Konzeptuelle Datenmodellierung für die inhaltsbasierte Suche in Kollektionen von digitalen Bildern

Diplomarbeit

Universität Rostock
Institut für Informatik

Vorgelegt von: Sebastian W.H. Czymaj
Geboren am: 22. Mai 1981 in Rostock
Erstgutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter: Prof. Dr.-Ing. habil. Peter Forbrig
Betreuer: Dipl.-Ing. Temenushka Ignatova
Dipl.-Inf. Ilvio Bruder
Abgabetermin: 31.05.2005

Zusammenfassung

In der vorliegenden Diplomarbeit wurde ein generisches Datenmodell entwickelt, mit dessen Hilfe Inhaltsmerkmale digitaler Bilder beschrieben werden können. Zu diesem Zweck wird ein Framework aus UML-Klassen und Beziehungen bereitgestellt. Ausgehend von diesem abstrakten Modell kann der Nutzer seine konkreten Bilddaten beschreiben.

Im Anschluss an die Modellierung erfolgt eine Transformierung der modellierten Klassen in das Datenmodell einer konkreten Zielplattform, wie beispielsweise einer objektrelationalen Datenbank. Diese Transformation wurde in dieser Arbeit prototypisch implementiert.

Als Zielplattformen wurden dafür die Objektrelationale Datenbank *IBM DB2 Version 8.1* sowie der *IBM Content Manager Version 8.1* verwendet. Als Beispiel diente die Speicherung digitalisierter Bilder von Notenblättern aus dem eNoteHistory-Projekt, deren Inhalt hier modelliert und transformiert wurden.

Abstract

In this diploma thesis a generic image data model was developed. Its purpose is describing content characteristics of digital images. Therefore, a framework consisting of UML classes and associations is provided. The user can build her/his own model describing the application specific image content based on this abstract model.

After designing the model a transformation step follows. In this step a model for a specific target platform is built from the users model, e.g. for an object-relational database. This transformation has been implemented as a prototype in this thesis.

As target platforms the object-relational database *IBM DB2 Version 8.1* and *IBM Content Manager Version 8.1* were used. The digital images of music manuscript pages from the eNoteHistory project were used as an example for an application specific model.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Konventionen

Abkürzungen und spezielle Fachbegriffe werden bei ihrem ersten Vorkommen in *kursiver Schrift* notiert und gegebenenfalls erklärt. Eine Übersicht aller Abkürzungen ist im Abkürzungsverzeichnis (Anhang A) zu finden.

Literaturverweise werden in der Regel unter Angabe der ersten drei Buchstaben des Autors, bzw. unter Angabe der ersten Buchstaben der Autoren, gefolgt vom Erscheinungsjahr notiert.

Begriffe aus konkreten Sprachen bzw. Modellen, z.B. SQL-Befehle, werden, sofern es an entsprechender Stelle sinnvoll ist, in einer `proportionalen Schrift` dargestellt.

Alle Bezeichnungen im konkreten Datenmodell und der Transformation (ab Kapitel 4) sowie der zugehörigen Komponenten sind in englischer Sprache verfasst.

1.2 Motivation

Für die Extraktion von Bildinhalten werden komplexe Algorithmen aus dem Bereich Computer Vision eingesetzt. Die Ergebnisse dieser Verfahren stellen verschiedene Inhaltsmerkmale dar. Sie ermöglichen die Suche nach inhaltlichen Ähnlichkeiten in digitalen Bildern. Hierzu wurden verschiedene Maße für die Ähnlichkeit zwischen Bildern entwickelt, die den Kern inhaltsbasierter Bild-Retrieval-Anwendungen ausmachen.

Um die Skalierbarkeit, die Performance und die Robustheit solcher Anwendungen zu gewährleisten, bieten sich Datenbanken an. Für eine erfolgreiche Integration inhaltsbasierter Retrieval-Anwendungen in Datenbanken sind jedoch einige Fragestellungen zu beantworten.

Der Nutzer des hier entwickelten Modells zur Bildbeschreibung ist der Entwickler einer Datenbank. Mit Hilfe des Modells wird es dem Nutzer ermöglicht, den Inhalt seines Bildes mit UML zu beschreiben. Für die Modellierung wird dem Nutzer ein Framework zur Verfügung gestellt, das

er für seine konkrete Anwendung anpassen kann. Anschließend wird das Modell transformiert, so dass es auf einer konkreten Zielplattform zum Einsatz kommen kann.

Ein generisches Datenmodell ist notwendig, weil die untersuchten existierenden Möglichkeiten zur Modellierung von Bildinhalten nicht ausreichend sind. So fehlen beispielsweise im Entity Relationship Model oder MPEG-7 einige wichtige Konzepte, die bei der Bildbeschreibung von Interesse sind. Mit UML-Klassendiagrammen sind die Möglichkeiten bereits sehr umfangreich, nur leider auch zu umfangreich, um jedes beliebige Klassendiagramm auf eine Zielplattform abzubilden.

Mit dem generischen Datenmodell wird ein Modell auf Basis von UML angeboten, das den Mittelweg findet zwischen den Freiheiten bei der Modellierung und den Möglichkeiten der Transformation.

1.3 Aufgabenstellung

Im Rahmen dieser Diplomarbeit ist ein Datenmodell für die Speicherung und das Retrieval von Bildern sowie ihrer Inhaltsmerkmale zu definieren. Die entsprechenden Algorithmen zur Merkmalsextraktion und Ähnlichkeitssuche sind zu integrieren.

Für die Modellierung sind existierende konzeptuelle Modelle, ER, UML und MPEG-7, und die Möglichkeiten der Abbildung auf Datenbank- (OR/OO-Schema) und CM-Modelle zu untersuchen und vergleichen. Ihre Eignung für die Modellierung speziell von Bildinhalten für verschiedene Zielplattformen ist zu analysieren. Ein generisches Modell für die Darstellung von Bildinhalten ist zu erstellen. Dabei sind Anforderungen wie Interoperabiliät, Plattformunahbhängigkeit und Erweiterbarkeit zu berucksichtigen.

Das erstellte Modell ist innerhalb des eNoteHistory-Projektes prototypisch umzusetzen. Die dafür benötigten Klassifikations- und Computer Vision Algorithmen werden bereitgestellt. Es ist das Datenbankmanagementsystem IBM ORDBMS DB2 und das Content Management System IBM Content Manager V.8.1 zu verwenden.

Folgende Schritte sind im Rahmen der Arbeit durchzuführen:

- Einarbeitung in das Thema, Literaturrecherche
- Untersuchung und Vergleich existierender konzeptueller Modelle und Ansätze für die Modellierung von Bilddaten
- Entwicklung eines Modells für die Darstellung von Bilddaten und ihrer Merkmale
- Abbildung auf OR-Schema und CM-Datenmodell
- Prototypische Implementierung des entwickelten Ansatzes

1.4 Aufbau der Arbeit

Den Anfang der vorliegenden Diplomarbeit bildet das Kapitel 2 mit der Vorstellung des eNote-History-Projekts. Dieses Projekt dient als fortlaufendes Beispiel und Anwendungsfall für die in dieser Arbeit entwickelten Konzepte.

Anschließend werden in Kapitel 3 verschiedene existierende Modelle zur Darstellung von Bildinhalten näher betrachtet: das Entity Relationship Model, MPEG-7 und die Unified Modeling Language. Dabei wird untersucht, inwiefern diese Modelle in der Lage sind, Bildinhalte digitaler Bilder zu beschreiben. Es werden bei der Untersuchung der einzelnen Modelle Ideen und Ansätze für ein generisches Datenmodell gesammelt.

Das Kapitel 4 wird diese Ideen aufgreifen. Es werden Anforderungen an das Datenmodell formuliert und verschiedene Aspekte der Modellierung betrachtet, wie beispielsweise Darstellung der Bildeigenschaften und Unterteilung des Bildes in Regionen. Daraus wird im Anschluss ein generisches Datenmodell zur Beschreibung von Inhalten digitaler Bilder entwickelt. Es werden außerdem verschiedene Probleme diskutiert, wie beispielsweise die Integration von Operationen.

Im Anschluss daran behandelt das Kapitel 5 die Abbildung der konzeptuell modellierten Daten in das Datenmodell einer spezifischen Zielplattform. Dabei sind zwei Systeme als Zielplattformen vorgegeben: das ORDBMS IBM DB2 und der IBM Content Manager V.8.1. Neben der Umsetzung der statischen Anteile, d.h. der Klassen, Attribute und Beziehungen, werden auch Wege untersucht, die modellierten Operationen auf den Zielplattformen einzubinden. Die Idee dahinter ist, Algorithmen zur Extraktion der Features eines Bildes server-seitig zu integrieren.

Das Kapitel 6 zeigt die Ansätze für einen Prototypen, der die vorgestellten Konzepte implementiert. Neben den wesentlichen Programmabläufen werden auch die Anbindung an die UML-Entwicklungsumgebung sowie die Zielplattformen vorgestellt.

Den Abschluss bildet Kapitel 7 mit einem Fazit und ein Ausblick auf mögliche zukünftige Arbeiten an diesem Projekt.

Kapitel 2

Das Projekt eNoteHistory

In diesem Kapitel wird das Projekt eNoteHistory[1] vorgestellt. Es handelt sich dabei um eine Anwendung, die sich im unter anderem mit der Speicherung von Metadaten digitaler Bilder beschäftigt. Zunächst wird das Projekt kurz beschrieben, anschließend wird auf die Feature-Extraktion der gespeicherten Bilder näher eingegangen. Abschließend werden die Aspekte, die für das generische Datenmodel interessant sind, zusammengefasst.

2.1 Begriffe

Bevor das Projekt vorgestellt wird, sollen einige Begriffe aus der Musik kurz erklärt werden:

- *Notensystem:* Ein Notensystem besteht aus 5 waagerechten Strichen, auf denen Noten, Pausen, Taktstriche und sonstige Symbole notiert werden. Es dient dazu die Tonhöhe der Noten eindeutig festzulegen.
- *Notenschlüssel:* Der Notenschlüssel steht linken Rand eines Notensystems. Er gibt die Grundtonlage an.
- *Note:* Eine Note besteht aus einem Notenkopf, einem Notenhals, und kann außerdem mit einem oder mehreren Fähnchen versehen sein. Der Notenkopf kann dabei entweder ausgefüllt oder leer sein. Der Notenhals wird normalerweise rechts am Notenkopf angesetzt und, je nach Tonhöhe der Note, nach oben oder nach unten gezeichnet. Die optionalen Fähnchen am anderen Ende des Notenhalses sowie die Füllung des Notenkopfes bestimmen die Spieldauer der Note. Man spricht bei Notenlängen von ganzen, halben, viertel und achtel Noten, usw.
- *Pause:* Es gibt verschiedene Symbole, die eine Pause bestimmter Länge anzeigen. Die Länge wird dabei wiederum als ganze, halbe, etc. Pause angegeben.

[1] Die Homepage des Projektes ist www.enotehistory.de. Allgemeine Informationen zum Projekt findet man in [GV03] oder [Goe03].

- *Takt:* Ein Takt besteht aus mehreren Noten und/oder Pausen. Takte werden durch Taktstriche am Ende begrenzt. Taktstriche sind senkrechte Linien, die über alle fünf Notenlinien reichen.

2.2 Projektbeschreibung

Im 17. und 18. Jahrhundert wurden Notenhandschriften durch so genannte Kopisten per Hand vervielfältigt. Der Universität Rostock steht eine umfangreiche Sammlung solcher Handschriften zur Verfügung. Das eNoteHistory-Projekt erfasst diese Dokumente elektronisch und bietet die Möglichkeit zum Zugriff auf die zugehörigen Metadaten sowie auf die digitalen Bilder der eingescannten Notenblätter an. Ein weiterer Teil des Projektes beschäftigt sich mit der automatischen sowie manuellen Analyse der eingescannten Bildseiten. Dabei werden die Notenschriftmerkmale des Kopisten extrahiert, wodurch auch Seiten, bei denen der Kopist nicht bekannt ist, ihrem Schreiber zugeordnet werden können.

Das eNoteHistory-Projekt ist ein Gemeinschaftsprojekt mit folgenden Partnern:

- Frauenhofer Institut für Grafische Datenverarbeitung (IGD)
- Lehrstuhl Datenbanken- und Informationssysteme (DBIS) der Universität Rostock
- Institut für Musikwissenschaften der Universität Rostock

Das IGD erstellt für das Projekt ein Modul zur automatischen Extraktion von Schriftmerkmalen aus einer eingescannten Notenseite. Es werden das Notensystem, einzelne Noten, Notenhälse, Taktschriche und andere Symbole erkannt. Die erkannten Objekte werden vermessen und entsprechend klassifiziert.

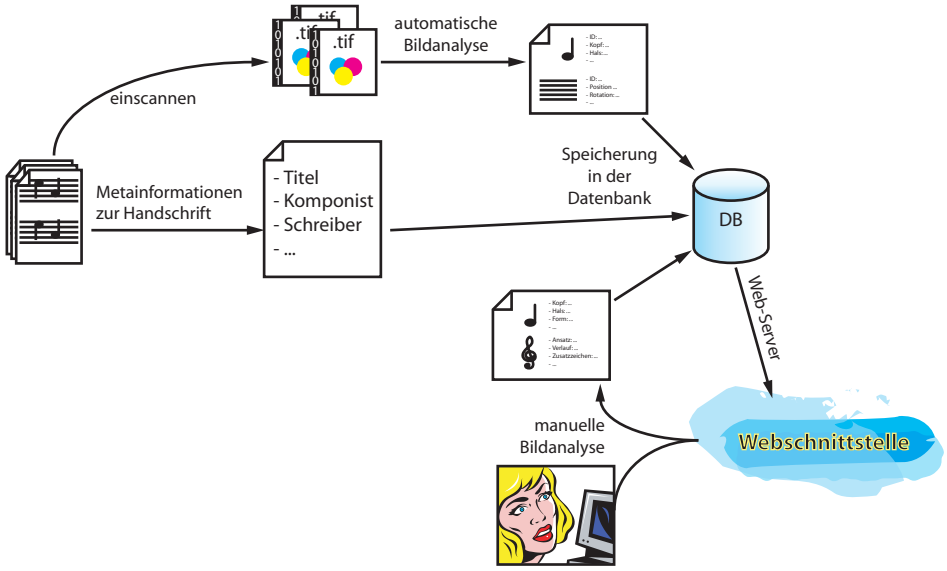
Der Lehrstuhl DBIS ist verantwortlich für die Speicherung der eingescannten Bilder sowie der Metadaten. Außerdem wird eine Web-Schnittstelle bereitgestellt, mit deren Hilfe der Katalog durchsucht und korrigiert sowie die Analyse einer vorliegenden Handschrift vorgenommen werden kann.

In Abbildung 2.1 sind die Zusammenhänge im eNoteHistory Projekt vereinfacht dargestellt. Das Bild zeigt, wie Notenhandschriften in die Datenbank eingefügt werden, die Bildanalyse und die Webschnittstelle. Die Abbildung vereinfacht einige Vorgänge, deshalb sei für an dieser Stelle weiterführende Informationen auf die Internetseiten des Projektes verwiesen.

2.3 Die Beschreibung der Features

Bei der Bildanalyse ist die Eingabe stets ein Notenblatt in digitaler Form. Dieses wird entweder manuell analysiert, oder eingescannt und dem Computer zur Analyse überlassen. Der Computer arbeitet bei der Analyse mit messbaren Größen wie Pixelabständen und Schwellenwerten. Die

Abbildung 2.1: Prozesse in eNoteHistory



manuelle Analyse basiert auf der Beschreibung der Form der einzelnen Elemente (z.B. der Notenfähnchen). Sie wird an der Webschnittstelle mit Hilfe von Schaubildern zur Klassifizierung durchgeführt. Für die automatische Analyse steht derzeit nur ein externes Tool zur Verfügung.

2.3.1 Manuelle Bildbeschreibung

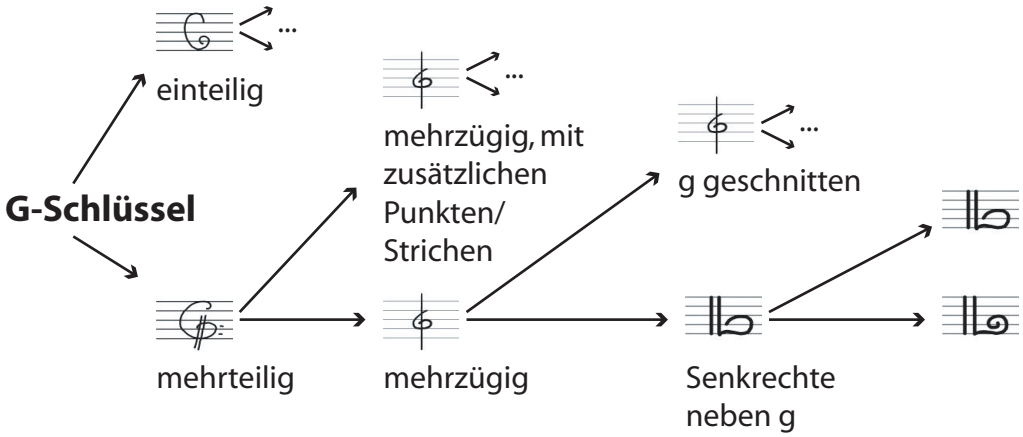
Bei der manuellen Beschreibung einer Notenhandschrift klassifiziert der Nutzer einige Elemente der Notenhandschrift, wie beispielsweise Notenschlüssel, Fähnchen, Neigung der Schrift oder Form der Notenköpfe. Dafür wird der Nutzer baumartig durch verschiedene Eigenschaften geleitet, wodurch das untersuchte Feature verfeinert wird. Abbildung 2.2 soll diesen Vorgang verdeutlichen. Sie zeigt die Schritte zur Klassifikation eines G-Schlüssels.

