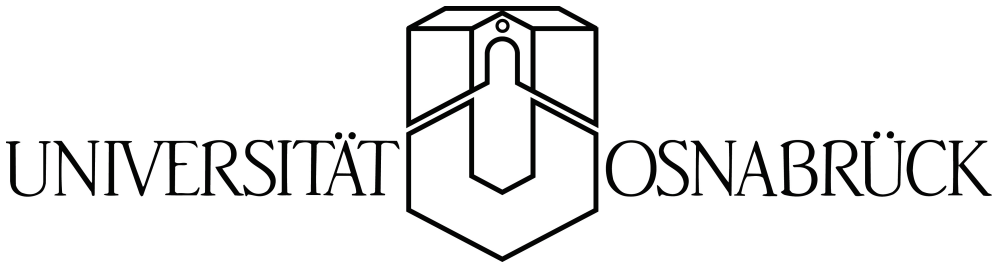

Börsendaten per Digital Audio Broadcasting

Bachelorarbeit
von
Denis Janak Bose

Gutachter:
Prof. Dr. Oliver Vornberger
Prof. Dr.-Ing. Werner Brockmann

Fachbereich Mathematik/Informatik
Universität Osnabrück

09.09.2006

Danksagung

Ich möchte mich bei allen bedanken, die mich bei dieser Bachelorarbeit unterstüzt haben. Insbesondere bei:

- Herrn Prof. Dr. Oliver Vornberger für die Betreuung und besonders für die sehr interessante Aufgabenstellung.
- Patrick Fox für die gute Betreuung und das Korrekturlesen der Arbeit.
- Herrn Prof. Dr.-Ing. Werner Brockmann für die Tätigkeit als Zweitgutachter.
- Ingrid Bose und Carola Bose für das Korrekturlesen der Arbeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Seit schon bald 100 Jahren gibt es das Radio. In der Zwischenzeit gab es einige technische Verbesserungen, dazu gehören beispielsweise die Übertragung des Analogsignals in Stereoqualität oder das *Radio Data System* (RDS), welches 1988 eingeführt wurde und ein erster digitaler Übertragungskanal ist. Mit Hilfe des RDS werden dem Radioempfäner digitale Daten übertragen, die zum Beispiel die Namen der Radiosender enthalten oder Informationen, die dem Empfänger ermöglichen, beim Wechesel eines Sendegebiets, ohne Unterbrechung auf den neuen Sendeplatz des zuvor gehörten Radiosenders automatisch umzuschalten.

Die neueste technische Weiterentwicklung ist die vollständige Digitalisierung des Radios. Derzeit gibt es weltweit zwei verschiedene Ansätze. In Europa und auch in Deutschland wird ein System verwendet, das *Digital Audio Broadcasting* (DAB) genannt wird. Neben dem digitalen Audiosignal, welches Radio in CD-Qualität ermöglicht, können fast beliebige Daten - es gibt Einschränkungen in der Größe und der Anzahl - über das DAB-Signal transportiert werden. Dem Fachbereich Medieninformatik der Universität wurde von der DRN Digital Radio Nord GmbH, die das digitale Sendernetz in Norddeutschland betreibt, eine geringe Übertragungsrate innerhalb des DAB-Signals zu Verfügung gestellt, um Daten zu übertragen. Diese Möglichkeit wird in dieser Bachelorarbeit genutzt.

1.1 Aufgabenstellung

Ziel dieser Arbeit ist die Implementation eines Protokolls, mit dem Dateien über DAB verschikt werden. Das sogennante *Multimedia Object Transfer Protocol* (MOT) legt fest, wie Dateien in das DAB-Signal eingebettet werden. Um das implementierte MOT-Protokoll zu testen, werden Börsendaten über das DAB-Signal verschickt. Dazu sind zwei Applikationen erforderlich.

Die Anwendung auf der Server Seite holt regelmäßig (ca. einmal in der Minute) Börsendaten von einem frei zugänglichem Web Service [Moka2006] im Internet und lädt die Daten dann auf einen speziellen Server der DRN Digital Radio Nord GmbH. Von dort werden sie in das norddeutsche DAB-Signal eingebettet. Dieser technische Vorgang ist zwar Grundlage, jedoch nicht Gegenstand dieser Arbeit. In der anderen Anwendung wird ein Teil des DAB-Signals decodiert und die daraus gewonnenen Kurse grafisch dargestellt.

Der „Empfänger“ - die Anwendung läuft nicht direkt an einem DAB-Receiver, sondern erhält die Daten von einem speziellen Web Server - besteht aus dem so genannten MOT-Decoder, der einen MOT *Stream* decodiert, d. h. aus einzelnen DAB-Paketen wieder ganze Dateien zusammensetzt, die empfangenen Dateien interpretiert und die Dateien, die die Aktienkurse enthalten, grafisch darstellt.

Nachfolgend eine schematisierte Zusammenfassung :

- in regelmäßigen Abständen (1 min) sammelt ein Programm Börsenkurse mit Hilfe eines Web Service
- die gesammelten Daten werden umgehend auf einen FTP-Server der Digital Radio Nord GmbH geladen
- von diesem FTP-Server werden die Daten in das norddeutsche DAB-Signal eingebettet und übertragen (dieser Vorgang ist nicht Gegenstand dieser Arbeit)
- ein DAB-Receiver - Terratec Box DR 1 - empfängt das DAB-Signal und decodiert einen Teil dieses Signals. Die Daten werden über einen Web-Server in das Internet als Packet Stream übertragen (dieser Teil ist auch nicht Gegenstand dieser Arbeit)
- der Packet Stream, in dem die per DAB übertragenen Börsenkurse enthalten sind, wird decodiert
- die Börsenkurse werden grafisch dargestellt

1.2 Aufbau der Arbeit

Im ersten Kapitel der Arbeit wird die technische Grundlage der *Digital Audio Broadcasting* Technik kurz darstellt. Im zweiten Kapitel wird dann das MOT-Protokoll genauer vorgestellt. Dieses Protokoll wurde im Rahmen dieser Arbeit in einfacher Version in Java implementiert. Im Kapitel „Börsenkurse“ wird erläutert, woher die Börsenkurse stammen und wie die Kurse verarbeitet werden. Im fünften Kapitel wird ein FTP- und ein HTTP-Klient vorgestellt, die von der Apache Software Foundation [Apac2006c] bereit gestellt werden. Im sechsten Kapitel wird erklärt, wie die Kurse übertragen werden, also wie die Dateiverwaltung aussieht. Im Anschluss wird aufzeigt, wie die Koordinaten zur graphischen Darstellung der Börsenkurse berechnet werden. Im letzten Kapitel wird die Implementation der entwickelten Applikationen dargelegt und erläutert.

2 Digital Audio Broadcasting

In diesem Kapitel wird eine kurze Einführung in das DAB-System gegeben. Dabei wird vor allem erklärt, wie die Bitstruktur des DAB-Systems aussieht. Es ist nicht beabsichtigt, das DAB-System detailliert aus technischer Sicht darzustellen. Vielmehr wird versucht, einen Überblick zu vermitteln, wie ein DAB-Decoder die empfangenen Signale interpretieren und decodieren muss.

Das DAB-Signal ist in *Transmission Frames* aufgeteilt, in denen die Daten enthalten sind. Die *Transmission Frames* sind in drei Teile aufgeteilt, die jeweils eine andere Aufgabe haben. Der DAB-Receiver empfängt die *Transmission Frames* in regelmässigen Abständen und extrahiert die darin übertragenen Daten.

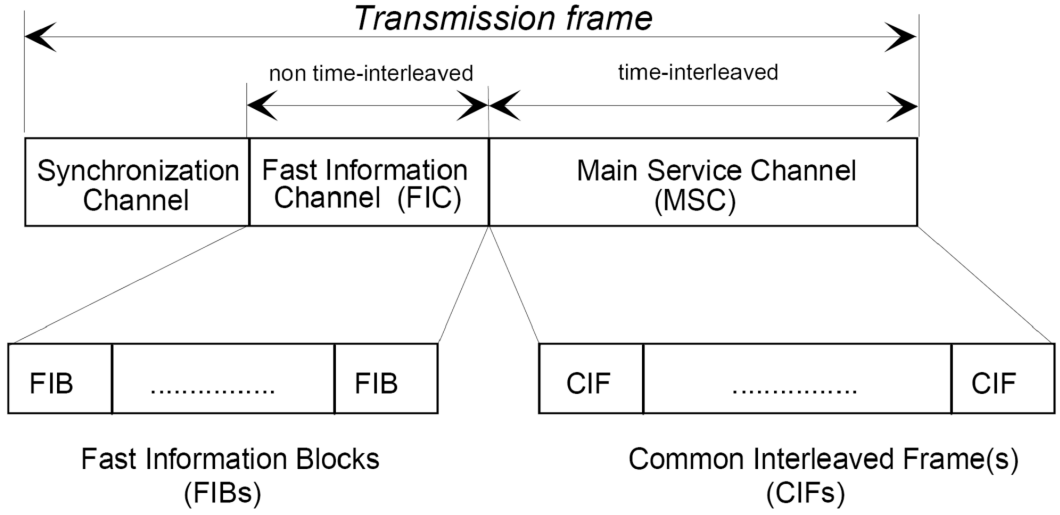

Abbildung 2.1: Transmission Frame

Abbildung 2.1 stellt einen *Transmission Frame* unabhängig vom *Transmission Mode* dar. Ein DAB-Receiver empfängt so einen *Transmission Frame* bis zu 42 mal in der Sekunde. DAB stellt vier verschiedene *Transmission Modes* zur Verfügung, die jeweils die besonderen Gegebenheiten bei der Übertragung berücksichtigen (Antenne, Satellit etc.), auf die hier aber nicht weiter eingegangen wird. Der grundsätzliche Aufbau ist immer identisch. Ein *Transmission Frame* besteht aus einem *Synchronisation Channel*, einem *Fast Information Channel* (FIC) und einem *Main Service Channel* (MSC). Der *Synchronisation Channel* dient der Synchronisation des Receivers mit dem DAB-Signal. Der FIC besteht aus mehreren *Fast Information Blocks* (FIB), deren Anzahl vom verwendeten *Transmission Mode* abhängt. Im FIC können normale Daten übertragen werden, seine eigentliche Aufgabe ist jedoch die Übertragung der *Multiplex Konfiguration*. Mit Hilfe der *Multiplex Konfiguration* stellt sich der DAB-Receiver auf das DAB-Signal und ein decodiert den MSC. Der MSC wiederum besteht aus mehreren *Common Inter-*

leaved Frames (CIF) und enthält die eigentlichen Audio- und Multimediadaten, welche in mehrere *Sub-Channels* aufgeteilt sind.

2.1 Main Service Channel

Der MSC besteht, wie oben erwähnt, je nach Übertragungsmode aus einem oder maximal vier CIFs. Ein CIF besteht aus 864 *Cappacity Units* (CU). Eine CU ist die kleinste adressierbare Einheit im DAB-Signal. Eine CU besteht wiederum aus 64 Bits. In Abbildung 2.2 ist der Aufbau eines CIFs dargestellt. Eine ganze Zahl zusammenhängender CUs bildet einen *Sub-Channel*, das heißt, zwei *Sub-Channels* können sich nicht eine CU teilen und die CUs eines *Sub-Channels* müssen nebeneinander liegen. Es werden mehrere Radiosender in einem Gebiet (zum Beispiel Norddeutschland, Bayern etc.) über einen Kanal übertragen, jeder in einem eigenen *Sub-Channel* innerhalb des MSC. In einem *Sub-Channel* können daneben auch multimediale Inhalte übertragen werden. Alle *Sub-Channel* zusammen werden *Ensemble* genannt.

Um Daten im DAB-Signal zu übertragen, sind zwei verschiedene Übertragungsmodi innerhalb der *Sub-Channel* definiert worden. Zum einen der *Stream Mode*, der für Dienste verwendet wird, die ein gleichmäßiges Datenaufkommen haben und einen ganzen *Sub-Channel* belegen. Dies kann zum Beispiel ein Radiokanal sein, der zu jeder Zeit eine gleiche feste Bitrate hat. Zum anderen gibt es den *Packet Mode*, der verwendet wird, um mehrere Dienste in einem *Sub-Channel* zu übertragen. Auf diesem *Packet Mode* setzt das MOT-Protokoll auf, welches im Rahmen der vorliegenden Arbeit implementiert wurde.

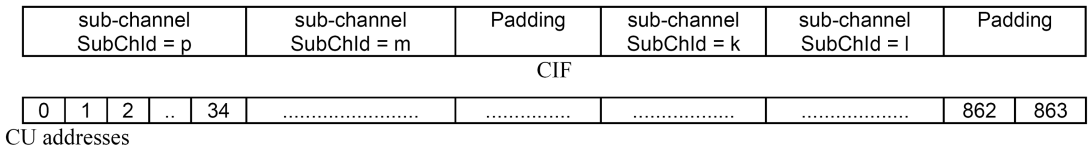

Abbildung 2.2: Common Interleaved Frame

2.2 Fast Information Channel

Der FIC ermöglicht dem DAB-Receiver einen schnellen Zugriff auf das DAB-Signal. Insbesondere wird in dem FIC die *Multiplex Konfiguration* übermittelt, also die Konfiguration des MSC. Im FIC steht, welche *Capacity Units* ein *Sub-Channel* belegt, und welche Art von Daten er enthält (Packet oder Stream, Audio oder Dateien). Daneben können aber auch in geringem Umfang normale Daten im FIC übertragen werden. Darüber hinaus werden im FIC *Labels* zur Darstellung

auf dem Display, der *Traffic Channel* sowie *Emergency Warning System* (EMS) Nachrichten übertragen.

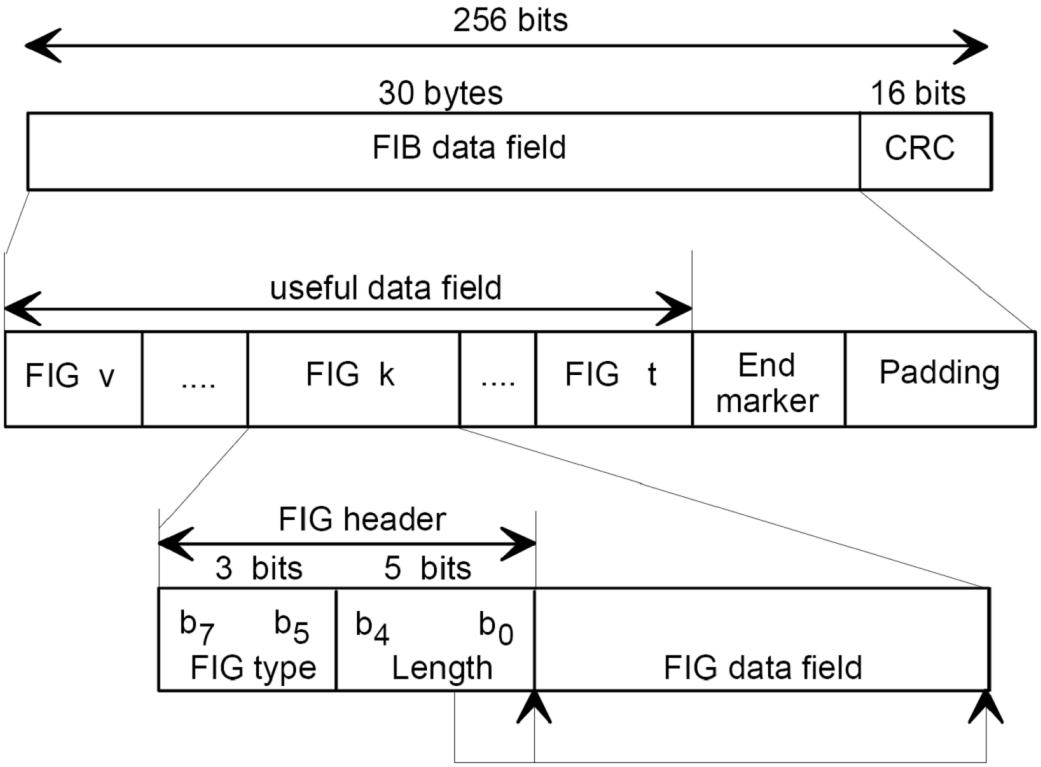

Abbildung 2.3: Fast Information Block

FIG type number	FIG type	FIG application
0	000	MCI and part of the SI
1	001	Labels, etc. (part of the SI)
2	010	Labels, etc. (part of the SI)
3	011	Reserved
4	100	Reserved
5	101	FIC Data Channel (FIDC)
6	110	Conditional Access (CA)
7	111	Reserved (except for Length 31)

Abbildung 2.4: Fast Information Group Typen

Der FIC besteht je nach Übertragungsmodus aus mehreren *Fast Information Blocks* (FIB). Abbildung 2.3 stellt einen FIB dar. Ein FIB ist 256 Bits (32 Bytes) groß, von denen die zwei letzten Bytes für den *Cyclic Redundancy Check* (CRC) verwendet werden, mit dem überprüft werden kann, ob Daten fehlerhaft

übertragen worden sind. Die restlichen Bytes werden in mehrere *Fast Information Groups* (FIG) aufgeteilt. Die ersten drei Bits einer FIG geben den FIG-Typ an, die folgenden fünf Bits geben die Länge der FIG in Bytes an. Diese Längenangabe bestimmt, wieviele der folgenden Bytes zu dieser FIG gehören. In Abbildung 2.4 ist eine Tabelle zu sehen, die alle FIG-Typen auflistet. Unterschiedliche FIG-Typen werden verwendet, um verschiedene Informationen (*Multiplex Konfiguration, Labels* etc.) zu übertragen.

2.3 Multiplex Konfiguration

In Abbildung 2.5 ist das *Data Field* eines FIG-Typ 0 dargestellt. In FIGs Typ 0 wird die *Multiplex Konfiguration* übermittelt. Exemplarisch wird hier anhand dieser FIG erklärt, wie der weitere Aufbau einer FIG aussieht. Andere Typen haben jeweils einen anderen Aufbau.

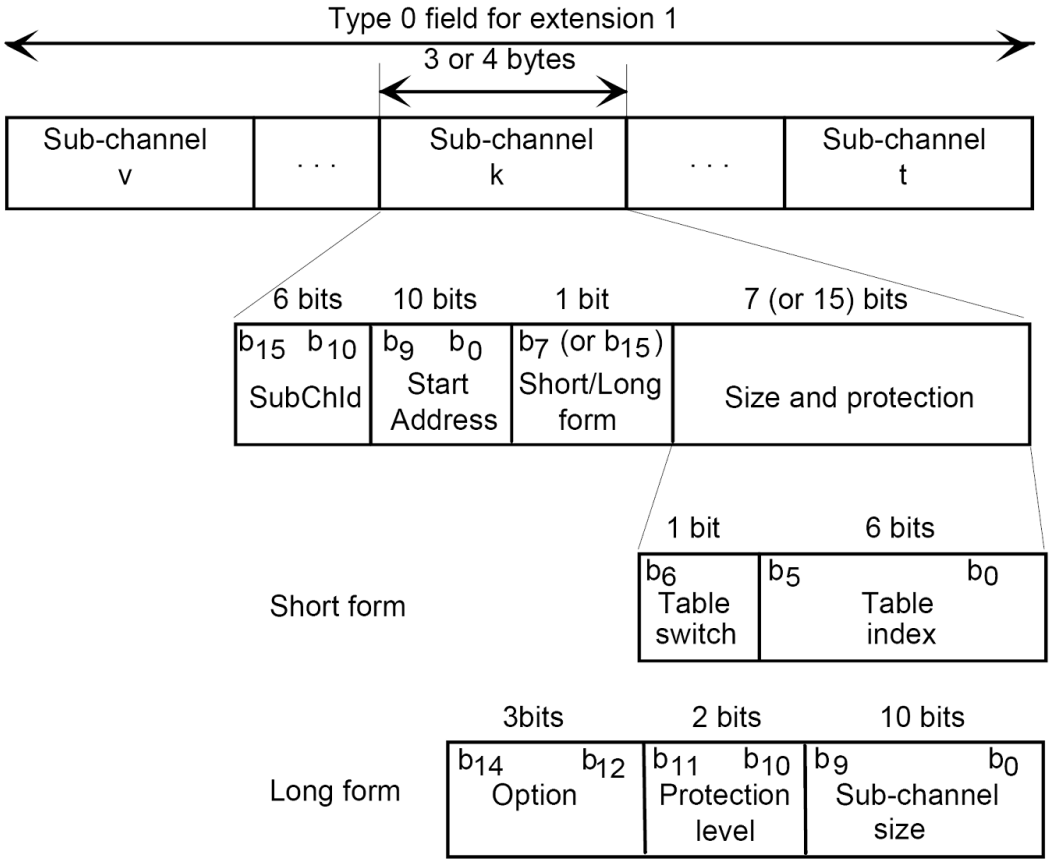

Abbildung 2.5: Fast Information Group Data Field

Das *Data Field* dieser FIG enthält mehrere *Sub-Channel*-Einträge. Jeder Eintrag hat eine von zwei möglichen Formen, eine *Short form* oder eine *Long form*, so dass

die *Sub-Channel* Einträge auseinander gehalten werden können. Die ersten sechs Bits eines Eintrages geben die *Sub-Channel ID* an. Die folgenden zehn Bits geben die Startadresse des *Sub-Channels* an, also die Adresse der *Capacity Unit* (CU). Das nächste Bit bestimmt, ob nachfolgend die *Short* oder die *Long Form* zur Beschreibung angegeben wird. Bei der *Short Form* folgen zusätzlich sechs weitere Bits, die einen Tabellenindex darstellen. In einer Tabelle der DAB-Spezifikation [Etsi2006] stehen alle weiteren Daten. Ist die *Long Form* angegeben, stellen die drei folgenden Bits einen Eintrag dar, anhand derer das Fehlerkorrekturverfahren bestimmt wird. Die folgenden zwei Bits geben den *Protektion Level* an. Die letzten zehn Bits geben schließlich an, wie viele CUs der *Sub-Channel* belegt.

