jedoch nicht verschwiegen, dass die Klärung der Ursache für die meisten dieser Probleme oftmals kompliziert war und einen guten Teil der verfügbaren Zeit in Anspruch genommen hat. Dass die Umstellung eines vorhandenen Systems auf einen ursprünglich nicht vorgesehenen Einsatzzweck nicht vollkommen reibungslos verlaufen würde, wurde von Anfang einkalkuliert. Überraschenderweise traten bereits beim ersten Versuch, die originale Anwendung auf den eigenen Testrechner zum Laufen zu bringen, einige Schwierigkeiten auf.

10.1 Generierung der Richtungsangaben

10.1.1 Korrektheit der Ausgabe

Als wesentliche Einschränkung erwies sich im späteren Verlauf dieser Diplomarbeit das Verfahren, mit dem beim ursprünglichen System die Route in Richtungsanweisungen umgesetzt wird. Zum besseren Verständnis soll der Algorithmus an dieser Stelle kurz skizziert werden:

Zu Beginn wird stets zuerst eine ”walk across”-Anweisung erzeugt, es sei denn, der Benutzer befindet sich bereits im Zielraum. Das System nimmt somit an, dass die Blickrichtung des Anwenders bereits bekannt ist beziehungsweise dieser in die Richtung blickt, in die er gehen muss.

Im weiteren Verlauf werden Richtungsänderungen wie folgt bestimmt: es wird eine hypothetische Linie vom Zentrum des zuletzt besuchten Raumes zum Mittelpunkt der Türe, durch die der Benutzer soeben gegangen ist, gezeichnet. Anschließend wird eine zweite Linie vom Zentrum der eben erwähnten Türe zum Mittelpunkt zu der als nächstes zu durchschreitenden Tür erstellt. Der Algorithmus betrachtet danach den Schnittwinkel zwischen den beiden Geraden. Dreht sich dieser im Uhrzeigersinn, so erhält man als nächste zu verfolgende Richtung ”Move to your right” oder ”turn to right” als Ausgabe, anderenfalls ”Move to your left” beziehungsweise ”turn to left”, bei minimalen Änderungen jedoch nur ”Reach Door”. Dieses Verfahren wird in Abbildung 10.1 veranschaulicht, wobei die Linie, die das Zentrum des zuletzt besuchten *Space*-Objekts und die Tür, die der Anwender gerade passiert hat, schneiden, schwarz eingezeichnet ist. Die Tür-zu-Tür-Gerade ist grün markiert.

Bei dieser Vorgehensweise kann es jedoch in bestimmten Fällen zu einer Verwechslung von rechts und links kommen, wie in Abbildung 10.2 dargestellt. Auf dem Bild ist klar zu erkennen, dass die zweite, grün eingezeichnete Linie im Verhältnis zur ersten gegen den Uhrzeigersinn gedreht ist und demzufolge als nächste Anweisung ”Move to your left” ausgegeben wird, tatsächlich aber sich der nächste Raum eigentlich in der entgegen gesetzten Richtung befindet.

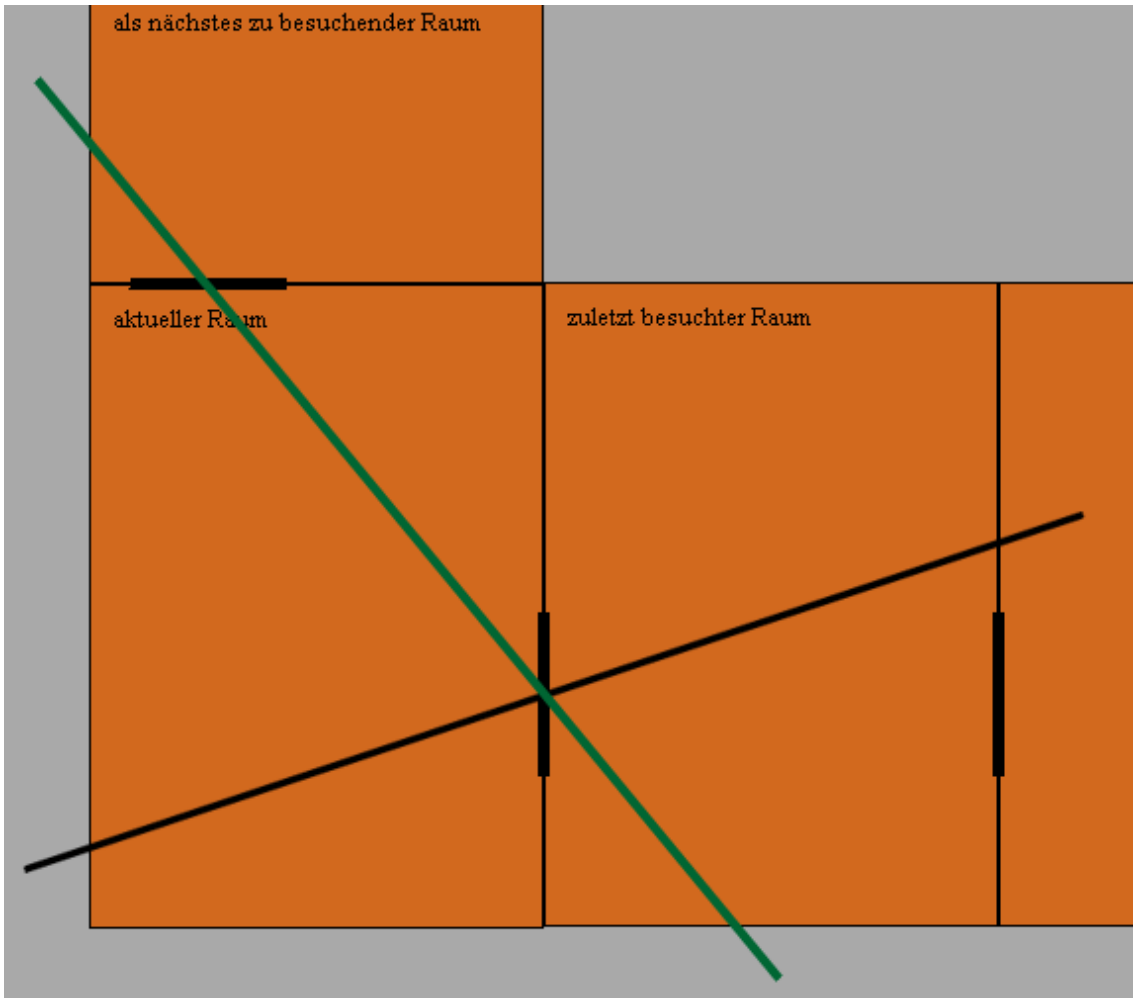

Abbildung 10.1: Befehlsanweisung für ”Move to your right”

Ebenso schwerwiegend war die Tatsache, dass unter Umständen äußerst ungenaue oder sogar wirklich falsche Instruktionen ausgegeben worden sind, beispielsweise in mehreren Fällen ”Go to the end of the floor”, obwohl sich der Raum gar nicht am Ende des Ganges, sondern in der Mitte befand. Dies trat in der Regel bei sehr großen Räumen oder Korridoren auf und wird ebenfalls durch diese Art der Berechnung verursacht.

10.1.2 Auswirkungen auf die aktuelle Arbeit

Der Ansatz, die Instruktionen nur von Raum zu Raum beziehungsweise Tür zu Tür zu erzeugen, führt noch zu einem weiteren Problem. Es werden nämlich lediglich bei Richtungsänderungen beziehungsweise beim Betreten eines neuen Raumes Anweisungen erzeugt und ausgegeben. Während die Umsetzung der generierten Textanweisungen in visuelle Symbole ein zufrieden stellendes Ergebnis lieferte, hat sich beim Versuch, das System dynamisch zu machen, jedoch herausgestellt, dass die so berechneten Anweisungen für diesen Zweck viel zu spärlich und lückenhaft sind. Einen Ausdruck wie ”Reach Door” kann man nicht 1:1 und ohne Informationsverlust in Symbolform umsetzen, wenn der Benutzer, übertrieben dargestellt, noch 15 Meter von der entsprechenden Türe entfernt ist. Es wäre hier vielmehr ein kontinuierlicher Informationsfluss zwingend notwendig gewesen, der sich jedoch nur dann erbringen lässt, wenn für jeden einzelnen Übergang von Zelle zu Zelle eine entsprechende Instruktion erstellt wird.