Mit Hilfe dieser Angaben können unbekannte Notenhandschriften anhand der Datenbasis und einer Distanzfunktion mit den vorhandenen Daten verglichen werden. Durch Clustering-Techniken kann dann die Zugehörigkeit eines Notenblattes zu einem Schreiber festgestellt werden.

Das Ergebnis der Analyse lässt sich als eine Menge von Knoten in einem Baum darstellen. Jeder Knoten wird durch eine Zahl, ähnlich wie in einer Dokumentgliederung (z.B. 1.1.2.3.1.5), identifiziert, wobei von jedem Knoten ausgehend jeder abwärts abgehende Zweig durchnummeriert wird.

Der Nutzer muss nicht alle Eigenschaften der Handschrift eines Kopisten angeben. Er kann beispielsweise die Charakterisierung von Notenschlüsseln auslassen. Diese Eigenschaft wird im

Abbildung 2.2: Manuelle Klassifizierung eines G-Schlüssels



Feature-Vektor als nicht vorhanden gekennzeichnet und bei Suche und Vergleich nicht berücksichtigt. In dieser Arbeit wird nicht näher auf die Features der manuellen Analyse eingegangen. Für nähere Informationen hierzu sei auf die Diplomarbeit von Lars Milewski, [Mil04], verwiesen.

2.3.2 Automatische Bildanalyse

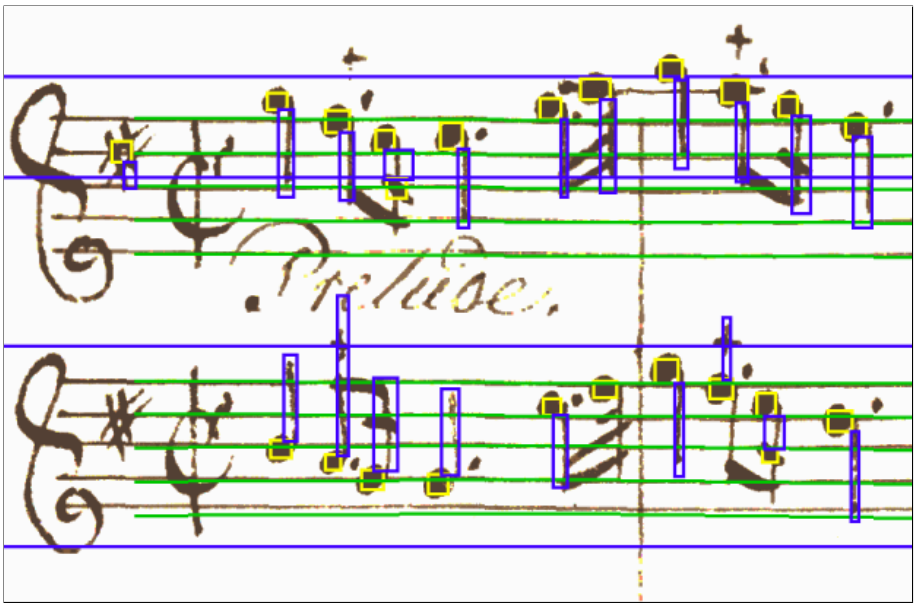
Für die automatische Analyse müssen die Bilder in einer relativ hohen Auflösung eingescannt werden. Anschließend werden verschiedene Filter auf das Bild angewandt, um Störungen zu beseitigen und die relevanten Bildbereiche zu verbessern. Das Bild wird danach auf den Bereich, der die Noten enthält, zugeschnitten. Abbildung 2.3 zeigt die erkannten Objekte auf einer Notenseite. [Goe03] beschreibt den Vorgang der automatischen Bildanalyse ausführlicher.

Im ersten Analyseschritt werden die Notenlinien identifiziert, da diese als Basis für die Notenerkennung dienen. Danach werden vertikale Linien und Notenköpfe erkannt. Bei vertikalen Linien gilt es Taktstriche und Notenhälse zu unterscheiden. Mittels Vorlagen (Templates), die von den Musikwissenschaftlern erstellt werden, kann die Lage komplexerer Objekte, wie beispielsweise Pausen und Notenfähnchen, erkannt werden.

Das Resultat ist ein Feature-Vektor, der mit dem Ergebnis der manuellen Analyse nicht verglichen werden kann, da die Klassifizierung jeweils aufgrund unterschiedlicher Kriterien erfolgt. Es ist wahrscheinlich nur mit recht hohem Aufwand möglich, die beiden Feature-Vektoren zumindest teilweise aufeinander abzubilden, da dafür Metrik und Elemente der manuellen Analyse ineinander überführt werden müssen. Vielmehr sollen beide Feature-Vektoren einander ergänzen.

Als Resultat der automatischen Analyse liegen neben einem vorverarbeiteten Bild im TIF-Format (Tagged Image File Format) mehrere Textdateien vor. Die erste Textdatei enthält Daten, die in

Abbildung 2.3: Beispiel einer automatischen Bildanalyse



der Vorverarbeitung extrahiert werden, z.B. den relevanten Bildausschnitt, der notwendig ist da dunkle Ränder unter Umständen als Notensysteme erkannt werden, und deshalb abgeschnitten werden müssen, die Drehung des Bildes, um Fehler beim Einscannen zu korrigieren, aber auch Informationen über Position und Dicke der Linien im Notensystem, da diese die Basis für die folgenden Analysen bilden. In weiteren Textdateien werden Informationen über Notenköpfe, Notenhälse, usw. abgelegt.

Eine Schnittstelle zur Datenbank, mit der die ermittelten Daten abgelegt werden können, existiert noch nicht. Die Anwendung zur automatischen Bildanalyse wurde jedoch bereits erweitert um den relevanten Bildbereich, der bisher von Hand festgelegt werden musste, selbstständig zu erkennen. Bilder können somit vollautomatisch analysiert werden. Es existieren außerdem prototypische Algorithmen, die eine automatische Zuordnung von Notenhälsen zu zugehörigen Notenköpfen ermöglichen sollen.

2.4 Ideen für das generische Datenmodell

Die manuelle Analyse sammelt Eigenschaften der Noten, welche Elemente aus mehreren Mengen von hierarchisch angeordneten Eigenschaften sind. Eine solche Menge wäre dabei beispielsweise die Menge aller möglichen Notenschlüssel, in denen die Eigenschaften wie in Abbildung 2.2 angeordnet sind.

Die automatische Analyse extrahiert metrische Informationen über die Handschrift, und sammelt diese. Hierbei werden auf dem Bild relevante Bereiche (*region(s) of interest*, *ROI(s)*) gekennzeichnet. Deren Eigenschaften werden ermittelt und der Region zugeordnet. Features lassen sich

meist mit Standard-Datentypen wie Ganzen- und Fließkommazahlen beschreiben.

Dieser Ansatz ist auch im generischen Datenmodell wiederzufinden. Die Aufteilung in Bildregionen und anschließende Zuordnung von relevanten Eigenschaften ist ein intuitiver Weg um relevante Daten auf einem Bild zu beschreiben.

Da sich Regionen auf dem Bild einer Notenhandschrift überlappen, bleibt zu untersuchen, inwiefern eine Forderung nach einer disjunkten Zerlegung sinnvoll ist, wie sie beispielsweise in [GS00] vorgeschlagen wird. Es wird auch von Interesse sein, zu klären, ob Informationen über die Region und die zugeordneten Features getrennt modelliert werden sollten.

Bisher wurde auch die Form einer Region nicht festgelegt. Weit verbreitet sind umschließende Rechtecke (*bounding rectangles/boxes*), möglich sind aber auch Polygonzüge, Ellipsen und andere geometrische Objekte. Die Klärung dieses Problems wird daher ebenfalls bei der Entwicklung des generischen Datenmodells zu untersucht.

Da auch die Implementierung der automatischen Analyse in geeigneter Form im generischen Modell wiederzufinden sein soll, wird es notwendig sein, Methoden zu modellieren und abzubilden.

Kapitel 3

Verschiedene Datenmodelle

In diesem Kapitel werden verschiedene Modelle untersucht, speziell im Hinblick auf ihre Eignung zur Modellierung von Bilddaten.

Dabei wird zunächst das Entity Relationship Modell vorgestellt, das in der Informatik eine große Bekanntheit erlangt hat und eine gute Basis für die Datenmodellierung bildet. Anschließend wird mit MPEG-7 ein neuerer Standard vorgestellt, dessen Aufgabe speziell die Speicherung von Metainformationen verschiedenster Art ist. Zum Abschluss wird die Unified Modeling Language näher betrachtet.

Neben der kurzen Vorstellung der einzelnen Modelle wird, jeweils im Fazit, das Modell im Zusammenhang mit der Aufgabenstellung untersucht, indem es unter dem Aspekt der Eignung zur Bildbeschreibung betrachtet wird.

Den Abschluss dieses Kapitels bildet eine Zusammenfassung, in der auch ein Ausblick auf das resultierende Modell zur Darstellung der Bilddaten gegeben wird.

3.1 Das Entity Relationship Modell

Das Entity Relationship Modell (ERM) wurde 1976 von Peter Pin-Shan Chen aus dem Relationalen-, dem Netzwerk- und dem Entity-Set-Modell entwickelt (siehe [Che76]).

3.1.1 Elemente des Modells

Grundlage des Modells bilden *Entitäten* (*entities*; e). Diese stellen Objekte der Realwelt dar. Entitäten mit gleicher Struktur werden in *Entitätsmengen* (*entity sets*; E mit $e \in E$) zusammengefasst. Entitätsmengen müssen nicht disjunkt sein.

Zwischen Entitäten existieren *Beziehungen* (*relationships*; rel). Jeder Entität, die an einer Beziehung teilnimmt, wird eine *Rolle* (*role*; r) zugewiesen. Eine Beziehung wird nun definiert als ein Vektor $(r_1/e_1, .., r_n/e_n)$, wobei die e_i die Entitäten und die r_i die ihnen zugewiesenen Rollen

innerhalb der Beziehung sind. Beziehungen zwischen den selben Entitätsmengen und mit den selben Rollen werden zu *Beziehungsmengen* (*relationship sets*; R mit $rel \in R$) zusammengefasst.

Weiterhin sind *Wertemengen* (*value sets*; V) sowie *Attribute* definiert. Attribute sind definiert als Abbildungsfunktionen, die Entitäts- oder Beziehungsmengen auf eine bzw. das kartesische Produkt mehrerer Wertemengen abbilden.

Im ERM werden Attribute oder Attributmengen, die die Entitäten einer Entitätsmenge identifizieren, *Schlüssel* genannt, wobei der semantisch sinnvollste als Primärschlüssel ausgezeichnet wird. Es können außerdem schwache Entitäten und schwache Beziehungen definiert werden.

Chen hat für das ERM auch die grafische Darstellung mittels Entity Relationship Diagrammen (ERD) festgelegt. Darin werden Beziehungen mit Kardinalitäten ausgezeichnet, die als Teilnahmezahlen in der Beziehung gedeutet werden können. Existiert eine Beziehungsmenge R zwischen zwei Entitäten E_1 und E_2, so kann jede Seite der Beziehung als Kardinalität 1 oder N erhalten. Die Notation von 1 bei E_1 und N bei E_2 bedeutet, dass eine Entität aus E_1 mit n (n=0,1,2,...) Entitäten aus E_2 in Beziehung stehen kann.

3.1.2 Fazit

Mit dem ERM können Daten und einfache Zusammenhänge verhältnismäßig realitätsnah modelliert werden. Das Modell geht aus dem Datenbanken-Umfeld hervor, und somit liegt der Schwerpunkt auf der Modellierung zu speichernder Daten. Beziehungen im ERM können zwar Attribute haben, allerdings ist die Angabe von Kardinalitäten beschränkt auf 1 und N, was die Modellierung einschränkt. Als Attribute kommen im ERM neben einfachen Attributen nur Tupel als einziger komplexer Datentyp vor. Da jedoch auch die Wertemengen (value sets) nicht näher spezifiziert wurden, müssten komplexe Datentypen zumindest als Elemente einer Wertemenge darstellbar sein.

Zur Darstellung von Features eines Bilddokumentes müssen verschiedene Features als Entitäten modelliert werden. Diese müssen anschließend mit Beziehungen der Entität Bild zugeordnet werden. Komplexere Daten können nur mit weiteren Entitäten dargestellt werden, sofern die Darstellung als Tupel nicht genügt. Prinzipiell sollte eine Modellierung von Features eines digitalen Bildes möglich sein, jedoch sind wohl fehlende Kardinalitäten und mangelnde Unterstützung komplexer Datentypen zweifellos ein Nachteil. Auch ein Operationenteil existiert im ERM nicht.

Dennoch kann das ERM als Grundidee zur Modellierung von Bilddaten dienen, denn die Darstellung des Modells ist einfach, für den Nutzer leicht nachzuvollziehen und die Modellierung ist einfach auf relationale Speicherstrukturen abbildbar (siehe [HS00]).

Mit Erweiterungen des ERM ist es sogar möglich, fehlende Konstrukte einzubinden. Dadurch wäre eine Modellierung in vollem Umfang möglich. Leider existieren viele Erweiterungen für das ERM, jedoch nur wenige gehen über eine syntaktische Erweiterung heraus. Es wird auch schwierig ein passendes Entwurfswerkzeug zu finden, das ERM-Erweiterungen unterstützt. Aber

selbst wenn ein solches Werkzeug vorliegt, so steht noch nicht fest, wie viele Entwickler die Erweiterung auch beherrschen, da nicht jede Erweiterung dafür weit genug verbreitet ist. Ein Beispiel für eine gut fundierte Erweiterung ist das *Higher-Order Entity Relationship Model* (HERM) von Prof. B.Thalheim. Einen Überblick über HERM bietet [Tha00].

3.2 Der MPEG-7-Standard

MPEG-7 ist ein von der *Motion Picture Experts Group* (MPEG) ins Leben gerufener Standard zur Speicherung von Metadaten. Diese Metadaten können zu verschiedenen Arten multimedialer Daten gehören, z.B. Audio, Video und Bilddaten. Eine ausführliche Einführung in den Standard findet sich in [MSS02].

Da die MPEG ein zu schnelles Veralten ihres Standards verhindern will, wurden lediglich Syntax und Semantik des Formats sowie dessen Dekodierung standardisiert. Dadurch wird sichergestellt, dass die Sprache trotz Erweiterbarkeit auf allen kompatiblen Systemen gelesen und verarbeitet werden kann. Das bedeutet speziell, dass mit MPEG-7 lediglich ein „Container“ zur Speicherung von Metadaten zur Verfügung gestellt wird. Somit stellt sich die Frage, ob MPEG-7 in der Lage ist, semantische Aspekte, also die Bedeutung der im Container enthaltenen Objekte, optimal zu transportieren, da der Schwerpunkt in diesem Modell auf der Syntax liegt.

Als Datendefinitionssprache (*Data Definition Language*, DDL) von MPEG-7 wurde nach Untersuchung mehrerer in Frage kommender Sprachen *XML-Schema* ausgewählt. Da XML-Schema die Anforderungen der MPEG nicht vollständig erfüllt hat, musste die Sprache erweitert werden, z.B. um Datentypen zur Darstellung von Vektoren und Matrizen. Um die MPEG-7 DDL zu parsen wird demnach ein „normaler“ XML-Schema-Parser nicht ausreichen.

Aufbauend auf den Standard-Datentypen von XML-Schema ist in MPEG-7 eine umfangreiche Hierarchie einfacher und komplexer Typen enthalten, die in den verschiedenen Schemata Verwendung finden können. Diese Typen sind mit einem führenden „mpeg7:“ im Namespace gekennzeichnet.

Auf Basis des MPEG-7-Schemas werden nun Dokumentbeschreibungen für multimediale Daten erstellt. Diese Dateien liegen entweder in der für XML üblichen textuellen Form vor, oder als eine binäre Datei, deren Inhalt 1:1 auf eine XML-Datei abgebildet werden kann. Die binäre Form dient einer schnelleren Verarbeitung und kompakteren Speicherung, die für Geräte mit begrenzten Ressourcen vorgesehen ist.

3.2.1 Aufbau einer MPEG-7-Datei zur Bildbeschreibung

Für den Aufbau eines MPEG-7-Dokumentes gibt es prinzipiell zwei Möglichkeiten. Neben einer optionalen Metadaten-Beschreibung, die unter anderem Version, letztes Update, Autor und

Rechte enthält, kann der Inhalt entweder mittels einer einzelnen *Description Unit* beschrieben werden, oder mit Hilfe eines vollständigen Schemas zur Inhaltsbeschreibung[1].

Description Units sind beliebige MPEG-7-Elemente, die auf dem Typ „Mpeg7BaseType“ basieren. Dieser Typ ist Obertyp für eine Vielzahl von Features, z.B. Farbinformationen, Histogramme, aber auch High-Level Features wie z.B. das Ergebnis einer Handschriftenanalyse.