Die bisher vorgestellten Informationen sind notwendig, um den gesammten DAB-Stream zu decodieren. Das im Rahmen dieser Arbeit entwickelte Programm decodiert aber nur die Daten eines *Sub-Channels*. Die in den folgenden Abschnitten vorgestellten DAB-Konzepte wurden, im Gegensatz zu den bis hier vorgestellten Konzepten, implementiert.

2.4 Packet Mode

Innerhalb eines *Sub-Channels* können die Daten im *Packet* oder im *Stream Mode* übertragen werden. Wie oben bereits erwähnt, wird der *Stream Mode* hauptsächlich zur Übertragung der eigentlichen Radiodaten verwendet. Die Daten eines Radiokanals belegen dann einen ganzen *Sub-Channel*. Die Radio-Audiodaten sind im *MPEG Audio Layer II*-Format kodiert. Ein im DAB-System verwendeter *MPEG Audio Layer II Frame* enthält zusätzlich die Möglichkeit, andere Daten zu übertragen. Zum Beispiel kann der *Packet Mode*, der im folgenden Abschnitt genauer erläutert wird, auch in einem oder mehreren *MPEG Audio Layer II Frames* eingebettet werden. So können Daten, die in Zusammenhang mit dem Radiokanal stehen, im gleichen *Sub-Channel* übertragen werden, zum Beispiel eine Auflistung der gespielten Musikstücke.

In dieser Arbeit wird nicht weiter auf den *Stream Mode* eingegangen, da die Börsendaten im *Packet Mode* übertragen werden. Im *Packet Mode* liegen mehrere *Packets* aneinandergereiht in einem *Sub-Channel*. *Packets* haben eine von vier Standard Größen: 96, 72, 48 oder 24 Bytes. Anhand einer Adresse im *Packet Header* können *Packets* unterschiedlichen Diensten zugeordnet werden. Abbildung 2.6 zeigt ein *Packet*.

Der drei Byte große *Packet Header* enthält folgende Informationen:

- **Packet length:** Hier wird die Länge des *Packet* angegeben.
- **Continuity index:** Dieser *2 Bit modulo 4* Zähler wird für jedes *Packet* mit gleicher Adresse, das hintereinander empfangen wird, um eins erhöht.

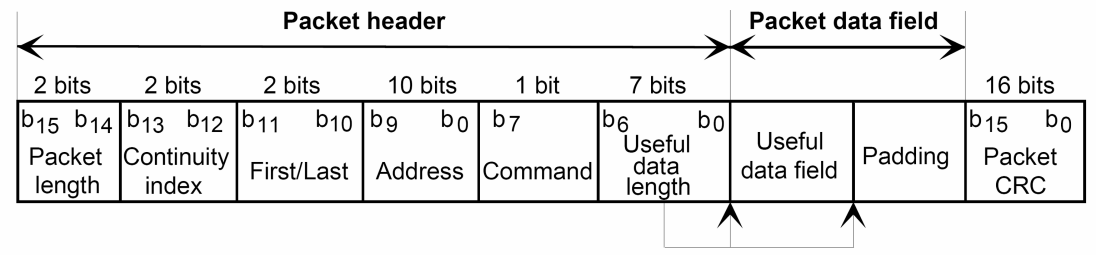

Abbildung 2.6: Packet

- **First/Last:** Hier wird angegeben, ob das *Packet* an erster oder letzter Stelle, mittendrin ist oder ob es das einzige *Packet* ist.
- **Address:** Packets gleicher Adresse werden dem gleichen Dienst in einem *Sub-Channel* zugeordnet. So können bis zu 1023 verschiedene Dienste in einem *Sub-Channel* transportiert werden. *Packets* mit der Adresse 0 sind Stopf-*Packets* und werden nicht für Dienste verwendet.
- **Command:** Dieses Bit gibt an, ob das *Packet* normale Daten enthält oder ob spezielle Anweisungen enthalten sind.
- **Useful data length:** Hier wird die nutzbare Datenlänge in Bytes (0-91) angegeben.

Das *Packet data field* enthält die eigentlichen Daten. Falls die gesamte *Packet*-Länge nicht ausgenutzt wird, müssen Stopf-Bytes verwendet werden. *Packets* sind in dem *Packet Stream* die kleinsten zusammenhängenden Einheiten. Diese sind Ausgangspunkt und werden zu größeren Einheiten zusammengesetzt bis eine Datei vollständig ist.

2.5 Data Groups

Packets werden zu *Data Groups* zusammengesetzt, die die nächst größere Einheit bilden. In Abbildung 2.7 ist der Aufbau einer *Data Group* zu sehen.

2.5.1 Data Group Header

Folgende Informationen sind im *Data Group Header* enthalten.

- **Extension flag:** Hier wird angegeben, ob es das *Extension Field* gibt.
- **CRC flag:** Hier wird angegeben, ob der CRC-Wert gesetzt ist.

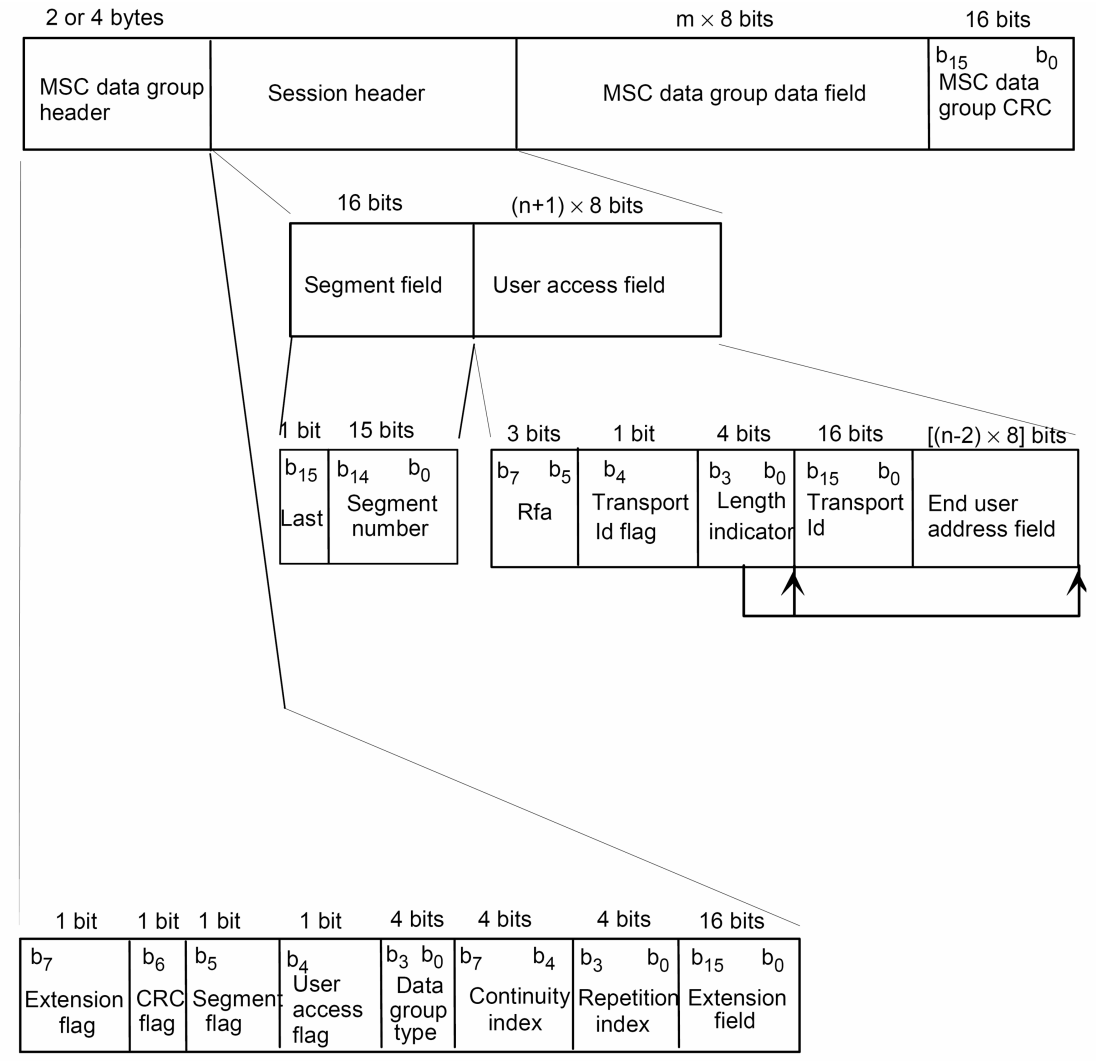

Abbildung 2.7: Data Group

- **Segment flag:** Hier wird angegeben, ob es das *Segment Field* gibt.
- **User access flag:** Hier wird angegeben, ob es das *User Access Field* gibt.
- **Data group type:** Hier wird angeben, um was für einen *Data Group*-Typ es sich handelt. Je nach übertragenen Daten, wird eine andere *Data Group* verwendet (*Header, Directory, Body* etc.).
- **Continuity index:** Dieser vier Bit große Wert wird immer dann erhöht, wenn die zuletzt empfangene *Data Group* gleichen Typs einen anderen Inhalt hat.
- **Repetition index:** Dieser Wert gibt an, wie oft einer *Data Group* Typ

x noch weitere *Data Groups* Typ x mit dem gleichen Inhalt folgen. Dazwischen können *Data Groups* anderen Typs übertragen werden. Der Wert 1111 zeigt an, dass die Wiederhohlungen nicht festgelegt sind.

- **Extension field:** Dieses Feld soll für *Conditional Access* (CA) verwendet werden. Bei anderen *Data Groups* kann dieses Feld für zukünftige Änderungen in der DAB-Spezifikation benutzt werden.

2.5.2 Session Header

Der *Session Header* einer *Data Group* setzt sich sich aus dem *Segment Field* und dem *End User Address Field* zusammen, die beide optional sind.

- **Last:** Her steht, ob dieses Segment das letzte ist oder ob noch Segmente folgen.
- **Segment number:** Hier wird vermerkt, das wievielte Segment dieses ist (wird später für MOT benötigt).
- **User access field:**
  - **Rfa (Reserved for future addition):** Diese drei Bits sind für zukünftige Änderungen reserviert.
  - **Transport Id flag:** Hier wird angegeben, ob eine *Transport Id* gesetzt ist.
  - **Length indicator:** Diese vier Bits geben die Länge der *Transport Id* und des *End User Address Fields* an.
  - **Transport Id (Identifier):** Dieses Feld stellt eine *Transport Id* dar, die ein Objekt eindeutig identifiziert.
  - **End user address field:** Diese Bits sind die Adresse des End-Benutzers.

Dem Session Header folgt dann das eigentliche Datenfeld, welches die Nutzdaten enthält und ein eventueller CRC-Wert.

Mehrere zusammenhängende, hintereinander ankommende *Packets* mit gleicher Adresse ergeben eine *Data Group*. *Data Groups* sind über den *Packets* angeordnet und sind vor allem größer als *Packets*. Abbildung 2.8 stellt den Zusammenhang zwischen mehreren *Packets* und einer *Data Group* dar. Um *Packets* zu einer *Data Group* zusammenzusetzen, werden einfach die nutzbaren Datenbytes der *Packets* in exakt der Reihenfolge zusammengesetzt, in der sie empfangen werden. Das erste empfangene *Packet* enthält den *Data Group Header*.

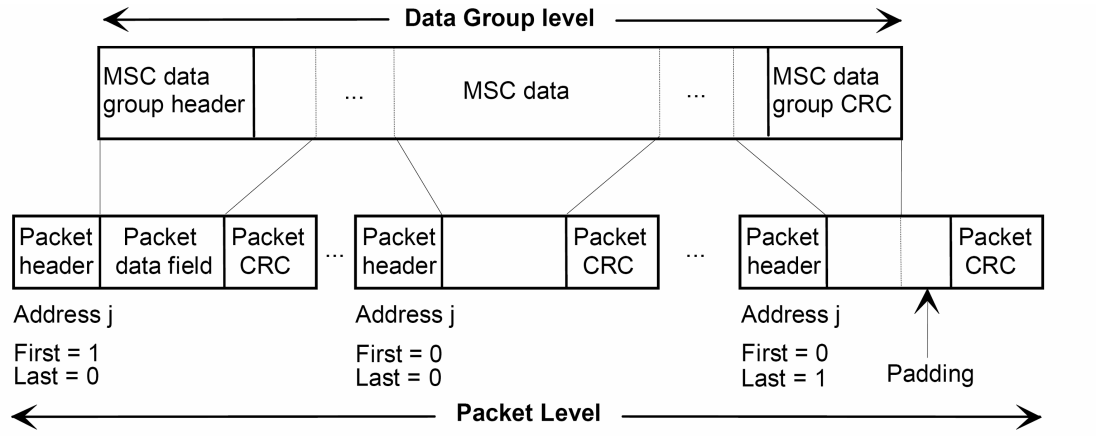

Abbildung 2.8: Zusammenhang zwischen einer Data Group und mehreren Packets

Das MOT-Protokoll, welches als Teil dieser Arbeit implementiert wurde, setzt auf das Konzept der *Packets* und *Data Groups* auf. Im folgenden Kapitel wird das MOT-Protokoll vorgestellt. Dateien, die mit Hilfe des MOT-Protokolls über DAB verbreitet werden, müssen quasi zestückelt werden, da sie nicht als ganzes übertragen werden können. Zerstückelt werden die Dateien in Segmente, die jeweils in eine *Data Group* passen.

3 Multimedia Object Transfer Protokoll

Das *Multimedia Object Transfer* (MOT) Protokoll ist ein speziell für DAB entwickeltes Protokoll, mit dem Dateien übertragen werden. Dabei werden zwei verschiedene Übertragungsarten unterstützt: zum einen der *Header Mode*, bei dem nur ein gültiges Objekt zu einem Zeitpunkt übertragen werden kann und zum anderen der *Directory Mode*, bei dem viele Objekte simultan übertragen werden können. MOT berücksichtigt dabei die besonderen Bedingungen eines Radiosystems, wie beispielsweise die unidirektionale Übertragung. Dabei entsteht im Vergleich zu Datenprotokollen, die im Internet eingesetzt werden, ein deutlich größerer Overhead. Das MOT-Protokoll kann sowohl im *Packet Mode* betrieben werden als auch als *Programm Associated Data* (PAD) übertragen werden. Mit PAD werden Daten bezeichnet, die in enger Beziehung zum Radioprogramm stehen (zum Beispiel eine Auflistung der gespielten Musikstücke). Solche Daten können in den *MPEG Audio Layer II Frame* des betreffenden Radiokanals eingebettet werden und werden im gleichen *Sub-Channel* übertragen. Im Folgenden wird nun beschrieben, wie MOT im *Packet Mode* verwendet wird.

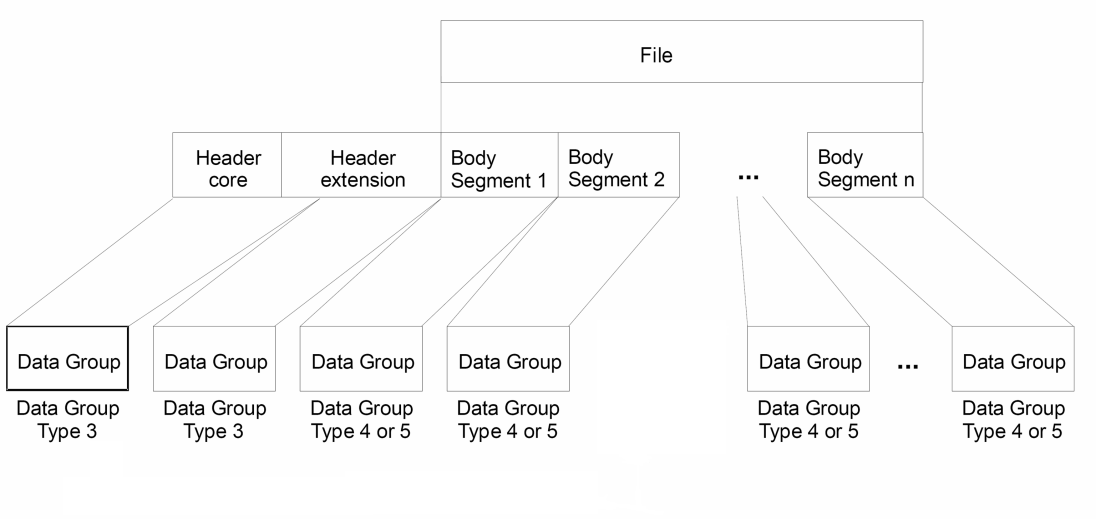

Abbildung 3.1: Segmentierung eines MOT-Objektes

In Abbildung 3.1 ist zu sehen, wie ein MOT-Objekt aufgeteilt wird und je nach Inhalt des Segments unterschiedlichen *Data Groups* zugeordnet wird. Im Wesentlichen gibt es drei verschiedene MOT-Teile, die in *Data Groups* verpackt werden müssen: der MOT *Body* (Datei), der MOT *Header* und das MOT *Directory*. Die unterschiedlichen Arten von Daten werden in verschiedene *Data Group*-Typen (ist im *Data Group Header* angegeben) verpackt, so dass das Decoder-Programm anhand des *Data Group*-Typs erkennt, welche Art von Daten in der *Data Group* transportiert werden. Die *Data Groups* werden, wie im vorherigen Kapitel beschrieben, in *Packets* übertragen.

Jedes Segment bekommt einen *Segmentation Header*, wie er beispielsweise in Abbildung 3.2 zu sehen ist.

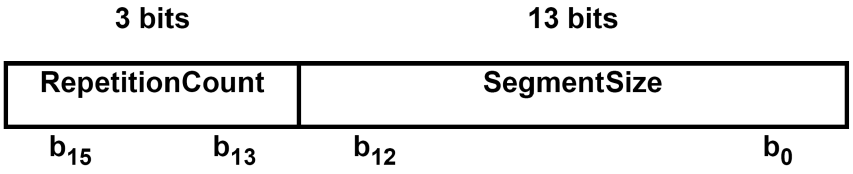

Abbildung 3.2: Segmentation Header

- **RepetitionCount:** Diese drei Bits geben an, wie oft dieses Segment noch übertragen wird (im *Header Mode*). Im *Directory Mode* geben diese Bits an, wie oft das dazugehörige Objekt als ganzes von jetzt an übertragen wird. Die Bits 111 bedeuten, dass die Anzahl weiterer Übertragungen nicht festgelegt ist.
- **SegmentSize:** Diese Bits geben an, wie groß dieses Segment ist.

Wie oben schon erwähnt, werden Segmente je nach Inhalt in unterschiedlichen *Data Groups* transportiert. Teile des MOT *Bodys* (Datei) kommen in *Data Groups*-Typ 4. Wenn *Conditional Access* (CA), ein Verfaheren, mit dem Daten, die über DAB übertragen werden, verschlüsselt werden und so beispielsweise nur gegegen Bezahlung genutzt werden können, verwendet wird, werden *Data Groups* Typ 5 verwendet. Für unkomprimierte MOT *Directorys* werden *Data Groups* Typ 6 benutzt und für ein komprimiertes MOT *Directory* werden *Data Groups* Typ 7 verwendet. Für den MOT *Header* werden *Data Groups* Typ 3 verwendet.

3.1 MOT Header

Der MOT *Header* besteht aus zwei Teilen, dem *Header Core*, der immer den gleichen Aufbau hat und aus vier Parametern besteht, und aus einer *Header Extension*, die variabel aus beliebig vielen Parametern bestehen kann. Dabei sind einige Parameter, wie beispielsweise der Name des MOT-Objektes zwingend erforderlich, andere Parameter hingegen nicht.