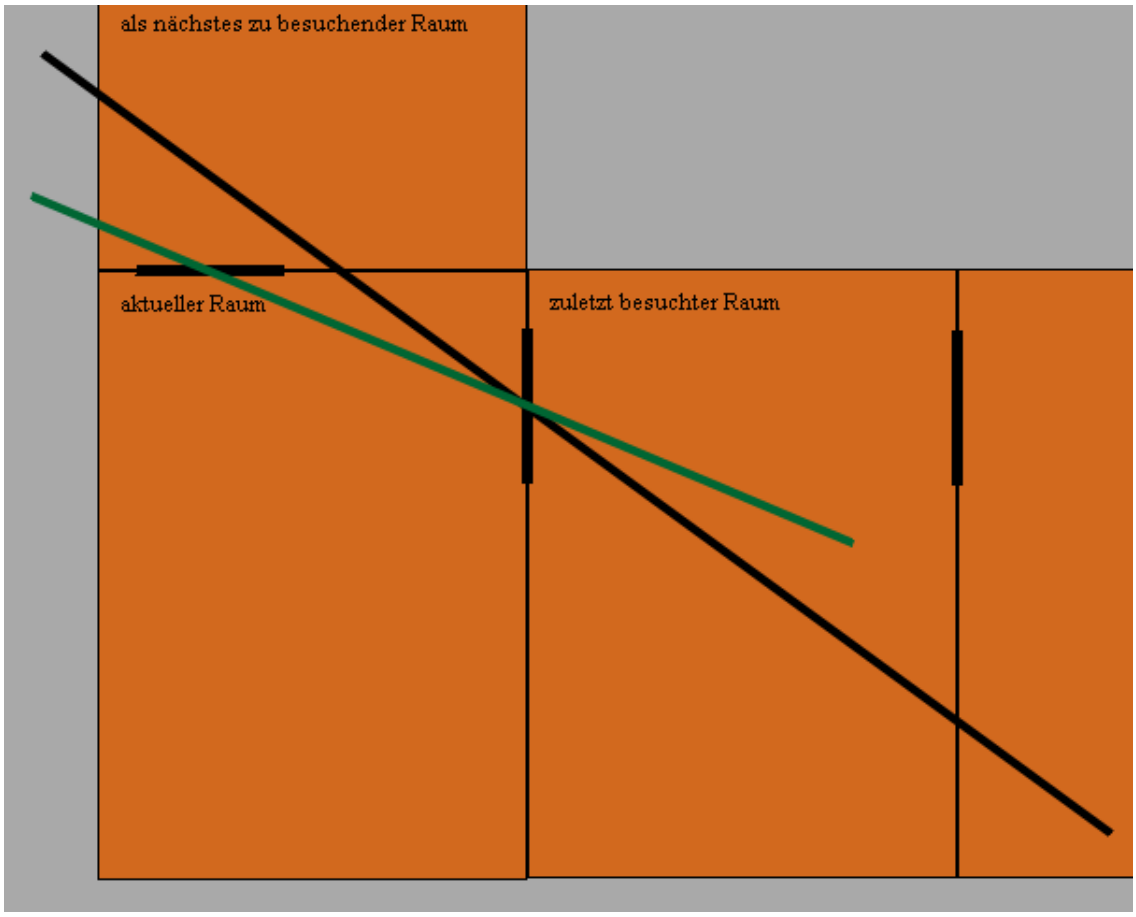

Abbildung 10.2: Beispiel für fehlerhafte Navigationsanweisung

Um dieses Verhalten zu erreichen, wurde zuerst daran gedacht, ein entsprechendes Modul neu zu entwerfen und einzusetzen. Da jedoch Zweifel bestanden, ob die Neuentwicklung und Implementierung einer solchen Komponente innerhalb des gegebenen Zeitrahmens zu schaffen ist, ist entschieden worden, das bisherige *GenerationInstructions*-Subsystem trotz der nachgewiesenen Unzulänglichkeiten in modifizierter Form weiterzuverwenden.

10.2 Bluetooth

Auch beim Testen der Bluetooth-Komponente kamen Probleme zum Vorschein. Trotz korrekter Installation, deren Vorgehensweise im Handbuch der originalen Arbeit beschrieben steht, lies sich die Client-Anwendung nicht dazu bringen, eine entsprechende Verbindung zum Server aufzubauen. Als erste Ursache wurde ausgemacht, dass auf dem zum Testen verwendete Laptop irrtümlicherweise der *WIDCOMM Protokoll Stack* installiert war. Der Austausch gegen den benötigten *Microsoft Stack* führte wider Erwarten jedoch auch nicht zum erwünschten Erfolg. Aus Zeitgründen auf eine weitere Suche nach der Ursache verzichtet und stattdessen entschieden worden, die Bluetooth-Bibliothek teilweise neu zu implementieren.

Dabei wurde sofort mit dem gewünschten Zielsystem, dem in [Ric06] entwickelten LPS gearbeitet. Eine Analyse der Originalbibliothek ergab, dass diese weitestgehend korrekt war, jedoch ausschließlich den Bluetooth-Port 15 für die Kommunikation verwendet hat. Nachdem dieses Manko beseitigt wurde, konnte erfolgreich eine Verbindung zum Zielgerät aufgebaut werden.

10.3 Fehler beim Laden der Karte

Beim Laden der Stockwerkkarte trat ebenfalls ein weiterer Fehler auf. Dieser wurde bei Tests mit selbst erstellten Karten zufällig entdeckt. Wird beim Erstellen eines Etagenplans eine Türe eingezeichnet, die keine Verbindung zu einem angrenzenden Raum hat, also sozusagen ”frei in der Luft hängt” und wird der Plan abgespeichert, so kann er später nicht mehr geladen werden. Die Anwendung stürzt in einem solchen Fall unwiderruflich ab.

Diese Anomalie konnte glücklicherweise rasch lokalisiert werden. Beim Speichern der Daten im XML-Format wird der Wert *null* an entsprechender Stelle in die Datei eingetragen, wenn bei einer Türe der links- beziehungsweise rechtsseitige Raum fehlt. Beim Laden erwartet das Programm aber den Identifikator des *Space*-Objekts, also eine Integerzahl. Der Versuch, den String *null* in eine Integerzahl zu konvertieren, schlägt logischerweise fehl und verursacht eine entsprechende Ausgabe.

Dieser Bug konnte beseitigt werden, indem beim Speichern an Stelle von *null* der Wert *-255* verwendet wurde. Dieser Lösungsansatz kam in der Folge auch bei *Stair* und *Lift* zu Anwendung, wo ähnliche Probleme hätten auftreten können. Auf die weitere Verarbeitung hat diese Änderung keine Auswirkungen.

10.4 Ländereinstellung

Das Laden einer bei der originalen Arbeit mitgelieferten Stockwerkkarte in der *Environment Builder*-Anwendung führte stets zu einem kompletten Absturz des Programms, welches dann nur noch über den Taskmanager beendet werden konnte. Die ausgegebene Fehlermeldung gab keinen Hinweis auf die Ursache und trug somit nicht zur Fehlerfindung bei. Der Grund für diesen Bug wurde erst mit Unterstützung der Autorin [Miz04] gefunden.

Dieser liegt in den unterschiedlichen Ländereinstellungen von englischen und deutschen Windows-Rechnern begründet. Während bei den Ersteren ein ”.” als Dezimaltrennzeichen verwendet wird, ist es bei deutschsprachigen Betriebssystemen ein ”,”. Somit können auf einem System erstellten Karten auf dem anderen nur dann geladen werden, wenn das Dezimaltrennzeichen in den Ländereinstellungen korrekt gesetzt ist.

Eine kurze Vorgehensweise, um diesem Problem Abhilfe zu schaffen (Windows XP):

”Start -> Systemsteuerung -> Datums-, Zeit-, Sprach- und Regionaleinstellungen -> Regions- und Sprachoptionen -> regionale Einstellungen -> Anpassen -> Dezimaltrennzeichen”.

Kapitel 11

Zusammenfassung und Ausblick

Hauptziel dieser Diplomarbeit war die Nutzbarmachung von Indoor-Positionierungsdaten zur mobilen Wegplanung. Um dies zu erreichen, sollte einerseits ein bereits vorhandenes, aber nur beschränkt einsetzbares, Navigationssystem geeignet modifiziert und eine Anbindung an ein Indoor-*Location Positioning System (LPS)* realisiert werden.

Die originale Anwendung wurde ursprünglich in erster Linie zum Auffinden von Büchern in einer Bibliothek entwickelt, ist lediglich innerhalb eines einzelnen Stockwerks einsetzbar und die Ausgabe der Navigationsanweisungen erfolgt in Textform. Um dennoch einen möglichst hohen Komfort und Verständlichkeit zu erzielen, findet ein Verfahren namens *Natural Language Generation (NLG)* Verwendung, welches natürlichsprachlich klingende Sätze erstellt.

Aus Zeitgründen war bei der ursprünglichen Anwendung auf die Entwicklung eines eigenen LPS beziehungsweise die korrekte Anbindung an eines solchen verzichtet und dieses zu Testzwecken durch ein Simulationsprogramm ersetzt worden.

Das in [Miz04] vorgestellte System sollte nun im Verlauf dieser Diplomarbeit zu einem weitestgehend vollwertigen *Navigational Assistance System (NAS)* erweitert und statt der LPS-Simulation eine Schnittstelle für die Anbindung eines externen Positionierungssystems implementiert werden. Die dabei für notwendig befundenen Veränderungen betreffen die Erweiterung des Navigationsprozesses, Schaffung einer grafischen Ausgabe auf Seiten des mobilen Clients, sowie ein dynamisches Verhalten. Abgesehen von letzterer Anforderung wurden diese Ziele auch in vollem Umfang erreicht.

11.1 Anbindung an ein externes LPS

Ein Navigationssystem benötigt naturgemäß aktuelle Positionsdaten. Um diese für das NAS nutzbar zu machen, musste eine Möglichkeit für die Anbindung an ein externes LPS gesucht werden. Bei der Festlegung der passenden Schnittstelle fiel die Wahl auf den auch in GPS-Empfängern verwendeten *NMEA-0183* Standard, um ein Maximum an Kompatibilität zu erzielen. Der Anschluss zum Ortungssystem erfolgt dabei über eine *Bluetooth*-Verbindung. Hierbei traten anfänglich einige Schwierigkeiten auf, denn das Bluetooth-Protokoll des originalen Systems erwies sich wider Erwarten als nicht funktionsfähig. Hier brachte erst eine teilweise Neuimplementierung Abhilfe. Danach klappte die Anbindung des Positionierungssystems und die Nutzung der von diesem gelieferten Daten jedoch reibungslos.