Der Standard bietet außerdem verschiedene Möglichkeiten zur Referenzierung von Elementen, die innerhalb und außerhalb der Beschreibung liegen, an:

- Referenzierung mit X-Path
  Im Standard kann eine Teilmenge der Sprache X-Path zur Referenzierung auf andere Elemente innerhalb des XML-Dokumentes verwendet werden.
- Verwendung von Identifikatoren
  Es ist ebenfalls möglich auf Identifikatoren zu verweisen. Dazu sind entsprechende Attribute im MPEG-7-Schema als IDREF-Attribute gekennzeichnet. Es existiert außerdem der Datentyp mpeg7:UniqueIDType, der verschiedene internationale Standards zur Identifikation unterstützt.
- Externe Referenzen
  Unter Verwendung von bereits aus HTML bekannten Hyper-Referenzen (href) ist es möglich Verweise auf externe Dokumente zu erstellen. Dafür wird das Ziel in Form seiner URI angegeben.

Alle diese Referenztypen werden in einem Datentyp (mpeg7:ReferenceType) zusammengefasst. Somit können die Referenztypen beliebig eingesetzt werden.

Beziehungen sind in MPEG-7 unidirektional, Quelle und Ziel werden als URI angegeben. Die inverse Beziehung ist im Standard als Umkehrung der ursprünglichen Beziehung definiert, wird jedoch nicht explizit notiert.

Um die Features zu strukturieren bietet MPEG-7 *Description Schemes* an. Description Schemes enthalten Description Units und/oder andere Description Schemes. Daher ist eine Bildbeschreibung im Standard nichts anderes als ein sehr umfangreiches Description Scheme.

In den Fällen, in denen nur ein einziges beschreibendes Element benötigt oder sinnvoll ist, kann statt einer vollständigen Beschreibung auch ein einziges Feature als Beschreibung genügen. Im Falle einer vollständigen Bildbeschreibung sieht MPEG-7 unter anderem folgende Metadaten vor:

- Ein Header
- Informationen über das Speichermedium

[1] Anstelle einer Inhaltsbeschreibung sieht MPEG-7 an dieser Stelle auch Möglichkeiten zur Verwaltung von Inhalten vor. Darauf wird im Zuge dieser Arbeit jedoch nicht näher eingegangen, stattdessen sei auf die Literatur zum MPEG-7-Standard verwiesen.

- Informationen über die Erstellung, unter anderem
  - Titel
  - Autor
  - Erstellungswerkzeuge
  - Klassifikation
  - Bezüge zu anderen Materialien
- Informationen zur Nutzung, unter anderem
  - Zugriffsrechte
  - Verfügbarkeit
- Anmerkungen, unter anderem
  - als freier Text
  - strukturiert
- Semantische Informationen
  In den semantischen Informationen werden Objekte oder Ereignisse in Zusammenhang zueinander gebracht. Dies geschieht über die oben genannten Möglichkeiten zur Referenzierung.
- eine Menge von Beziehungen, mit denen Bezüge zu anderen Bildelementen angegeben werden können
- Features, bestehend aus:
  - der Feature-Beschreibung, die folgendes sein kann:
    * Visuelle Description Unit; vordefiniert sind Informationen über folgende Inhalte: Histogramme, Formen (2D, 3D, Konturen), Ergebnisse einer Gesichtserkennung, Texturen, Farben und Bewegungen.
    * ein Description Scheme
    * eine Zerlegung in ein gleichmäßiges Gitter (Grid), wobei für beide Dimensionen in bis zu 255 Teile untergliedert werden können, und für jede Gitterzelle eine Beschreibung möglich ist. Die Elemente des Gitters können wiederum unterteilt werden.
  - Informationen über die Zerteilung des Bildes
    Das Bild kann in beliebig viele Regionen unterteilt werden, die wiederum mit Features versehen werden können, da sie vom selben komplexen Datentyp wie auch das eigentliche Bild sind.

3.2.2 Fazit

MPEG-7 bietet die Möglichkeit, verschiedene Audio- und/oder Visuelle Medien zu beschreiben. Es wurden viele Metadaten berücksichtigt, wie beispielsweise die umfangreichen Informationen zur Erstellung oder die Informationen über das Medium. Es wurden auch sehr viele Low-Level-Features berücksichtigt, für die jeweils eine eigene Description Unit definiert ist. Weitere Features dürften mit Entwicklung des Standards kein Integrationsproblem darstellen. Eine Erweiterung des Modells ist daher unproblematisch.

Im Bereich der High-Level-Features ist zunächst eine Definition des Features erforderlich, die dann in der Typhierarchie unterhalb des abstrakten Typs für visuelle Description Units, im Schema definiert unter mpeg7:VisualDType, angeordnet sein muss. Ist dies geschehen, so kann das Feature wie jedes andere in einer MPEG-7-Beschreibung verwendet werden. Bei der Definition des Features hat der Nutzer alle Möglichkeiten von XML-Schema zur Verfügung, und kann außerdem auf die im Standard vordefinierten Typen zurückgreifen. XML-Schema unterstützt auch Kardinalitäten (mit den Attributen min/maxoccurs).

Da MPEG-7-Beschreibungen XML-Dateien sind, werden praktisch alle Daten als Strings gespeichert, deren Syntax mittels Pattern festgelegt ist. Außerdem sind in XML-Schema bereits diverse Typen vordefiniert, z.B. numerische Typen oder Zeitangaben. Eine Auswertung des Schemas ermöglicht somit eine korrekte Interpretation der vorliegenden Daten.

Nachteil von MPEG-7 ist die Darstellung der modellierten Daten. Durch die oftmals sehr tiefe Hierarchie innerhalb eines MPEG-7-Dokumentes, die beispielsweise durch Aufteilung in Regionen und Angabe von Beziehungen entsteht, ist zur Darstellung einer Beschreibung mehr als nur ein Texteditor notwendig. Dadurch wird allerdings der Vorteil, dass XML durch seine textuelle Form menschenlesbar sein soll, zu einem Nachteil, denn da die Hierarchie der Elemente zu stark ausgebildet ist, wird in der textuellen Form eher Speicherplatz verschwendet. Auch ist bei eigenen Features eine vorherige Definition der Daten erforderlich, da sonst kein gültiges MPEG-7-Dokument vorliegt. Version 1.0 des Standards unterstützt die Definition eigener Deskriptoren nicht, somit können noch keine anwendungsspezifischen Features realisiert werden.

Die Vielfalt der Datentypen, die MPEG-7 bietet, ist für die Beschreibung von Bilddaten von Vorteil. Neben den Grunddatentypen können ohne weiteres verschiedene komplexe Typen verwendet werden.

3.3 Die Unified Modeling Language

Die Unified Modeling Language (UML) ist eine Sprache, die Softwareentwicklern eine einheitliche Notation für eine Vielzahl von Aspekten der Anwendungsentwicklung zur Verfügung stellt. UML kann sowohl statische als auch dynamische Vorgänge darstellen. Die Sprache ist nicht durch Grammatik, wie es beispielsweise für Programmiersprachen üblich ist, definiert, sondern ist durch ein Metamodell, das wiederum mit UML beschrieben ist, definiert und wird meist exemplarisch beschrieben.

Für die Bildbeschreibung kommen nur UML-Klassendiagramme in Frage, die ein Mittel zur Darstellung von Strukturen sind. Es gibt viele Parallelen zum (nicht erweiterten) ERM, jedoch bietet UML wesentlich mehr Möglichkeiten zur Verfeinerung der Eigenschaften und der Zusammenhänge im Modell.

Der folgende Abschnitt soll lediglich einen Überblick über den Funktionsumfang von UML geben. Für weiterführende Informationen zu UML sei auf die Literatur, z.B. [HK99] oder den Standard, [OMG03], verwiesen.

3.3.1 Elemente der Sprache

Klassen

Als Ausgangspunkt für die Darstellung betrachten wir Klassen. Eine Klasse ist vergleichbar mit einer Entitätsmenge im ERM. Klassen beschreiben den Aufbau von Objekten. Objekte, die Instanzen der selben Klasse sind, haben demnach gleiche Eigenschaften.

Eine Klasse besteht aus einem Namen, Attributen und Operationen. Zusätzlich zum Namen können auch weitere Eigenschaften der Klasse notiert werden, beispielsweise ein Stereotyp oder Informationen über Persistenz oder Vererbung.

Mit Stereotypen können erweiterte Informationen über die Klasse angegeben werden. Stereotypen können beispielsweise signalisieren, dass eine Klasse ein Objekt des Problembereiches darstellt (*entity*), ein Schnittstellenobjekt ist (*boundary*) oder eine Steuerungsfunktion im Anwendungsprogramm darstellt (*control*).

Das Konzept der Attribute ist bereits aus dem ERM bekannt. Mit Attributen werden Eigenschaften, die diese Klasse von Objekten hat, festgehalten. Dies geschieht unter Angabe von Name und Typ des Attributs. Außerdem können unter anderem folgende Eigenschaften angegeben werden:

- Sichtbarkeit der Attribute für andere Klassen
- Klassenattribut-Eigenschaft
- Kann das Attribut aus anderen Attributen abgeleitet werden?
- Multiplizität[2]
- Einschränkungen des Wertebereiches
- Typ des Attributs
- Initialwert

[2] innerhalb dieser Arbeit werden die Begriffe Multiplizität und Kardinalität als Synonyme füreinander verwendet

Da Objekte neben ihren Eigenschaften auch bestimmte Aktionen durchführen können, bietet UML Möglichkeiten an, Operationen näher zu spezifizieren. Jede Operation hat neben ihrem Namen eine Parameterliste, für die Typ und Standardwerte angegeben werden können sowie einen Ergebnistyp, der den Rückgabewert der Operation näher spezifiziert. Operationen können abstrakt sein, und auch Klassenoperationen sind möglich.

Zu jeder nicht-abstrakten Klasse kann es zugehörige Objekte geben, die der Beschreibung der Klasse entsprechen. Diese Objekte werden *Instanzen* der Klasse genannt. Die Menge der Instanzen zu einer Klasse, also der Objektvorrat, wird *Extension* genannt.

Assoziationen

Um verschiedene Klassen in Beziehung zueinander zu bringen, bietet UML *Assoziationen* an. In ihrer einfachsten Form ist eine Assoziation eine binäre Relation zwischen zwei Klassen. Es ist allerdings üblich Assoziationen zu benennen und eine Leserichtung anzugeben. Wie bereits im ERM können auch hier Rollen angegeben werden, die die Beteiligung der Klassen in der Beziehung näher erläutern. Normalerweise gehört zu Assoziationen auch die Angabe von Multiplizitäten. Diese werden in Intervallen notiert und geben an, mit wie vielen Objekten einer Klasse ein Objekt der assoziierten Klasse mindestens in Beziehung stehen muss und höchstens darf. Dabei sind auch komplexere Angaben wie $[1, 3, 5..*]$ möglich, wobei * für unendlich steht. Es kann außerdem angegeben werden, dass, einmal etabliert, die Assoziation nicht mehr geändert werden kann, oder nur noch weitere Assoziationen hinzugefügt werden können.

Einer Assoziation kann eine Klasse zugeordnet werden. Diese *Assoziationsklasse* besteht dann aus Attributen, die (vgl. ERM) Eigenschaften der Assoziation enthält.

Assoziationen können verfeinert werden, indem eine Navigationsrichtung angegeben wird. Dadurch wird dargestellt, das die Realisierung des Modells zwar in die eine Richtung der Assoziation navigieren kann, aber nicht zwangsweise in die Rückrichtung.

Normalerweise besteht eine Assoziation zwischen zwei nicht zwangsweise unterschiedlichen Klassen. Sollte es notwendig sein mehr als zwei Klassen in Beziehung zu bringen, so kann eine n-äre Assoziation notiert werden. Diese darf allerdings keine Navigationsrichtung enthalten.

Es besteht auch die Möglichkeit eine XOR-Auswahl zwischen zwei Assoziationen anzugeben. Dies wurde beispielsweise bei der Modellierung von MPEG-7 verwendet. Ein MPEG-7-Dokument ist entweder eine einzelne Description Unit, oder eine Vollständige Beschreibung (siehe Abschnitt 3.2.1).

Aggregation, Komponenten, Generalisierung

Eine spezielle Form der Assoziation ist die *Aggregation*. Diese beschreibt eine Teil-Ganzes-Beziehung. Die Aggregation drückt aus, dass die zwei beteiligten Klassen nicht gleichwertig sind. Allerdings ist die Existenz von Instanzen beider Klassen nicht von vornherein abhängig von der Existenz eines assoziierten Objektes der jeweils anderen Klasse, es sei denn, es wird durch die Multiplizität gefordert.

Verlangen wir nun aber eine Abhängigkeit der Existenz, so ergibt sich eine *Komponentenbeziehung* (auch Komposition). Komponenten eines Objekts existieren nicht ohne das Ganze. Dies ist vergleichbar mit schwachen Entitäten aus dem ERM.

Eine andere Form der Beziehung ist die *Generalisierung*, bzw. als Umkehrung davon die *Spezialisierung*. Damit können Klassen in eine Vererbungshierarchie eingeordnet werden. Es können dabei verschiedene Eigenschaften notiert werden, z.B. die Forderung nach disjunkten spezialisierten Klassen.

3.3.2 Fazit

UML ist in der Lage Modelle des ERM darzustellen, und kann darüber hinaus vieles im Modell wesentlich genauer ausdrücken, als es im ERM möglich war. Speziell im Hinblick auf Assoziationen ist die Vielfalt möglicher Eigenschaften ein klarer Vorteil der Sprache. Da UML der Softwareentwicklung dient finden viele Sprachelemente auch eine entsprechende Umsetzung. So kann beispielsweise die Modellierung einer Assoziation mit Navigationsrichtung dazu führen, dass die Beziehung in der Realisierung des Modells ebenfalls nur unidirektional definiert wird, wobei „normale“ Assoziationen hier die Berücksichtigung der inversen Beziehung fordern. Viele UML-Konstrukte sind allerdings auf die Umsetzung einer Objektorientierten Programmiersprache ausgelegt. So ist beispielsweise eine Einkapselung von Attributen nicht sehr sinnvoll, zumindest auf Systemen die keinen, oder nur einen sehr einfach ausgelegten Operationenteil besitzen, der speziell im Hinblick auf die Umsetzung von OO-Konzepten nur wenig oder keine Möglichkeiten bietet.

Ein weiterer Punkt, der für UML spricht, ist der Operationenteil der Sprache. Mit Hilfe von Operationen können die Extraktionsalgorithmen der Bildeigenschaften bereits in das Modell integriert, und deren Verwendung auf Seite der Zielplattform soweit wie möglich vorbereitet werden.

Wir haben uns bereits auf die Klassendiagramme von UML beschränkt, aber angesichts der umfangreichen Ausdrucksmöglichkeiten sind auch hier nicht alle Elemente von Interesse für die Modellierung von Bilddaten. Nicht jedes UML-Sprachelement ist auch problemlos auf eine Zielplattform zu transformieren.

3.4 Zusammenfassung

Mit den vorgestellten Modellen lassen sich Features digitaler Bilder verschiedenartig darstellen. Zwar ist der direkte Vergleich der Modelle nicht möglich, da es sich zum einen um konzeptuelle Modelle, zum anderen mit MPEG-7 um eine Speichervorschrift handelt, aber folgendes Beispiel soll die Fähigkeiten der Modelle verdeutlichen und zeigen, an welchen Stellen ein Modell gegenüber den anderen im Vorteil ist.

3.4.1 Vergleichsbeispiel

In Anlehnung an das eNoteHistory-Projekt (siehe Kapitel 2) soll folgendes Beispiel zum Vergleich der drei vorgestellten Modelle dienen. Wir gehen von einer Menge von digitalisierten Notenhandschriften aus, die uns bereits vorliegen. In diesen sind die Notensysteme die relevanten Features. Auf den Notensystem wiederum sind die darauf stehenden Noten von Interesse.

Bei den Notensystemen interessiert uns die Lage auf dem Papier sowie die Größe, jeweils angegeben durch Pixelkoordinaten. Für Noten wollen wir die Lage, Ausdehnung und Rotation einer umschließenden Ellipse speichern. Die Daten werden automatisch bestimmt und sollen in einer relationalen Datenbank gespeichert werden.

3.4.2 Umsetzung im ERM

Das ERM ist ein konzeptionelles Modell und soll somit relevante Objekte und Beziehungen zwischen ihnen auf einem abstrakten Niveau darstellen. Ein Nutzer würde daher mehrere Entitäten modellieren, nämlich Notenhandschrift, Notensystem und Note. Notenhandschriften enthalten Notensysteme, diese wiederum enthalten die Noten. Dies lässt sich mit Beziehungen darstellen. Eine Darstellung mit schwachen Entitäten wäre ebenfalls möglich.