3.1.1 Header Core

In Abbildung 3.3 ist der Aufbau des MOT *Header Cores* zu sehen. Folgende Felder beschreiben den *Header Core*:

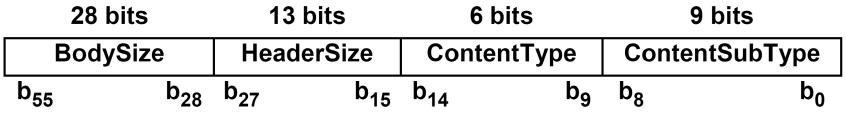

Abbildung 3.3: MOT-Header Core

- **BodySize:** Hier wird die gesamte Größe der Datei angegeben. Laut MOT-Spezifikation sollen übertragene Dateien, deren tatsächliche Größe von der hier angegebenen Größe abweicht, verworfen werden.
- **HeaderSize:** Hier wird angegeben, wie groß der gesamte *Header* ist, dies ist für die Decodierung der Parameter wichtig.
- **ContentType:** Hier wird angegeben, um welche Daten es sich bei diesem MOT-Objekt handelt.
- **ContentSubType:** Hier kann der Inhalt des MOT-Objektes spezifiziert werden.

3.1.2 Header Extension

Die *Header Extension* ist einfach eine Parameterliste, die hinter dem *Header Core* angehängt ist.

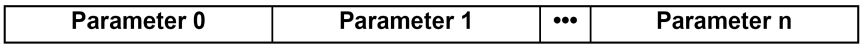

Abbildung 3.4: Parameterliste der Header Extension

Die einzelnen Parameter der Parameterliste können unterschiedlich aufgebaut sein. Die ersten beiden Bits eines Parameters legen fest, wie das weitere Format dieses Parametes ist. Vier verschiedene Formate für die Parameterangabe sind festgelegt. In Abbildung 3.5 sind die Formate dargestellt.

Ein Parameter besteht aus folgenden Teilen:

PLI (Parameter Length Indicator): Hier wird festgelegt, welches Format für diesen Parameter genutzt wird.

ParamId (Parameter Identifier): Dieser Wert gibt an, um welchen Parameter es sich handelt.

Ext (ExtensionFlag): Diese Bits legen die Länge des *Data Field Length Indicators* fest, 7 oder 15 Bits.

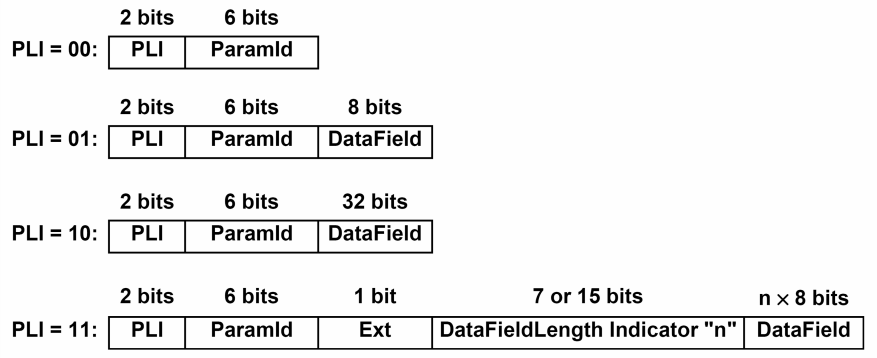

Abbildung 3.5: MOT Header Extension

DataFieldLength Indicator: Diese 7 oder 15 Bits legen fest, wie lang das *Data Field* ist. In der Abbildung 3.5 wird dieser Wert mit n bezeichet.

DataField: Hier werden die Daten des Parameters angegeben.

3.1.3 MOT Extension Parameter

In diesem Abschnittt wird die Kodierung einiger wichtiger Parameter genauer darstellt. Alle übrigen Parameter sind in Tabelle 3.7 aufgeführt und im Originaldokument [Etsi2006a] vorzufinden.

3.1.3.1 Content Name Der Parameter *Content Name* identifiziert ein MOT-Objekt eindeutig. Dieser Parameter verwendet das Parameterformat, das für den Wert $PLI = 11$ definiert ist. In Abbildung 3.6 ist das *Data Field* dieses Parameters zu sehen.

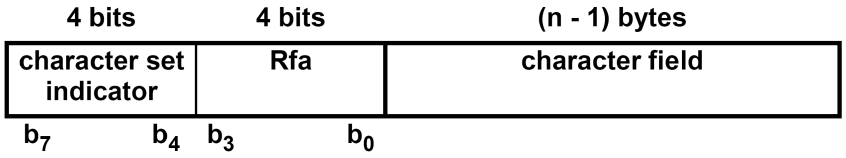

Abbildung 3.6: Parameter Content Name

Der Parameter *Content Name* enthält die Angabe der benutzen Codierung für den Namen, die in den ersten vier Bits angeben ist (*character set indicator*). Es wird in der DAB Spezifikation empfohlen, das *Character Set ISO Latin1* zu verwenden, dies ist auch das *Character Set*, das Java bei der Kodierung von *Charactern* verwendet. Es folgen vier Bits, die für spätere Änderungen reserviert sind. Den Abschluss bildet der Dateiname. Um den Namen zu decodieren, müssen

die empfangenen Bits einfach byteweise nach *Character* gecastet werden. Die Länge ist dabei die angegebene Länge des *Data Fiedls Length Indicators* - 1. Der Name muss als einziger Parameter immer angegeben werden.

3.1.3.2 Compression Type Dieser Parameter zeigt an, ob das MOT-Objekt komprimiert übertragen wird. Es wird das Parameterformat verwendet, das für den Wert $PLI = 01$ definiert ist. Das *Data Field* enthält ein Byte, welches den Kompressionsalgorithmus anhand einer Tabelle identifiziert [Etsi2006b]. Zur Zeit wird nur der gzip-Algorithmus unterstützt. Die acht Bits, die zur Verfügung stehen, um den verwendeten Kompressionsalgorithmen anzuzeigen, lassen aber zuküftig noch viele andere Kompressionsalgorithmen zu. Dieser Parameter ist Pflicht, wenn das MOT-Objekt komprimiert wurde. Textbasierte Dateien werden in der Regel alle komprimiert übertragen.

3.1.3.3 MIME Mit dem MIME-Parameter (*Multi purpose Internet-Mail Extension*, welches aus dem HTTP-protokoll bekannt ist) wird dem Decoder mitgeteilt, welche Daten in dem MOT-Objekt enhalten sind. Mit Hilfe dieser Information kann der Decoder die empfangenen Daten korrekt darstellen, zum Beispiel „image/jpeg“, „application/octet-stream“, „text/xml“. *Das Data Field* des *MIME*-Parameters ist n Bytes lang und enthält den *MIME*-Typ als String codiert, wobei n der *Data Field Length Indicator* ist.

3.1.3.4 Tabelle aller Extension Parameter Abbildung 3.7 ist eine Tabelle der *MOT Extension Parameter*. Die meisten der Parameter werden nicht verwendet, daher wurden diese Parameter auch nicht in der entwickelten Applikation implementiert.

Ein MOT-Objekt besteht aus zwei Teilen: dem MOT *Header* und dem MOT *Body*, der die eigentliche Datei beinhaltet. Der MOT *Body* wird in *Data Groups*-Typ 4 oder 5 und der MOT-Header in *Data Groups*-Typ 3 übertragen.

Parameter Id b_5 b_0	Parameter	Definition	Possible occurrences	Usage mandatory for content provider	Support mandatory for MOT decoders
00 0000	**reserved for MOT protocol extensions**				
00 0001	PermitOutdatedVersions	6.2.3.1.2	only once	no	no
00 0010 00 0011 00 0100	**reserved for MOT protocol extensions**				
00 0101	TriggerTime (user application specific parameter)	see [5]	see [5]	see [5]	see [5]
00 0110	**reserved for MOT protocol extensions**				
00 0111	RetransmissionDistance	6.2.3.1.5	only once	no	no
00 1000	**reserved for MOT protocol extensions**				
00 1001	Expiration	6.2.3.1.1	only once	no	yes, if receiver provides "MOT caching support"
00 1010	Priority	6.2.3.1.4	only once	no	no
00 1011	Label (user application specific parameter)	see [6]	only once	no	no
00 1100	ContentName	6.2.2.1.1	only once	yes	yes
00 1101	UniqueBodyVersion	6.2.3.1.3	only once	no	no
00 1110 00 1111	**reserved for MOT protocol extensions**				
01 0000	MimeType	6.2.2.1.2	only once	user application specific	user application specific
01 0001	CompressionType	6.2.2.1.3	only once	yes (if body is compressed)	**yes;** every receiver must check if an object is compressed
01 0010 ••• 01 1111	**reserved for MOT protocol extensions**				
10 0000	AdditionalHeader (user application specific parameter)	see [6]	once or several times	see [6]	see [6]
10 0001	ProfileSubset	6.2.3.3.1	only once	no	no
10 0010	**reserved for MOT protocol extensions**				
10 0011	CAInfo	6.2.3.2.1	only once	yes (if CA is used)	**yes**; every receiver must check if an object is scrambled
10 0100	CAReplacementObject	6.2.3.2.2	only once	no	no
10 0101 ••• 11 1111	**reserved for user application specific parameters**				

Abbildung 3.7: Tabelle der MOT Extensions

3.2 MOT Transport Modes

MOT stellt zwei *Transport Modes* zur Verfügung, um die Dateien zu übermitteln: den *Header Mode* und den *Directory Mode*. Im *Header Mode* kann stets nur ein MOT-Objekt gleichzeitig übertragen werden (In den unterschiedlichen *Sub-Channels* können natürlich mehrere Dateien übertragen werden.). Im *Directory Mode* können eine Vielzahl von MOT-Objekten simultan übertragen werden,

dieser Mode wird auch *Data Karussell* genannt, da die Dateien immer wieder übertragen werden.

3.2.1 Header Mode

Der MOT *Header Mode* wird verwendet, wenn nur eine Datei übertragen wird. Der MOT *Stream* besteht aus einem MOT *Header* und dem zugehörigen MOT *Body*. Der *Header* und der *Body* können mehrere Male übertragen werden. Das Europäische Institut für Telekommunikationsnormen hat in einem Dokument [Etsi2006c] beschrieben, wie dieser Mode genutzt werden kann.

Im *Header Mode* können hintereinander verschiedene MOT-Objekte übertragen werden. Zu jedem gegebenen Zeitpunkt kann es jedoch immer nur einen gültigen *Header* und einen gültigen *Body* geben. Um dem Decoder mitzuteilen, dass ein neues MOT-Objekt übertragen wird, wird eine neue *Transport Id* verwendet. Empfängt der MOT-Decoder eine neue *Transport Id*, muss er alle empfangenen MOT-Segmente mit der alten *Transport Id* verwerfen und nur noch das neue Objekt sammeln.

3.2.2 Directory Mode

Im *Directory Mode* wird in einem *Sub-Channel* eine ganze Dateistruktur übertragen. Die *Header* aller MOT-Objekte und einige Zusatzinformationen werden zusammen in einem *Directory* übertragen. Es kann auch hier immer nur ein gültiges *Directory* geben. Ändert sich die Dateistruktur, so wird ein neues *Directory* übertragen. Dieses wird anhand einer neuen *Transport Id* (wird im *Header* der *Data Group* angegeben) erkannt. Dateien, die unverändert sind, brauchen nicht noch einmal zusammengesetzt werden, da sie auch im neuen *Directory* die gleiche *Transport Id* haben und somit erkannt werden können.

3.2.2.1 MOT Directory Struktur Abbildung 3.8 stellt die Struktur des MOT *Directorys* dar. Das *Directory* wird im *Packet-Stream* - zerlegt in *Data Groups* und *Packets* - übertragen. Empfängt der Decoder eine *Data Group* vom Typ 6 oder 7, so weiss er, dass hier ein MOT *Directory* zusammengesetzt werden muss. Sobald das *Directory* vollständig gesammelt ist, sind dem Decoder alle Dateien, inklusive zugehöriger Headerinformationen wie der Dateiname, die Größe etc.., in diesem *Data Karussell* bekannt. Wenn der Decoder auf einer Maschiene mit begrentzden Ressourcen läuft, läst er beispielsweise große Dateien aus, wenn nicht genügend Speicherplatz zur Verfügung steht.

- **CF (CompressionFlag):** Dieses Bit sollte 0 sein.

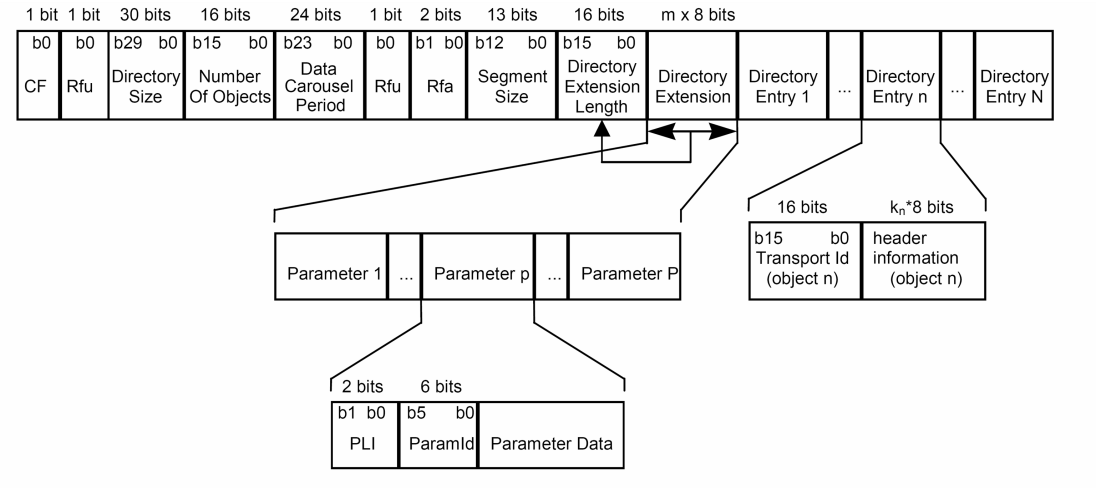

Abbildung 3.8: MOT Directory-Struktur

- **Rfu:** Dieses Bit ist für spätere Änderungen reserviert.
- **DirectorySize:** Diese Bits geben die gesamte Größe des MOT *Directorys* in Bytes an.
- **NumberOfObjects:** Hier wird die Anzahl der übertragenen MOT-Objekte angegeben.
- **DataCarouselPeriod:** Hier steht die maximale Zeit (in 1/10 Sekunden) die es braucht, um das MOT *Directory* einmal komplett zu übertragen.
- **Rfu:** Dieses Bit ist spätere Änderungen reserviert.
- **Rfa:** Diese Bits sind für spätere Änderungen reserviert.
- **SegmentSize:** Hier wird die Größe der Segmente der MOT-Objekte angegeben. Steht hier 0, so ist die Größe nicht weiter festgelegt und kann in den einzelnen Segmenten unterschiedlich sein.
- **DirectoryExtensionLength:** Diese Bits geben die Länge der *Directory Extension* in Bytes an.
- **DirectoryExtension:** Hier ist eine Parameterliste enthalten, die das MOT *Directory* genauer spezifiziert, diese ist ähnlich aufgebeut wie die *Objekt Header Extension*.
- **DirectoryEntry:** Ein *Entry* enthält die Beschreibung eines MOT-Objektes.
- **TransportId:** Mit der *Transport Id* wird ein MOT-Objekt eindeutig identifiziert.

- **HeaderInformation:** Hier ist der vollständige MOT *Header* enthalten. Anhand der *Transport Id* wird der *Header* dem richtigen *Body* zugeordnet.

3.2.2.2 MOT-Directory Extensions Die Kodierung der MOT *Directory Extension* ist ähnlich der Kodierung der *Header Extension*, wie in Abbildung 3.5 auf Seite 20 zu sehen ist. Abbildung 3.9 zeigt eine Liste der möglichen Parameter. Da die Parameter nicht für die Dekodierung der MOT-Objekte erforderlich sind, werden sie hier auch nicht weiter vorgestellt.

Parameter Id b_5 b_0	Parameter	Definition	Possible occurrences	Usage mandatory for content provider	Support mandatory for MOT decoders
00 0000	SortedHeaderInformation	7.2.4.1	only once	no	no
00 0001	DefaultPermitOutdatedVersions	7.2.4.2	only once	no	no
00 0010	**reserved for MOT protocol extensions**				
•••					
00 1000					
00 1001	DefaultExpiration	7.2.4.3	only once	no	yes, if receiver provides "MOT caching support"
00 1010	**reserved for MOT protocol extensions**				
•••					
01 1111					
10 0000	**reserved for user application specific parameters**				
10 0001					
10 0010	DirectoryIndex (user application specific parameter)	see [6]	once or several times	see [6]	see [6]
10 0011	**reserved for user application specific parameters**				
•••					
11 1111					

Abbildung 3.9: MOT Directory Extensions

Die in den ersten beiden Kapiteln vorgestellten Konzepte beschreiben die Codierung/Decodierung eines MOT *Streams*. Hier wird noch einmal kurz zusammengefasst, welche Schritte erforderlich sind, um aus dem MOT *Stream* die eigentlichen Daten zu extrahieren:

- Zusammenhägende *Packets* werden zu *Data Groups* zusammengesetzt.
- Die *Data Groups* werden nach ihren Typen unterschieden.
- *Data Groups* Typ 6 werden gesammelt und zu einem *Directory* zusammengestzt. In diesem sind alle Dateien, die in dem *Stream* enthalten sind, vermerkt. Das *Directory* enthält auch alle MOT-Objekt *Header*. Der Zusammenhang zwischen den *Headern* und den MOT *Body* wird über eine *Transport Id* hergestellt.
- Es wird in den *Data Groups Type* 4, die den MOT *Body* enthalten, noch den *Tranport Ids* ausschau gehalten, für die bereits im *Directory* ein *Header* übertragen wurde.

- Sind alle Teile eines MOT *Bodys* empfangen, kann die empfangene Datei zusammengesetzt werden.
- Kommen neue MOT-Objekte in den MOT *Stream*, wird ein neues *MOT Directory* übertragen. Das alte verliert seine Gültigkeit.

4 Die Börsenkurse

Die Börsenkurse, die über DAB übertragen werden, werden von einem frei zugänglichen Web Service abgerufen. Mit Hilfe von Web Services können heute Daten zwischen Programmen ausgetauscht werden. Ein Server stellt im Internet oder in einem lokalen Netzwerk eine Schnittstelle zur Verfügung, die nach genau definierten Regeln angesprochen werden kann und die in der Regel Funktionen bereitstellt. Web Services funktionieren ähnlich wie Webseiten. Während bei Webseiten Daten und Funktionen sichtbar dargestellt werden, damit ein Mensch damit arbeiten kann, werden die Daten und Funktionen bei Web Services so bereitgestellt, dass Programme die automatische Weiterverarbeitung übernehmen können.

Der zunächst in dieser Arbeit verwendete Web Service (www.invesbot.com), der in einem Web Service Verzeichnisdienst [Xmet2006] aufgelistet war, wurde kurzfistig vom Netz genommen. Der Web Service, der zur Zeit in der Server-Applikation benutzt wird, wird von Swanand Mokashi [Moka2006] bereitgestellt. Anscheinend wird dieser Web Service von Swanand Mokashi als kostenloser Ableger eines kostenpflichtigen Web Service betrieben [Stri2006].

Dieser Web Service hat einen deutlich geringeren Umfang als der zuerst verwendete. Der vom Netz genommene Web Service hat beispielsweise eine umfangreiche Beschreibung der Firma, für die der Börsenkurs abgefragt wurde, geliefert. Diese Zusatzinformationen sollten zunächst auch über das DAB-System übertragen und auf Empfängerseite darstellt werden.

Derzeit verfügbare kostenlose Web Services bieten nur Börsenkurse von Unternehmen, die an amerikanischen Börsen gelistet sind. Dies hat zur Folge, dass ein aktueller Tageschart in der Zeit von ca. 15:50 bis 22:15 (UTC+1, entspricht 09:50 bis 16:15 UTC-5.) übertragen wird, da zu diser Zeit die Börsen in Amerika geöffnet haben.

Grundlage eines Web Service ist das SOAP-Protokoll, welches auf XML (*eXtensible Markup Language*) basiert. Der gesamte Austausch von Nachrichten zwischen dem Server, der den Web Service bereitstellt und dem Klienten, der den Web Service nutzen möchte, findet in XML statt.

4.1 XML

Um Daten unabhängig zwischen verschiedenen Systemen und Plattformen auszutauschen, wurde Mitte der neunziger Jahre XML entwickelt. XML ist eine vom W3C[1] entwickelte Metasprache, die eine Reihe von Regeln zur Verfügung stellt, mit denen individuelle Dokumententypen entwickelt werden können, die zur Datenspeicherung, Übertragung, oder Verarbeitung verwenden werden. XML ist in

[1] World Wide Web Consortium

dieser Hinsicht sehr flexibel einsetzbar und bietet den Vorteil, dass man sich nur wenig Gedanken über ein Datenformat machen muss. Zudem gibt es bereits eine Anzahl von Werkzeugen, die Entwickler bei der Verarbeitung von XML verwenden können.

Eine XML-Datei bildet eine baumartige Struktur. Sie besteht aus einem Kopf, der angibt, wie die XML-Datei kodiert ist und einem Rumpf mit den eigentlichen Daten. Der Rumpf besteht aus einer beliebigen Zahl von Elementen. Zwingend erforderlich ist immer ein Wurzelelement. Die darunter liegenden Elemente können beliebig tief geschachtelt werden. Ein Element kann entweder nutzbare Daten beinhalten oder eine beliebige Anzahl weiterer Elemente. Darüber hinaus kann es Attribute enthalten. Jedes Element besteht aus einem öffnenden und einem schließenden *Tag*.