11.2 Erweiterung für den Einsatz in mehrstöckigen Gebäuden

Die ursprüngliche Beschränkung auf eine einzelne Etage war ein wesentlicher Hemmschuh der originalen Anwendung. Um dies im Institut für Informatik der LMU München einsetzen zu können, sollte es auch mehrere Stockwerke korrekt handhaben. Die dafür erforderlichen Modifikationen waren teilweise äußerst tiefgreifend und ihre Implementierung nahm einen großen Teil der verfügbaren Zeit in Anspruch. Im Einzelnen musste die ursprüngliche Datenstruktur für den Stockwerkplan so angepasst werden, dass sich mit ihr eine komplette mehrstöckige Karte für ein Gebäude erzeugen ließ. Um darin eine Navigation zu ermöglichen, waren zwischen den einzelnen Etagen geeignete Verbindungselemente einzubauen, wofür die Klassen *Lift* und *Stair* gewählt wurden. Zur Handhabung dieser neuen Datenstruktur war eine anschließende Anpassung der grafischen Oberfläche unerlässlich. Zu den dabei durchgeführten Veränderungen zählten Hinzufügen neuer Stockwerke, Zeichnen von Treppen und Aufzügen, das Setzen von Referenzpunkten für die Wegeberechnung, das Eintragen von Etagenbezeichner beziehungsweise Höhen und einiges mehr.

Nachdem dadurch die notwendige Grundlage erschaffen worden war, konnte der eigentliche Navigationsprozess modifiziert werden. Dieser Prozess besteht einerseits aus der Wegeberechnung und andererseits aus Erzeugung von Anweisungen, die den Benutzer diesen Weg verfolgen lassen. Der in der ursprünglichen Arbeit eingesetzte *A*-Algorithmus* wurde nach ausführlicher Literaturrecherche als ungeeignet erkannt und in der Folge davon beschlossen, ihn durch den Algorithmus von *Dijkstra* zu ersetzen.

Letzterer benutzt für die Wegeberechnung ebenfalls Kantengewichte. Somit ergab sich die Gelegenheit, Lifte und Treppen mit verschiedenen Kosten zu versehen, die bei der Bestimmung der Route zum Ziel entsprechend berücksichtigt wurden. Wie sich herausgestellt hat, können die jeweiligen Gewichte allerdings nicht allgemeingültig angegeben, sondern müssen für jede Gebäudearchitektur separat empirisch bestimmt werden.

Die Umsetzung des Weges in Anweisungen für den Anwender erfolgt beim originalen System in der *GenerationInstructions*-Komponente. Die von ihr gelieferten Instruktionen gelten jeweils von Raum zu Raum beziehungsweise von Tür zu Tür. Bei ersten Tests mit diesem Modul wurde rasch klar, dass die generierten Hinweise nicht nur gelegentlich fehlerhaft sind, sondern sich diese auch nicht 1:1 in einen kontinuierlichen Informationsstrom konvertieren lassen, welcher für ein vollständig dynamisches Verhalten Voraussetzung ist. Aus diesem Grunde wurde deshalb die Möglichkeit ins Auge gefasst, die entsprechende Komponente durch eine eigene Entwicklung auszutauschen. Aufgrund der zu dem Zeitpunkt jedoch bereits knapp bemessenen Zeit musste darauf verzichtet und stattdessen mit den nicht voll befriedigenden Resultaten des originalen Moduls weitergearbeitet werden.

11.3 Grafische Ausgabe

Das dritte Unterfangen, welches im Verlauf dieser Arbeit unternommen werden sollte, war die Umstellung des Ausgabemodus von Anweisungen in Textform auf visualisierte Richtungsangaben. Zur Darstellung derartiger Informationen auf einem Computerbildschirm oder PDA Display existieren drei grundlegende Verfahren, nämlich *2D route sketches*, *2D-* oder *3D maps*.

Bei diesem Ansatz werden zur Ausgabe vorwiegend mit Bemerkungen (*labels*) versehene Pfeile eingesetzt, welche die einzuschlagende Richtung angeben. Die Informationen sind also stark abstrahierter, aber bei korrekter Verwendung dennoch sehr gut verständlich. Nachteilig gestaltet sich jedoch, dass die Informationen stark kontextabhängig sind, also beispielsweise dem System die Blickrichtung des Anwenders bekannt sein muss.

Die letzten beiden Vorgehensweisen verwenden eine mehr oder weniger abstrakt gehaltene zwei- beziehungsweise dreidimensionale Karte der Umgebung. Die Navigationsinstruktionen werden direkt in der Karte selbst angezeigt, so dass sich der benötigte Kontext von selbst ergibt. Während der Benutzer Gebäude und sonstige auffällige Objekte mit einer dreidimensionalen Ansicht besser erkennen kann, stellt eine zwei-

dimensionale Karte der Umgebung aus der Vogelperspektive wesentlich moderatere Anforderungen an die Rechenleistung und Bildschirmauflösung.

Um die Anwendung auf einem mobilen Kleingerät (PDA etc.) verwenden zu können, musste auf eine Anzeige als 3D map von vornherein verzichtet werden. Diese Entscheidung fiel allerdings leicht, da solche Karten beim Einsatz in einem Gebäude ohnehin keinen hohen Wiedererkennungswert aufweisen und in der Hinsicht den zweidimensionalen Umgebungsplänen unterlegen sind. Somit kamen nur 2D route sketches und 2D maps in die engere Wahl. Als nur scheinbarer Vorteil erwies sich hierbei, dass das originale System bereits mit Stockwerkplänen in Form von zweidimensionalen Karten arbeitet. Wie sich bei näherer Betrachtung rasch zeigte, besteht keine sinnvolle Möglichkeit, diese auf Clientseite für die Ausgabe nutzbar zu machen. Auch die vom *GenerationInstructions*-Modul erzeugten Anweisungen können nicht ohne weiteres für die Anzeige im Kontext einer 2D map verwendet oder angepasst werden. Die hierzu notwendigen Änderungen hätten den Aufwand für die Entwicklung neuer Komponente überstiegen und dies musste aus Zeitgründen unterbleiben.

Somit kam hier letztlich nur der Einsatz von 2D route sketches in Frage. Diese erlauben eine relativ simple 1:1 Abbildung der Textanweisung auf visuelle Symbole, wobei eine Anweisung wie beispielsweise ”go straight ahead” durch einen geraden und ”move to your right” durch einen nach rechts gebogenen Pfeil dargestellt werden kann. Die restlichen Anweisungen des originalen Systems konnte man auf analoge Art und Weise in Symbolform transformieren.

11.4 Dynamische Wegefindung

Der letzte Meilenstein, den es zu erreichen galt, war die Implementierung einer dynamischen Wegefindung. Bei der ursprünglichen Anwendung wurde der Weg anfänglich einmal berechnet und anschließend in passenden Teilstücken als Text ausgegeben. Um das nächste Textsegment zu erhalten, hatte der Benutzer bei der Clientanwendung auf einen Button zu klicken. Seine aktuelle Position wurde nicht verifiziert und es erfolgte bis zum Ziel auch keine Neuberechnung des Weges.

Vollwertige Navigationssysteme können jedoch - abgesehen von der Eingabe des Zielortes - gewöhnlich weitestgehend auf eine Interaktion mit dem Benutzer verzichten. Sein Standort wird automatisch erkannt und seine Bewegungen laufend verfolgt, während er zu seinem gewünschten Ziel geführt wird. Weicht der Anwender bewusst vom Weg ab oder interpretiert er diese Anweisungen falsch und verläuft oder verfährt sich, korrigiert das System dies automatisch. Ein berühmt-berüchtiges Beispiel hier ist das ”Bitte wenden Sie bei der nächsten Gelegenheit” von GPS-basierten Autonavigationssystemen. Ferner werden die Instruktionen automatisch, kontinuierlich und zum passenden Zeitpunkt angezeigt, ohne dass hierfür irgendwelche Eingaben getätigt werden müssen.

Da nunmehr auf Indoor-Positionierungsdaten von einem externen LPS zurückgegriffen werden konnte, sollte nun ein solches Verhalten auch mit der bestehenden Anwendung realisiert werden. Hier zeigten sich jedoch nun rasch die Grenzen des originalen Systemdesigns. Da die *GenerationInstructions*-Komponente nicht dazu ausgelegt war, einen kontinuierlichen Anweisungsstrom zu generieren, sondern - wie bereits erläutert — nur für die Strecke von einer Tür zur nächsten Instruktionen ausgab, konnte kein sauberes dynamisches Verhalten erzielt werden. Zudem waren die Richtungsangaben nun häufig nicht mehr aussagekräftig.

11.5 Fazit

Bei der Erweiterungen und Verbesserungen des bestehenden NAS nahm das Verstehen des zur Verfügung gestellten Quellcodes einen großen Teil der Zeit in Anspruch. Die Dokumentation war hierbei leider oftmals wenig hilfreich, so dass viel Zeit und Energie in diese Arbeit gesteckt werden musste. Da die Funkti-

onsweise des Systems erst später komplett überblickt werden konnte, blieben grundlegende konzeptionelle Mängel teilweise zu lange verborgen.