Für die Entitäten werden folgende Attribute notiert:

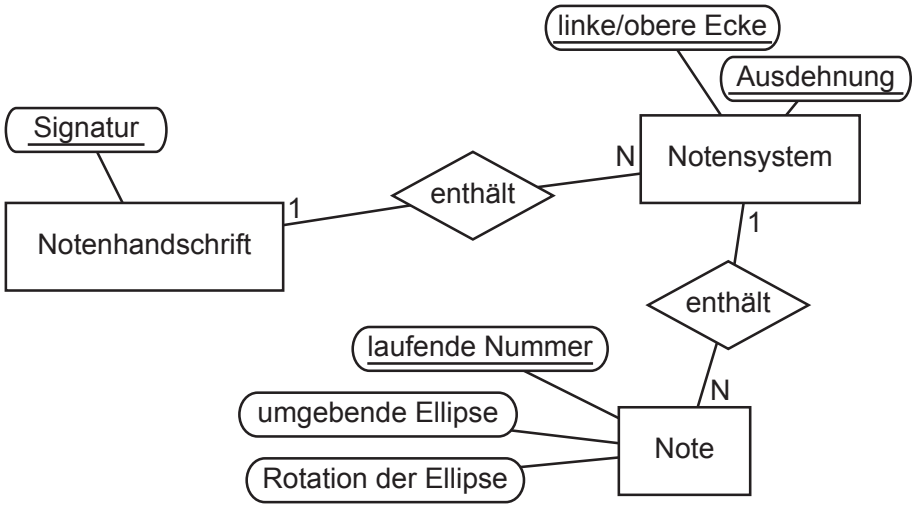
- für Notenhandschriften eine identifizierende Signatur,
- für Notensysteme eine Koordinate (bestehend aus x und y), die die linke/obere Ecke markiert sowie ein Attribut das die Ausdehnung (Höhe und Breite) des Notensystems enthält,
- bei den Noten benötigen wir zur Definition einer Ellipse einen Punkt und zwei Längenangaben sowie einen Wert zur Speicherung der Rotation. Zur Identifikation der Note sei eine laufende Nummer vorgesehen.

Abbildung 3.1 zeigt das entstehende ERD.

Hat der Nutzer das Modell erstellt, so kann aus dem Modell ein Datenbankschema abgeleitet werden. Dieses dient dann der Speicherung der eigentlichen Daten.

Dieses Modell kann mit sehr geringem Aufwand erstellt werden, und da es Algorithmen zur Umwandlung eines ER-Modells in ein Relationales Datenbankschema gibt, ist der Schritt der Transformierung nicht sehr kompliziert. Allerdings wurden in diesem Beispiel nur sehr einfache Konzepte verwendet. Soll beispielsweise noch eine Operation berücksichtigt werden, die Notensysteme in einer Handschrift ermittelt, und eine weitere Operation, die Noten in einem Notensystem aufspürt und erkennt, so kann dies mit dem ERM nicht modelliert werden. Ein anderes Beispiel, das mit dem ERM nicht modelliert werden kann, wäre die Verwendung von Kardinalitäten. Soll beispielsweise festgehalten werden, dass jedes Notensystem mindestens eine Note enthalten muss, aber höchstens 50, so kann auch dies im ERM nicht dargestellt werden. Ein weiterer, wichtiger Nachteil beim Entwurf mit dem ERM ist die fehlende Typisierung. Um

Abbildung 3.1: Beispiel im ERM



eine Transformation auf eine Zielplattform wie DB2 überhaupt möglich zu machen, ist es unbedingt notwendig Informationen über die Datentypen der Attribute zu definieren. Dies müsste dann durch nähere Spezifizierung der Wertemengen der Attribute erfolgen.

Es wird hier schnell deutlich, dass das ERM in seiner Urform nicht alles darstellen kann. Ohne Erweiterungen ist das ERM daher nicht in der Lage, Bilddaten ausreichend zu beschreiben.

3.4.3 Umsetzung mit MPEG-7

MPEG-7 beschreibt ein Speicherformat, das die Metadaten aufnehmen soll. Der Nutzer benötigt lediglich einen Extraktionsalgorithmus, der die beschreibenden Daten MPEG-7-konform generiert. Dabei ist die Beschreibung wiefolgt aufgebaut:

Das Wurzelelement ist vom Typ mpeg7:Mpeg7. Darunter ist als Kindknoten eine Inhaltsbeschreibung angehängt (mpeg7:ContentEntityType). Dieser Knoten enthält als Kindknoten eine Bildbeschreibung (mpeg7:ImageType), welche wiederum eine Beschreibung einer Region (mpeg7:StillRegionType) enthält. Bei dieser Region muss es sich um das gesamte Bild handeln, oder einen relevanten Bildausschnitt, da es sich um die Wurzel-Region handelt. Die Signatur des Bildes wird im Attribut ID gespeichert.

Innerhalb des Bildes wird nun eine Zerlegung angegeben, die das Bild in weitere, kleinere Regionen aufteilt. Diese sind wieder vom Typ mpeg7:StillRegionType. Diese Regionen sollen die Notensysteme darstellen. Um die Ausdehnung der Notensysteme zu speichern, benutzen wir das Attribut SpatialLocator, und speichern eine Box. Diese wird zwar mit zwei Koordinaten definiert, aber da dies gleichwertig zur Speicherung einer Ecke und der Ausdehnung ist, wird der Wert auf diese Art gespeichert.

Zu jedem Notensystem wird nun eine Zerlegung in weitere Regionen erzeugt, die für die Noten

vorgesehen sind. Diese Regionen sollten eigentlich durch eine Ellipse beschrieben werden, was allerdings mit dem Attribut SpatialLocator, welches vom Typ mpeg7:RegionLocatorType ist, nicht möglich ist. Eine Alternative wäre die Angabe eines Polygons, oder umgebenden Rechtecks anstelle der Ellipse. Eine andere Alternative ist die Erweiterung des MPEG-7-Schemas, damit Ellipsen an dieser Stelle gültig wären. Letzteres würde nur mit einer Weiterentwicklung des Standards funktionieren.

Wir entscheiden uns daher dafür, einen Polygonzug zu definieren, der die Ellipse approximiert. Der Polygonzug hat hierfür vier Punkte, wobei die jeweils gegenüberliegenden eine Halbachse der Ellipse definieren. Somit wäre ein Anwendungsprogramm in der Lage, aus dem Polygonzug die Ellipse wieder zu rekonstruieren.

Da die entstehende Beschreibung eine XML-Datei ist, kann diese in einer geeigneten XML-Datenbank abgelegt werden. Bietet die Datenbank nun auch noch Methoden zur Suche auf den Daten an, so können die gespeicherten Features auch direkt ausgewertet werden. Ist keine Speicherung der XML-Datei in einer Datenbank möglich, so muss mittels Transformationen der Inhalt der XML-Datei auf ein relationales Schema erfolgen, damit ein Retrieval der Bildeigenschaften möglich ist. Abbildung 3.2 zeigt, wie die zugehörige MPEG-7-Datei aufgebaut ist.

Der Nutzer ist hierbei auf einen Extraktionsalgorithmus oder ein Tool angewiesen, der oder das die Erzeugung der MPEG-7-Datei realisiert. Der Arbeitsaufwand hierbei hängt von der Qualität des Tools ab. Ist das Tool in der Lage aus den Vorgaben des Nutzers die MPEG-7-Speicherstruktur zu erzeugen, so sollte der Aufwand bei der Beschreibung der Daten nicht wesentlich höher sein als beim ERM. Ein Problem von MPEG-7 ist jedoch, dass es nur wenige Tools gibt, mit denen Beschreibungen erzeugt werden können. Somit ist MPEG-7 gegenüber dem ERM und vor allem UML im Nachteil, für die gute und weit verbreitete Entwurfswerkzeuge existieren. Außerdem erfolgt der Entwurf von ER- und UML-Modellen grafisch, was für den Nutzer meist intuitiver ist als eine strukturierte, eher textbasierte Bildbeschreibung in Form von MPEG-7.

Im Beispiel wird auch deutlich, dass MPEG-7-Dateien eine hohe Strukturierung besitzen. Dies kann bei der Speicherung und bei Anfragen von Nachteil sein, da Datenbanken, die XML-Dateien vollständig speichern, auch jedes Element speichern müssen, wobei den relevanten Daten unter Umständen ein recht langer Pfad durch die Speicherstruktur vorangeht. Vorteil dieses Formats ist die Tatsache, dass die Transformation in eine Speicherstruktur entfallen kann, sofern die Zielplattform, auf der das Dokument gespeichert wird, XML-Dateien effektiv speichern und Anfragen realisieren kann. Auf Basis von XSLT könnten Transformationen für andere Zielplattformen vorgenommen werden, die XML nicht direkt speichern können.

Durch das MPEG-7 zugrunde liegende XML-Schema liegt bereits ein großer Umfang vordefinierter Datentypen vor. Sobald MPEG-7 eine Erweiterung des Schemas unterstützt, können auch eigene Datentypen integriert werden. Zwar können auch das ERM und UML verschiedenste Datentypen darstellen, aber anders als bei MPEG-7 ist bei diesen Modellen die Nutzbarkeit des Datentyps nicht von den Fähigkeiten der Zielplattform abhängig, denn die Daten werden bei XML immer als Zeichenketten abgespeichert. Die Interpretation der Daten wird dann durch XML-Schema vorgegeben.

Abbildung 3.2: Beispiel als MPEG-7-Beschreibung

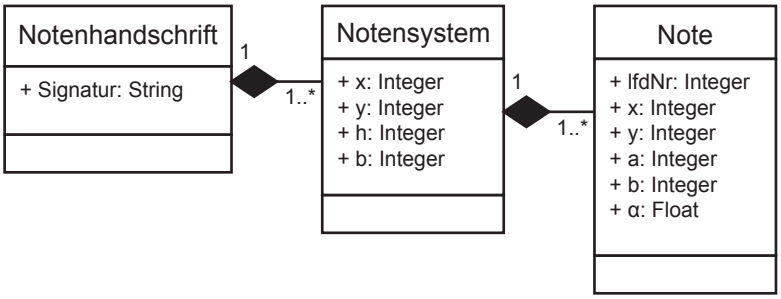
```
<mpeg7>
  <Description>
     <MultimediaContent>
        <Image ID=" ... ">
          <SpatialDecomposition>
             <StillRegion>
                <SpatialLocator>
                  <Box> ... </Box>
                </SpatialLocator>
                <SpatialDecomposition>
                  <StillRegion>
                    <SpatialLocator>
                      <Polygon> ... </Polygon>
                    </SpatialLocator>
                  </StillRegion>
                </SpatialDecomposition>
             </StillRegion>
             <StillRegion>
               ...
             </StillRegion>
             ...
          </SpatialDecomposition>
        </Image>
     </MultimediaContent>
  </Description>
</mpeg7>
```

3.4.4 Umsetzung mit UML

Die Vorgehensweise bei UML ist ähnlich wie beim ERM. Der Nutzer definiert drei Klassen, Notenhandschrift, Notensystem und Note. Diese werden nun mit Komponentenbeziehungen in Beziehung gebracht. Die Attribute werden wie im ERM notiert. Abbildung 3.3 verdeutlicht diesen Sachverhalt.

Anders als im ERM können hier konkrete Datentypen zu den Attributen notiert werden. Dies ist für die spätere Transformation unbedingt notwendig, da sonst beispielsweise die Generierung einer Datenbanktabelle nicht möglich wäre. In UML gibt es weitere interessante Konzepte, wie beispielsweise unidirektionale Beziehungen, die auch vom IBM Content Manager unterstützt werden. Das Konzept der Vererbung und die Definition von Operationen können hilfreich sein. Allerdings fehlt bei UML die Auszeichnung identifizierender Attribute (Schlüssel). Diese spielen in der Modellierung eine nicht unwichtige Rolle, und müssen somit bei der späteren Realisierung des generischen Datenmodells berücksichtigt werden.

Abbildung 3.3: Beispiel mit UML



Da UML in der Industrie ein anerkannter, geförderter und verbreiteter Standard ist, und viele Entwurfswerkzeuge dafür existieren, liegt es nahe, für das generische Datenmodell UML zu verwenden. Für die Umsetzung des Prototyps (Kapitel 6) wird das ebenfalls weit verbreitete Modellierungswerkzeug *Rational Rose* verwendet, welches neben guten Entwurfsmöglichkeiten auch eine direkte Anbindung der Transformationskomponente ermöglicht. UML selbst bietet sehr umfangreiche Entwurfsmöglichkeiten, und ist standardisiert, was viele Erweiterungen des ERM nicht sind. Es ist daher von vielen Nutzern verwendbar, und, da es grafisch dargestellt wird, auch sehr gut lesbar.

Kapitel 4

Ein generisches Datenmodell zur Bildbeschreibung

Wir haben bereits verschiedene Möglichkeiten zur Modellierung von Inhalten digitaler Bilder kennen gelernt. In diesem Kapitel wird ein neues Datenmodell entwickelt und diskutiert, das dem Nutzer, also dem Entwickler einer Bilddatenbank, eine konzeptuelle Modellierungsmöglichkeit zur Verfügung stellt, mit deren Hilfe später die eigentliche Datenbank erstellt werden kann.

Zunächst werden die Ziele für das Modell festgehalten, und Anforderungen an das Modell gestellt. Anschließend werden verschiedene Aspekte der Bildbeschreibung diskutiert, aus denen dann das konkrete Datenmodell entsteht. Um den Schreib-/Lese-Aufwand zu reduzieren, wird das hier erarbeitete generische Datenmodel nachfolgend mit *GDM* abgekürzt. Eine Zusammenfassung über das GDM ist Anhang B zu entnehmen.

4.1 Anforderungen und Ziele

Bevor wir mit dem Aufbau des Modells beginnen, ist es nötig einige Voraussetzungen für das Modell zu klären.

Das Grundmodell, das dem Nutzer in Form eines Frameworks zur Verfügung gestellt wird, ist in UML verfasst, da sich bei der Betrachtung der verschiedenen Modellierungsmöglichkeiten UML als die am besten geeignete herausstellte.

Die Aufgabe des GDM ist es, einem Nutzer die Möglichkeit zu geben, den Inhalt von Bilddaten konzeptuell zu beschreiben. Das bereitgestellte Framework enthält einige Klassen und Beziehungen, die den Kern des GDM ausmachen, und mit deren Hilfe der Nutzer ein konkretes, anwendungsspezifisches Modell erstellen kann.

Dabei soll die folgende Sichtweise umgesetzt werden: Das Bild wird in Regionen unterteilt. Diese Regionen grenzen relevante Bildinhalte räumlich ein. Den Regionen werden anschließend Eigenschaften (Features) zugeordnet, die das enthaltene Objekt beschreiben.

Da das GDM auf eine Vielzahl verschiedener Anwendungsfälle angewendet werden soll, müssen Flexibilität und Erweiterbarkeit eine wichtige Eigenschaft des Modells sein.

Letztendlich soll es möglich sein, mittels eines Transformationsschrittes aus den modellierten Daten eine Speicherstruktur abzuleiten. Die Zielplattform der Transformation ist dabei beliebig. Da jedoch nicht alle Konstrukte aus UML ohne weiteres abbildbar sind – speziell auf die in dieser Arbeit verwendeten Zielplattformen – muss es Einschränkungen bei dem im GDM verwendbaren UML-Sprachumfang geben. In dieser Arbeit werden folgende Systeme als Zielplattformen verwendet: DB2 Universal Database v8.1, ein objektrelationales Datenbanksystem von IBM, und der IBM Content Manager V.8.1.

4.2 Entwicklung des Modells

Einige Ideen, die im Zusammenhang mit dem eNoteHistory-Projekt entstanden, sind bereits in Abschnitt 2.4 genannt worden. Diese werden hier aufgegriffen, und zusammen mit anderen Aspekten der Bildbeschreibung detailliert untersucht.

4.2.1 Bildsegmentierung

Bei der Beschreibung eines Bildes wird es stets in relevante Teile zerlegt. Jeder Teil kann dann einzeln beschrieben und ausgewertet werden. In diesem Abschnitt sollen verschiedene Aspekte einer Bildsegmentierung betrachtet und deren Eignung für das GDM geklärt werden.

Vollständige/Disjunkte Zerlegung

Wird ein Bild restlos in Regionen zerlegt, ohne das es Bereiche im Bild gibt, die zu keiner Region gehören, so sprechen wir von einer vollständigen Zerlegung. Diese Art der Bildteilung ist interessant, wenn für jeden Punkt auf dem Bild eine Beschreibung existieren soll. Für diesen Fall müsste auch festgelegt werden, dass jeder Region mindestens ein Feature zuzuordnen ist. Die Forderung nach einer vollständigen Zerlegung ist jedoch nicht intuitiv, da es (aus der Sicht des Nutzers) auf jedem Bild Bereiche gibt, in denen keine relevanten Informationen liegen. Bei der Speicherung bringt die vollständige Zerlegung ebenfalls keine Vereinfachungen und Vorteile. Im GDM wird demnach keine vollständige Zerlegung des Bildes gefordert.

Eine andere Frage im Zusammenhang mit der Zerlegung ist, ob Regionen sich überlappen dürfen oder nicht. Anders als bei der vollständigen Zerlegung hat dieses Kriterium Auswirkung auf die Anfragen und Updates in der Datenbank. Diese Probleme treten beispielsweise auch bei R-Bäumen auf (siehe dazu [SH99]). Lässt man eine Überlappung zu, so verschlechtern sich die Anfragen, bei denen eine Region zu einem bestimmten Punkt gesucht werden. Sind Überlappungen verboten, so kann zwar zu jedem Punkt genau eine Region bestimmt werden, jedoch müssen beim Einfügen neuer Regionen alte eventuell aufgeteilt werden.