Folgendes Beispiel stellt eine einfache XML-Struktur dar, ähnlich derer, die später per DAB übertragen werden soll. Die Tatsächliche Struktur kann im Anhang B nachgeschlagen werden.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Kurse>
  <AktienKurs Symbol="msft">
    <Firma>Microsoft </Firma>
    <Kurs> 26,54  </Kurs>
  </AktienKurs>
  <AktienKurs Symbol="intc">
    <Firma>Intel</Firma>
    <Kurs> 243,54  </Kurs>
  </AktienKurs>
</Kurse>
```

Die erste Zeile beinhaltet den Kopf. Angegeben werden hier die XML-Version und die verwendetet Zeichenkodierung. Das Element `Kurse` ist das Wurzelelement. Die zweite Zeile beinhaltet einen Kommentar. Weiter besteht das XML-Dokument aus zwei Elementen `Aktienkurs`, die jeweils ein Element `Firma` und ein Element `Kurs` haben. Die Elemente `Firma` und `Kurs` enthalten die eigentlichen Daten. Wichtig sind das Wurzelelement und die jeweils schließenden *Tags*, da andernfalls XML-Dateien nicht korrekt verarbeitet werden können.

Oben genanntes Beispiel stellt ein willkürliches XML-Beispiel dar. Mit Hilfe der sogenanten *Dokument Type Definition* (DTD) kann festgelegt werden, wie ein XML-Dokument aussehen soll, damit ein Programm ein Dokument verarbeiten kann. Anhand der DTD können ohne weitere Kenntnisse des Programms XML-Dokumente erzeugt werden, die das Programm korrekt verarbeiten kann. Eine DTD legt fest, welche Elemente und Attribute ein XML-Dokument haben darf und haben muss.

```
<?xml version='1.0' encoding="UTF-8" ?>
<!ELEMENT Kurse                  (AktienKurs*)>
<!ELEMENT AktienKurs              (Firma,Kurs)>
<!ATTLIST AktienKurs Symbol       ID #REQUIRED>
<!ELEMENT Firma                      (#PCDATA)>
<!ELEMENT Kurs                       (#PCDATA)>
```

Das obige Beispiel zeigt eine einfache DTD für das weiter oben aufgeführte XML-Dokument. In der ersten Zeile wird wieder die verwendetet XML-Version und Zeichenkodierung angegeben. In der zweiten Zeile wird das Wurzelelement, hier `Kurse`, angegeben und die/das Kind(er)Element(e). Im vorliegenden Fall besteht das Wurzelelement aus einer beliebigen Anzahl Elemente mit dem Namen `Aktienkurs`. Das Element `Aktienkurs` wiederum hat das Attribut `Symbol` und jeweils ein Kinderelement `Firma` und `Kurs`. Die Elemente `Firma` und `Kurs` enthalten die eigentlichen Daten vom Typ PCDATA (*Parsed Character Data*), also Daten, die von einem Parser analysiert werden (beispielsweise wird „&ouml“ als ö angezeigt).

Daneben besteht die Möglichkeit, die Struktur eines XML-Dokuments mit Hilfe eines XML-Schemas festzulegen. Folgendes Beispiel stellt ein äquivalentes XML-Schema für die oben angegebene DTD dar.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Kurse">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Kurs" minOccurs="0" maxOcurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Kurs">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Firma"/>
         <xs:element ref="Kurs"/>
       </xs:sequence>
    </xs:complexType>
    <xsd:attribute name="persnr" type="xsd:string"/>
  </xs:element>
  <xs:element name="Firma" type="xs:string"/>
  <xs:element name="Kurs" type="xs:string"/>
</xs:schema>
```

XML-Schemas haben gegenüber einer DTD den Vorteil, dass sie selber in XML formuliert sind und daher von XML-Parsern gelesen werden können und bieten zusätlich noch die Möglichkeit, die Datentypen und Werte der angegebenen Elemente genau festzulegen. Um Konflikte zwischen unterschiedlichen XML-Schemas und XML-Dokumenten zu vermeiden, gibt es verschiedene *Namespaces*. Ein *Namespace* wird durch einen „:“ gebildet. In dem oben definierten XML-Schema wird der Namespace „xs:“ definiert. Alle Elemente, die mit „xs:“ beginnen, gehören dem gleichen *Namespace* an. Für unterschiedliche *Namespaces* können innerhalb eines Dokumentes jeweils andere XML-Schemata festgelegt werden. XML ist Grundlage von Web Services, die kurz in den nächsten Kapiteln vorstellet werden. Außerdem werden die Börsenkurse als XML-Dokument verarbeitet.

XML bietet bei der Verarbeitung von Daten den Vorteil, dass eine Reihe von Entwicklerwerkzeugen zur Verfügung steht. Dadurch fällt bei der Entwicklung von Anwendungen ein erheblicher Arbeitsaufwand weg. Darüberhinaus können XML-Dokumente sehr gut komprimiert werden, was für die Übertragung per DAB nützlich ist.

Um XML-Dokumnete zu verarbeiten, stehen zwei verschiedene Parsertypen zur Verfügung, zum einen die *Sax*-Parser und zum anderen die *Dom*-Parser. *Sax*-Parser arbeiten ereignisorientiert, d. h. diese Parser arbeiten ein XML-Dokument sequentiell ab und teilen dem Anwendungsprogramm ein auftretendes Ereignis (Anfang eines *Tags*, Ende eines *Tags* etc.) mit.

Daneben gibt es die *Dom*-Parser, die aus einem XML-Dokument ein *Dokument Objekt Model* (DOM) erzeugen, welches das gesamte Dokument in einer Baumstruktur enthält. Das DOM stellt darüber hinaus Methoden bereit, mit denen das XML-Dokument ausgelesen und verändert werden kann.

```
Documentbuilder db;

Document dom = null;
try {
   dom = db.parse("Aktienkutrse.xml");
} catch (SAXException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
}
Element docEle = dom.getDocumentElement();

NodeList nl;

nl = docEle.getElementsByTagName("Firma");
```

```
if(nl != null && nl.getLength() > 0) {

   for (int i = 0; i y nl.length; i++) {
        System.out.println(nl.item(0).getTextContent());
   }
}
```

Dies Beispiel zeigt, wie einfach XML-Dateien mit einem *DOM*-Parser verarbeitet können. Nachdem mit dem `Documentbuilder` das *Document Object Model* aus einer XML-Datei erzeugt wird, wird das Wurzelelement geholt. Von dem Wurzelelement können dann beliebig alle weiteren Elemente über den jeweiligen Namen als `NodeList` abgefragt werden (`docEle.getElementsByTagName(Firma)`). Die `NodeList` kann `null` sein, dann existieren keine Elemente mit dem gesuchten Namen, oder die Liste ist nicht `null`, dann kann die Liste in einer Schleife durchlaufen und der Inhalt des jeweiligen Elements ausgegeben werden (`nl.item(0).getTextContent()`). Die gefundenen Elemente in der Knotenliste können natürlich weitere Elemente enthalten.

4.2 Web Services

Web Services sind Klient/Server-Anwendungen, die auf SOAP und XML basieren. Sowohl der Nachrichtenaustausch zwischen Server und Klienten, als auch die Beschreibung der zur Verfügung stehenden Dienste finden in XML statt. Die SOAP-Nachrichten können mit Hilfe eines beliebigen Transportprotokolls (HTTP, FTP, SMTP etc.) übertragen werden.

4.2.1 SOAP

SOAP legt fest, wie Daten zwischen Client und Server ausgetauscht werden. Eine SOAP-Nachricht besteht aus drei Teilen: einem Umschlag (SOAP-*Envelope*), einem optionalen Kopf (SOAP-*Header*) und den eigentlichen Daten (SOAP-*Body*). Das folgende Beispiel ist ein SOAP-*Envelope* und umschließt die gesamte Nachricht.

```
<soap:Envelope xmlns:s0="http://swanandmokashi.com"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <!--hier Header und Body -->
</soap:Envelope>
```

Die wichtigste Information im Envelope ist diese Zeile:

```
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
```

Hier wird das XML-Schema für den Namespace `soap` festgelegt, welches das XML-Schema für ein SOAP-Dokument ist. Weiterhin werden noch die Namensräume `xs` und `s0` festgelegt. Das *Envelope* Element einer SOAP-Nachricht ist das Wurzelelement.

Nach dem *Envelope* kann optional ein *Header* folgen. Dieser beginnt mit dem gleichen *Namespace* wie der *Envelope*. Der *Header* kann zum Beispiel Informationen zur Authentifizierung enthalten.

```
<soap:Envelope xmlns:s0="http://swanandmokashi.com"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <soap:Header>
     <auth:authentication xmlns:auth="'http://auth.de/auth>
       <auth:user>dabunios<auth:user>
       <auth:password>xxxxxxxx<auth:password>
     </auth:authentication>
   </soap:Header>
<!--hier Body -->
</soap:Envelope>
```

Im *Body* der SOAP-Nachricht sind die eigentlichen Daten enthalten, die beispielsweise festlegen, welche Methode der Server ausführen soll. Das *Body* Element muss dem gleichen *Namespace* zugeordnet sein wie der *Envelope*. Folgendes Beispiel zeigt die gesammte SOAP-Nachricht, die zum Web Service Server geschickt wird, um die Börsenkurse der Firmen Microsoft (msft) und SAP AG (sap) abzufragen. Eine Liste aller verfügbaren Symbol-Zuordnungen kann über die Webseite der New York Stock Exchange eingesehen werden [Nyse2006].

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:s0="http://swanandmokashi.com"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <soap:Body>
     <s0:GetQuotes>
       <s0:QuoteTicker>msft sap</s0:QuoteTicker>
     </s0:GetQuotes>
   </soap:Body>
</soap:Envelope>
```

Im SOAP *Body* wird dem Server mitgeteilt, welche Aufgabe er übernehmen soll. Im obigen Beispiel soll der Server eine Methode mit dem Namen `GetQuotes` ausführen, was durch das erste Kindelement (`s0:GetQuotes`) des *Bodys* ausgedrückt wird. Das Kindelement von `GetQuotes, Quoteticker`, enthält die Funktionsparameter der Methode (msft sap).

Die Antwort des Servers sieht folgendermaßen aus:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soap:Body>
     <GetQuotesResponse xmlns="http://swanandmokashi.com">
       <GetQuotesResult>
         <Quote>
           <CompanyName>MICROSOFT CP</CompanyName>
           <StockTicker>MSFT</StockTicker>
           <StockQuote>25.94</StockQuote>
           <LastUpdated>8/31/2006 2:40pm</LastUpdated>
         </Quote>
         <Quote>
           <CompanyName>SAP AKTIENGESELL</CompanyName>
           <StockTicker>SAP</StockTicker>
           <StockQuote>47.74</StockQuote>
           <LastUpdated>8/31/2006 2:40pm</LastUpdated>
         </Quote>
       </GetQuotesResult>
     </GetQuotesResponse>
   </soap:Body>
</soap:Envelope>
```

Der wesentliche Unterschied zum *Request* ist, dass das Element mit der aufzurufenden Funktion durch ein Element mit dem gleichen Namen ersetzt wurde, ergänzt um den Zusatz *Response*. Die eigentlichen Informationen sind die Kindelemente des `GetQuotesResult` Elements.
Falls bei dem Aufruf eines Web Services ein Fehler auftritt, wird dies dem Klienten durch eine Fehlernachricht mitgeteilt. Tritt ein Fehler auf, enthält die SOAP-Nachricht als einziges Kindelement ein Faultelement (`soap:Fault`), welches eine genaue Fehlerbeschreibung liefert.

4.2.2 WSDL

Um auf einen Web Service zu greifen zu können, muss bekannt sein, wie mit diesem kommuniziert werden muß. Dazu wurde die *Web Service Definition Language* (WSDL) entwickelt, die ebenfalls auf XML basiert. Eine WSDL beschreibt im XML-Format, wo der Web Service zu finden ist, welche Funktionen bereitgestellt werden, welche Argumente eine Funktion erwartet, wie die Antwort aussieht und welches Trägerprotokoll (HTTP, FTP etc.) eingesetzt wird. Im Anhang A ist die WSDL des Web Services, der in der Aplikation genutzt wird, um die Börsenkurse abzufragen, dargestellt. Der Aufbau einer WSDL-Datei sieht folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BeispielWSDL">
  <types>  ...  </types>

  <message>  ...   </message>

  <portType>
     <operation> ...  </operation>
  </portType>

  <binding>  ...   </binding>

  <service>
     <port> ... </port>
  </service>
</definitions>
```

- **definitions:** Das `definitions`-Element ist das Wurzelelement einer WSDL-Datei und kann dazu genutzt werden, verschiedene *Namespaces* zu definieren.
- **types:** Innerhalb des `types`-Elements können alle Datentypen definiert werden, die nicht im Standard XML-Schema enthalten sind. Folglich alle Datentypen, die aus anderen Datentypen zusammengesetzt sind (*complexType*).
- **message:** Im `message`-Element wird festgelegt, wie die Nachrichten aussehen, die zwischen dem Klienten und dem Server ausgetauscht werden. Es wird pro Funktion jeweils eine Nachricht beschrieben, die der Klient an den Server schickt und für den Fall, dass dieser antwortet, eine Nachricht, die der Server an den Klienten schickt.
- **portType:** `portType`-Elemente beschreiben die Methoden, die bei einem Web Service aufgerufen werden können.

- **operation:** Das `operation`-Element als Kindelement eines `portType`-Elementes beschreibt den Nachrichtenverlauf der zugehörigen Methode (*One Way, Request-Response, Solicit-Response, Notification*). Das `operation`-Element hat die möglichen Kindelemente `input` und `output`, die festlegen, welche `message`-Elemente jeweils ausgetauscht werden.
- **binding:** Das `binding`-Element legt fest, wie der Klient mit dem Server kommuniziert, insbesondere, welches Trägerprotokoll (HTTP etc.) eingesetzt wird.
- **service:** Im `service`-Element wird festgelegt, unter welcher Adresse ein Web Service zu erreichen ist.

Mit Hilfe einer solchen WSDL kann auf den Web Service ohne Kentnisse der internen Programmstruktur zugegriffen werden. Ist es auch egal, auf welcher Plattform der Web Service läuft und in welcher Sprache der Web Service implementiert ist.

Dank bestehender Frameworks (Axis, .NET, PEAR SOAP etc.) müssen Web Services nicht von Grund auf neu implementiert werden. Dies mindert den Programmieraufwand erheblich. Insbesondere bei der Implementation eines Web Service-Klienten ist es wegen der erwähnten Frameworks möglich, den Klienten fast ohne Kenntnisse der internen Struktur der Web Services zu benutzen. Um auf den Web Service zu zugreifen, der die Börsenkurse bereit stellt, wurde Axis verwendet, das im nächsten Abschnitt vorstellt wird.

4.2.3 Axis

Axis ist eine von Apache [Apac2006c] bereitgestellte SOAP *Engine*. Also ein Framework, mit dem auf einfache Weise auf SOAP basierende Programme erstellt werden können. Axis stellt aber noch mehr zur Verfügung:

- Einen einfachen Standalone Server
- Einen Server, der mit Servlet-Servern wie Tomcat zusammenarbeiten kann
- Umfangreiche WSDL Unterstützung
- Ein Tool, welches Java-Klassen aus WSDL-Dateien erstellt
- Beispielprogramme
- Ein TCP/IP Monitoring-Tool

Im Wesentlichen ist Axis ein Framewerok, mit dem auf effiziente und einfache Weise Web Services erstellt und betrieben werden können. Axis bewahrt einen Web Service-Entwickler davor, sich zu sehr in die theoretischen Grundlagen der Web Services einarbeiten zu müssen, da Axis viele Aufgaben bei der Implementation übernimmt.

Im Rahmen dieser Bachelorarbeit wurde Axis genutzt, um die Klientseite eines Web Services zu implementieren. Axis stellt dafür ein Tool namens *WDSL2Java* und einige Klassen bereit, welche aus einer WSDL-Datei/URL die erforderlichen Klassen erzeugen, um den Web Service zu benutzen. Auf die bereitgestellten Funktionen des Web Services kann anschließend ganz einfach über Java-Methoden zugegriffen werden, ohne sich in irgendeiner Form mit dem Übertragungsprotokoll SOAP auseinander zusetzen. Beim Aufruf der Methoden ist zu beachten, dass ein entfernter Methodenaufruf über ein Netzwerk stattfindet, wodurch gegebenenfalls eine gesonderte Fehlerbehandlung erforderlich ist.

Folgender Code erzeugt aus der URL einer WSDL-Datei die erforderlichen Klassen um den Web Service zu nutzen, der in der WSDL beschrieben wird:

```
Emitter e = new Emitter();

try {
   e.run("http://www.swanandmokashi.com/"    +
         "HomePage/WebServices/StockQuotes.asmx?WSDL");
} catch (Exception e1) {
    e1.printStackTrace();
}
```

Im Allgemeinen erzeugt Axis aus einer beliebigen-WSDL Datei folgende Java-Code:

- Pro *type*-Eintrag eine Java Klasse
- Für jeden *portType*-Eintrag ein Interface
- Für jeden *binding*-Eintrag eine Stubklasse
- Für jeden *service*-Eintrag ein Service-Interface
- Einen Service Locator

Für den im Rahmen dieser Arbeit benutzten Web Service (WSDL in Anhang A) sieht das Resultat folgendermaßen aus:

- GetQuotes.java

  `GetQuotes` repräsentiert das Type Element `GetQuotes`, das in der WSDL Datei aufgeführt ist.

- GetQuotesResponse.java

  `GetQuotesResponse` repräsentiert das Type Element `GetQuotesResponse`, das in der WSDL Datei aufgeführt ist.

- Quote.java

  `Quote` repräsentiert das Type Element `Quote`, das in der WSDL Datei aufgeführt ist.

- StockQuotes.java

  Diese Klasse legt das Interface für das Service-Element fest.

- StockQuotesSoap.java

  Diese Klasse definiert das *Service Definition Interface* (SDI) für ein `portType` Element. Über dieses Interface wird auf eine Operation des Web Service zugegriffen.

- StockQuotesSoapStub.java

  Diese Klasse implementiert das SDI und ist der Web Service-Stub. In dieser Klasse ist die Web Service Logik gekapselt. Hierüber wird auf den Web Service wie auf ein lokales Objekt zugegriffen.

- StockQuotesLocator.java

  Diese Klasse implementiert den Service Locator, über den der Stub instantiiert wird.

Der Web Service kann dann sehr einfach über die erzeugten Klassen abgerufen werden. Das unten aufgeführte Beispiel veranschaulicht, wie die Aktienkurse von dem Web Service abgefragt werden.

```
StockQuotesLocator loc = new StockQuotesLocator();

StockQuotesSoap soap = loc.getStockQuotesSoap();

GetQuotes quotes = new GetQuotes("msft sap");

GetQuotesResponse quotesResponse = soap.getStockQuotes(quotes);

Quote[] quote = quotesResponse.getGetQuotesResult();
```

Das `Quote[]` Array enthält zu jedem abgefragten Symbol (msft, sap) ein `Quote` Objekt. `Quote` ist als *type* Element in der WSDL genau festgelegt worden, daher kann auf die Felder (`Quote, Company, LastUpdated` etc.) über Methoden zugegriffen werden.

5 FTP- / HTTP-Klient

Damit die Börsenkurse, die von dem Web Service abgefragt werden, in das DAB-Signal gelangen, müssen sie auf einen FTP-Server der Digital Radio Nord GmbH geladen werden. Von dort sorgt dann spezielle Software und Hardware dafür, dass die auf dem Server gespeicherten Daten in das DAB-Signal eingebettet werden. Um Börsendaten mit Hilfe von DAB zu übertragen, müssen daher zunächst die Kurse im Minutentakt auf den Server der Digital Radio Nord GmbH geladen werden. Um dieses Problem zu lösen, wird in der Applikation ein FTP-Klient der Apache Software Foundation [Apac2006c] verwendet. Es ist zwar möglich, über die im SDK [Micro2006] implementierte `URL`-Klasse eine FTP-Verbindung herzustellen, diese hat aber das RFC 2396 [Ietf1998] implementiert und bietet daher nicht den vollen Umfang eines Standard FTP-Programms. Der von Apache bereit gestellte FTP-Klient ist nach dem RFC 959 [Ietf1985] implementiert und bietet daher den von den gängigen FTP-Programmen bekannten Funktionsumfang und ist intuitiv zu benutzen.

5.1 FTP-Klient

Der FTP-Klient ist im Jakarta Commons Projekt der Apache Software Foundation [Apac2006c] enthalten, welches eine Reihe von wiederverwendbaren Java-Komponenten zur Verfügung stellt. Das Package `NET` [Apac2006a] enthält eine Reihe von in Java implementierten Netzwerkprotokollen, unter anderem das FTP-Protokoll. Folgende Zeilen Code sind selbsterklärend und zeigen, wie der FTP-Klient benutzt wird.