Die Datenstruktur für die Umgebungspläne konnte ohne ernstere Schwierigkeiten für die Verwendung in einem mehrstöckigen Gebäude angepasst werden. Auch die Anbindung an ein Ortungssystem ließ sich nach Behebung der anfänglichen Probleme mit der Ansteuerung des verwendeten Bluetoothgerätes erfolgreich bewerkstelligen. Gleiches gilt für die Nutzung der Indoor-Positionierungsdaten und der modifizierten Wegeberechnung im mehrdimensionalen Raum.

Erste potentielle Designfehler kamen zum Vorschein, als das Modul für die Umsetzung des gefundenen Weges in Richtungsanweisungen examiniert wurde. Während das dort implementierte Verfahren für Textinstruktionen vermutlich ausreichend war, bestanden hinsichtlich des anvisierten neuen Verwendungszweckes jedoch gewisse Bedenken. Diese wurden durch die Erkenntnis, dass die Komponente unter bestimmten Umständen falsche Angaben liefert, noch weiter verstärkt.

Der in diesem Stadium ins Auge gefasste Plan, dieses und die restlichen benötigten Module von Grund auf neu zu entwickeln, wurde wieder verworfen, da nicht sicher war, ob dies innerhalb der verbliebenen restlichen Zeit erfolgreich durchgeführt werden konnte. Wie im späteren Verlauf klar wurde, bedeutete diese Entscheidung jedoch, dass statt einer zweidimensionalen Karte für die grafische Ausgabe auf die schlichteren 2D route sketches zurückgegriffen werden musste. Schwerwiegender war allerdings, dass sich eine dynamische Wegefindung nicht mehr wie ursprünglich gewünscht realisieren ließ.

Hätte unbegrenzt Zeit zur Verfügung gestanden, wäre mit Sicherheit die Entscheidung zugunsten einer Neuentwicklung und -implementation der entsprechenden Module gefallen. Dies war jedoch nicht der Fall und somit kann auch in nachträglicher Betrachtung nicht endgültig entschieden werden, ob der Entschluss zur Weiterverwendung der originalen Komponente nun richtig oder falsch war.

Trotz dieser Stolpersteine wurden die Vorgaben dieser Arbeit zum überwiegenden Teil erreicht. Für zukünftige Verbesserungen bieten sich freilich noch viele Ansätze, die im Folgenden noch kurz angesprochen werden sollen.

11.6 Ausblick

Ein modernes NAS sollte optimalerweise ein voll automatisiertes und dynamischen Verhalten zeigen. Dies ließ sich in der vorliegenden Arbeit nicht mehr zufrieden stellend realisieren, da das zugrunde liegende Systemdesign dafür nicht ausgelegt war. Jedoch konnte hiermit zumindest eine solide Basis gelegt werden, auf deren Fundament in zukünftigen Arbeiten dieses Vorhaben in Angriff genommen werden kann. Notwendig hierfür wäre in erster Linie die Umsetzung des berechneten Pfades in feingranulare, kontextbezogene und kontinuierliche Navigationsinstruktionen.

Mit diesen ließe sich anschließend vermutlich auch eine grafische Ausgabe unter Verwendung der bewährten zweidimensionalen Kartenansicht realisieren. Hierfür dürften allerdings noch eine ganze Reihe weiterer Modifikationen erforderlich sein, da die Clientanwendung nicht für die Anzeige derartiger Umgebungspläne ausgelegt ist.

Als weitere wesentliche potentielle Verbesserungsmöglichkeit sei an dieser Stelle auch noch die geeignete Anbindung des Servers an eine Datenbank für die Gebäudekarten genannt. In seiner momentanen Form arbeitet dieser nämlich nur mit einer einzigen Karte, die zudem in seiner Konfigurationsdatei fest vorgegeben werden muss.

Im Hinblick auf die in dieser Arbeit gesammelten Erfahrungen sei jedoch davor gewarnt, um jeden Preis bereits vorhandene Module in modifizierter Form wieder verwenden zu wollen. Gerade bei hochspezialisierten Komponenten gelingt eine Anpassung an eigene Bedürfnisse nicht immer zufrieden stellend oder aber der dafür nötige Aufwand übersteigt den für die Entwicklung neuer Bauteile unter Umständen deutlich.

Anhang A

Handbuch

Um dem Administrator einen reibungslosen Einstieg zu ermöglichen, wurde ein kleines Handbuch erstellt, wie das System zu bedienen ist.

A.1 Installation

Systemanforderungen:

Server:

- PC mit Betriebssystem Windows 2000 oder höher
- installiertes Microsoft *.NET-Framework*
- 1 MB freier Speicherplatz
- Wireless LAN mit installiertem TCP/IP-Protokoll

Client:

- Pocket PC mit Windows CE
- 5 MB freier Speicherplatz
- Wireless LAN mit installiertem TCP/IP-Protokoll
- Bluetooth mit installiertem *Microsoft Bluetooth Stack*

Um die Anwendung zu installieren, genügt es, die Verzeichnisse EnvBuilder/bin/Release und NavAssist-Server/bin/Release auf den Server beziehungsweise PersAssist/bin/Release auf den Client zu kopieren. Konfiguration und Ausführung werden im Folgenden beschrieben:

A.2 Environment Builder GUI

Gestartet wird der *Environment Builder* mit einem Doppelklick auf die Datei EnvBuilder.exe. Diese GUI (Abbildung A.1) dient zum Erstellen, Anzeigen und Modifizieren von Gebäudeplänen.

In der Befehlsleiste unter ”File” (siehe Abb. A.2) besteht für den Benutzer die Möglichkeit, vorhandene Umgebungspläne im XML-Format zu laden, zu speichern und zu schließen. Unter ”Set Reference Points” können die beiden Referenzpunkte, die für die Koordinatentransformation (vgl. Abschnitt 6.3) notwendig sind, gesetzt werden. Die Höhenangaben und Namen der betreffenden Stockwerke gibt man in dem

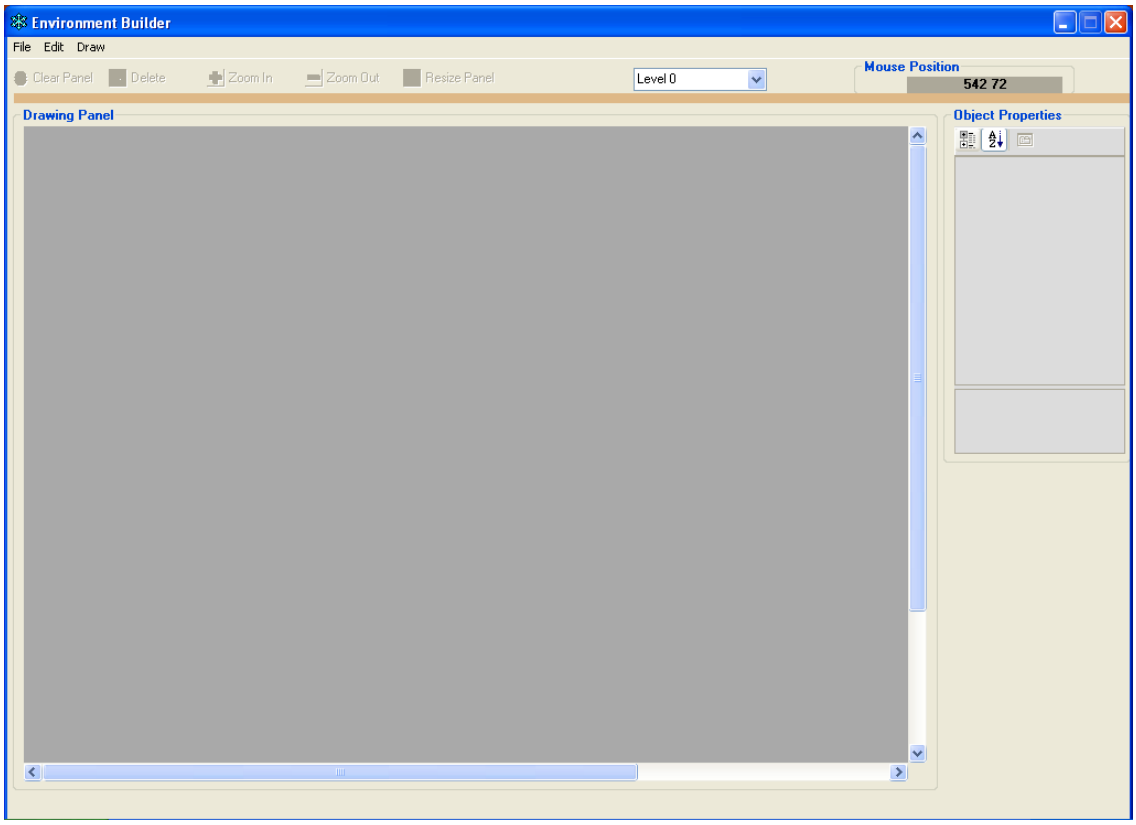

Abbildung A.1: leere Benutzeroberfläche von Environment Builder

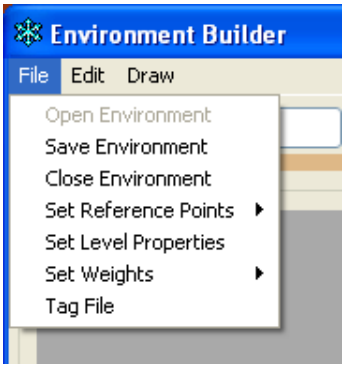

Abbildung A.2: die Befehlsleiste ”File”.

Menüunterpunkt ”Set Level Properties” ein. Dazu kann der Administrator unter ”Set Weights” die allgemeingültigen Gewichte von Treppe und Lift manuell eintragen.