Da Anfragen nach Regionen zu bestimmten Pixeln nicht zu den Anforderungen an das GDM gehören, und in der Praxis sich überlappende Regionen nicht ausgeschlossen werden können, ist im GDM eine Überlappung erlaubt. Es wäre mit disjunkten Regionen beispielsweise nicht möglich die Noten und Notenlinien beim eNoteHistory-Projekt vernünftig abzubilden, da die Noten mit einer Wahrscheinlichkeit von ca. 50% immer auf einer Notenlinie liegen werden, und selbst dann muss sehr präzise gearbeitet werden um die restlichen Noten zwischen, über und unter den Linien des Notensystems in Regionen einzugrenzen.

Gruppierung von Regionen, Aufbau einer Hierarchie

Es kann sinnvoll sein, mehrere Regionen zu gruppieren, und der Gruppe dann Features zuzuordnen. Um bei eNoteHistory zu bleiben, sei folgendes Beispiel betrachtet: Jede Note und jede Pause wird durch eine Region umschrieben. Es ist nun möglich mehrere Noten zu einem Takt zu gruppieren, anschließend mehrere Takte zu einer Melodie. So könnte eine Notenerkennung unterstützt werden, da relevante Objekte innerhalb eines Takts bereits zusammengehörig gespeichert werden und so mit einer einzigen Anfrage verfügbar werden. Auch könnten Takten und Melodien dann Eigenschaften zugeordnet werden.

Für die Modellierung bedeutet das, dass Features nicht nur Regionen zugeordnet werden können, sondern auch Gruppen, und das wir eine Definition von Gruppen benötigen, die neben Regionen auch wiederum Gruppen beinhalten kann. Bei der Transformation eines konkreten Modells in eine relationale Datenbank müssten Tabellen zur Zuordnung von Features zu Gruppen sowie Regionen existieren. Außerdem werden zwei Tabellen zur Gruppendefinition benötigt, eine für Gruppen von Regionen, eine andere für Gruppen von Gruppen. Wollen wir nun auch noch zulassen, dass Gruppen gleichzeitig Regionen und andere Gruppen enthalten können, so wird die Transformation etwas komplizierter, da nun eine weitere Tabelle in der Datenbank benötigt wird, die eine Beziehung zwischen den letzteren beiden Tabellen herstellt.

Letztendlich führt die Einführung von Gruppen dazu, dass eine Hierarchie auf Regionen definiert werden kann. Statt Gruppen und Regionen getrennt zu betrachten, ist sinnvoller eine Menge von Regionen wiederum als Region aufzufassen. Es ließe sich so eine Hierarchie aufbauen, dessen sämtliche Knoten Regionen sind. Streng genommen erlauben solche Hierarchien keine Überlappung von Regionen, also das Enthaltensein der selben Region in verschiedenen übergeordneten Regionen, da das Resultat dann kein Baum, sondern eine Art geordnetes Netzwerk wäre. Im konzeptuellen Modell können solche Überlappungen allerdings durchaus zugelassen werden, jedoch wird dann die Effizienz der Anfragen auf einer Zielplattform in jedem Fall verschlechtert. Es sollte daher dem Nutzer überlassen werden, ob er Regionen als disjunkt definiert, oder Überlappungen zulässt. Dies kann mit Hilfe der Kardinalität im UML-Modell gesteuert werden.

Geordnete Bildsegmentierung

In MPEG-7 wird zur Bildsegmentierung neben freier Regionsdefinition auch eine Bildunterteilung mittels eines gleichmäßigen *Gitters* (*Grid*) angeboten. Auf diese Weise ist die explizite Definition einer Region überflüssig, die Zuordnung eines Features würde dann unter Angabe einer

Gitterzelle erfolgen. Diese Zerlegung kann zwar simuliert werden indem mehrere rechteckige Regionen über dem Bild definiert werden, aber der Speicheraufwand im Vergleich zu einem Grid würde sich erheblich vergrößern, denn es müssen im Grid nur dessen Gesamtgröße und die Anzahl der Unterteilungen in jeder Richtung angegeben werden, während jedes Rechteck durch zwei Koordinaten definiert wäre.

Die Möglichkeit der Grid-Segmentierung ist demnach sinnvoll und ist daher auch im GDM untergebracht.

Form freier Regionen

Als freie Region sei jede Region bezeichnet, die sich durch Position und Ausdehnung definiert, und nicht beispielsweise durch die Position in einem Grid. Bisher wurde viel von Regionen bzw. ROIs gesprochen, aber noch wurde die Form von Regionen nicht konkretisiert.

Die gängigste Form ist wohl das umschließende Rechteck (bounding rectangle/box). Denkbar sind jedoch auch Polygonzüge, oder andere geometrische Objekte wie Kreise und Ellipsen, oder auch Bezier-Kurven oder B-Splines. Nicht zu vernachlässigen ist auch die Möglichkeit Regionen um einen bestimmten Winkel zu drehen, um sie dem Bildobjekt anzupassen. Eine andere Form der Region ergibt sich aus der Betrachtung der High-Level Features (siehe dazu Abschnitt 4.2.4 zur Objekterkennung, sowie Abbildung 4.3), wo das Ergebnis einer Objekterkennung als Maske dargestellt wird. Auch Masken eignen sich zur Definition von Regionen. Sicher lassen sich noch viele andere Arten von Regionen finden, denn die Form variiert je nach Anwendung.

Es wäre demnach nicht sehr sinnvoll die Regionsformen zu beschränken. Deshalb ist im GDM eine abstrakte Oberklasse für Regionen definiert, die Assoziationen zu anderen Klassen besitzt, sowie einige konkrete Unterklassen, die bereits gängige Ausprägungen von Regionen darstellen. Der Nutzer hat die Möglichkeit durch die Bildung eigener Unterklassen neue Regionen zu definieren.

Attribute einer Region

Jeder Region sollen letztendlich Eigenschaften (Features) zugeordnet werden. Dabei können Features sehr einfach oder auch sehr kompliziert strukturiert sein. Hat das Feature nur ein einziges Attribut, weil es z.B. ein dominierender Farbwert ist, so könnte man diesen Wert statt als Feature auch als Attribut der Region betrachten. Dadurch würde sich die Speicherung vereinfachen, denn es muss nur ein weiteres Attribut bei der Region gespeichert werden.

Da die Modellierung jedoch konzeptuell erfolgen soll, ist es sauberer Regionen und Features klar zu trennen. Dies bringt zwar später einen gewissen Overhead bei der Speicherung mit sich, aber das Modell ist dadurch besser strukturiert, da Features eindeutig zu erkennen sind, während sonst auch jede Region implizit ein Feature sein könnte. Im GDM sollten Regionen demnach nur Attribute haben, die zur Festlegung ihrer Position und Ausdehnung dienen.

4.2.2 Typisierung

Es ist für eine Abbildung auf eine Speicherstruktur unbedingt notwendig, Datentypen in das Modell einzubeziehen. Anderenfalls wäre beispielsweise eine Abbildung auf eine relationale Datenbank nicht möglich, oder es müsste jede Spalte als (VAR)CHAR definiert sein, und eine externe Anwendung würde diese Daten in die entsprechenden Typen umwandeln. Hierbei kann man jedoch kaum von effizienter Speicherung sprechen.

UML hat nur ein sehr einfaches Typsystem (siehe [OMG03]), lässt jedoch als Datentypen neben Klassen auch die Typen einer Programmiersprache zu. UML liefert auch eigene primitive Datentypen mit, diese dienen allerdings hauptsächlich der Definition von UML selbst, also der Verwendung im UML-Metamodell, z.B. für Constraints und Kardinalitäten. Die folgenden Typen sind in UML vordefiniert:

- Boolean: die Wahrheitswerte `true` und `false`
- Integer: ganzzahlige Werte im Intervall von $-\infty$ bis $+\infty$; Dieser Wert kann auf den größten `Integer`-Datentyp abgebildet werden. Allerdings ist es sehr wohl denkbar, dass der Nutzer den Wertebereich bereits im Vorfeld einschränken will. Dies kann mit dem UML-Datentyp nicht ohne den Umweg über ein Constraint dargestellt werden. Bei der Transformation kann ohne Einschränkungen des Wertebereichs nur auf den größtmöglichen Integer-Typ abgebildet werden.
- String: eine beliebige Zeichenkette, gebildet aus den Zeichen eines „passenden“ Zeichensatzes.
- UnlimitedNatural: dieser Typ stellt eine beliebige, natürliche Zahl (≥ 0) dar; unendlich wird mit einem * notiert. Für diesen Datentyp gelten die selben Anmerkungen wie für Integer

Neben Standard-Datentypen wie Zeichenketten, verschiedenen Zahlentypen und Wahrheitswerten ist es insbesondere bei Low-Level Features interessant spezielle komplexe Datentypen, wie beispielsweise Histogramme, zu definieren. Außerdem müssen auch Matrizen, Mengen und Listen im Typsystem berücksichtigt werden. Im GDM sind deshalb diverse Datentypen vordefiniert. Die Umsetzung in die Zielplattformen wird Aufgabe der Transformation sein, die in Kapitel 5 ausführlich erklärt wird.

Eine Erweiterung des Modells ist möglich, erfordert aber eine Anpassung der Implementierung. Neue Typen erfordern eine Anpassung der Grammatik sowie der Transformation.

4.2.3 Modellierung des Bildes

Das eigentliche Bild wird als Klasse definiert. Diese Klasse wird einige physische Metadaten als Attribute enthalten, wie beispielsweise URL und Bildgröße. Die Features werden jedoch nicht als Attribute gespeichert, denn konzeptuell sollen Features von den Bilddaten getrennt werden.

Alle anderen Elemente werden anschließend als eigene Klassen modelliert und in Beziehung zum Bild gebracht. So können beispielsweise Regionen als Komponenten des Bildes dargestellt werden.

Außerdem wird es nicht möglich sein von der Bild-Klasse zu erben. Die Klasse der Bilder ist Ausgangspunkt für alle weiteren Beziehungen, eine Erweiterung in Form von Vererbung ist nicht sinnvoll. Technisch gesehen ist ein Bild stets eine rechteckige Region in der grafische Elemente den Inhalt definieren. Das Bild kann einerseits in Form einer Vektorgraphik vorliegen, d.h. als Bild, dessen Koordinaten sich im reellen Zahlenbereich im Intervall 0.0 bis 1.0 je Raumachse bewegen, und somit beliebig verlustfrei skalierbar sind. Geläufiger ist die Speicherung als Rasterbild, d.h. als Matrix von Pixeln. Beide Typen unterscheiden sich nur in der Art der Speicherung und Qualität der Darstellung, sind aber auf gleiche Weise zu beschreiben. Es spricht jedoch nichts dagegen, zusätzliche Attribute in die Klasse Bild aufzunehmen, sofern es sich um Eigenschaften des Bildes, und nicht dessen Inhalts handelt, also entweder um physische Informationen oder um Metadaten.

Im GDM ist die Klasse der Bilder mit der Einschränkung `final` gekennzeichnet, was semantisch das Ende einer Vererbungshierarchie anzeigt. Dies erfolgt in Anlehnung an die Programmiersprache Java, in der Klassen, die nur in den Blättern der Klassenhierarchie vorkommen dürfen, ebenfalls als `final` gekennzeichnet werden.

4.2.4 Features

Nachdem wir nun das Bild und dessen Unterteilung geklärt haben, ist es Zeit, über die eigentlichen Daten, die Features, zu sprechen. Bei Features sind zwei Arten zu unterscheiden:

- Low-Level-Features: Diese Features lassen sich in der Regel automatisch bestimmen, und beschreiben eher technische Einzelheiten des Bildes. Typische Beispiele für Low-Level-Features sind Farbhistogramme und Farbwerte.
- High-Level-Features: Diese Features beschreiben den Inhalt des Bildes, wie beispielsweise Objekte, die sich im Bild befinden. Sie lassen sich nicht immer automatisch bestimmen, da dafür sehr spezielle Algorithmen notwendig sind.

Die folgenden Abschnitte werden sich mit den Feature-Typen genauer auseinander setzen sowie Ansätze zur Modellierung erarbeiten. Es wird Features geben, die unter gewissen Gesichtspunkten auch zu den Features des jeweils anderen Levels zugehörig angesehen können. Als Kriterium für die Einordnung dient hier die Qualität des Ergebnisses und der Aufwand der Berechnung. Die Erkennung von Farbwerten ist beispielsweise einfach zu berechnen und liefert ein eindeutiges, nie verfälschtes Ergebnis. Auch Kantenerkennung kann als eine sehr akkurat arbeitende Technik angesehen werden, im Bild vorkommende Kanten werden durch Auffinden von Kontrasten in benachbarten Pixeln gefunden. Anders ist es beispielsweise bei einer Text- oder Objekterkennung, die nicht immer akkurat ist, und dessen Algorithmus als verhältnismäßig komplex bezeichnet werden kann. Im GDM werden Low- und High-Level Features nicht unterschiedlich behandelt, die Kategorisierung und mögliche Fehler dabei bleiben daher ohne Folgen.

Low-Level Features

Zu den Low-Level-Features zählen unter anderem folgende Bildeigenschaften:

- dominierende Farbe: Dieses Feature lässt sich mittels eines Farbwertes speichern. Farbwerte sind, je nach verwendetem Farbsystem, aus verschiedenen Komponenten bestehende Werte.

  In der Bildverarbeitung ist *RGB[A]* weit verbreitet, wobei die Farbe unter Angabe des Rot-, Grün- und Blau-Anteils, gegebenenfalls zuzüglich eines Transparenz-Wertes (Alpha-Wert), gespeichert wird.

  Ein anderes Farbsystem ist *CMYK*, bei dem die Farben in Zyan-, Magenta-, Gelb- und Schwarz-Komponenten zerlegt angegeben werden.

  Ein weiteres Farbsystem ist *HSV* (Hue Saturation Value). Es schlüsselt Farben in Farbwert entlang des Spektrums (hue), Farbsättigung (saturation) und Farbhelligkeit (value) auf. Statt Farbhelligkeit findet man auch die Begriffe Intensität oder Brightness, was zu den Abkürzungen HSI und HSR führt. Oftmals werden RGB bzw. CMYK auf HSV abgebildet, da es eine bessere Farbmetrik hat, so dass Abstände zwischen zwei Farben besser, d.h. für den Betrachter besser nachvollziehbar, berechnet werden können.

  Jeder einzelne Farbwert kann mittels 8-bit Ganzzahlen, oder als Fließkommawert zwischen 0 und 1 dargestellt werden. Die allgemeinere Darstellung ist ein Vektor aus Fließkommazahlen, der auch abweichende Farbauflösungen abbilden kann.

  Ähnliche Features zur dominierenden Farbe sind beispielsweise hellste, dunkelste, seltenste oder häufigste Farben, weitere Möglichkeiten sind denkbar.

- Farbhistogramme: Hierbei handelt es sich um Diagramme, auf denen die Häufigkeit einer bestimmten Farbe abzulesen ist. In der Praxis sind Histogramme für die jeweiligen Farbkanäle (z.B. Rot, Grün und Blau) üblich, sowie Histogramme über Helligkeitswerte, Farbtöne, Sättigung oder Graustufen.

  An der y-Achse liegen Häufigkeitswerte, also ganzzahlige Werte. An der x-Achse die Werte, nach denen das Histogramm erstellt wird, z.B. die Farbwerte. Liegen die Werte der x-Achse als Fließkommazahlen vor, so muss mit Intervallen gearbeitet werden. Folgendes Beispiel verdeutlicht warum: Angenommen, es existieren im Intervall 0.0 bis 0.1 der Rotkomponente 1000 Werte. Jeder Wert ist verschieden, es existieren keine zwei gleichen Werte. Im Histogramm wäre demnach die Häufigkeit jedes Rottons jeweils 1. Neben dem enormen Speicheraufwand zur Speicherung dieser Information ist auch ihr praktische Nutzen zu bezweifeln. Es ist besser ein, optimaler weise nutzerdefiniertes, Intervall vorzugeben, und die Häufigkeit der in diesem Intervall vorkommenden Werte zu speichern. Im Beispiel würde ein Intervall der Größe 0.1 aus 1000 zu speichernden Wertepaaren ein einziges machen. Im GDM können Histogramme als Typen definiert werden, in der Transformation müssen dann die passenden Speicherstrukturen gefunden werden.

  Die Abbildungen 4.1 und 4.2 zeigen ein Beispiel für Histogramme. Das Beispielbild zeigt eine Grafik, bei der auf weißem Hintergrund mehrere Pixel in der selben Farbe dargestellt

werden. Deren Rotkomponente hat den Wert 100 auf einer Skala von 0 bis 255. Auf dem Bild der Größe 100x100 Pixel sind 3973 farbige Pixel enthalten. Wird die Anzahl weißer Pixel auf den Wert 1.0 normiert, so ergibt sich im Histogramm ein Wert von 0.659 den Rot-Wert 100.