```
FTPClient ftp  = new FTPClient() ;
ftp.connect(127.0.0.1);
ftp.login("Name","xxxxxxxx");

ftp.makeDirectory(directory);
ftp.removeDirectory(directory);
ftp.storeFile(file ,location);
ftp.deleteFile(file);
ftp.listfiles();

ftp.disconnect();

//Zum Abfragen der Antwort des FTP

int reply = ftp.getReplyCode();
```

5.2 HTTP-Klient

Werden neue Dateien auf den FTP-Server geladen, werden diese nicht umgehend in das DAB-Signal eingebettet. Bevor dies geschieht muss, zuerst eine Authentifizierung über eine gesicherte HTML-Seite durchgeführt werden. Nach dieser Authentifizierung startet der von dem Frauenhofer Institut entwickelte Multimedia Data Server mit der Aufbereitung der auf dem Server liegenden Daten und passt die Daten so an, dass sie in das DAB-Signal eingebettet werden könne. Wird diese Authentifizierung nicht durchgeführt, werden die auf dem FTP-Server liegenden Dateien nicht per DAB übertragen.

Auch die Authentifizierung ist über die `URL` möglich, die im SDK implementiert ist, bietet aber wie auch bei einer FTP-Verbindung nur geringeren Funktionsumfang. Daher wurde, ebenfalls aus dem Apache Jakarta Commons Projekt, ein HTTP-Framework [Apac2006] verwendet, um diese Authentifizierung durch zuführen.

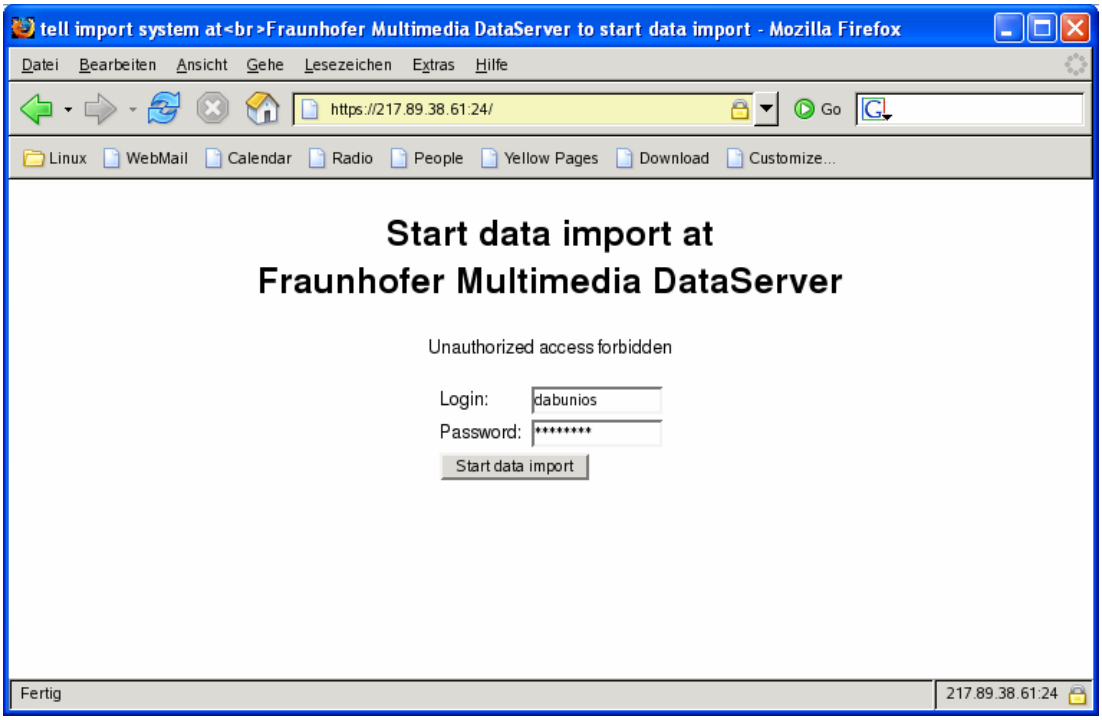

Abbildung 5.1: Data Server

Die nachfolgenden Zeilen Code zeigen, wie mit dem Framework ein einfacher HTTP-Klient implementiert und genutzt werden kann, um sich, wie in Abbildung 5.1 zu sehen ist, zu authentifizieren.

```
HttpClient https = new HttpClient();

PostMethod post = new PostMethod("https://127.0.0.1");

NameValuePair[] data = {
    new NameValuePair("login", "dabunios"),
    new NameValuePair("password",  "xxxxxxxx"};

post.setRequestBody(data);

int respond = https.executeMethod(post);

// Die Antwort des Servers als InputStream

InputStream res = post.getResponseBodyAsStream();
```

Da der Web Server der Digital Radio Nord GmbH ein nicht signiertes SSL-Zertificat verwendet, muss dieses zuerst abgespeichert werden und ein neues *Truststore* erstellt werden, das dieses Zertificat importiert und als vertrauenswürdig deklariert. Dies geschieht mit dem im SDK [Micro2006] enthaltenen Dienstprogramm *keytool*. Dieses *Truststore* hat den Namen `truststore` und muss später im gleichen Ordner liegen wie das Programm, das die Börsenkurse von dem Web Service abfagt und auf den FTP-Server lädt. Im eigentlichen Programm muss auf folgende Weise die System-Property `javax.net.ssl.trustStore` geändert werden.

```
System.setProperty("javax.net.ssl.trustStore", "truststore");
```

`truststore` stellt das mit dem *keytool* erzeugte *Truststore* dar. Dieses stellt eine Möglichkeit dar, das Problem zu lösen. Eine andere wäre gewesen, wie aus den gängigen Web-Browsern bekannt ist, dem Benutzer des Programms mitzuteilen, dass der Server ein nicht zertifiziertes SSL-Zertificat benutzt. Der Benutzer müste dann bestätigen, dass die Verbindung trotzdem hergestellt werden soll.

6 Dateien

Um die Kurse mit DAB zu übertragen, muss festgelegt werden, wie die Dateistruktur der Börsenkurse aussehen soll. Dabei sind einige Gegebenheiten/Spezifikationen des DAB-Systems zu berücksichtigen. Die nahe liegende Lösung, alle Kurse in eine Datei zu schreiben und diese Datei einmal in der Minute neu auf den DAB-Server hoch zuladen, die gleichzeitig die vorherige Datei überschreibt, ist nicht möglich.

Dies liegt daran, dass am Ende eines Tages die Datei so groß wird, dass sie in den meisten Fällen nicht mehr empfangen werden könnte. Das zugrunde liegende Problem ist, dass die Datei jeweils nur eine Minute Zeit zur Übertragung hat, da sie nach einer Minute wieder ersetzt wird, tatsächlich aber in den meisten Fällen (vor allem bei großen Dateien) länger als eine Minute zur Übertragung benötigt.

Angesichts dieser Problematik, war der erste Ansatz, jeden Kurs in einer separaten Datei zu übertragen und nach Firma, Tag und Jahr in unterschiedliche Dateien zu speichern. So sollte verhindert werden, dass eine einzige Datei zu schnell aktualisiert wird und nicht mehr beim Empfänger ankommt. Bei dieser Lösung tritt jedoch das Problem auf, dass schon in sehr kurzer Zeit sehr viele Dateien generiert wurden. Bei 20 verschiedenen Aktienkursen, die jede Minute aktualisiert werden, werden schon nach 50 Minuten über 1000 Dateien übertragen. Werden in dem MOT *Stream* sehr viele Dateien übertragen, wird das MOT `Directory`, welches notwendig ist, um die Dateien zu decodieren, jedoch so groß, dass es länger als eine Minute zur Übertragung benötigt. Sollen aber einmal in der Minute neue Dateien in den MOT-Stream eingebettet werden, wird das MOT `Directory` auch einmal in der Minute erneuert. Dies hat zur Folge, dass das `Directory` nicht mehr vollständig beim Receiver ankommt und daher keine Dateien decodiert werden können.

Um diese beiden Probleme zu umgehen, werden die verschiedenen Kurse in einer Datei gespeichert und Stunde lang in einzelnen Dateien übertragen. Parallel werden die Kurse lokal[2] gemeinsam in einer Datei gespeichert. Beginnt eine neue Stunde, werden die Kurse der letzten Stunde nicht mehr separat pro Minute übertragen, sondern es werden alle Kurse der letzten Stunde in einer Datei übertragen. Außerdem wird dem DAB-Server nur alle 5 Minuten mitgeteilt, dass neue Dateien vorhanden sind, die in das DAB-Signal eingebettet werden sollen. Der Ablauf, der implementiert werden muss, sieht wie folgt aus:

1. Die Kurse werden einmal in der Minute bei dem Web Service abgefragt. Für jede Minute wird eine Datei auf den Server geladen. Lokal wird eine Datei angelegt, in der die abgefragten Kurse alle gemeinsam abgespeichert werden.

[2] Auf dem Computer, auf dem das Programm läuft, das sie Börsenkurse vom Web Service abfragt und auf den FTP-Server hochlädt.

2. Die Authorisation über das HTTP-Protokoll gegenüber dem DAB-Server, die notwendig ist, damit die neuen Dateien in das DAB-Signal eingebettet werden, wird ca. alle fünf Minuten durchgeführt. Dies hat zur Folge, dass nur alle 5 Minuten neue Dateien übertragen werden und gibt dem MOT-Decoder die Möglichkeit das MOT *Directory* in jedem Fall zu empfangen. Fünf Minuten ist ein willkürlicher festgelegter Wert, bei dem es keine Probleme gibt, aber die Verzögerung der Datenübertragung auch nicht zu groß ist.

3. Am Beginn einer jeden Stunde werden die Kurse der letzten Stunde vom Server gelöscht und die lokal abgespeicherte Datei mit den Kursen der letzten Stunde wird auf den Server hochgeladen. Diese Kurse werden nun in einer Datei übertragen. Die in der neu angefangenen Stunde abgefragten Kurse werden wieder in einzelnen Dateien übertragen. Für jede Stunde gibt es einen eigenen Ordner auf dem Server, so können die zu löschenden Dateien anhand des Ordners identifiziert werden.

4. Da dem Fachbereich Medieninformatik der Universität Osnabrück nur eine geringe Bandbreite des DAB-Signals (2 kBit/sec oder eine CU) zur Verfügung steht, können die gesamten Tageskurse nicht dauerhaft übertragen werden, da diese sonst die Übertragung neuer Kurse behindern würde. Daher werden am Ende eines Tage alle Minutenkurse vom DAB-Server gelöscht und es wird ein einzelner Tagesschlusskurs übertragen, so dass auch Wochen- und Jahrescharts beim Empfänger dargestellt werden können.

Das Programm, das die Kurse auf den Server lädt, läuft bei Abgabe der Arbeit erst seit ca. zwei Monaten. Daher ist die Darstellung eines kompletten Jahres-Chart derzeit nicht möglich.

7 Darstellung

Wenn die Börsenkurse beim Empfänger angekommen sind, müssen sie, bevor sie grafisch dargestellt werden können, an das Koordinatensystem des Monitors angepasst werden.

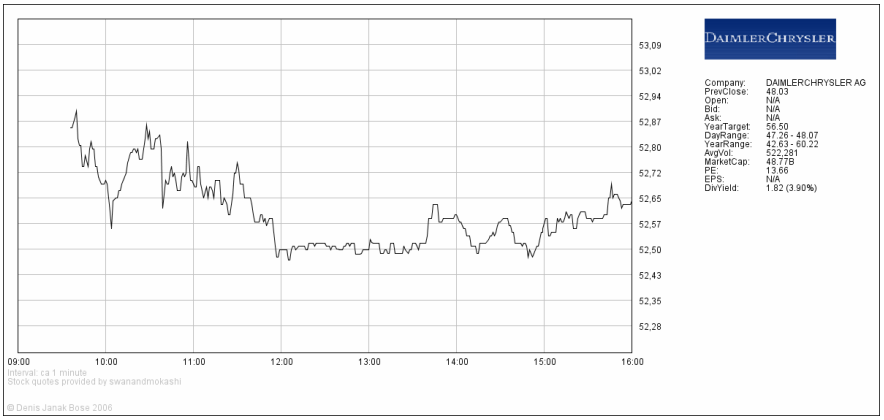

Abbildung 7.1: Tageschart

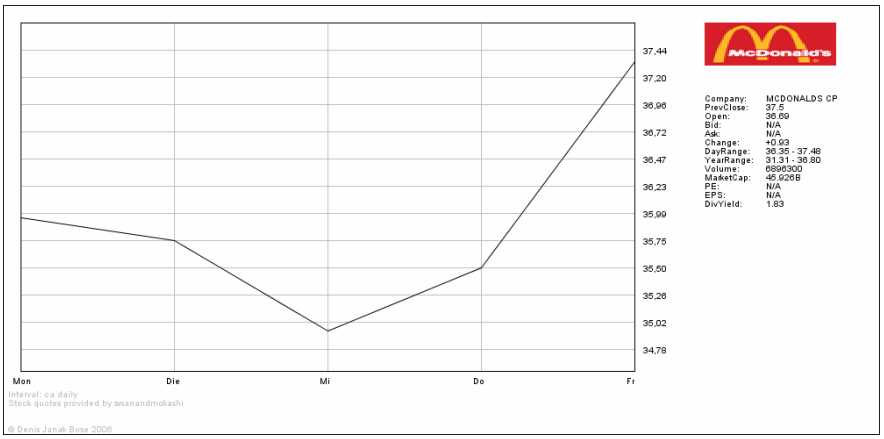

Abbildung 7.2: Wochenchart

Der zu übertragene Aktienkurs hat folgendes Format:

```
<Price>38.20</Price>
<Time>3:45pm</Time>
```

Eine beträchtliche Menge dieser Kurse, verteilt über den ganzen Tag, müssen bei der Berechnung des Tageshcharts korrekt auf das Koordinatensystem abgebildet werden. Zwei Konvertierungen sind hierfür erforderlich. Zum einen muss

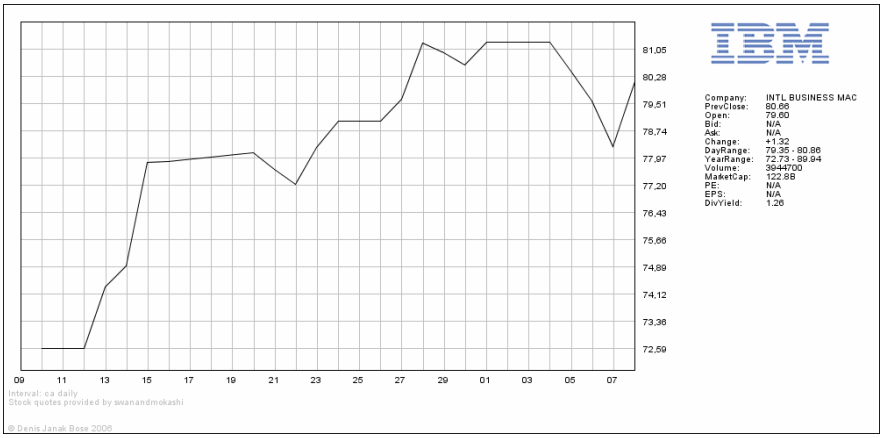

Abbildung 7.3: Monatschart

dem Wert `Price` ein passender Y-Wert zugeordnet und zum anderen dem Wert `Time` ein passender X-Wert zugeordnet werden. Bei der Darstellung mehrerer Tageskurse gilt dies für die einzelnen Tage.

8 Implementation

Im Rahmen dieser Bachelorarbeit wurden zwei Applikationen programmiert: Ein Server, der regelmäßig Börsenkurse von einem Web Service abfragt und diese auf den DAB-Server lädt und ein Klient, der einen MOT *Stream* decodiert und die daraus empfangenen Börsendaten grafisch darstellt. In den nächsten beiden Abschnitten wird erläutert, wie bei der Implementierung vorgegangen ist.

8.1 Server

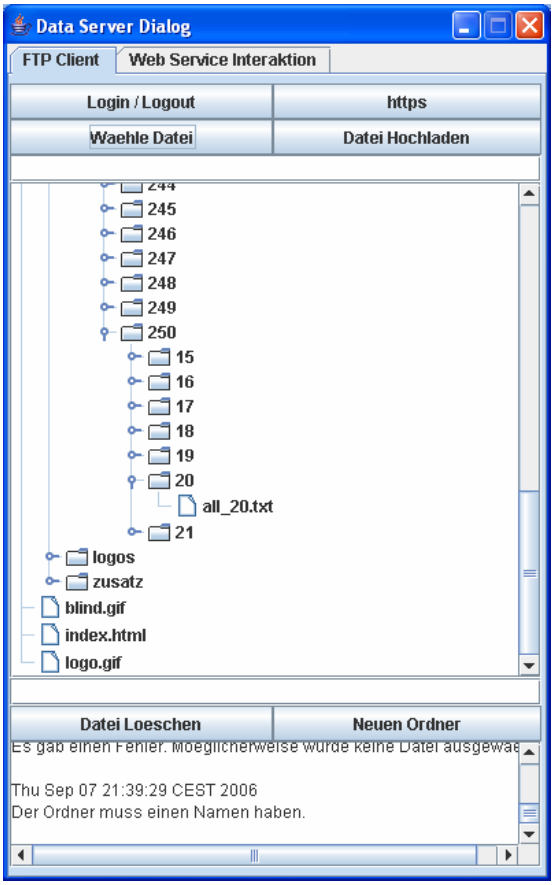

Abbildung 8.1: FTP-Klient

Bei der Serverseite handelt es sich nicht um einen Server im eigentlichen Sinne. Vielmehr besteht dieses Programm aus einem einfachen FTP-Browser und einem Teil, der die Interaktion mit dem Web Service und dem DAB-Server übernimmt. Die Web Service-Interaktion und der FTP-Klient sind auf zwei verschiedenen `JTabbedPanes` untergebracht.

8.1.1 Die Oberfläche

Die `JTabbedPane` mit dem FTP-Server, in der Abbildung 8.1 zu sehen, bietet folgende Funktionen:

- **Button Login/Logout:** Beim Drücken des Buttons *connected/disconnected* sich der FTP-Klient mit dem FTP-Server.
- **Button https:** Über diesen Button wird das HTTP-Skript aufgerufen und dem DAB-Server mitgeteilt, dass neue Daten auf dem Server liegen, die in das DAB-Signal eingebettet werden.
- **Button Waehle:** Mit Hilfe dieses Buttons wird ein `JFileChooser` geöffnet, um eine lokale Datei auszuwählen, die auf den FTP-Server geladen werden soll.
- **Button Datei hochladen:** Beim Drücken dieses Buttons wird die zuvor gewählte Datei auf den FTP-Server geladen.
- **Oberes Inputfeld:** In diesem Inputfeld steht der Name der Datei, die hochgeladen werden soll.
- **Unteres Inputfeld:** In dieses Inputfeld kann der Name eines Ordners eingegeben werden, der auf dem FTP-Server erstellt werden soll.
- **Button Neuen Ordner:** Dieser Button erstellt einen Ordner auf dem FTP-Server.
- **Button Datei Löschen:** Mit diesem Button wird eine auf dem FTP-Server ausgewählte Datei gelöscht.

Die `JTabbedPane` mit der Web Service-Interaktion, zu sehen in der Abbildung 8.2, bietet folgende Funktionen:

- **Button Start:** Mit diesem Button wird der `Thread` gestartet, über den die Börsenkurse von dem Web Service geholt werden und auf den FTP-Server geladen werden.
- **Button Ende:** Dieser Button beendet den `Thread`.
- **Symbol loeschen:** Diser Button löscht ein Symbol aus der Tabelle.

- **Symbol hinzufuegen:** Mit diesem Button wird ein Symbol in die Tabelle eingefügt. Eine Liste der verfügbaren Symbol kann auf der Web Seite der New York Stock Exchange eingesehen werden [Nyse2006].
- **Tabelle:** Hier wird eine Liste aller Symbole angezeigt, für die Börsenkurse übertragen werden.

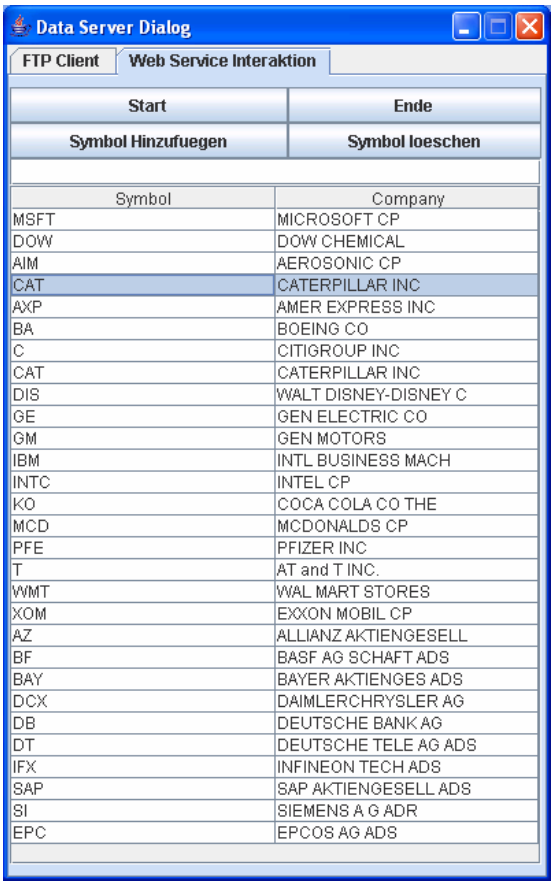

Abbildung 8.2: Web Service-Interaktion

Das Layout der Oberfläche wurde mit Hilfe des `GridBagLayout` erstellt.

8.1.2 Klassendiagramm

In Abbildung 8.3 ist die Klassenhierachie der Serverseite dargestellt. Es werden nur die wesentlichen Attribute, Methoden und Assiziationen dargestellt. Außerdem sind die *Listener* nicht aufgefürt. Andernfalls wäre das Diagramm zu unübersichtlich geworden.

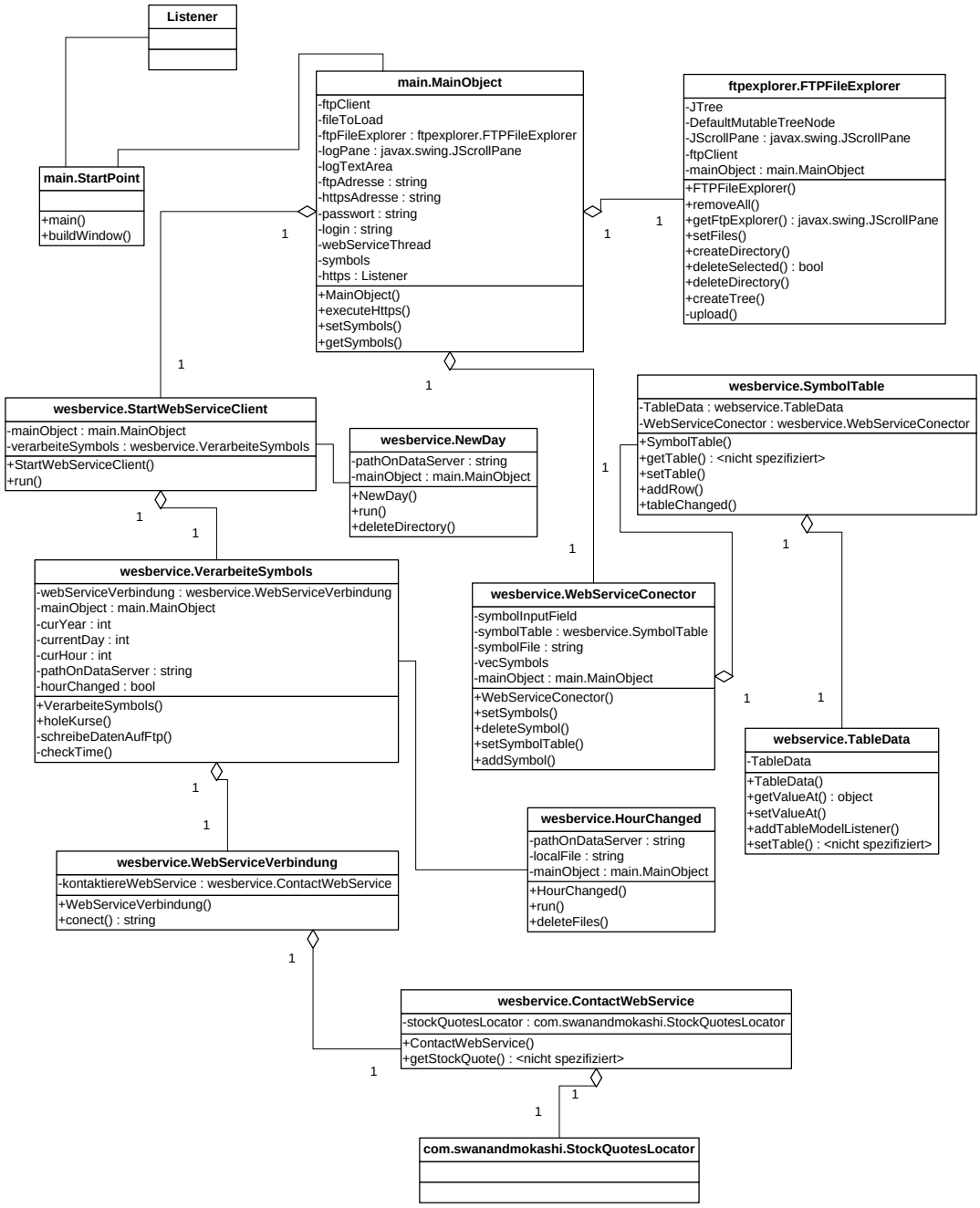

Abbildung 8.3: Klassendiagramm des Servers

8.1.3 Optionen

Das Programm, das die Börsenkurse auf den Server lädt, liest beim Starten eine XML-Datei ein, mit der der Server konfiguriert wird. Folgendes XML-Dokument zeigt die Optionen, die über die Datei „optionen.xml“ verändert werden können.