Mit Hilfe des Befehls ”Edit” kopiert und fügt man einzelne Objekte ein.

Das Menü ”Draw” (Abb. A.3) ist für das Zeichnen von verschiedenen Objekten zuständig, wobei berücksichtigt werden muss, dass Treppen und Aufzüge Teil keine selbständige Objekte sind, d.h. man hat sie in Räume bzw. Korridore einzuzeichnen. Um zwei Räume miteinander zu verbinden, kann eine Türe (*Door*) auf der Zeichenfläche erzeugt werden und anschließend hat man die Möglichkeit, sie im Eigenschaftsmenü auf der rechten Seite durch Ersetzen von ”Door” zu ”opening” in eine Öffnung umzuwandeln. Neue Stockwerke können ebenfalls in diesem Menü erstellt werden.

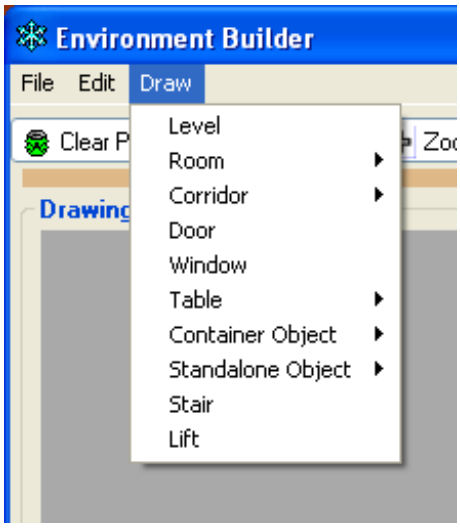

Abbildung A.3: die Befehlsleiste ”Draw”.

HINWEIS: Beim Zeichnen eines mehrstöckigen Gebäudeplans ist zu berücksichtigen, dass man immer mit dem am tiefsten gelegenen Stockwerk anzufangen hat. Will man jedoch mit einem höher gelegenem beginnen, muss eine entsprechende Anzahl von leeren Etagen erzeugt werden, denn späteres Löschen bzw. Hinzufügen neuer Level zwischen bereits vorhandenen ist nicht möglich.

Die Toolbar (Abb. A.4) stellt Funktionen für das Löschen von markierten Objekten und das Zurücksetzen der Zeichenfläche sowie für das Vergrößern und Verkleinern der Pläne zur Verfügung. Außerdem kann man hier die Größe der Zeichenoberfläche einstellen.

Abbildung A.4: die Toolbar.

Weiter rechts neben der Toolbar ist ein Drop-Drown-Menü, wo der Name des aktuellen Stockwerks eingetragen steht, diese macht auch einen Wechsel zwischen den Etagen möglich.

Im Feld noch weiter rechts wird die gegenwärtige Position des Mauszeigers angezeigt, allerdings nur, wenn dieser nicht über ein Objekt liegt.

Am rechten Rand der GUI (Abbildung A.5) erkennt man, wenn man ein entsprechendes Objekt angeklickt hat, seine Eigenschaften. Wofür die einzelnen Einträge stehen, wird bei nochmaligem Anklicken der Objekteigenschaften weiter unten preisgegeben.

HINWEIS: Beim Zeichnen des Gebäudeplanes ist folgender Maßstab einzuhalten: 100 Pixel auf der Zeichenfläche entsprechen 2,2 Meter, wobei das Setzen dieses Wertes direkt im Feld *ObjectProperties* möglich ist.

Weitere Erläuterungen kann man aus der originalen Arbeit [Miz04] entnehmen.

A.3 Anpassung des Servers

HINWEIS: Möglicherweise führt das Starten des Servers zum Absturz des Systems, wenn eine Firewall installiert und aktiv ist. Daher wird an dieser Stelle empfohlen, bevor der Server gestartet wird, die Firewall zu deaktivieren.

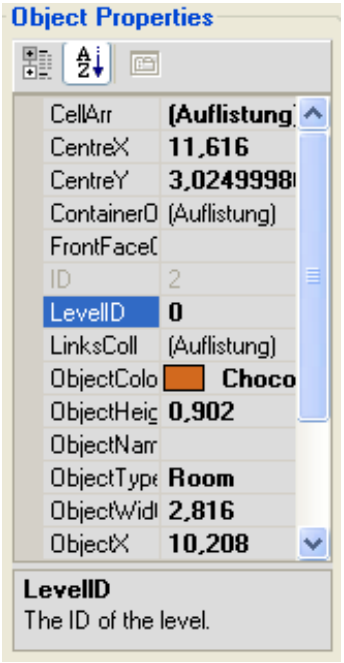

Abbildung A.5: die Objekteigenschaften.

Der Server wird mit Ausführen von NavAssistServer.exe gestartet. Vorher hat man die XML-Konfigurationsdatei *configuration.xml*, die wie folgt aufgelistet ist, korrekt anzupassen:

```
<?xml version="1.0" encoding="utf-8" ?>
<serverApp>
<serverIPAddress>127.0.0.1</serverIPAddress>
<serverPort>2040</serverPort>
<envDataFile>Data\VerySchönerGebäudeplan.xml</envDataFile>
<rulesFile>Data\rules.txt</rulesFile>
<lexiconFile>Data\lexicon.txt</lexiconFile>
<LPS>local</LPS>
<simulation>N</simulation>
</serverApp>
```

Dabei muss die IP und die Portnummer des Servers angegeben werden, der korrekte Pfad zum verwendeten Gebäudeplan, sowie im Falle einer Textausgabe die Pfade zu den beiden Dateien rules.txt und lexicon.txt. Letztere können unberücksichtigt bleiben, wenn man grafische Symbole als Navigationsinstruktionen bevorzugt. Die restlichen Einträge werden im aktuellen System nicht mehr ausgewertet, sie sind bedeutungslos.

A.4 Anpassung des Clients

Auf dem Pocket PC PDA muss der mobile Client installiert und seine *configuration.xml* ebenfalls angepasst werden:

```
<?xml version="1.0" encoding="utf-8" ?>
<clientapp>
<settings>
<ServerIPAddress>127.0.0.1</ServerIPAddress>
<ServerPort>2040</ServerPort>
```

```
6 <ConnectionType>wireless</ConnectionType>
  <Segmentation>false</Segmentation>
8 <VerifyPos>false</VerifyPos>
  <LPS>foreign</LPS>
10 <bluetoothDeviceName>NASBluetooth</bluetoothDeviceName> //LÖSCHEN?
  <LPSBluetoothDeviceName>lapmsp2</LPSBluetoothDeviceName>
12 </settings>
  </clientapp>
```

Die Server-IP und sein Port muss bekannt sein, dazu auch der Übertragungsmodus, wobei beim aktuellen System nur TCP/IP zur Verfügung steht (*wireless*). Auch hier haben die übrigen Einträge keine Bedeutung mehr mit Ausnahme von *LPSBluetoothDeviceName*, wo der Name des LPS einzutragen ist, von dem die Indoor-Positionsdaten bezogen werden sollen.

Anhang B

Codeausschnitte

B.1 Methoden der Klasse Stair

B.1.1 setRooms()

```
public string setRooms(EnvironmentCollection envColl)
{
 ObjectsCollection lowerLevel = envColl.getByLevelID(this.LevelID);
 ObjectsCollection upperLevel = envColl.getByLevelID(this.LevelID + 1);
 Point centre = new Point((this.ObjectLocation.X + (this.ObjectSize.Width/2)),
(this.ObjectLocation.Y + (this.ObjectSize.Height/2)));
//centre of stair object in pixel
 bool found = false;

 foreach (BaseObject obj in lowerLevel)
 {
  //if the object is a space
  if ((obj.GetType().BaseType != null) &&
(obj.GetType().BaseType.Equals(typeof(Space))))
  {
  Rectangle objRect = new Rectangle(obj.ObjectLocation,obj.ObjectSize);
  if ((objRect.Contains(centre)==true) && (obj.Equals(this) == false))
  {
   Space space = new Space();
   //remove stair from previous space
   //because it could have been moved
    if (this.LowerRoom != null)
    {
     space = (Space) LowerRoom;
     if (space.LinksColl.Contains(this))
     {
      space.LinksColl.Remove(this);
     }
    }
    //then add stair to new space
    space = (Space) obj;
    if (space.LinksColl.Contains(this) == false)
    {
     space.LinksColl.Add(this);
    }
```

```
36        //store pointer to new space
          this.LowerRoom = obj;
38        this.LowerRoomID = obj.ID;
          this.LowerLevelID = obj.LevelID;
40        found = true;
        }
42    }
    }
44  //if lower room doesn't exist
    if (!found)
46  {
      //store as a null pointer
48    this.LowerRoom = null;
      this.LowerRoomID = -255;
50  }
    //if upper level exist
52  if (upperLevel != null)
    {
54    //if upper level exist but is empty
      if (upperLevel.Count == 0)
56    {
        //store as a null pointer
58      this.UpperRoom = null;
        this.UpperRoomID = -255;
60      this.UpperLevelID = upperLevel.LevelID;
      }
62    //else upper level is not empty
      else
64    {
        found = false;
66      foreach (BaseObject obj in upperLevel)
        {
68        //if the object is a space
          if ((obj.GetType().BaseType != null) &&
70  (obj.GetType().BaseType.Equals(typeof(Space))))
          {
72          Rectangle objRect = new Rectangle(obj.ObjectLocation,obj.ObjectSize);
            if ((objRect.Contains(centre)==true) && (obj.Equals(this) == false))
74          {
              Space space = new Space();
76            //remove stair from previous space
              //because it could have been moved
78            if (this.UpperRoom != null)
              {
80              space = (Space) UpperRoom;
                if (space.LinksColl.Contains(this))
82              {
                  space.LinksColl.Remove(this);
84              }
              }
86            //then add stair to new space
              space = (Space) obj;
88            if (space.LinksColl.Contains(this) == false)
              {
90              space.LinksColl.Add(this);
              }
92            //store pointer to new space
              this.UpperRoom = obj;
```

```
94          this.UpperRoomID = obj.ID;
            this.UpperLevelID = obj.LevelID;
96          found = true;
          }
98       }
       }
100    //if upper room doesn't exist
       if (!found)
102    {
        //store as a null pointer
104     this.UpperRoom = null;
        this.UpperRoomID = -255;
106     this.UpperLevelID = upperLevel.LevelID;
       }
108   }
      //move stair on top of the level
110   if (!upperLevel.Contains(this))
      {
112    upperLevel.Add(this);
      }
114   else
      {
116    upperLevel.Remove(this);
       upperLevel.Add(this);
118   }
    }
120 //if upper level doesn't exist
    else
122 {
     //store as a null pointer
124  this.UpperRoom = null;
     this.UpperLevelID = -255;
126  this.UpperRoomID = -255;
    }
128 return "ok";
}
```