Abbildung 4.1: Beispielbild zum Histogramm

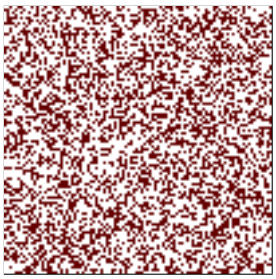
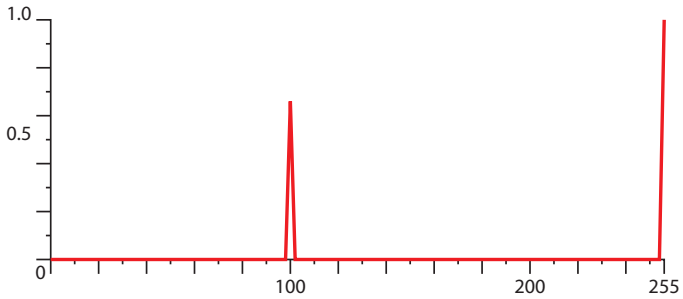


Abbildung 4.2: Histogramm zum Beispielbild (Abb.4.1)



- Erkennung von Kanten: Es gibt verschiedene Algorithmen um Kanten im Bild zu finden. Dies geschieht im Rasterbild durch Erkennung starker Helligkeitsunterschiede. In Vektorbildern können Kanten erkannt werden indem Konturen und Farbwerte angrenzender Objekte untersucht werden.

  Betrachten wir zunächst das Rasterbild. Bei der Kantenerkennung wird das Bild durch Anwendung eines Filters verändert. Das Resultat ist ein neues Bild, auf dem dort, wo Kanten im Bild erkannt werden, helle Pixel stehen, an anderen Orten dunkle Pixel. Das Ergebnis ist also wiederum ein neues Bild. Es muss demnach möglich sein als Feature wieder ein Bild anzugeben. Dies kann in Form eines Attributes erfolgen, dessen Typ die Klasse Bild ist. Alternativ kann eine Beziehung zwischen dem Feature „Kante“ und dem Bild hergestellt werden. Beide Varianten sind mit dem GDM umsetzbar.

  Erkennen wir Kanten im Vektorbild, so ist das Resultat eine Menge von Polygonzügen. Diese bestehen aus Abschnitten, die wiederum durch eine Linie oder Kurve bestimmt sind. Der Grundtyp für die Speicherung ist eine Liste, deren Elemente Vektoren von Grunddatentypen sind. Für dieses Feature sind keine Anpassungen im GDM notwendig, es kann bereits mit den vorgesehenen Konzepten realisiert werden.

High-Level Features

- Erkennung von Formen: Das Ziel einer solchen Erkennung ist, geometrische Gebilde im Bild zu identifizieren. Dazu werden Bildausschnitte mit Vorlagen der geometrischen Objekte verglichen. Das Resultat sind Koordinaten und andere Parameter zur Bestimmung der Lage des jeweiligen Objektes. Diese können mittels Grunddatentypen dargestellt werden und sollten in einem komplexerem Datentyp, wie beispielsweise einer Menge, untergebracht werden können.

- Texturbeschreibung: Texturen beschreiben die Struktur einer Oberfläche. Ein Bild besteht oftmals aus vielen verschiedenen Texturen. Diese können in Form von Grauwerten angegeben werden. Eine Textur ist demnach als Graustufenbild beschreibbar, welches mit einem Attribut vom Typ Bild oder Matrix im GDM dargestellt werden kann.

  Die Eigenschaften von Texturen lassen sich im wesentlichen durch Matrizen und Skalare ausdrücken. Für Details hierzu sei auf die Literatur verwiesen. Weiterführende Informationen lassen sich beispielsweise in [HR04] finden. Für das GDM ist nur interessant, das es sich dabei um Grunddatentypen handelt, bzw. Anordnungen (Vektoren, Matrizen) von Grunddatentypen.

- Objekterkennung: In [FGLX04] wird eine Möglichkeit zu Erkennung bestimmter Objekte in Bildern beschrieben. Dafür werden im Bild so genannte *salient objects*, also wichtige Objekte, erkannt. Dem System zur Erkennung werden dafür mittels einer Trainingsmenge die Eigenschaften bestimmter Objekte, beispielsweise Himmel oder Sandstrand, beigebracht, woraufhin in neuen Bildern diese Objekte automatisch erkannt werden. Das Ergebnis der Erkennung kann ein Teil des Bildes sein, der nicht zusammenhängen muss und der mitunter eine sehr schwer beschreibbare Form hat. Liegt das Bild als Rasterbild vor, so kann die Angabe des Objekts mit Hilfe einer *Maske* erfolgen. Eine Maske ist eine Matrix mit der selben Größe wie das Bild, Inhalt der Maske sind Bits. Ist der Wert eines solchen Bits 1, so ist das Pixel mit den selben Koordinaten im Bild auch Teil des Objektes.

  Um dies zu verdeutlichen soll folgendes Beispiel helfen: Abbildung 4.1 soll dafür als Vorlage dienen. Abbildung 4.3 zeigt links eine Maske, wobei weiße Pixel diejenigen Bits mit Wert 1, und schwarze Pixel Bits mit Wert 0 darstellen. Das rechte Bild zeigt den Bildausschnitt, der durch diese Maske definiert ist, nicht selektierte Pixel werden grau dargestellt.

  Zu beachten ist, dass das Feature, also das Objekt im Bild, durch eine Region beschrieben wird. Eine Maske kann demnach sowohl als Feature, als auch als Region auftreten. Daraus ergibt sich die Frage, wie sich das Ergebnis der Objekterkennung modellieren lässt. Einerseits können innerhalb eines Bildsegments Objekte erkannt werden, die als Features modelliert werden. Eine Eigenschaft dieser Features wäre dann die Lage des Objektes, welche durch eine Maske beschrieben wird. Andererseits könnten die erkannten Objekte als Regionen modelliert werden, die hierarchisch unterhalb des betrachteten Bildsegments angeordnet werden. Die Features der Regionen, die die Objekte beschreiben, sind dann die Eigenschaften des jeweiligen Objektes. Beide Ansätze sind möglich, und das GDM

Abbildung 4.3: Beispiel für eine Maske und den entsprechenden Bildausschnitt

unterstützt auch beide, da eine Maske sowohl als Datentyp zur Verwendung in Features, als auch als Region definiert werden kann.

Modellierung von Features

Im GDM werden Features als eine Klasse definiert. Dieser Abschnitt hat jedoch gezeigt, dass Features sehr verschiedene Ausprägungen haben können. Diese Unterschiede bei den Features müssen auch im Modell berücksichtigt werden. Hierfür bietet sich das Konzept der Vererbung an. Ausgehend von der Klasse der Features kann der Nutzer seine Features davon ableiten und mit eigenen Attributen versehen. Alle Zusammenhänge mit Features, die im Modell gelten, werden mit der Oberklasse geknüpft. So wird sichergestellt, dass jedes Feature unabhängig von seiner Art, mit anderen Klassen in Beziehung stehen kann. Dies ist vor allem für die Erweiterbarkeit der modellierten Daten von Interesse. So können beispielsweise neue Regionsarten hinzugefügt werden, ohne dass Beziehungen neu geknüpft werden müssen.

4.2.5 Funktionen/Operationen

Ein wichtiges Konzept im objektorientierten Entwurf mit UML sind Methoden. Diese machen aus Klassen mehr als statische Attributsammlungen. Zunächst stellt sich jedoch die Frage, welche Rolle Methoden in einem Modell zur Bildbeschreibung spielen. Prinzipiell ist das GDM zur Beschreibung von Bildinhalten gedacht, wobei zunächst keine Operationen vorkommen. Allerdings werden Features in der Regel nicht manuell eingegeben, sondern von einem Algorithmus möglichst automatisch ermittelt. Unterstützt die Zielplattform Operationen auf den gespeicherten Daten, so können derartige Extraktionsprozesse integriert werden.

Für die Extraktion der Features gibt es mehrere Möglichkeiten:

- Für jedes Feature ist eine eigene Methode zur Extraktion definiert. Dadurch wird eine sehr starke Modularität erreicht. Ein Nachteil hierbei ist jedoch, das Extraktionen, die auf den selben Vorverarbeitungsschritten basieren, nicht zusammengefasst werden können, oder große vorverarbeitete Bilder gespeichert werden müssen.

- Alle Features des Bildes werden durch einen einzigen Prozess erzeugt. Die Extraktionsmethode ist in diesem Fall in der Klasse `Image` zu definieren. Dadurch wird der Nachteil der ersten Möglichkeit eliminiert, jedoch auch ihr Vorteil, denn da nun nur eine einzige Extraktionsmethode existiert, erfolgen Änderungen am Algorithmus stets am kompletten Code. Auch können einzelne Teile nicht ohne alle anderen verwendet werden, was speziell im Bereich der Low-Level-Features als eher nachteilig angesehen werden muss.

Welche dieser Möglichkeiten eingesetzt wird, hängt stark von der Anwendung ab. Es ist auch möglich beides zu kombinieren, indem beispielsweise einfach zu berechnende Features, wie z.B. Farbverteilungen, als Methode beim jeweiligen Feature modelliert werden, wobei die Extraktion komplexerer Features, wie z.B. die Erkennung verschiedener Objekte, als Methode des Bildes modelliert werden können.

Ein weiterer Punkt im Zusammenhang mit Operationen, speziell im Hinblick auf die Transformation, ist der Wirkungsbereich der Funktionen. In UML können Parameter und Rückgabetypen für Methoden spezifiziert werden. Es stellt sich jedoch die Frage, inwiefern es einer Funktion gestattet ist Daten zu manipulieren. Es wird üblicherweise Seiteneffektfreiheit bei Operationen gefordert, d.h. eine Funktion berechnet lediglich einen Ausgabewert aus verschiedenen Eingabewerten. In der Objektorientierung erweitert das Methodenkonzept dieses Vorgehen um Möglichkeiten zur Objektmanipulation. Darf aber eine Methode, die in der Klasse `Image` definiert ist Instanzen der Klassen `Region` und `Feature` bilden? Um eine Integration der Feature-Extraktion in das Modell zu ermöglichen muss diese Frage mit „ja“ beantwortet werden können. Der Anwender muss dies in jedem Fall bei der Integration des Extraktionsprozesses in die Zielplattform berücksichtigen.

Operationen können definiert werden um Regionen zu verändern. So könnten beispielsweise Methoden zum trennen und zusammenfügen von Regionen modelliert werden, die später auf der Zielplattform eine Manipulation von Regionen erlauben.

4.3 Das konkrete Datenmodell

4.3.1 Das Typsystem

Die Syntax

Da das Typsystem von UML nicht vorgegeben ist, ist es frei definierbar. Die in UML vordefinierten Typen dienen der Definition von UML selbst. Da UML eine Sprache zur Anwendungsentwicklung ist, wird im Standard die Verwendung der Typen einer Programmiersprache vorgeschlagen. Für das GDM existiert jedoch keine spezielle Zielplattform, wodurch ein Typsystem notwendig wird. Lediglich wenn die Zielplattform bereits bekannt ist kann auf dessen Typsystem zurückgegriffen werden.

Um eine optimale Flexibilität zu erreichen, werden die Typen mit Hilfe einer kontextfreien Grammatik definiert. Diese Grammatik wird in der Erweiterten Backus-Normalform (auch Erweiterte Backus-Naur Form, EBNF) notiert, wodurch sie für Parser einfach umzusetzen sein wird.

Durch Verwendung eines Parser-Generators kann das Typsystem mit moderatem Aufwand erweitert werden. Änderungen am Typsystem betreffen neben der Grammatik allerdings auch die Transformationsregeln.

Die Grundidee für das Typsystem ist folgende: Es soll eine Menge von Grunddatentypen zur Verfügung gestellt werden. Dies werden aus Programmiersprachen bekannte Typen sein, wie z.B. Integer, String und Boolean. Weiterhin werden komplexe Datentypen definiert, um Strukturen wie Listen, Mengen und Felder zu definieren. Der Nutzer hat dann die Möglichkeit mit Hilfe dieses Systems eigene Typen zu definieren, diese anschließend zu benennen und auf seine Attribute anzuwenden.

Es folgt nun die schrittweise Definition der Grammatik. Die vollständige Grammatik ist im Anhang B.1 zu finden. Es ist dabei zu beachten, dass von der Grammatik Groß- und Kleinschreibung unterschieden werden.

Bevor wir Typen definieren müssen noch einige Terminale definiert werden. Da jede Typdefinition sauber abgeschlossen sein sollte, sei zunächst folgendes Symbol definiert[1]:
`TypeDefTerminator = ';'.`
Es schließt eine Typdefinition ab. Ein weiteres Symbol ist notwendig, um die Parameter einiger Datentypen, z.B. die Größe eines Vektors, zu markieren. Dieses Symbol wird wiefolgt definiert:
`TypeDefSeparator = ':'.`
Da Größenangaben als Zahlen notiert werden, muss vorher definiert sein wie eine Zahl aufgebaut ist. Dazu definieren wir die Menge der Zeichen 1 bis 9 als Nicht-Terminal `PosDigit`. Wir definieren nun das Nicht-Terminal `Digit` als
`Digit = '0', PosDigit`,
welches jede Ziffer darstellt. Eine positive Zahl ist nun definiert als
`AnyNumber = PosDigit, {Digit}.`
Zusätzlich werden Vorzeichen, Länge von Integern und Präzision von Fließkommazahlen definiert.

Wir beginnen nun mit dem Wurzelelement der Typdefinition, `Type`, welches entweder ein Grunddatentyp oder ein komplexer Typ ist:
`Type = (SimpleType | ComplexType).`

Folgende Grunddatentypen sind definiert: Ganzzahlige Typen (`IntegerType`), Fließkommatypen (`FloatType`), Zeichentypen (`CharacterType`), Wahrheitswerte (`BooleanType`), Binärdaten (`BinaryType`) und Aufzählungen (`EnumType`). Die jeweiligen Definitionen dieser Typen sind dem Anhang B.1 zu entnehmen.

Folgende Komplexe Datentypen werden definiert:

- Menge (`SetType`): Eine Menge enthält verschiedene Elemente gleichen Typs ohne besondere Ordnung und ohne Duplikate.

[1] Die Angabe der EBNF-Regeln erfolgt im Text der Übersichtlichkeit halber ohne abschließendes Semikolon

- Multimenge (`BagType`): Eine Multimenge verhält sich wie eine Menge, allerdings sind Duplikate erlaubt.

- Vektor (`VectorType`): Ein Vektor, oder eine Liste, ist eine Datenstruktur mit festgelegter oder dynamischer Größe, bei dem alle Elemente gleichen Typs sind. Im Gegensatz zur Menge existiert eine Ordnung auf den Elementen.

- Matrix (`MatrixType`): Eine Matrix ist eine mehrdimensionale[2] Struktur, deren Elemente von gleichem Typ sind. Jedem Element kann eindeutig eine Position in der Matrix zugeordnet werden, und umgekehrt.

- Struktur (`StructType`): Eine Struktur enthält verschiedene Datentypen, die mit einem Namen referenziert werden.

Die Elemente der komplexen Typen können beliebige andere Typen sein. Dabei kann als Element eine Typdefinition, oder der Name eines bereits deklarierten Typs verwendet werden.

Die Deklaration eines Typs wird ebenfalls in der Grammatik festgelegt. Bei der Deklaration wird ein Name, gefolgt von einem Zuweisungssymbol und einer Typdefinition angegeben. Der Name eines Typs ist beliebig, darf jedoch keines der Zeichen enthalten, die in der Typdefinitionssprache verwendet werden.

Semantik

Als erstes werden die Grunddatentypen betrachtet. Hier wird die Definition des jeweiligen Typs gezeigt, die Parameter erklärt, gegebenenfalls Bedingungen an die Parameter formuliert und der Wertebereich angegeben.

- Ganze Zahlen
  Definition: `Int:`*n*`:`*s*`;`
  Parameter: *n*: Anzahl Bits, möglich sind 8, 16, 32 und 64; *s*: `signed` für eine vorzeichenbehaftete Zahl, oder `unsigned` für eine nicht vorzeichenbehaftete Zahl
  Wertebereich: bei Angabe von `signed`: $-2^{n-1}..2^{n-1}-1$; sonst: $0..2^n-1$

- Fließkommazahlen
  Definition: `Float:`*p*`;`
  Parameter: *p*: Präzision, möglich sind `single` und `double`, die Interpretation erfolgt nach [IEEE85]
  Wertebereich: Single Precision $\pm(2-2^{-23})^{127}$ (ca. $\pm 10^{38.53}$), Double Precision $\pm(2-2^{-52})^{1023}$ (ca. $\pm 10^{308.25}$)

[2] Eine Matrix kann hier, entgegen ihrer mathematischen Definition, durchaus mehr als zwei Dimensionen besitzen.

- Wahrheitswerte
  Definition: `Bool;`
  Wertebereich: {true, false}

- Zeichen
  Definition: `Char;`
  Wertebereich: ein einziges, beliebiges Zeichen aus dem Unicode-Zeichensatz

- Zeichenketten
  Definition: `String:`*n*`;`
  Parameter: *n*: maximale Länge
  Wertebereich: eine beliebige Zeichenkette mit höchstens n Zeichen des Unicode-Zeichensatzes

- Binäre Daten
  Definition: `Binary:`*n*`;` oder `FileRef;`
  Parameter: *n*: maximale Größe in Kilobytes
  Wertebereich: Binärdaten beliebiger Größe

- Aufzählungen
  Definition: `Enum{`*Element*$_1$`,...,`*Element*$_n$`};`
  Wertebereich: die Elemente *Element*$_1$ bis *Element*$_n$

Als nächstes betrachten wir komplexe Datentypen, wiederum mit Definition und Parameter der Typen, außerdem mit den darauf möglichen Operationen.