```
<?xml version="1.0"?>
<Optionen Beschreibung="Server-Side Optionen" >
  <Data-Server>
    <FTP-Server>217.89.38.61</FTP-Server>
    <HTTP-Server>217.89.38.61:24</HTTP-Server>
    <Login>dabunios</Login>
    <Passwort>xxxxxxxx</Passwort>
  </Data-Server>
</Optionen>
```

- **FTP-Server:** Hier wird die IP-Adresse des FTP-Servers angegeben (Server auf den die Börsenkurse gladen werden). Hier kann testweise die Adresse eines beliebigen FTP-Servers eingegeben werden, auf den die geholten Kurse geladen werden können.
- **HTTP-Server:** Die IP-Adresse und Port des Servers, auf dem die Authentifizierung durchgeführt werden musss, damit neue Daten in das DAB-Signal eingebettet werden. Diese Adresse sollte nicht geändert werden, da das Programm die System-Property `javax.net.ssl.trustStore` geändert hat. Daher ist es nur möglich eine SSL-Verbindung zu dem angegebenen Server aufzubauen, ausser das Zertificat wird in das verwendete *Truststore* importiert.
- **Login:** Hier steht der *Login* Name für den FTP/HTTP-Server.
- **Passwort:** Hier steht das Passwort für den FTP/HTTP-Server.

8.1.4 Programmablauf

Das Sequenzdiagramm in Abbildung 8.4 zeigt, wie die Klassen miteinader interagieren, um die Börsenkurse von dem Web Service zu holen und auf den FTP-Server zu laden. Dieses Diagramm stellt nicht alle Objekte dar, da es sonst zu unübersichtlich geworden wäre. Die wesentlichen Schritte sind jedoch enthalten. Die Funktion des FTP-Explorers ist nicht dargestellt.

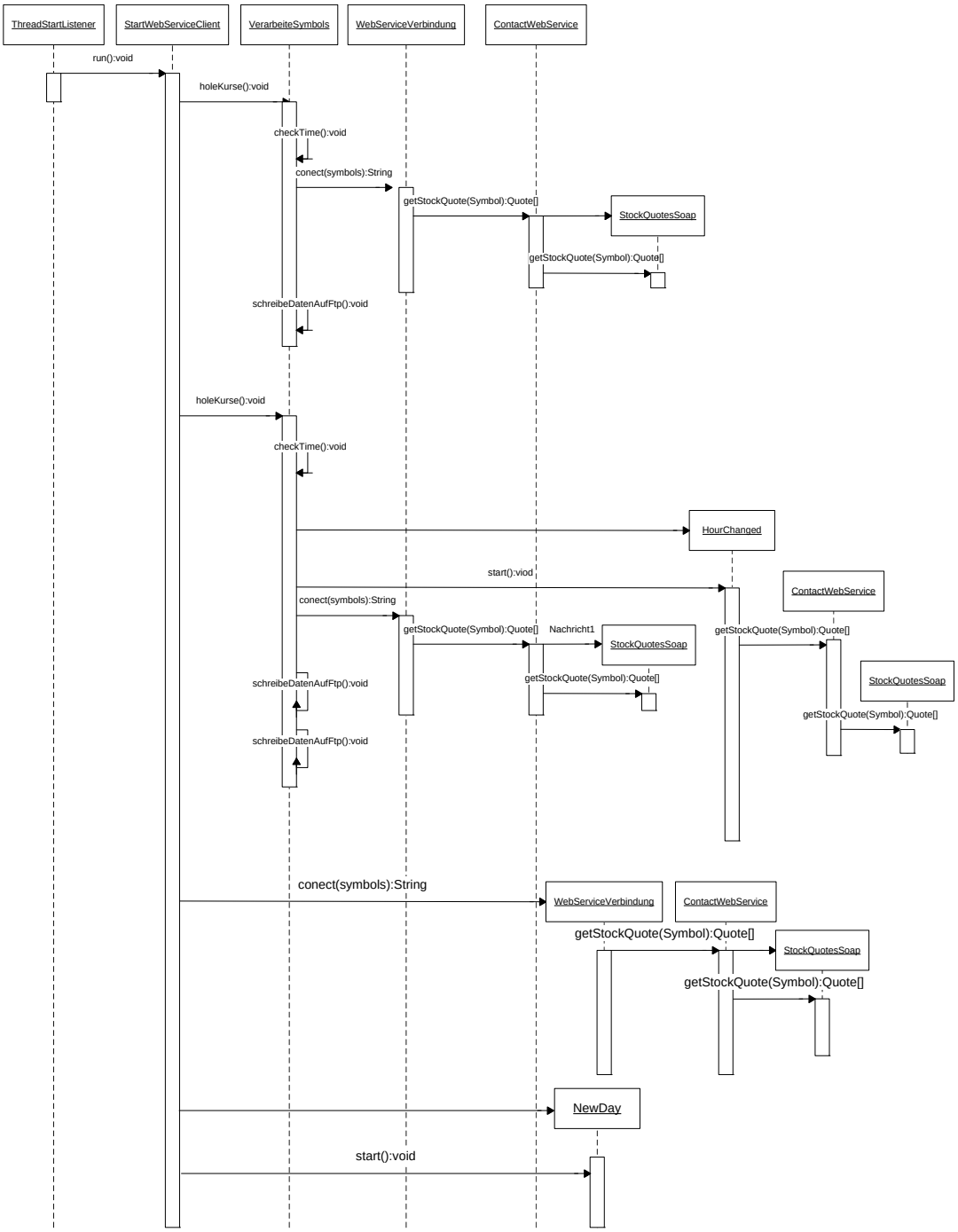

Abbildung 8.4: Sequenzdiagramm des Servers

Wird der Button „Start“ gedrückt, wird ein `Thread` gestartet. Dieser `Thread` bekommt im Konstrukor die Klasse `StartWebServiceClient` übergeben, die das Interface `Runnable` implementiert. In der Methode `run` läuft eine Endlos-Schleife, in der zwischen 09:50h und 16:20h (UTC-5) einmal in der Minute die entsprechenden Anweisungen durchgeführt werden, um Börsenkurse auf den DAB-Server zu laden. Ungefähr alle 5 Minuten wird über das Objekt `MainObjekt` (nicht im Sequenzdiagramm enthalten) das HTML-Skript aufgerufen, wodurch die auf dem FTP-Server liegenden Dateien in das DAB-Signal eingebettet werden. Auf `MainObjekt` hat quasi jedes andere Objekt eine Referenz. Dieses stellt die Verbindungen zwischen den Klassen her. Hier ist zum Beispiel das Passwort für den Sever gespeichert.

Ca. einmal in der Minute wird ausgehend von `StartWebServiceClient` die Methode `holeKurse` der Klasse `VerarbeiteSymbols` aufgerufen. In der Methode `holeKurse` wird zuerst überprüft, ob eine neue Stunde begonnen hat, dazu wird die Methode `checkTime` aufgerufen. Wenn es keinen Stundenwechsel gegeben hat, wird von dem Objekt `WebServiceVerbindung` die Methode `connect` aufgerufen. In der Methode `connect` wird zuerst die Methode `getStockQuote` des Objektes `ContactWebService` aufgerufen. Diese liefert ein `Quote[]` als Rückgabewert. In diesem `Quote[]` ist das gesamte Ergebnis der Web Service-Abfrage gespeichert.

Da ursprünglich geplant war, einen anderen Web Service zu benutzen, der jedoch unerwartet vom Netz genommen wurde, werden in dieser Methode die Werte vom neuen Web Service an die des alten angepasst. In den meisten Fällen haben die Elemente der beiden Web Services unterschiedliche Namen, die angepasst werden müssen. Zudem haben die Zeiten, die angeben, von wann die Kurse sind, ein anderes Format und müssen konvertiert werden (neues Format: 9/1/2006 4:00pm, altes Format: 10:46am). Diese „Umformatierung“ wird durchgeführt, da sonst eine Änderung der Klientseite erforderlich wäre, die die Börsendaten parsen muss.

In der Methode `getStockQuote` des Objektes `ContactWebService` wird über die mit Hilfe von Axis erzeugten Klassen auf den Web Service zugegriffen.

Als Ergebnis liefert die Methode `connect` an `VerarbeiteSymbols` den String zurück, der das XML-Dokument enthält, das per DAB übertragen wird. In der Methode `holeKurse` wird anschließend über die Methode `schreibeDatenAufFtp` das XML-Dokument auf den FTP-Server geladen, gleichzeitig wird dieses Dokument auch in eine lokale Datei geschrieben, die alle Kurse der aktuellen Stunde enthält. Der hier dargestellte Vorgang findet etwa einmal in der Minute statt.

Zu Beginn einer neuen Stunde wird der `Thread HourChanged` gestartet. In Abbildung 8.4 ist dies zu sehen, wenn zum zweiten Mal die Methode `holeKurse` aufgerufen wird. Der `Thread HourChanged` löscht auf dem Server alle Dateien der letzten Stunde und lädt diese Kurse in einer einzigen Datei wieder auf den Server, da sonst zu viele Dateien per DAB übertragen würden. Ausserdem werden einmal

pro Stunde weitere Zusatzinformatinen (siehe Anhang C), die ebenfalls übertragen werden, aktualisiert. Auf Grund der zur Verfügung stehenden Bandbreite des DAB-Signals, werden, um die Übertragung der eigentlichen Börsenkurse nicht zu behindern, diese Infomationen nur einmal in der Stunde aktualisiert. Die Zusatzinfomationen werden in einem separaten Ordner (aktienkurse/zusatz) gespeichert und enthalten auch die Zuordnung der Symbole zu den zugehörigen Firmen.

In Abbildung 8.4 ist als letzte Aktion zu sehen, wie am Ende eines Tages ein `Thread` gestartet wird. Dieser `Thread` bekommt im Konstruktor ein Objekt `NewDay` übergeben, das das Interface `Runnable` implementiert. In dem `Thread` werden sämtliche Einzelkurse gelöscht und durch einen Tageskurs ersetzt.

Die hier dargestellten Schritte werden jeden Tag automatisch durchgeführt: Wird der Startknopf auf der Web Service-Interaktionsoberfäche des Programms gedrückt, soll dieser Prozess solange stattfinden, bis der Stopknopf gedrückt wird.

8.2 Klient Seite

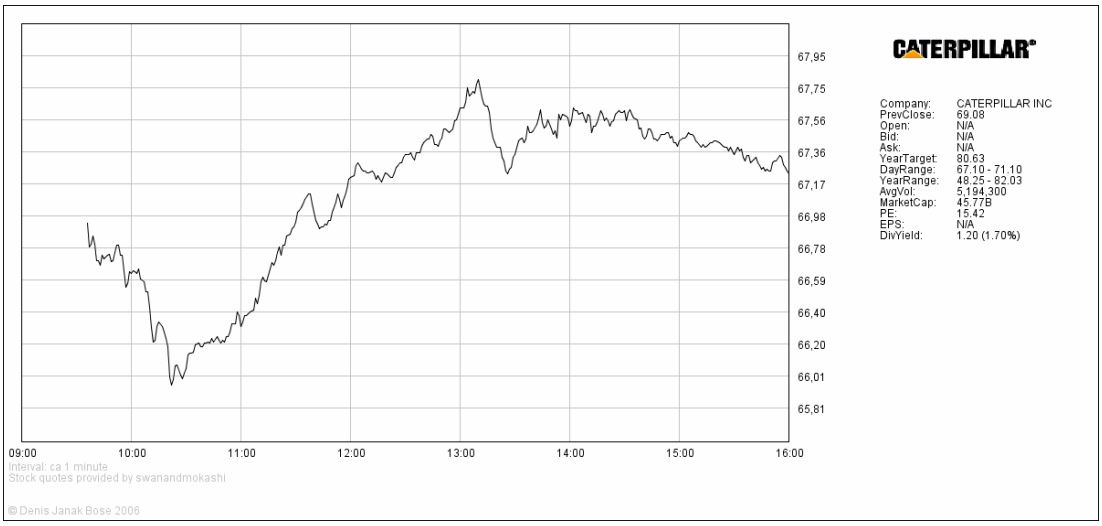

Abbildung 8.5: Klient Programm

Die Klientseite besteht aus zwei Teilen, zum einen dem Mot-Decoder und zum anderen aus einem `Thread`, der für die Darstellung und die Interaktion zuständig ist. Beide sind zwar in einem Programm untergebracht, aber der MOT-Decoder und die Darstellung wurden ursprünglich als zwei separate Programme geplant. Daher sind der MOT-Decoder und die Darstellung relativ unabhängig von einander und werden getrennt voneinander erläutert.

8.2.1 Packet Stream

Die Daten, die in dem vorliegenden Programm decodiert werden, kommen nicht unmittelbar von einem DAB-Receiver. Der DAB-Receiver wird an einem Web Server betrieben, der an der Technischen Universität München entwickelt wurde. Der Server greift die Daten von dem DAB-Receiver ab und stellt die Daten der einzelnen *Sub-Channel* über einen HTTP-Server als Stream zur Verfügung. Im Server ist ein eigener MOT-Decoder integriert und er bietet die decodierten Daten über eine HTTP-Schnittstelle an. Er stellt darüber hinaus die Rohdaten aus den *Sub-Channels* bereit, die dann als *Packet* Stream vorliegen. D. h. das implementierte Programm decodiert ab *Packet*-Level. Da auf *Packet*-Ebene kein Synchronisationsmechanismus vorliegt, fügt der Web Server in regelmäßigen Abständen vier Bytes ein (*0xff 0xfe 0xfd 0xfc*), danach beginnt relativ sicher ein *Packet*, sofern diese Bytes nicht in den Nutzdaten vorkommen.

Soll der MOT-Decoder direkt an einem Receiver angeschlossen werden, ist es erforderlich, das Programm zu erweitern. Je nach API des Receivers müsten noch einige Schritte zusätzlich implementiert werden. Im besten Fall stellt der Receiver eine Schnittstelle bereit, an der direkt der *Packet* Stream eines *Sub-Channels* abgegriffen werden kann.

8.2.2 Optionen

Folgendes XML-Dokument beschreibt die Option Datei (Options.xml), mit der einige Programmparameter eingestellt werden können. Wichtig ist, dass sie in dem gleichen Ordner wie das eigentlichen Decoderprogramm liegt. Die Datei wird beim Starten vom Klienten eingelesen.