B.1.2 fillRooms()

```
public void fillRooms(EnvironmentCollection envColl)
2 {
  ObjectsCollection objColl1 = envColl.getByLevelID(lowerLevelId);
4 if (lowerRoomId != -255)
  {
6  BaseObject obj1 = objColl1.getObjectWhoseID(this.LowerRoomID);
   this.LowerRoom = obj1;
8
   Space space1 = (Space) obj1;
10 if (space1.LinksColl == null)
   {
12  space1.LinksColl = new LinksCollection();
   }
14 space1.LinksColl.Add(this);

16 if ((upperLevelId != -255) && (upperRoomId != -255))
   {
18  ObjectsCollection objColl2 = envColl.getByLevelID(upperLevelId);
    BaseObject obj2 = objColl2.getObjectWhoseID(this.UpperRoomID);
```

```
20      this.UpperRoom = obj2;

22      Space space2 = (Space) obj2;
        if (space2.LinksColl == null)
24      {
          space2.LinksColl = new LinksCollection();
26      }
        space2.LinksColl.Add(this);
28    }
      else //case of no upper room exists (i.e. upper room id = -255)
30    {
        this.UpperRoom = null;
32    }
    }
34  else //case of no lower room exists (i.e. lower room id = -255)
    {
36    this.LowerRoom = null;
    }
38 }
```

B.2 Methoden der Klasse SearchCls

B.2.1 Auszug aus adjacentToCellsFromDifferentSpace()

```
  if (obj.GetType().Equals(typeof(Stair)))
2 {
    Stair stair = (Stair) obj;
4   if (stair.LevelID == objColl.LevelID)
    {
6     Space lowerRoom = (Space) stair.LowerRoom;
      ArrayList lowerRoomCells = new ArrayList();
8     if (lowerRoom != null)
      {
10      lowerRoomCells = lowerRoom.CellArr;
      }
12    else
      {
14      lowerRoomCells = null;
      }
16    Space upperRoom = (Space) stair.UpperRoom;
      ArrayList upperRoomCells = new ArrayList();
18    if (upperRoom != null)
      {
20      upperRoomCells = upperRoom.CellArr;
      }
22    else
      {
24      upperRoomCells = null;
      }
26    //check the cells of the two different spaces for adjacency
      if (lowerRoomCells != null)
28    {
        for (int i = 0; i < lowerRoomCells.Count; i++)
30    {
        cellCls cellClass = (cellCls) lowerRoomCells[i];
32      if (upperRoomCells != null)
        {
```

```
34       for (int j = 0; j < upperRoomCells.Count; j++)
       {
36        cellCls cellClass2 = (cellCls) upperRoomCells[j];
         if (cellClass.LinksColl != null)
38         {
           if (cellClass2.LinksColl != null)
40           {
             if (cellClass.LinksColl.Contains(stair) && cellClass2.LinksColl.
42 Contains(stair))
               {
44               //if it is not in adjacency list yet
                 //add it to its adjacency list
46               if (cellClass.AdjacencyList == null)
                 {
48                 cellClass.AdjacencyList = new ArrayList();
                 }
50               if (cellClass2.AdjacencyList == null)
                 {
52                 cellClass2.AdjacencyList = new ArrayList();
                 }
54               //adjust the adjacency list of cellClass
                 if (!cellClass.AdjacencyList.Contains(cellClass2))
56               {
                   cellClass.AdjacencyList.Add(cellClass2);
58                 lowerRoom.CellArr = cellClass.saveCellWhoseID(cellClass,
  lowerRoom.CellArr);
60               }
                 //adjust the adjacency list of cellClass2 too
62               if (!cellClass2.AdjacencyList.Contains(cellClass))
                 {
64                 cellClass2.AdjacencyList.Add(cellClass);
                   upperRoom.CellArr = cellClass2.saveCellWhoseID(cellClass2,
66 upperRoom.CellArr);
                 }
68             }
             }
70         }
         }
72       }
     }
74   }
  }
76 }
  if (obj.GetType().Equals(typeof(Lift)))
78 {
   Lift lift = (Lift) obj;
80 if (lift.LevelID == objColl.LevelID)
  {
82  for (int i = 0; i < lift.RoomColl.Count - 1; i++)
    {
84    Space lowerSpace = (Space) lift.RoomColl[i];
      ArrayList lowerSpaceCells = lowerSpace.CellArr;
86    Space upperSpace = (Space) lift.RoomColl[i + 1];
      ArrayList upperSpaceCells = upperSpace.CellArr;
88    //check the cells of the two different spaces for adjacency
      for (int j = 0; j < lowerSpaceCells.Count; j++)
90    {
       cellCls cellClass = (cellCls) lowerSpaceCells[j];
```

```
92        for (int k = 0; k < upperSpaceCells.Count; k++)
          {
94          cellCls cellClass2 = (cellCls) upperSpaceCells[k];
            if (cellClass.LinksColl != null)
96          {
              if (cellClass2.LinksColl != null)
98            {
                if (cellClass.LinksColl.Contains(lift) && cellClass2.LinksColl.
100 Contains(lift))
              {
102             //if it is not in adjacency list yet
                //add it to its adjacency list
104             if (cellClass.AdjacencyList == null)
                {
106               cellClass.AdjacencyList = new ArrayList();
                }
108             if (cellClass2.AdjacencyList == null)
                {
110               cellClass2.AdjacencyList = new ArrayList();
                }
112             //adjust the adjacency list of cellClass
                if (!cellClass.AdjacencyList.Contains(cellClass2))
114             {
                  cellClass.AdjacencyList.Add(cellClass2);
116               lowerSpace.CellArr = cellClass.saveCellWhoseID(cellClass,
lowerSpace.CellArr);
118             }
                //adjust the adjacency list of cellClass2 too
120             if (!cellClass2.AdjacencyList.Contains(cellClass))
                {
122               cellClass2.AdjacencyList.Add(cellClass);
                  upperSpace.CellArr = cellClass2.saveCellWhoseID(cellClass2,
124 upperSpace.CellArr);
                }
126           }
            }
128       }
        }
130   }
    }
132 }
}
```

B.2.2 getRequiredPathUsingDijkstra()

```
public ArrayList getRequiredPathUsingDijkstra(Point3D pt1, Point3D pt2,
2 EnvironmentCollection envColl, ArrayList edgeArr, int noOfPoints)
{
4  //overwrite noOfPoints
   this.noOfPoints = noOfPoints;
6  //get the required pts
   ArrayList ptsArr = this.getRequiredPts(pt1, pt2, envColl);
8  //compute path using shortest path algo
   ArrayList predArr = this.dijkstra((PtUsedInPathCls) ptsArr[0], (PtUsedInPathCls)
10 ptsArr[1], edgeArr);
   //return final path
12 ArrayList finalPath = new ArrayList();
   this.pathFromPred(predArr, (PtUsedInPathCls)ptsArr[0], (PtUsedInPathCls) ptsArr[1],
```

```
14 finalPath);
   return finalPath;
16 }
```