- Mengen
  Definition: `Set{`*typ*`};`
  Parameter: *typ*: eine Typdefinition, oder der Name eines bereits definierten Typs
  Operationen: Hinzufügen und Entfernen von Elementen, Test auf leere Menge, Test auf Enthaltensein eines Elements, Vereinigung, Differenz und Durchschnitt, Mächtigkeit

- Multimengen
  Definition: `Bag{`*typ*`};`
  Parameter: *typ*: eine Typdefinition, oder der Name eines bereits definierten Typs
  Operationen: Hinzufügen und Entfernen von Elementen, Test auf leere Multimenge, Test auf Enthaltensein eines Elements, Anzahl Vorkommen eines Elementes, Vereinigung, Differenz und Durchschnitt, Mächtigkeit

- Vektor
  Definition: `Vector:`*n*`{`*typ*`};`
  Parameter: *n*: Anzahl der Elemente, oder `dyn` falls der Vektor dynamisch ist, d.h. die Anzahl der Elemente variiert; *typ*: eine Typdefinition, oder der Name eines bereits definierten Typs

Operationen: Größe des Vektors, Hinzufügen und Entfernen von Elementen, Positionsbestimmung eines Elements, Enthaltensein eines Elements, Elementbestimmung zu einer Position

- Matrix
  Definition: `Matrix:`n_1`:...:`n_k{*typ*}`;`
  Parameter: n_i: Anzahl der Elemente in der jeweiligen Dimension, oder `dyn`, wenn die Anzahl dynamisch ist, *k* ist die Anzahl der Dimensionen der Matrix; *typ*: eine Typdefinition, oder der Name eines bereits definierten Typs
  Operationen: Größe einer Dimension (bei dynamischer Anzahl von Elementen), Hinzufügen und Entfernen von Elementen, Positionsbestimmung eines Elements, Enthaltensein eines Elements, Elementbestimmung zu einer Position, Schnitte (Slices, Hyperebenen) durch die Matrix

- Struktur
  Definition: `Struct`{$name_1$`:`typ_1`,...,`$name_k$`:`typ_k}`;`
  Parameter: $name_i$: Name des i-ten Elements; typ_i: eine Typdefinition, oder der Name eines bereits definierten Typs
  Operationen: Zugriff auf die Komponenten, die mittels Angabe der Komponente in „Dot-Notation“ erfolgt. Ist der Name des Typs beispielsweise „Datum“, und eine Komponente heißt „Monat“, so wird der Zugriff auf diese Komponente als „Datum.Monat“ notiert.

Beispiele

Nachfolgend sei die Definition von Datentypen anhand einiger Beispiele demonstriert:

- RGB-Farbwert: Ein Farbwert ist ein Vektor mit 3 Komponenten, die Werte zwischen 0 und 255 annehmen können. Die Definition kann wiefolgt aussehen:
  ```
  FarbKomponente:= Int:8:unsigned;
  Farbe:= Vector:3{FarbKomponente};
  ```

- Histogramm: Ein Histogramm ist ein zweidimensionales Diagramm, bei der auf der x-Achse diskrete Werte, bzw. Intervalle kontinuierlicher Werte, und auf der y-Achse Fließkommazahlen für normierte Farbhäufigkeiten stehen. Es handelt sich somit um eine Zuordnung von Werten (y-Achse) zu diskreten Werten (x-Achse), und kann somit als Vektor dargestellt werden. Ein Histogramm kann wiefolgt definiert werden:
  ```
  Histogramm:= Vector:256{Float:double};
  ```

- Maske: Eine Maske ist eine zweidimensionale Matrix aus Bits, oder Wahrheitswerten, welche eine äquivalente Darstellung von Bits ermöglichen. Die Maske für das Beispielbild in Abbildung 4.1 kann wiefolgt definiert werden: (zur Erinnerung: das Bild hat eine Größe von 100 x 100 Pixel)
  ```
  Maske:= Matrix:100:100{Bool};
  ```

4.3.2 Allgemeines, Gemeinsame Eigenschaften der Klassen

Vordefinierte Komplexe Datentypen

Folgende Datentypen sind vordefiniert:

- Datum: Im Datum sind Tag, Monat und Jahr enthalten, Tag und Jahr als 8-bit Ganzzahlen, Monat als Element einer Aufzählung von „Jan“ bis „Dec“.
- Zeit: Eine Zeitangabe erfolgt unter Angabe von Stunden, Minuten und Sekunden, wobei Stunden und Minuten jeweils 8-bit Ganzzahlen sind, und die Sekunden als Fließkommazahl angegeben werden.
- Zeitstempel: Ein Zeitstempel kombiniert Datum und Zeit in einer Struktur.
- Bildpunkte: Als Punkttyp im Rasterbild wird eine Struktur mit einer x- und einer y-Komponente verwendet. Die Werte sind jeweils ganzzahlig und nicht negativ.
- Ausdehnung: Eine Ausdenhung besteht ebenfalls aus einer x- und einer y-Komponente, die jedoch vom Typ Fließkommazahl sind.
- Farbwerte: Es sind Strukturen definiert, die Farbwerte des RGB, CMYK und HSV-Farbraums enthalten können.

Die jeweiligen Definitionen, unter Verwendung des GDM-Typsystems, sind Anhang B.2 zu entnehmen.

Objektidentifikation

Neben der Objektidentität (OID) ist es auch sinnvoll, identifizierende Attribute (Schlüsselattribute) zu kennzeichnen. Attribute, die als Teil eines Schlüssels gekennzeichnet werden, müssen stets einen eindeutigen Wert zugeordnet haben. Auf diesem Wege ist es möglich, einen effizienteren Zugriff auf die Objekte zu ermöglichen, beispielsweise in einer Datenbank durch Erzeugen eines Index. Dadurch verringert sich auch der Aufwand bei der Ausführung von Anfragen auf die Datenbank.

Im UML-Diagramm werden Attribute, die Teil eines Schlüssels sind, mit dem Stereotyp `<<ID>>` gekennzeichnet. Alle mit diesem Stereotyp ausgezeichneten Attribute bilden zusammen einen Schlüssel. Falls mehrere Schlüssel definiert werden, so muss dies mittels einer Einschränkung in der Form $\{\texttt{key=}x\}$ erfolgen, wobei $x \in 1, 2, \ldots$ die Nummer des Schlüssels ist. Der Schlüssel mit der kleinsten Nummer wird als Primärschlüssel verwendet.

Falls die Zielplattform, auf die das Modell anschließend abgebildet wird, keine OIDs unterstützt, so müssen OIDs künstlich erzeugt werden, d.h. in einem Surrogatattribut mit einem automatisch generierten Wert gespeichert werden.

Datenobjekte

Da UML eine objektorientierte Sprache ist, hat der Nutzer auch die Möglichkeit seine Daten als Objekte zu modellieren. Dies kann insbesondere bei High-Level Features von Interesse sein, beispielsweise bei der Objekterkennung, bei der ein Objekt eine komplexe Struktur sein kann und deshalb als eigene Klasse modelliert sein könnte. Diese Klasse muss anschließend mit einer Assoziation einem Feature zugeordnet werden. Alternativ kann im Falle einer 1:1-Beziehung das Attribut als Komponentenobjekt, d.h. als Datentyp eines Attributes im Feature, definiert werden.

Methoden

Methoden sind in jeder Klasse erlaubt. Die Implementierung der Methoden ist Aufgabe des Benutzers. Durch Definition von Methoden werden – sofern dies auf der Zielplattform möglich ist – entsprechende Definitionen vorgenommen, und/oder leere „Hüllen“ für die Implementierung erstellt. Da die Anwendungs- und Ausführungsmöglichkeiten von Methoden stark von den Fähigkeiten der Zielplattform abhängen, sind im GDM keine Rahmenbedingungen für Methoden spezifiziert.

4.3.3 Die Klassen `Image` und `ImageMetaData`

Der Ausgangspunkt für die Beschreibung eines Bildes ist das Bild selbst. Dies wird mit einer Klasse namens `Image` modelliert, die für das Bild relevante Attribute trägt, und der anschließend Regionen zugeordnet werden können. Dabei werden (vgl. [Sta99]) physische Bildattribute, z.B. URI des Bildes, sowie das Bild selbst, in der Klasse `Image` abgelegt. Metadaten zum Bild, wie beispielsweise Autor und Copyright, werden stattdessen in der Klasse `ImageMetaData` gespeichert. Die Klassen stehen in einer 1:0..1-Beziehung zueinander. Eine Sichtweise auf Bilder ist, dass Bilder aus Teilbildern zusammengesetzt sein können. Um dies zu modellieren wird eine Aggregation vom Bild zu sich selbst definiert.

Da die Klasse der Bilder den Einstiegspunkt in die Datenbank bildet, ist es sinnvoll, Schlüsselattribute zu definieren. Alle anderen Objekte können dann durch das Verfolgen von Referenzen erreicht werden.

Zu den physischen Angaben zum Bild zählen beispielsweise Speicherort oder verwendete Farbkodierung. Im GDM sind in der `Image`-Klasse folgende Attribute bereits vordefiniert:

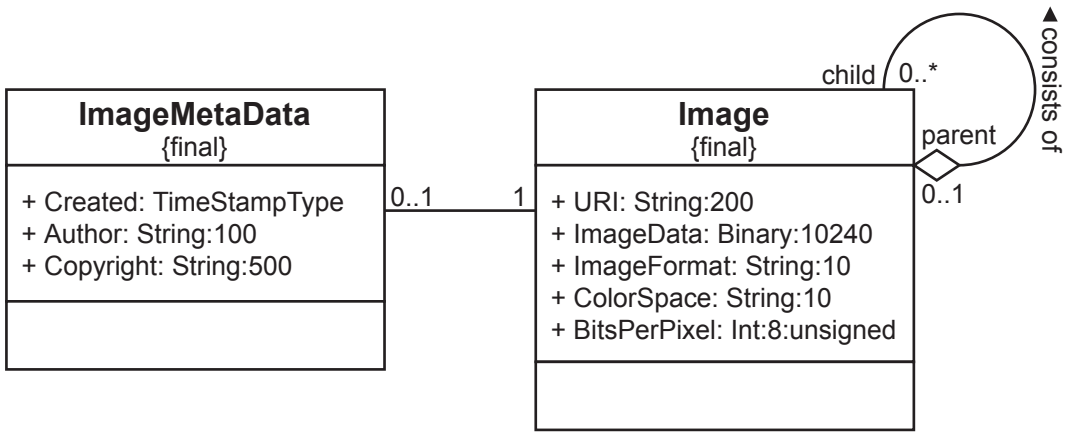
- Speicherort des Bildes unter Angabe einer URI
- Speicherformat, z.B. JPEG, TIFF, etc.
- das eigentliche Bild in Form von Binärdaten
- Farbkodierung, z.B. RGB, CMYK, etc.
- Anzahl Bits pro Farbkanal als ganzzahliger Wert

Weiterhin gibt es allgemeine Metainformationen zum Bild, die in der Klasse `ImageMetaData` gespeichert werden. Dies sind beispielsweise Angaben über Ursprung des Bildes, Zeit der Erstellung und letzten Bearbeitung sowie Angaben über den Autor und das Copyright. Diese Attribute sind optional und können, falls der Nutzer dafür keine Verwendung hat, aus der Klasse entfernt werden. Genauso kann der Nutzer der Klasse auch eigene Attribute hinzufügen. Folgende Attribute sind als Vorgabe bereits in der Klasse enthalten:

- Datum und Uhrzeit der Erstellung (Zeitstempel)
- Autor
- Copyright-Hinweise

Die Klasse Bild wird als `final` gekennzeichnet, von dieser Klasse kann nicht geerbt werden. Der Zweck der Klasse Bild ist es, allgemeine Informationen zum Bild zu speichern. Verschiedene Bildtypen können mit einem Attribut gekennzeichnet werden, verschiedene Eigenschaften können mittels Features dargestellt werden. Eine Vererbung ist an dieser Stelle daher nicht sinnvoll. Abbildung 4.4 zeigt die Klasse Bild (`Image`) mit den vordefinierten Attributen.

Abbildung 4.4: Die Klasse Image



4.3.4 Die Klasse Region

Ausgangspunkt für die Modellierung von Regionen bildet die abstrakte Klasse `Region`. Diese kann identifizierende Attribute, die mit `<<ID>>` gekennzeichnet werden, enthalten. Alle Referenzen und Assoziationen werden mit dieser Klasse hergestellt, wodurch Regionen sehr flexibel einsetzbar sind.

Gruppierung und Hierarchie

In Abschnitt 4.2.1 wurde bereits die Problematik der Gruppierung bzw. Hierarchie auf Regionen diskutiert. Im GDM wird diese wie die Strukturierung der Bilder umgesetzt. Es ist möglich Regionen mit Regionen in Aggregationsbeziehungen zu setzen. Somit kann eine Region aus mehreren Regionen bestehen. Da es sich nicht um eine Komponentenbeziehung handelt ist es nicht ausgeschlossen, dass dabei einzelne Regionen mehrfach in verschiedenen Aggregaten vorkommen können. Um dies zu vermeiden bzw. zu ermöglichen kann der Nutzer die Kardinalität verändern. Ist die Kardinalität 1:N, d.h. eine Region kann aus mehreren Regionen bestehen, mit jeder untergeordneten Region ist jedoch nur eine einzige Region assoziiert, so wird die Disjunktheit garantiert. Analog lässt die Kardinalität N:N Überlappungen zu. Als Vorgabe ist im Modell 1:N angegeben, der Nutzer kann dies allerdings ändern.

Konkrete Regionen

Konkrete Regionen werden nun von der abstrakten Klasse `Region` abgeleitet. Im Modell sind bereits vordefinierte Regionstypen enthalten, der Nutzer kann jedoch beliebig viele eigene hinzufügen. Folgende Regionen sind vordefiniert:

- Rechteck: Die Definition erfolgt über den linken oberen und rechten unteren Eckpunkt, sowie einen Winkel, um den das Rechteck gedreht ist. Die Drehung wird um den linken oberen Punkt entgegen dem Uhrzeigersinn in Grad angegeben.

- Ellipse: Eine Ellipse definiert sich durch Angabe des Mittelpunkts und die Längen der beiden Halbachsen. Weiterhin kann ein Winkel angegeben werden, der die Drehung der Ellipse um ihren Mittelpunkt angibt. Die Angabe der Drehung erfolgt wie beim Rechteck entgegen dem Uhrzeigersinn in Grad.

- Polygonzug: Ein geschlossener Polygonzug ist definiert als ein Vektor von Punkten. Enthält der Vektor k Punkte, so ist der jeweils i-te Punkt mit dem $((i + 1)$ modulo $k)$-ten verbunden.

- Maske: Wie bereits in Abschnitt 4.2.4 erklärt, können Masken auch als Definition für Regionen gelten. Da die Größe der Maske und die Bildgröße übereinstimmen müssen, die Bildgröße jedoch in jedem Bild verschieden sein kann, muss die Maske mittels einer dynamischen Matrix definiert werden: `MaskType:= Matrix:dyn:dyn{Bool};`

- Grid: Die Aufteilung eines Bildes in ein Gitter ist etwas komplizierter und lässt sich mit einer einzelnen Klasse nicht modellieren. Aus diesem Grund enthält das GDM eine Klasse `Grid`, die die Aufteilung des Bildes in ein Gitter festlegt. Als Kindklasse der `Region` steht die Klasse `GridCell`, die eine Zelle im Grid darstellt und eine Komponentenbeziehung mit `Grid` eingeht. Die Klasse `Grid` wird mit der Klasse `Image` in eine 1:N-Beziehung gebracht. Dies ist notwendig, da `Grid` keine Unterklasse von `Region` ist, und

sonst nicht in Bezug zu einem Bild stehen würde. Würde man `Grid` so definieren, so könnten Features auch einem Grid zugeordnet werden, was jedoch wenig Sinn macht, da diese Klasse lediglich die Segmentierung definiert, die konkreten Segmente jedoch der Klasse `GridCell` angehören. Da auch eine Unterteilung von Grid-Zellen in ein Grid möglich sein soll, ist eine zusätzliche Assoziation zwischen `GridCell` und `Grid` definiert. In Abbildung B.2 im Anhang ist das zugehörige UML-Klassendiagramm zu finden.