```
<?xml version="1.0"?>
<Optionen Beschreibung="MOT Optionen" >
  <Ordner>e:\mot\</Ordner>
  <DAB-Server>
    <URL>project.informatik.uni-osnabrueck.de</URL>
    <Port>8000</Port>
    <Stream>/stream/6</Stream>
  </DAB-Server>
  <Rahmen-Groesse>
    <X>1200</X>
  </Rahmen-Groesse>
  <Antialiasing>An</Antialiasing>
</Optionen>
```

- **Ordner:** Hier steht der Ordner, in dem die empfangenen Dateien gespeichert werden. OBACHT: Dateien innerhalb dieses Ordners, die nicht im MOT *Directory* enthalten sind, werden in regelmäßigen Abständen gelöscht.
- **DAB-Server:**
  - **URL:** Hier wird die URL des Web-Servers angegeben, von dem das Programm den *Packet Stream* bezieht.
  - **Port:** Hier steht der Port des Web-Servers.
  - **Stream:** Hier steht die genaue Bezeichnung des MOT *Streams*.
- **Rahmen-Groesse:** Hier steht die X-Größe des Fensters.
- **Antialiasing:** Hier kann eingestellt werden, ob Antialiasing zum Zeichnen der Aktienkurse verwendet werden soll. Der Wert „An“ bedeutet, dass Antialiasing benutzt wird. Bei jedem beliebigen anderen Wert wird kein Antialiasing benutzt.

8.2.3 Decoder

Da die Decodierung des MOT *Streams* und die Darstellung der Börsenkurse relativ unabhängig voneinader sind, werden die beiden Teile getrennt dargestellt. In diesem Abschnitt wird der Aufbau und die Funktionsweise des Decoders beschrieben.

8.2.3.1 Klassendiagramm des Decoders Abbildung 8.6 stellt das Klassendiagramm für den MOT-Decoder dar.

8.2.3.2 Programmablauf des Decoders In Abbildung 8.7 ist in einem vereinfachten Sequenzdiagramm der Ablauf des Programms vom Einschalten bis zum Speichern einer Datei auf der Festplatte dargestellt.

Zuerst wird in der Klasse `SynchronizeStream` ein `URL`-Objekt erzeugt, welches zu der in der „Option.xml“ angegebenen URL eine Verbindung aufbaut. Von dieser Verbindung kann ein `InputStream` geholt werden. In diesem Stream kommt der *Packet Stream* byteweise an. Im nächsten Schritt muss in diesem Stream der Einstiegspunkt gefunden werden, in Form des Bytes, bei dem ein *Packet* beginnt. Dazu wird mit Hilfe eines Zustandsautomaten nach den richtigen Bytes gesucht (*0xff 0xfe 0xfd 0xfc*). Wurde der Anfang eines *Packets* gefunden, werden solange Bytes gesammelt, bis die genannten vier Bytes erneut auftreten. Die gesammelten Bytes werden der statischen Methode `extractPackets` in der Klasse `PacketStreamDecoder` übergeben.

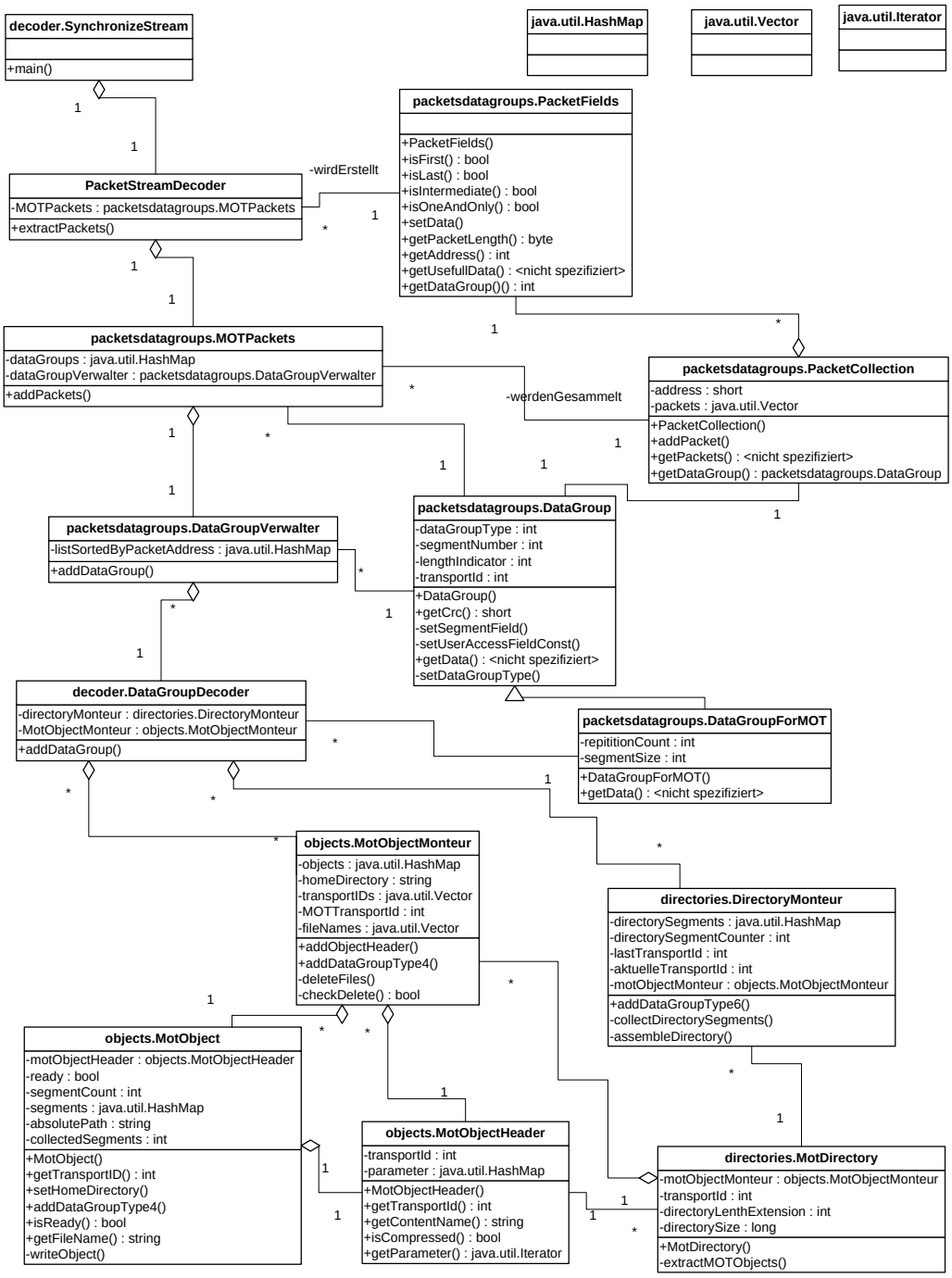

Abbildung 8.6: Klassendiagramm des Decoders

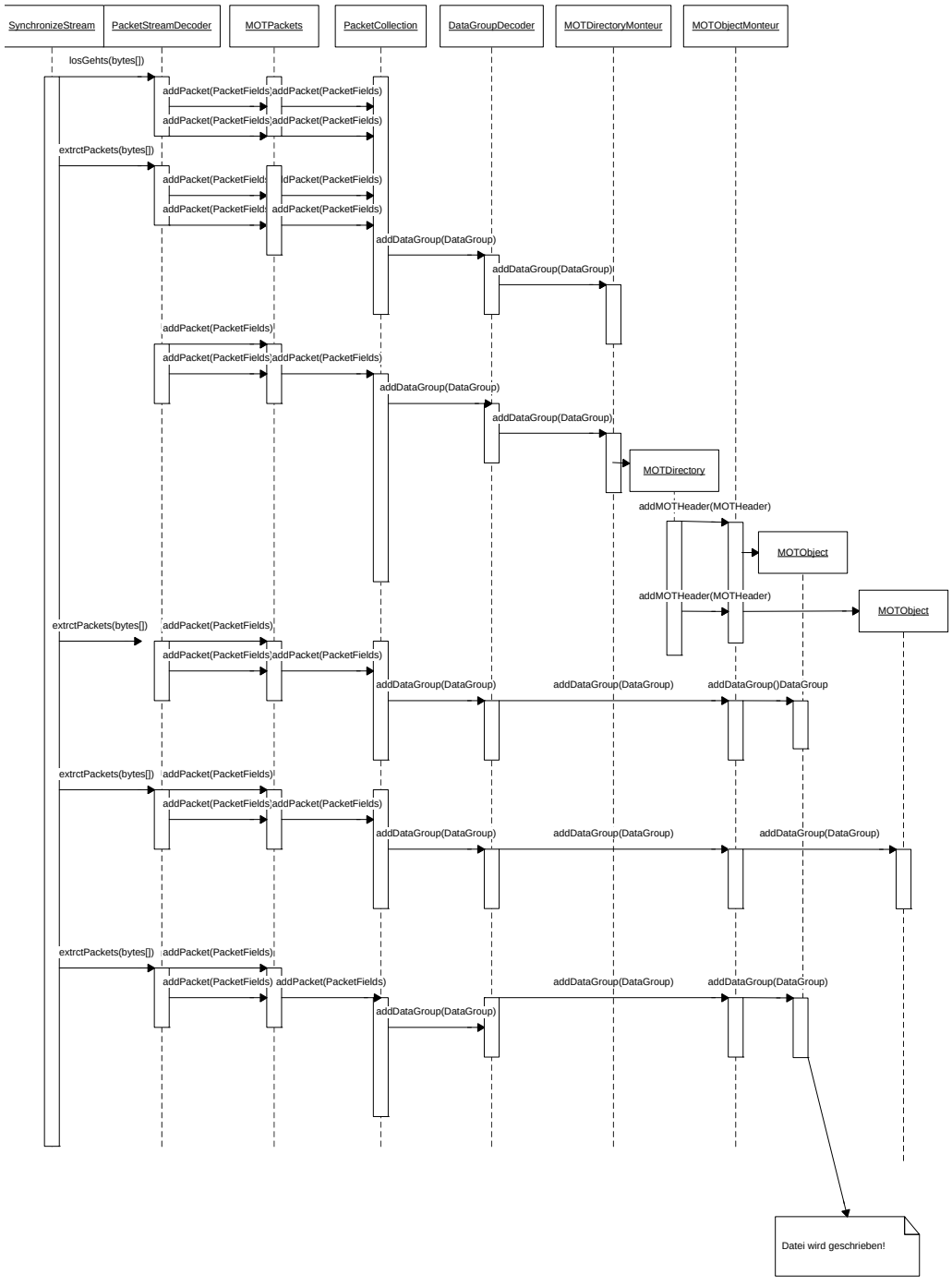

Abbildung 8.7: Sequenzdiagram des Decoders

In der Methode `extractPackets` werden aus dem Array die einzelnen *Packets* extrahiert. In der Regel ist das `Array` 192 Bytes groß und enthält meistens zwei bis vier *Packets*. Da bekannt ist, dass das erste Byte des Arrays der Anfang eines *Packets* ist, muss dieses nur in seine Bitdarstellung umgewandelt werden. Die ersten beiden Bits geben die Länge des *Packets* an. Auf diese Weise können dann die *Packets* aus dem Array extrahiert werden. Aus den Bytes, die zu einem *Packet* gehören, wird ein `PacketFields`-Objekt, welches nicht im Sequenzdiagramm zusehen ist, erzeugt. Über dieses Objekt kann per get-Methode auf alle Felder des *Packets* zugegriffen werden. Um die *Packet*-Felder zu bestimmen, werden die Bytes sukzessive in ihre Bitdarstellung umgewandelt. Die Klasse `PacketStreamDecoder` hat ein static Objekt `MOTPackets`, dem die *Packets* übergeben werden.

In der Klasse `MOTPackets` werden die *Packets* jeweils nach *Packet*-Adresse in einer `PacketCollection` zusammengefasst in einer `HashMap` gespeichert. Wie in in Abschnitt zwei erläutert, ergeben mehrere zusammenhängende *Packets* mit gleicher Adresse eine *Data Group*. Sobald ein *Packet* ankommt, bei dem das Feld *First/Last* anzeigt, dass dieses das letzte *Packet* einer *Data Group* ist, wird ein `DataGroupForMOT`-Objekt erzeugt (dieses ist nicht in dem Sequenzdiagramm zu sehen) und bekommt im Konstruktor die `PacketCollection` mit den zusammenhängenden *Packtes* übergeben. In dem `DataGroupForMOT`-Objekt werden die Nutzdaten der *Packets* zusammengehängt, der *Data Group Header* bestimmt, in dem die die jeweiligen Bytes in ihre Bitdarstellung zerlegt werden und in einem Byte-Array die Nutzdaten der *Data Group* gespeichert. Die `DataGroupForMOT` wird dem `DataGroupVerwalter` übergeben. Dort wird der CRC-Wert der *Data Group* überprüft und die *Data Group* gegebenenfalls verworfen.

Der `DataGroupVerwalter` hat für jede *Packet*-Adresse einen `DataGroupDecoder`. In dem `DataGroupDecoder` gibt es für die verschiedenen *Data Group*-Typen eigene Decoder. Implementiert sind derzeit zwei Decoder

1. **Type 4:** Für ein nicht komprimiertes MOT *Directory*.
2. **Type 6:** Für die eigentlichen MOT-Objekte (Dateien).

Offenbar werden zur Zeit nur diese *Data Group* Typen in Deutschland verwendet. Andere Typen werden für CA, den *Header Mode* und für komprimierte MOT *Directories* verwendet. Für die Decodierung der Börsenkurse sind die beiden Decoder ausreichend.

Je nach *Data Group* wird die *Data Group* weiter gereicht. Im Sequenzdiagramm werden zuerst einige *Data Groups* Type 4 empfangen, also MOT *Directory*-Segmente, die dem `DirectoryMonteur` übergeben werden. Der `DirectoryMonteur` sammelt jetzt so lange *Data Groups*, bis er genügend hat, um ein *Directory* zu zusammenzusetzen. Anhand der *SegmentNumber* im *Session Field* der *Data Group*

und der Angabe, ob es sich um das letzte Segment handelt, kann die Gesamtanzahl der zu einem MOT-*Directory* gehörenden *Data Groups* bestimmt werden.

Sind alle *Data Groups* eines MOT *Directories* empfangen, werden ihre Nutzdaten geordnet in ein Byte-Array abgelegt und ein `MOTDirectory`-Objekt erzeugt, welchem im Konstruktor das Byte-Array übergeben wird. Das `MOTDirectory` bestimmt zuerst den *Directory Header*, in dem die Bytes wieder in ihre Bitdarstellung zerlegt werden und dann den Feldern zugeordnet werden. Dann werden nach und nach die MOT-Header aus den restlichen Bytes extrahiert und es wird jeweils ein `MotObjectHeader`-Objekt erstellt. Die *Header* werden dem `MotObjectMonteur` übergeben. Dies ist das gleiche Objekt wie im `DataGroupDecoder`.

Der `MotObjectMonteur` erzeugt zu jedem `MotObjectHeader` ein `MotObject`, welches nach und nach die Daten für die später zu schreibende Datei sammelt. Ankommende *Data Groups* vom Typ 4 werden im `DataGroupDecoder` dem `MotObjectMonteur` übergeben. Die *Data Groups* können anhand der *Tranport Id* im *User Access Field* zugeordnet werden. Im `MotObjectHeader` ist die jeweilige *Transport Id* angegeben. Die *Data Groups* werden in in ihrem `MotObject` gespeichert. Sind alle *Data Groups* eines `MotObject` gesammelt, schreiben die MOT-Objekte sich sozosagen selbst auf die Festplatte.

Einige Dateien, vor allem textbasierte Dateien, werden im gzip-Format übertragen. Ob eine Datei komprimiert ist, steht im `MotObjectHeader`. Das SDK [**?**] hat den gzip-Algorithmus im Package `java.util.zip.GZIPInputStream` implementiert. Komprimierte Dateien können so sehr einfach entpackt werden.

Dieser Vorgang wiederholt sich nun immer wieder. Wird ein neues *Directory* empfangen, werden nur die neuen Dateien dieses *Directories* im `MOTObjektMonteur` gesammelt.

8.2.4 Darstellung

In diesem Abschnitt wird erläutert, wie die Börsenkurse dargestellt und verwaltet werden.

8.2.4.1 Klassendiagramm der Darstellung Die Klassen, die die grafische Darstellung der Kurse und die Logik für die Interaktion übernehmen, sind in Abbildung 8.8 zu sehen.

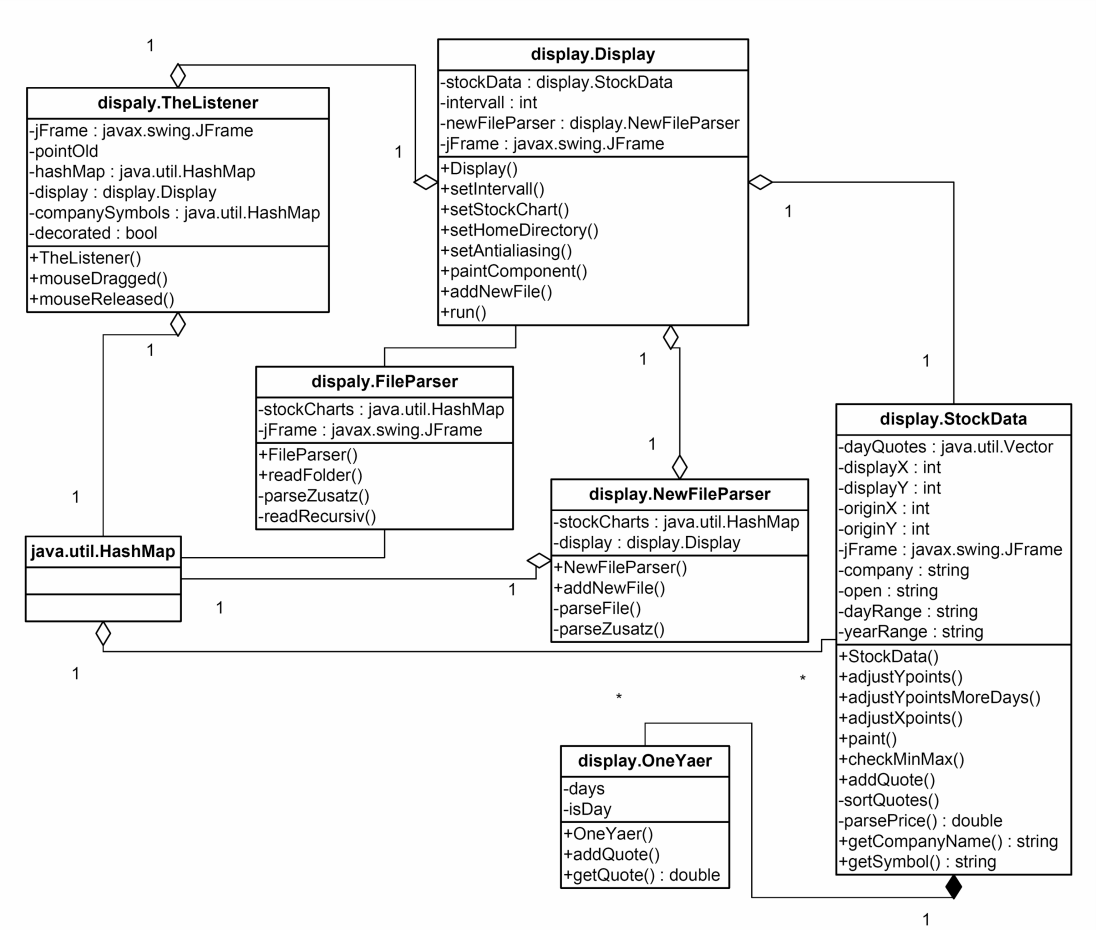

Abbildung 8.8: Klassendiagramm der Darstellung

8.2.4.2 Programmablauf der Darstellung Der bisher beschriebene Vorgang dient ausschließlich zur Decodierung der Dateien. In der Klasse `MOTObject-Monteur` besteht die Verbindung zwischen der Darstellung und dem Decoder. Die Darstellung wird in einem anderen `Thread` ausgeführt, die Klasse `Display` implementiert das Interface `Runnable` und ist Ausgangspunkt des `Threads`. Wenn ein neues Objekt vollständig empfangen wurde, ruft der `MOTObjectMonteur` beim `Display`-Objekt eine Methode auf und übergibt dem `Display` den Dateinamen der neuen Datei. Der Vorgang und die darauf folgenden Methodenaufrufe sind im Sequenzdiagramm in Abbildung 8.9 zu sehen.

Für die grafische Darstellung und die Interaktion sind die Klassen `Display`, `StockData`, `FileParser`, `NewFileParser` und `TheListener` zuständig. `Display` ist von `JPanel` abgleitet, um Zugriff auf das `Graphics`-Objekt zu erlangen und hat das Interface `Runnable` implementiert.

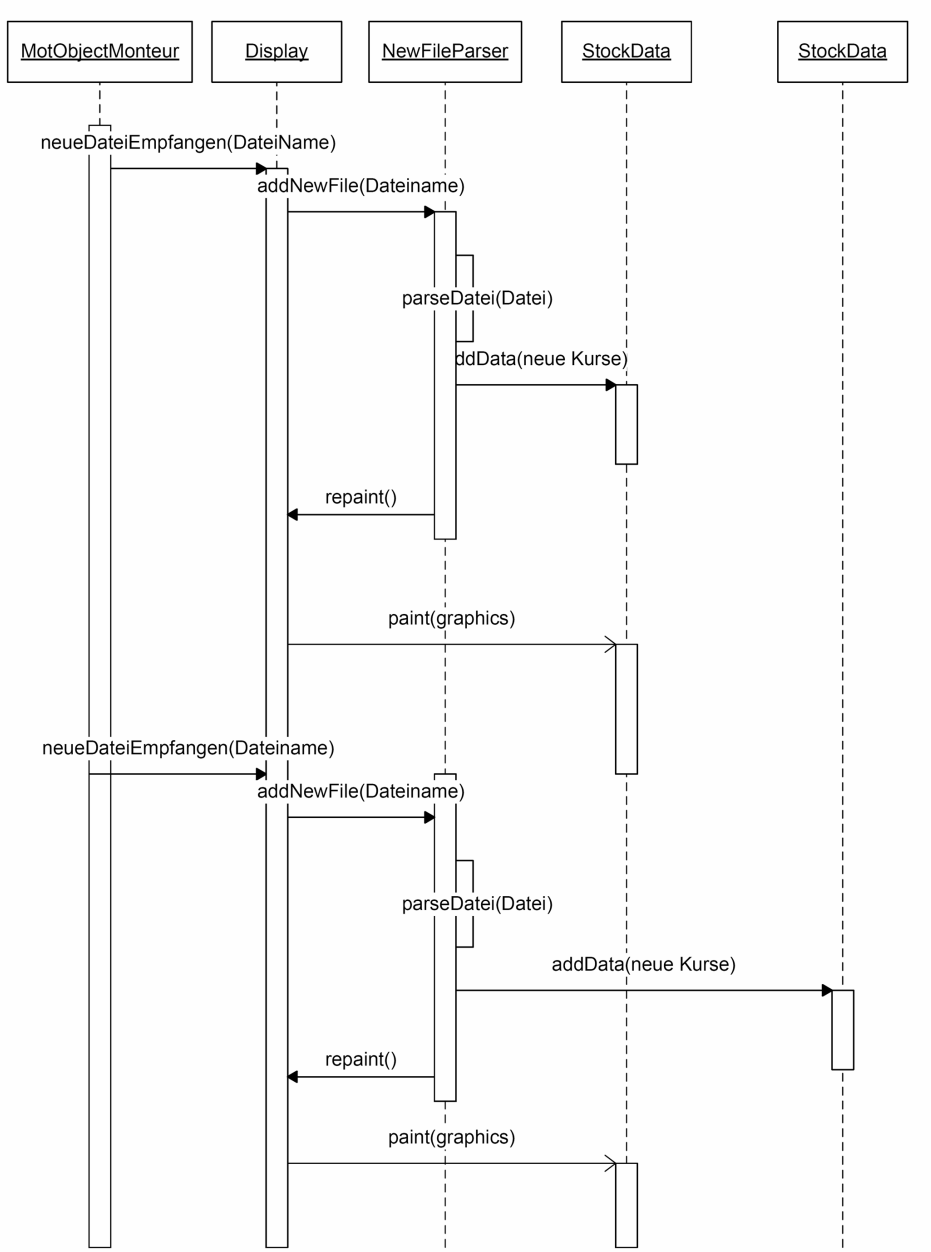

Abbildung 8.9: Sequenzdiagram des Displays

Wird die `run`-Methode des `Displays` ausgeführt, wird ein Objekt `FileParser` erstellt. Dieses durchsucht alle Dateien des Ordners, in dem der MOT-Decoder die empfangenen Dateien speichert, um schon vorhandene Börsenkurse darzustellen. Alle vorhandenen XML-Dateien in den Ordnern „all“ und „zusatz“ werden mit

einem `DocumentBuilder` geparst, der aus den XML-Dateien jeweils ein *Document Object Model* erzeugt. Für jedes gefundene Firmen-Symbol (beispielsweise msft für Microsoft) wird ein `StockData`-Objekt erstellt. Diese Objekte enthalten alle Daten, die bezüglich eines Firmensymbols vorliegen. Alle Objekte `StockData` werden in einer `HashMap` mit ihrem jeweilegen Firmensymbol als Schlüssel gespeichert. Eine Referenz auf diese `HashMap` erhält `NewFileParser` und `TheListener`.

`NewFileParser` funktioniert ähnlich wie `FileParser`, parst im Unterschied dazu jedoch Dateien, die während des Betriebs neu empfangen werden. Der `TheListener` ist für die Interaktion mit der Maus zuständig und implementiert `MouseMotionListener` und `MouseListener`. Folgende Funktionen hat der `TheListener` implementiert:

- **Doppelklick:** Mit einem Doppelklick kann der Rahmen aktiviert/ deaktiviert werden.
- **Rechte Maustaste:** Mit der rechten Mausttste wird navigiert: Wahl der darzustellenden Firma, Wahl der Zeitperiode des Aktienkurses und Exit.
- **Maus Bewegung:** Ist der Rahmen deaktiviert, so kann durch Drücken einer Maustaste und gleichzeitigem Bewegen das Fenster verschoben werden.

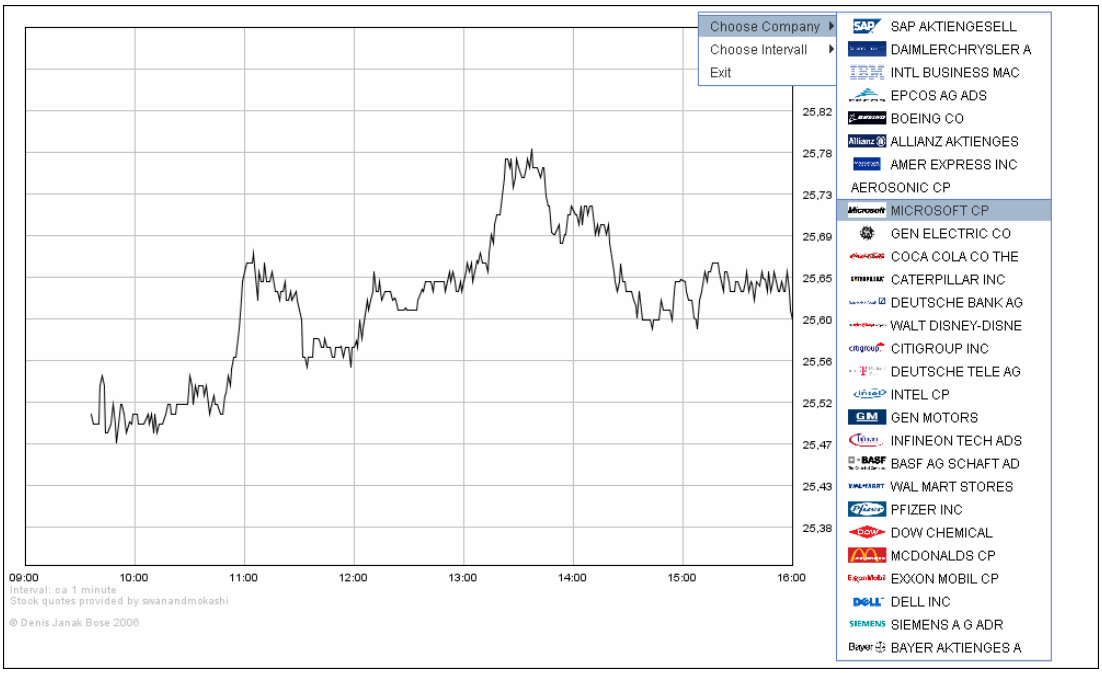

Abbildung 8.10: Interaktion mit dem Klienten

Der `TheListener` hat eine Referenz auf den `Display` und übergibt diesem das gerade aktive `StockData`-Objekt, welches über die rechte Maustaste ausgewählt

werden kann. Über ein `int` wird dem `Display` mitgeteilt, welcher Chart (Tag, Jahr etc.) gezeichnet werden soll.

Das Zeichnen der Kurse übernehmen die Objekte `StockData` selbst. Das `Display` übergibt dem `StockData`-Objekt, auf das es gerade eine Referenz hat, das `Graphics`-Objekt zum Zeichnen. Zuerst wird das Raster mit Hilfe der `drawLine`-Methode des `Graphics`-Objekt gezeichnet. Dann werden die Angaben an der Seite mit der Methode `drawString` gezeichnet. Bevor der Chart gemalt werden kann, müssen die Kurse, wie in Kapitel 7 bereits erwähnt, erst an das Koordinatensystem des Monitors angepasst und gegebenfalls sortiert werden. Für den Tageschart wird für die Y-Koordinaten eine eigene Methode verwandt: `adjustYpoints`, da die Minutenkurse des aktuellen Tages anders gespeichert werden als die Kurse früherer Tage. Pro Jahr existiert in jedem `StockData`-Objekt ein Objekt `OneYaer`, welches die früheren Tageskurse enthält. Für die Perioden, in denen mehrere Tage dargestellt werden, und für die X-Achse geschieht dies in derselben Methode (`adjustYpointsMoreDays` und `adjustXpoints`). Der Chart wird dann mit der Methode `drawPolyline` des `Graphics`-Objektes gemalt.

Die Logos der Firmen werden mit Hilfe der Klasse `Toolkits` erstellt. Folgender Code zeigt, wie mit Hilfe der Klasse `Toolkit` eine Grafik gemalt werden kann:

```
Garphics g;

Image image = Toolkit.getDefaultToolkit().getImage("logo.gif");

g.drawImage(image, 100, 100, 50 , 50, jFrame);
```

Die Parameter der Methode `drawImage` geben die Position und Größe des Bildes an.

Insgesamt wurde angestrebt, die Größen relativ zu halten. Die gesamte grafische Darstellung wird immer relativ zur Größe des umgebenen `JFrames` dargestellt.

9 Fazit

Ziel dieser Bachelorarbeit war es, einen MOT-Decoder zu entwickelen. Um diesen zu testen, werden Börsenkurse per DAB verschickt, die der Decoder aus einem MOT *Stream*, der über das Internet übertragen wird, decodiert und grafisch darstellt. Hiezu wurden zwei Apllikationen entwickelt: Ein Server, der jeden Tag Börsenkurse von einem Web Service auf den DAB-Server lädt und ein Decoder, der einen *Packet Stream* decodiert und die Kurse grafisch darstellt.

Obwohl der Decoder derzeit mit Daten, die über das Internet übertragen werden, arbeitet, könnte sehr wahrscheinlich ein Grossteil der entwickelten Klassen auch auf einem Java-fähigem mobilen DAB-Receiver genutzt werden. Es müste je nach API des Receivers noch eine Schnittstelle zwischen dem Receiver und dem MOT-Decoder implementiert werden. Es wäre sicher auch ein interessante Aufgabe, einen vollständigen DAB-Decoder zu implementieren, der den *Fast Information Channel* decodiert und die Daten der einzelnen *Sub-Channel* bereit stellt.

Es wurde gezeigt, dass die Übertragung aktueller Daten über das DAB-System relativ gut automatisiert möglich ist. Die Kurse kommen in der Regel innerhalb weniger Minuten bei dem Empfänger an. Sollen in noch kürzeren Abständen (weinger als ca. drei Minuten) aktuelle Daten verbreitet werden, kann dies mit DAB möglicherweise nicht mehr geeignet bewerkstelligt werden. Der Teil der Arbeit, der für die Darstellung der Börsenkurse verwendet wird, kann auf einem mobilen Gerät auf Grund der normalerweise sehr kleinen Displays mobiler Geräte wohl nicht verwendet werden.

Insgesammt wurde das Ziel der Aufgabenstellung erreicht.

A WSDL Datei des Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:s0="http://swanandmokashi.com"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 targetNamespace="http://swanandmokashi.com"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://swanandmokashi.com">
      <s:element name="GetQuotes">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="QuoteTicker" type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetQuotesResponse">
        <s:complexType>
          <s:sequence>
            <s:element maxOccurs="1" name="GetQuotesResult" type="s0:ArrayOfQuote"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="ArrayOfQuote">
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="unbounded" name="Quote" type="s0:Quote"/>
        </s:sequence>
      </s:complexType>
      <s:complexType name="Quote">
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="CompanyName" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="StockTicker" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="StockQuote" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="LastUpdated" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="Change" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="OpenPrice" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="DayHighPrice" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="DayLowPrice" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="Volume" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="MarketCap" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="YearRange" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="ExDividendDate" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="DividendYield" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="DividendPerShare" type="s:string"/>
          <s:element minOccurs="0" maxOccurs="1" name="PercentChange" type="s:string"/>
        </s:sequence>
      </s:complexType>
      <s:element name="ArrayOfQuote" nillable="true" type="s0:ArrayOfQuote"/>
    </s:schema>
  </types>
  <message name="GetQuotesSoapIn">
    <part name="parameters" element="s0:GetQuotes"/>
  </message>
  <message name="GetQuotesSoapOut">
    <part name="parameters" element="s0:GetQuotesResponse"/>
  </message>
  <message name="GetQuotesHttpGetIn">
    <part name="QuoteTicker" type="s:string"/>
  </message>
```

```
<message name="GetQuotesHttpGetOut">
  <part name="Body" element="s0:ArrayOfQuote"/>
</message>
<message name="GetQuotesHttpPostIn">
  <part name="QuoteTicker" type="s:string"/>
</message>
<message name="GetQuotesHttpPostOut">
  <part name="Body" element="s0:ArrayOfQuote"/>
</message>
<portType name="StockQuotesSoap">
  <operation name="GetStockQuotes">
    <input name="GetQuotes" message="s0:GetQuotesSoapIn"/>
    <output name="GetQuotes" message="s0:GetQuotesSoapOut"/>
  </operation>
</portType>
<portType name="StockQuotesHttpGet">
  <operation name="GetStockQuotes">

    <input name="GetQuotes" message="s0:GetQuotesHttpGetIn"/>
    <output name="GetQuotes" message="s0:GetQuotesHttpGetOut"/>
  </operation>
</portType>
<portType name="StockQuotesHttpPost">
  <operation name="GetStockQuotes">

    <input name="GetQuotes" message="s0:GetQuotesHttpPostIn"/>
    <output name="GetQuotes" message="s0:GetQuotesHttpPostOut"/>
  </operation>
</portType>
<binding name="StockQuotesSoap" type="s0:StockQuotesSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/http" style="document"/>
  <operation name="GetStockQuotes">
    <soap:operation soapAction="http://swanandmokashi.com/" style="document"/>
    <input name="GetQuotes">
      <soap:body use="literal"/>
    </input>
    <output name="GetQuotes">
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<binding name="StockQuotesHttpGet" type="s0:StockQuotesHttpGet">
  <http:binding verb="GET"/>
  <operation name="GetStockQuotes">
    <http:operation location="/GetQuotes"/>
    <input name="GetQuotes">
      <http:urlEncoded/>
    </input>
    <output name="GetQuotes">
      <mime:mimeXml part="Body"/>
    </output>
  </operation>
</binding>
<binding name="StockQuotesHttpPost" type="s0:StockQuotesHttpPost">
  <http:binding verb="POST"/>
  <operation name="GetStockQuotes">
    <http:operation location="/GetQuotes"/>
    <input name="GetQuotes">
      <mime:content type="application/x-www-form-urlencoded"/>
    </input>
    <output name="GetQuotes">
      <mime:mimeXml part="Body"/>
    </output>
  </operation>
</binding>
```

```
  <service name="StockQuotes">
    <port name="StockQuotesSoap" binding="s0:StockQuotesSoap">
      <soap:address location="http://www.swanandmokashi.com/WebServices/StockQuotes.asmx"/>
    </port>
    <port name="StockQuotesHttpGet" binding="s0:StockQuotesHttpGet">
      <http:address location="http://www.swanandmokashi.com/WebServices/StockQuotes.asmx"/>
    </port>
    <port name="StockQuotesHttpPost" binding="s0:StockQuotesHttpPost">
      <http:address location="http://www.swanandmokashi.com/WebServices/StockQuotes.asmx"/>
    </port>
  </service>
</definitions>
```

B XML-Struktur der zu übertragenden Börsenkurse

```
<?xml version="1.0" encoding="UTF-8" ?>
<begin>
<StockQuotes>
 <Date>
   <Day>250</Day>
   <Year>2006</Year>
 </Date>
 <StockQuote>
   <Symbol>MSFT</Symbol>
   <Price><big><b>25.63</b></big></Price>
   <Time>9:45am</Time>
 </StockQuote>
 <StockQuote>
   <Symbol>DOW</Symbol>
   <Price><big><b>37.0453</b></big></Price>
   <Time>9:45am</Time>
   </StockQuote>
 <StockQuote>
   <Symbol>AIM</Symbol>
   <Price><big><b>7.50</b></big></Price>
   <Time>9:45am</Time>
 </StockQuote>
</StockQuotes>
<StockQuotes>
 <Date>
   <Day>250</Day>
   <Year>2006</Year>
 </Date>
```

```
 <StockQuote>
   <Symbol>MSFT</Symbol>
   <Price><big><b>25.61</b></big></Price>
   <Time>9:47am</Time>
 </StockQuote>
 <StockQuote>
   <Symbol>DOW</Symbol>
   <Price><big><b>37.02</b></big></Price>
   <Time>9:47am</Time>
 </StockQuote>
 <StockQuote>
    <Symbol>AIM</Symbol>
    <Price><big><b>7.50</b></big></Price>
    <Time>9:47am</Time>
  </StockQuote>
</StockQuotes>
</begin>
```

C XML-Struktur der Zusatzinformationen

```
<?xml version="1.0" encoding="UTF-8" ?>
<begin>
  <StockQuote xmnls="">
     <Symbol>MCD</Symbol>
     <Company>MCDONALDS CP</Company>
     <Price>36.52</Price>
     <Time>9/7/2006 3:41pm</Time>
     <Change>+0.30</Change>
     <Open>36.12</Open>
     <DayRange>36.05 - 36.61</DayRange>
     <YearRange>31.31 - 36.80</YearRange>
     <Volume>3554500</Volume>
     <MarketCap>44.785B</MarketCap>
     <DivYield>1.85</DivYield>
  </StockQuote>
</begin>
```

D Inhalt der CD-Rom

arbeit	Bachelorarbeit
docs	ETSI Dokumente
KursDecoder	**Decoder**
KursDecoder/ausführbar	ausführbares Programm
KursDecoder/source	Quellcode
KursServer	**KursServer**
KursDecoder/ausführbar	ausführbares Programm
KursDecoder/source	Quellcode
libs	**Bibliotheken**
libs/apache	Bibliotheken
software	**benötigte Software**
software/java	Java 2 SDK 1.5.0.0.8

E Literaturverzeichnis

Literatur

[Apac2006] **Apache Software Foundation**
Apache Jakarta Commons HTTPCLIENT
http://jakarta.apache.org/commons/httpclient/

[Apac2006a] **Apache Software Foundation**
Apache Jakarta Commons Net
http://jakarta.apache.org/commons/net/

[Apac2006b] **Apache Software Foundation**
Apache Web Services Axis
http://ws.apache.org/axis/

[Apac2006c] **Apache Software Foundation**
Apache Software Foundation
http://apache.org/

[Etsi2006] **ETSI**
ETSI EN 300 401 v1.4.1 Radio Broadcasting Systems;
Digital Audio Broadcasting (DAB) to mobile,portable and fixed receivers
http://www.etsi.org/

[Etsi2006a] **ETSI**
ETSI EN 301 234 V2.1.1 Digital Audio Broadcasting (DAB);
Multimedia Object Transfer (MOT) protocol
http://www.etsi.org/

[Etsi2006c] **ETSI**
ETSI TS 101 499 V2.1.1 Digital Audio Broadcasting (DAB);
MOT Slide Show; User Application Specification
http://www.etsi.org/

[Etsi2006b] **ETSI**
ETSI TS 101 756 V1.3.1 Digital Audio Broadcasting (DAB);
Registered Tables
http://www.etsi.org/

[KuWo2002] **Kuschke, Michael; Woelfe Ludger**
Web Services kompakt
Spektrum Akademischer Verlag, Heidelberg Berlin 2002

[Ietf1985] **IETF**
Request for Comments: 959
FILE TRANSFER PROTOCOL (FTP)
http://www.ietf.org/rfc/rfc0959.txt?number=959

[Ietf1998] **IETF**
Request for Comments: 2396
Uniform Resource Identifiers (URI): Generic Syntax
http://www.ietf.org/rfc/rfc2396.txt?number=2396

[Lang2003] **Langer, Torsten**
Web Services mit Java
Neuentwicklung und Refactoring in der Praxis
Markt + Technik Verlag, München 2003

[Laut1996] **Lauterbach, Thomas**
Digital Audio Broadcasting
Grundlagen, Anwendungen und Einführung von DAB
Franzis-Verlag, Feldkirchen 1996

[Micro2006] **Sun Microsystems**
JavaTM 2 Platform Standard Edition 5.0
API Specification
http://java.sun.com/j2se/1.5.0/docs/api/

[Micro2006a] **Sun Microsystems**
Installing and Configuring SSL Support
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security6.html

[Micro2006b] **Sun Microsystems**
Monitoring and Management Using JMX
http://java.sun.com/j2se/1.5.0/docs/guide/management/agent.html

[Mind2006] **Mindreef**
Testservice für Web Services
http://www.mindreef.net/

[Moka2006] **Mokaschi, Swanand**
Börsendaten Web Service
http://www.swanandmokashi.com/HomePage/WebServices/

[Norg2003] **Norguet, Jean-Pierre**
Java FTP client libraries reviewed
http://www.javaworld.com/

[Nyse2006] **NYSE**
New York Stock Exchange, Listed Company Directory
http://www.nyse.com/about/listed/listed.html

[SeWa1996] **Sedgewick, Robert; Wayne, Kevin**
Introduction to Computer Science
http://www.cs.princeton.edu/introcs/home/

[Stri2006] **STRIKEIRON**
Kostenpflichtige Web Service
http://www.strikeiron.com/

[Ulle2006] **Ullenboom, Christian**
Java ist auch eine Insel
Programmieren für die Java 2-Plattform in der Version 5
http://www.galileocomputing.de/openbook/javainsel5/

[Vorn2005] **Vornberger, Oliver**
Datenbanksysteme
http://www-lehre.inf.uos.de/ dbs/2005/skript/skript.html

[Xmet2006] **XMETHODS**
Web Service Verzeichnisdienst
http://www.xmethods.com/

Erklärung

Hiermit erkläre ich, dass ich die Bchelorarbeit selbstständig angefertigt und keine anderen Quellen und Hilfsmittel außer den in der Arbeit angegebenen benutzt habe.

Osnabrück, den 09.09.2006 .....

Denis Janak Bose