B.2.3 dijkstra()

```
   //Dijkstra algorithm for path finding
2  public ArrayList dijkstra(PtUsedInPathCls startPt, PtUsedInPathCls destPt, ArrayList
   edgeArr)
4  {
   ArrayList nodeQueue = new ArrayList(); //contains the nodes who have to be visited
6                                                          //next
   ArrayList predecessor = new ArrayList(); //list of predecessors
8  ArrayList color = new ArrayList(); //marks if a node is already visited (black),
                                                  //have to (grey) or not yet (white)
10 ArrayList distance = new ArrayList(); //contains the distance of a node to the
                                                    //starting point
12
   for (int i=0; i <= noOfPoints; i++)
14 {
    predecessor.Add(-1);
16  distance.Add(-1.0d);
    color.Add("white");
18 }
   int node = 0;
20 int childnode = 0;
   PtUsedInPathCls fnode = new PtUsedInPathCls(); //the actually visited node
22 PtUsedInPathCls fchildnode = new PtUsedInPathCls(); //the next node to visit

24 color[startPt.ID] = "gray";
   predecessor[startPt.ID] = -1;
26 distance[startPt.ID] = 0.0d;
   //add the start point to the unvisited nodes
28 nodeQueue.Add(startPt);

30 while ((nodeQueue.Count != 0) && (fchildnode.Equals(destPt) == false))
   {
32  fnode = extractMin(nodeQueue,distance);
    node = fnode.ID;
34  //if nothing found, break
    if (node == 0)
36  {
     break;
38  }
    nodeQueue.Remove(fnode);
40
    //get the adjacenct nodes of fnode
42  ArrayList adjArr = AdjacencyList(fnode,edgeArr);
    double dist = 0.0d;
44  //fill the nodeQueue with the adjacent nodes (fchildnodes)
    //of the current node (fnode)
46  for (int i=0; i<adjArr.Count; i++)
    {
48   fchildnode = (PtUsedInPathCls) adjArr[i];
     childnode = fchildnode.ID;
50   string colr = (string) color[childnode];
     //if it hasnt been visited
52   if (colr.Equals("white"))
```

```
      {
54      color[childnode] = "gray";
        dist = (double) distance[node] + getWeight(fnode, fchildnode, edgeArr);
56      if ((dist < (double) distance[childnode]) || ((double) distance[childnode] ==
-1.0d))
58      {
          distance[childnode] = dist;
60        predecessor[childnode] = fnode;
        }
62      nodeQueue.Add(fchildnode);
      }
64    }
      color[node] = "black";
66  }
    if (nodeQueue.Count == 0)
68  {
      //MessageBox.Show("Path not found");
70  }
    return predecessor;
72 }
```

B.2.4 getWeight()

```
public double getWeight(PtUsedInPathCls pt1, PtUsedInPathCls pt2, ArrayList edgeArr)
2 {
    for (int i = 0; i < edgeArr.Count; i++)
4   {
      PathClass path = (PathClass) edgeArr[i];
6     if (((path.Pt1.ID == pt1.ID) && (path.Pt2.ID == pt2.ID)) || ((path.Pt1.ID == pt2.ID)
&& (path.Pt2.ID == pt1.ID)))
8     {
        return path.Weight;
10    }
    }
12  return -1.0d;
}
```

B.3 Methoden der Klasse TextPlanner

B.3.1 finalizePath()

```
private ArrayList finalizePath(ArrayList sourcePath)
2 {
    ArrayList finalizedPath = new ArrayList();
4   PtUsedInPathCls prevPt = null;
    PtUsedInPathCls currPt = null;
6
    for (int i = 0; i < sourcePath.Count; i++)
8   {
      currPt = (PtUsedInPathCls) sourcePath[i];
10    //if not first point of path
      if (prevPt != null)
12    {
        //if current point is on the same level as the previous
14      if (currPt.ptZ == prevPt.ptZ)
        {
```

```
16        finalizedPath.Add(currPt);
          prevPt = currPt;
18      }
        else //if they r not on the same level
20      {
          //test if it is a lift
22        LevelLink link = (LevelLink) this.getNextLevelLink(sourcePath, prevPt);
          if (link.GetType().Equals(typeof(Lift)))
24        {
            while (currPt.ptZ != prevPt.ptZ)
26          {
              prevPt = currPt;
28            i++;
              currPt = (PtUsedInPathCls) sourcePath[i];
30          }
            finalizedPath.Add(prevPt);
32          finalizedPath.Add(currPt);
          }
34        else //else if its a stair
          {
36          finalizedPath.Add(currPt);
            prevPt = currPt;
38        }
        }
40    }
      else //if its the first point of path
42    {
        finalizedPath.Add(currPt);
44      prevPt = currPt;
      }
46  }
    return finalizedPath;
48 }
```

B.3.2 getNextSpace()

```
    public BaseObject getNextSpace(ArrayList finalPath, LevelLink nextLink,
2   PtUsedInPathCls currPt)
    {
4     int index = finalPath.IndexOf(currPt);

6     if (nextLink.GetType().Equals(typeof(Stair)))
      {
8       if (finalPath.Count > index+1)
        {
10        PtUsedInPathCls nextPt = (PtUsedInPathCls) finalPath[index+1];
          Space nextSpace = (Space) nextPt.obj;
12        return nextSpace;
        }
14    }
      else if (nextLink.GetType().Equals(typeof(Lift)))
16    {
        bool stop = false;
18      while (!stop)
        {
20        stop = true;
          if (finalPath.Count > index+1)
22        {
```

```
      PtUsedInPathCls nextPt = (PtUsedInPathCls) finalPath[index+1];
      if (currPt.ptZ != nextPt.ptZ)
      {
        currPt = nextPt;
        index++;
        stop = false;
      }
      else
      {
        Space nextSpace = (Space) currPt.obj;
        return nextSpace;
      }
    }
  }
 }
 return null;
}
```

B.3.3 Ausschnitt aus pathProcessing()

```
if ((nextLink != null) && (nextSpace != null))
{
 //if the space of the pt is a stairs do StairsMsg
 if (nextLink.GetType().Equals(typeof(Stair)))
 {
  StairsMsg stairsmsg = new StairsMsg();
  stairsmsg.stairs = (Stair) nextLink;
  //get largest id
  stairsmsg.ID = largestID + 1;
  ++largestID;
  //find out direction
  if (firstPathPt.obj.ObjectZ < nextSpace.ObjectZ)
  {
   stairsmsg.direction = "up";
  }
  else
  {
   stairsmsg.direction = "down";
  }
  ObjDescMsg objDescMsg = new ObjDescMsg();
  objDescMsg = objDescMsg.getObjDescOfObject(firstPathPt.obj);
  objDescMsg.type = "Stair";
  objDescMsg.name = "Stair";
  stairsmsg.msgArr.Add(objDescMsg);
  otherMsgArr.Add(stairsmsg);
  stairsCheck = true;
 }
 else if (nextLink.GetType().Equals(typeof(Lift)))
 {
  LiftMsg liftmsg = new LiftMsg();
  liftmsg.lift = (Lift) nextLink;
  liftmsg.nextSpace = (Space) nextSpace;
  //get largest id
  liftmsg.ID = largestID + 1;
  ++largestID;
  //find out direction
  if (firstPathPt.obj.ObjectZ < nextSpace.ObjectZ)
  {
```

```
      liftmsg.direction = "up";
40    }
      else
42    {
       liftmsg.direction = "down";
44    }
      ObjDescMsg objDescMsg = new ObjDescMsg();
46    objDescMsg = objDescMsg.getObjDescOfObject(firstPathPt.obj);
      objDescMsg.type = "floor "+nextSpace.LevelID;
48    objDescMsg.name = "floor "+nextSpace.LevelID;
      liftmsg.msgArr.Add(objDescMsg);
50    otherMsgArr.Add(liftmsg);
      liftCheck = true;
52   }
  }
```

B.4 Methoden der Klasse GuidingServerCls

B.4.1 Aussschnitt aus ClientRead()

```
  if (parsedmsg[0] == "NMEA")
2 {
   //parse the message received
4  string[] nmeamsg = parsedmsg[1].Split(new Char[] {',’});
   //replace "." with ","
6  StringBuilder strBuilder = new StringBuilder(nmeamsg[2]);
   strBuilder.Replace(".", ",");
8  double latitude = double.Parse(strBuilder.ToString());
   if (nmeamsg[3].Equals("S"))
10 {
    latitude = latitude * (-1);
12 }
   //replace "." with ","
14 strBuilder = new StringBuilder(nmeamsg[4]);
   strBuilder.Replace(".", ",");
16 double longitude = double.Parse(strBuilder.ToString());
   if (nmeamsg[5].Equals("W"))
18 {
    longitude = longitude * (-1);
20 }
   //replace "." with ","
22 strBuilder = new StringBuilder(nmeamsg[9]);
   strBuilder.Replace(".", ",");
24 double sealevel = double.Parse(strBuilder.ToString());
   //transform the gps-coordinates into pixel-coordinates
26 CoordTrans coordTrans = new CoordTrans();
   double latInDecimal = coordTrans.convert(latitude);
28 double longInDecimal = coordTrans.convert(longitude);
   //transform latInDecimal and longInDecimal to pixel
30 Point currentPos = coordTrans.transform(envColl.GpsPt1_x, envColl.GpsPt1_y,
envColl.PixelPt1, envColl.GpsPt2_x, envColl.GpsPt2_y, envColl.PixelPt2,
32 latInDecimal, longInDecimal);
   //create the corresponding 3D point
34 Point3D current3DPos = new Point3D(currentPos.X, currentPos.Y, (float) sealevel);
   //tell the navAssist the current user position
36 state.navAssist.userPos = current3DPos;
   //get the nmea checksum
```

```
38  string nmeaChecksum = nmeamsg[14];
    //cut the character "*" of the string
40  //calculate the checksum
    string calculatedChecksum = this.getChecksum(parsedmsg[1]);
42  //reply to the client
    if (calculatedChecksum == nmeaChecksum)
44  {
     Send(serverHandler, "OK#");
46  }
    else Send(serverHandler, "ERROR#");
48 }
```

B.4.2 getChecksum()

```
  //calculates the nmea checksum
2 public string getChecksum(string sentence)
  {
4  // Loop through all chars to get a checksum
   int Checksum = 0;
6  foreach (char Character in sentence)
   {
8   if (Character == '$')
    {
10   // Ignore the dollar sign
    }
12  else if (Character == '*')
    {
14   // Stop processing before the asterisk
     break;
16  }
    else
18  {
     // Is this the first value for the checksum?
20   if (Checksum == 0)
     {
22    // Yes. Set the checksum to the value
      Checksum = Convert.ToByte(Character);
24   }
     else
26   {
      // No. XOR the checksum with this character's value
28    Checksum = Checksum ^ Convert.ToByte(Character);
     }
30  }
   }
32 // Return the checksum formatted as a two-character hexadecimal
   return Checksum.ToString("X2");
34 }
```

B.5 Methoden der Klasse NavAssistantCls

B.5.1 getDirectionExpression()

```
public string getDirectionExpression(MessageCls msgCls)
2 {
  string msg = "";
```

```
 msg = msgCls.ToString().ToLower();
 if ((msg.IndexOf("across") != -1) || (msg.IndexOf("along") != -1) ||
(msg.IndexOf("straight") != -1) || (msg.IndexOf("end") != -1))
 {
  return "STRAIGHT";
 }
 else if (msg.IndexOf("through") != -1)
 {
  return "DOOR";
 }
 else if (msg.IndexOf("right") != -1)
 {
  return "RIGHT";
 }
 else if (msg.IndexOf("left") != -1)
 {
  return "LEFT";
 }
 else if (msg.IndexOf("stair") != -1)
 {
  if (msg.IndexOf("up") != -1)
  {
   return "STAIRSUP";
  }
  else return "STAIRSDOWN";
 }
 else if (msgCls.GetType().Equals(typeof(LiftMsg)))
 {
  LiftMsg liftMsg = (LiftMsg) msgCls;
  int floorId = liftMsg.nextSpace.LevelID;
  if (liftMsg.direction.Equals("up"))
  {
   return "LIFTUP|" + floorId;
  }
  else return "LIFTDOWN|" + floorId;
 }
 else if (msg.IndexOf("you are in") != -1)
 {
  return "SMILE";
 }
 else return null;
}
```

B.6 Methoden der Klasse assistantClientCls

B.6.1 getNMEAPos()

```
//used for connection to the LPS
//gets the NMEA string and returns it
public string getNMEAPos()
{
 string nmea = "";

 try
 {
  lpsSocket = this.connect(lpsBTDeviceName); //connect to BT device
  if (lpsSocket == 1) //we have got an error if the value of lpsSocket = 1
```

```
    {
12    return "connFailed";
    }
14  while (nmea.IndexOf("$GPGGA") == -1) //get the correct NMEA string
    {
16    nmea = "";
      while (nmea.IndexOf("\n") == -1) //while the end of the string hasnt been reached
18    {
        nmea += bluetoothDll.readRequest(lpsSocket);
20    }
    }
22
    bluetoothDll.closeConnection(lpsSocket); //close the connection
24  return nmea;
  }
26 catch (SocketException ex)
  {
28  bluetoothDll.closeConnection(localSocket); //close the connection
    return "connFailed";
30 }
}
```

Literaturverzeichnis

[Ama04] AMATO, NANCY: *Approximate Cell Decomposition Methods*. Verfügbar im World Wide Web, 2004. http://parasol.tamu.edu/ amato/Courses/padova04/lectures/L4.approx-decomp.pdf.

[ASt05] WWW.GEOSIMULATION.DE: *Routenoptimierung / Kostenbasierte Suche / A-Stern Algorithmus*. Verfügbar im World Wide Web, August 2005. http://www.geosimulation.de/umsetzungen/Beschreibungen/Routenoptinierung.htm.

[Bau01] JÖRG BAUS, CHRISTIAN KRAY, ANTONIO KRÜGER UND WOLFGANG WAHLSTER: *A resource-adaptive mobile navigation system*. 2001.

[Ber01] BERNREUTHER, M.: *Kürzester-Weg-Suche mit dem Dijkstra-Algorithmus*. Verfügbar im World Wide Web, 2001. http://www.uni-stuttgart.de/iv-kib/generic/download/lehre/einfuehrung/ws00_01/vorlesung/mb_dijkstra.pdf.

[Col94] COLLBERG, CHRISTIAN: *Natural Language*. Verfügbar im World Wide Web, October 1994. www.cs.arizona.edu/ collberg/Teaching/07.363/Handouts/handout14.ps.

[Dij05] WIKIPEDIA: *Edsger Wybe Dijkstra*. Verfügbar im World Wide Web, October 2005. http://de.wikipedia.org/wiki/Dijkstra.

[Dot06] MICROSOFT: *Basics of .NET*. Verfügbar im World Wide Web, January 2006. http://www.microsoft.com/net/basics.mspx.

[Dro04] DROBNIK, PROF. DR. O.: *Mobile Computing*. Verfügbar im World Wide Web, 2004. http://www.tm.informatik.uni-frankfurt.de:8080/Plone/Lehre/07_SoSe04/MobileComputingHandouts/MoCo-04-WirelessLAN.

[ELK06] ELEKTRONIK KOMPENDIUM DAS: *IEEE 802.11 / Wireless LAN / Wireless Fidelity (WiFi)*. Verfügbar im World Wide Web, 2006. http://www.elektronik-kompendium.de/sites/net/0610051.htm.

[Gal05] WIKIPEDIA: *Galileo (Satellitennavigation)*. Verfügbar im World Wide Web, August 2005. http://de.wikipedia.org/wiki/Galileo_%28Satellitennavigation%29.

[Gie03] GIER, MARCUS: *Knowledgebase Wireless LAN*. Verfügbar im World Wide Web, Juli 2003. http://www.abcdata.de/wlan_Standards.htm.

[Gm99] GMÜR, CHRIGI: *Bluetooth*. Verfügbar im World Wide Web, 1999. http://www.elektronik-kompendium.de/public/chrigi/bluetooth.htm.

[GPS05] WIKIPEDIA: *Global Positioning System*. Verfügbar im World Wide Web, August 2005. http://de.wikipedia.org/wiki/Global_Positioning_System.

[Hol03] HOLTKAMP, HEIKO: *Einführung in Bluetooth*. Verfügbar im World Wide Web, November 2003. http://www.rvs.uni-bielefeld.de/ heiko/bluetooth/bluetooth.pdf.

[Kra03] CHRISTIAN KRAY, KATRI LAAKSO, CHRISTIAN ELTINIG UND VOLKER COORS: *Presenting route instructions on mobile devices*. 2003.

[Les05] LESTER, PATRICK: *A* Pathfinding for Beginners*. Verfügbar im World Wide Web, July 2005. http://www.policyalmanac.org/games/aStarTutorial.htm.

[Leu97] JOSEF LEUNG, CHING-LONG YEH: *Natural Language Processing for Verbatim Text Coding and Data Mining Report Generation*. Verfügbar im World Wide Web, 1997. www.cse.ttu.edu.tw/chingyeh/papers/esomar.pdf.

[Lis06] LIST, MARKUS: *Drahtlose lokale Netze*. Verfügbar im World Wide Web, 2006. http://www.rn.inf.tu-dresden.de/scripts_lsrn/Lehre/prosem_rn/print/print_ws05-06/Drahtlose_LAN.pdf.

[McD87] MCDONALD, DAVID D.: *Encyclopedia of Artificial Intelligence*, Kapitel Natural language generation, Seiten 642–655. John Wiley and Sons Verlag, New York, NY, 1987.

[Miu02] MIU, ALLEN KA LUN: *Design and Implementation of an Indoor Mobile Navigation System*. Diplomarbeit, Massachusetts Institute of Technology, January 2002.

[Miz04] MIZZI, DOREEN: *A Mobile Navigational Assistance System Using Natural Language Generation*. Diplomarbeit, Department of Computer Science and AI, University of Malta, June 2004.

[Moe04] MICHAEL MÖSER, UNIV. PROF. DR.-ING. HABIL.: *Geodäsie für Verkehrsingenieure*. Verfügbar im World Wide Web, 2004. http://wwwgi.geo.tu-dresden.de/ig/formulare/vl_verkehr.pdf.

[MSP06] MICROSOFT: *Microsoft Platform SDK Update*. Verfügbar im World Wide Web, January 2006. http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdk-full.htm.

[NAS05] WIKIPEDIA: *Navigationssystem*. Verfügbar im World Wide Web, August 2005. http://de.wikipedia.org/wiki/Navigationssystem.

[Nav05] WIKIPEDIA: *Navigation*. Verfügbar im World Wide Web., August 2005. http://de.wikipedia.org/wiki/Navigation.

[Nme06] WIKIPEDIA: *National Marine Electronics Association*. Verfügbar im World Wide Web, January 2006. http://de.wikipedia.org/wiki/NMEA.

[Per06] PERSON, JON: *Writing Your Own GPS Applications: Part 2*. Verfügbar im World Wide Web, February 2006. http://www.codeproject.com/netcf/WritingGPSApplications2.asp.

[Rei97] REITER E., DALE R.: *Building Applied Natural Language Generation System*. Natural Language Engineering, 3(1), 1997.

[Ric06] RICKINGER, THOMAS: *Eine Client/Server-Architektur zur endgerätebasierten Ortung auf Basis von WLAN Scene Analysis*. Diplomarbeit, Institut für Informatik, Ludwig-Maximilian-Universität, München, March 2006.

[Tom05] TOMTOM: *TomTom, tragbare GPS-Fahrzeugnavigationssysteme*. Verfügbar im World Wide Web, August 2005. http://www.tomtom.com.

[Wah01] WOLFGANG WAHLSTER, JÖRG BAUS, CHRISTIAN KRAY UND ANTONIO KRÜGER: *REAL: Ein ressourcenadaptierendes mobiles Navigationssystem*. 2001.