Abbildung 4.5: Die abstrakte Klasse Region und einige vordefinierte Regionentypen

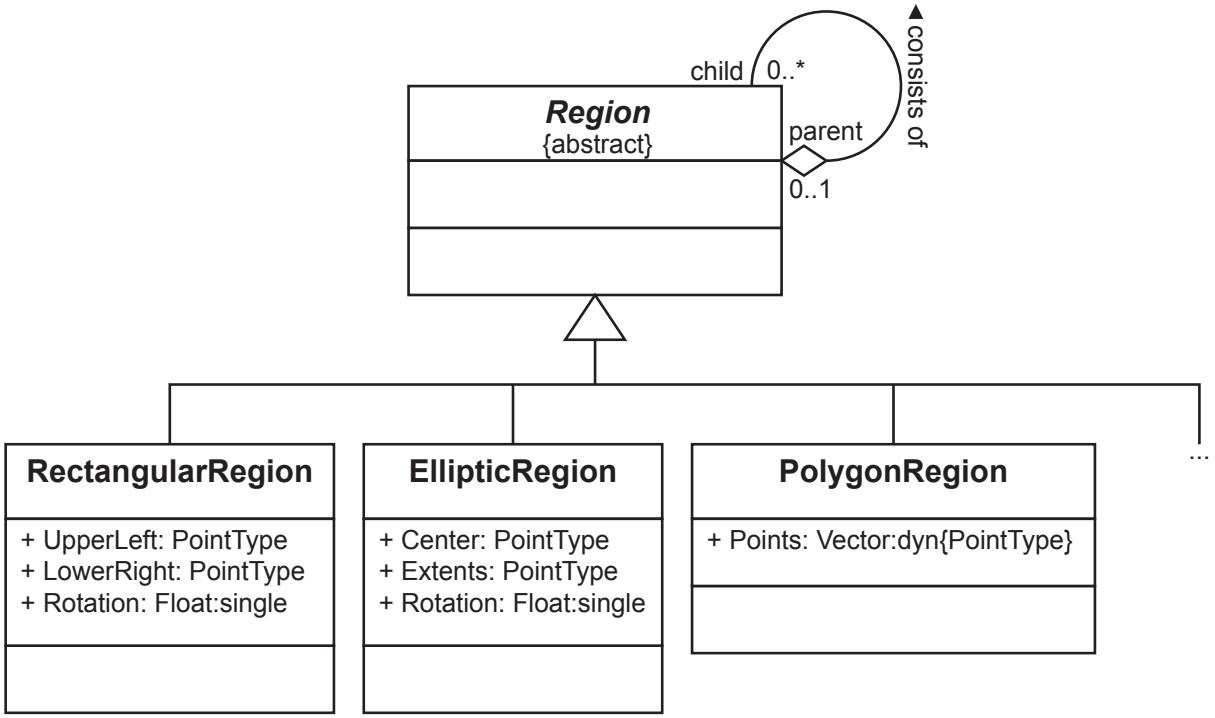
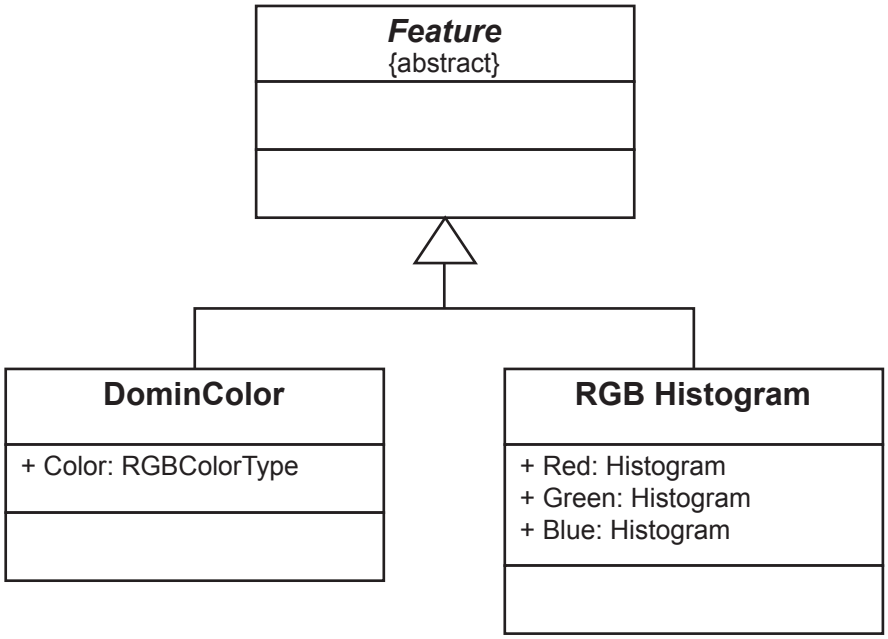


Abbildung 4.5 zeigt die Definition einer Region und einige der vordefinierten konkreten Regionen. Im Anhang B.3.2 ist eine Übersicht über alle vordefinierten Regionen zu finden.

4.3.5 Die Klasse Feature

Auch Features werden zunächst als abstrakte Klasse modelliert, da über diese Klasse die Beziehungen zu den Regionen hergestellt werden. Der Nutzer hat anschließend die Aufgabe von dieser Klasse zu erben und sich eigene Features abzuleiten. Definiert der Nutzer Schlüsselattribute, so sollte dies ebenfalls in der abstrakten Klasse Feature geschehen, damit jedes Feature durch Anfragen gefunden werden kann. Allerdings ist es auch möglich, nur in bestimmten Features Schlüssel zu definieren, in diesem Fall sind auch nur diese Features in Anfragen direkt ermittelbar. Abbildung 4.6 zeigt die abstrakte Feature-Klasse, mit einigen konkreten Features, in diesem Fall ein dominierender Farbwert und ein Histogramm.

Abbildung 4.6: Die abstrakte Klasse Feature mit zwei konkreten Features



4.3.6 Zusammenhänge

Ausgangspunkt der Betrachtung ist die Klasse Bild. Die Zuordnung der Metadaten zum Bild wurde bereits in Abbildung 4.4 gezeigt. Da unsere Sicht auf das Bild vorschreibt, dass auf dem Bild Regionen identifiziert werden, deren Eigenschaften anschließend als Features zu modellieren sind (siehe Abschnitt 4.1), ist der nächste Schritt die Verknüpfung des Bildes mit den Regionen.

Die Beziehung zwischen Bildern und Regionen ist eine Aggregation, da Bilder aus Regionen bestehen, die semantisch nicht mit Bildern gleichgestellt, sondern untergeordnet sind. Da es möglich sein soll Bilder ohne zugehörige Regionen zu definieren, kann an dieser Stelle keine Komposition verwendet werden, da dann die Kardinalität 0..* nicht zulässig gewesen wäre.

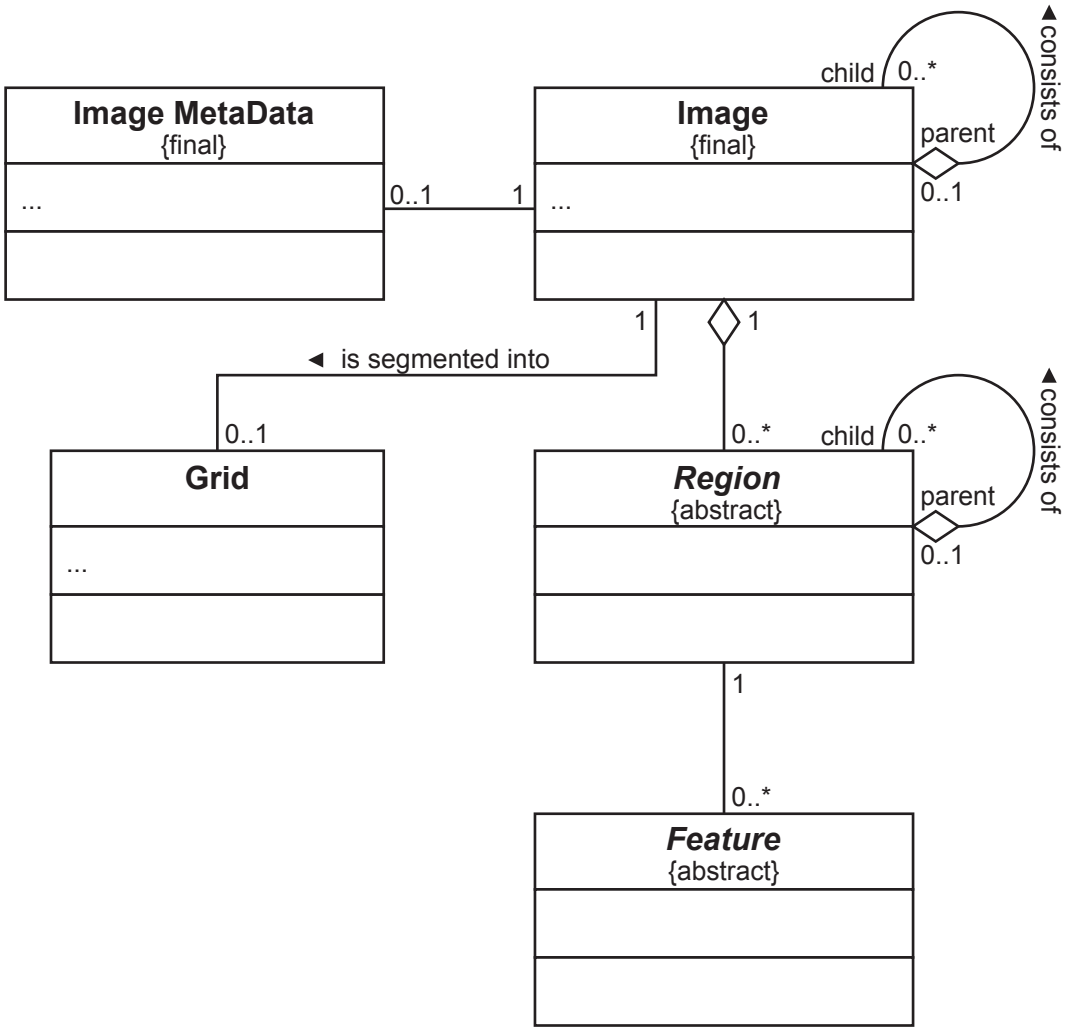
Um Regionen hierarchisch anzuordnen, ist eine Aggregation der Klasse Region mit sich selbst möglich. Dies ist bereits in Abbildung 4.5 dargestellt worden. Auf diesem Weg ist es möglich, Regionen in Beziehung zueinander zu bringen. Dabei ist die Multiplizität jeweils ein Intervall welches bei 0 beginnt, da nicht jede Region Teil einer anderen Region sein kann, z.B. die Wurzel der Regionshierarchie, genauso wie nicht jede Region andere Regionen beinhalten kann, wie z.B. die Blätter der Regionshierarchie. Die andere Seite des Intervalls kann vom Nutzer geändert werden, die Grundeinstellung ist folgende: Die `parent`-Region kann beliebig viele `child`-Regionen haben, während umgekehrt jede `child`-Region höchstens einer `parent`-Region zugeordnet ist.

Jeder Region kann nun eine Menge von Features zugeordnet werden. Die Verknüpfung erfolgt mittels einer Assoziation. Dabei ist die Multiplizität wiefolgt: Eine Region kann beliebig vielen

Features zugeordnet werden (Multiplizität 1..*), ein Feature gehört jedoch zu genau einer Region (Multiplizität 1). Es ist vorstellbar, dass in einigen Anwendung ein Feature auch zu mehreren Regionen gehören kann, z.B. falls sich zwei Regionen überlappen und dadurch eine gemeinsame Eigenschaft besitzen. In diesem Fall wäre eine m-n-Beziehung sinnvoll. Dies kann allerdings auch durch die Gruppierung von Regionen realisiert werden, indem alle betroffenen Regionen in eine übergreifende Region gruppiert werden und dieser dann das Feature zugeordnet wird.

Abbildung 4.7 zeigt alle vorgegebenen Assoziationen. Die Beziehungen zwischen der Klasse `Grid` und der hier nicht dargestellten Klasse `GridCell` sind Abbildung B.2 im Anhang zu entnehmen.

Abbildung 4.7: Beziehungen zwischen den einzelnen Klassen



4.3.7 Erweiterbarkeit

Da es sich beim GDM um ein Framework handelt, ist eine gute Erweiterbarkeit von vornherein gegeben. Ein Framework stellt dem Benutzer eine Vorlage zur Verfügung, von der ausgehend er sein eigenes Problem beschreiben kann.

Mit dem GDM werden einige Klassen und Beziehungen bereitgestellt, die bereits grundlegende Zusammenhänge vorgeben. Der Nutzer hat die Möglichkeit eigene Klassen zu definieren, und er kann beliebige Beziehungen erstellen. Er kann außerdem an beliebiger Stelle Operationen hinzufügen. Mit dem Typsystem ist eine sehr flexible Typisierung der Attribute möglich. Das Typsystem kann gegebenenfalls erweitert werden, was jedoch nur durch Änderung der zugrunde liegenden Grammatik möglich ist, und somit für den eigentlichen Nutzer dieses Systems nicht zugänglich ist, da dafür implementierungsspezifische Kenntnisse erforderlich sind. Außerdem erfordert dieser Schritt die Anpassung der Transformation, da sonst die Erweiterung nicht nutzbar wäre.

Der Nutzer entwirft und arbeitet mit seinem Modell mit einem UML-Tool. Im Prototypen zu dieser Arbeit ist dieses Tool Rational Rose. Dabei werden einige Einschränkungen getroffen, die notwendig sind um die Transformierbarkeit des Systems zu gewährleisten. Dem Nutzer steht also UML in einer etwas eingschränkten Form zur Verfügung.

4.3.8 UML-Einschränkungen

UML als zugrunde liegende Sprache ist sehr flexibel und ermöglicht dem Nutzer sehr vielfältige Entwurfsmöglichkeiten. Allerdings kann nicht jede dieser Möglichkeiten bei der Transformation in eine konkrete Zielplattform umgesetzt werden. Um zu vermeiden, dass die Transformation zu komplex oder stellenweise nicht möglich ist, sind folgende UML-Entwurfsmöglichkeiten nicht zugelassen:

- Kardinalitäten bei Attributen: Multiplizität kann durch die Wahl eines entsprechenden Datentyps erreicht werden.
- Klassenattribute: Da im GDM nur sehr wenige Klassen enthalten sind, ist es nicht notwendig Klassenattribute zu definieren. Ein Klassenattribut, beispielsweise in einer Region, kann als Attribut in den Metadaten des Bildes modelliert werden. Außerdem kann das Konzept der Klassenattribute in DB2 und im ICM nur simuliert werden, entweder durch eine zusätzliche Tabelle, wodurch Anfragen komplizierter werden, oder mittels Trigger, die Änderungen eines Tupels auf die gesamte Tabelle propagieren, was jedoch einen enormen Update-Aufwand und hohe Redundanz bei der Speicherung zur Folge hätte.
- abgeleitete Beziehungen: Da in einem Datenmodell, dessen Instanzen durch einen Algorithmus erzeugt werden, ohnehin jede Beziehung praktisch aus dem Bild abgeleitet ist, ist es nicht sinnvoll abgeleitete Beziehungen separat anzugeben. Stattdessen sollte die Beziehung regulär spezifiziert werden. Gegebenenfalls kann ein Algorithmus in Form einer

Operation angegeben werden, der diese Beziehung ermittelt und speichert. Insbesondere da abgeleitete Beziehungen in DB2 nicht abgebildet werden können, sind abgeleitete Beziehungen im GDM nicht erlaubt. In einer Datenbank wäre eine berechnete Beziehung eine Selektion, die stets neu zu berechnen ist, und das würde in der Datenbank zu einem hohen Anfrageaufkommen führen. Um dies zu umgehen, müsste die Beziehung gespeichert werden, nur dann kann auch von vornherein eine nicht abgeleitete Assoziation verwendet werden.

- Mehrfachvererbung: Mehrfachvererbung ist weder in SQL/99, noch in DB2 vorgesehen. Es wäre demnach notwendig, die Vererbungshierarchie und entsprechende DDL- und DML-Methoden durch einen eigenen Client zu realisieren, der die Mehrfachvererbung auf relationale Tabellen abbildet. Auch werden in [Heu97] verschiedene Probleme mit Mehrfachvererbung genannt, beispielsweise Namenskonflikte und Probleme mit der Substituierbarkeit von Super- und Subtypen, die zu lösen wären. Da Mehrfachvererbung für die Realisierung des eNoteHistory-Datenmodells (siehe 4.3.9) nicht benötigt wird, und um zusätzlichen Mehraufwand bei der Transformation zu vermeiden, ist Mehrfachvererbung im GDM nicht zugelassen.

UML bietet umfangreiche Möglichkeiten Beziehungen (Assoziationen) zu definieren. Folgende Konzepte sind bei der Modellierung von Assoziationen erlaubt:

- Benennung von Beziehungen
- Angabe einer Navigationsrichtung
- beliebige Kardinalitäten
- Attribute der Beziehung
- Mehrstellige Assoziationen

4.3.9 Das eNoteHistory-Datenmodell

Beschreibung des Modells

Ausgangspunkt für das eNoteHistory-Datenmodell sind die eingescannten Bilder der Notenhandschriften. Auf diese Bilder kann ein Extraktionsprozess angewendet werden, der verschiedene Features aus dem Bild extrahiert. Diese gilt es anschließend zu speichern.

Zum Bild gehören werden folgende Metadaten gespeichert:

- Library Code: Kürzel der Bibliothek, aus der das Notenblatt stammt
- Shelf Mark: Signatur der Bibliothek, welche das Werk identifiziert
- Section Number: Nummer des Teils des Werkes

- Page Number: Seite des Teils, deren Bild hier vorliegt

Diese haben identifizierenden Charakter, werden daher mit dem Stereotyp `<<ID>>` gekennzeichnet. Die Kardinalität der Beziehung zwischen `Image` und `ImageMetadata` wird auf 1:1 gesetzt, damit die Metadaten auch jedes Bild identifizieren.

Folgende physische Eigenschaften werden für das Bild gespeichert:

- Größe: Höhe und Breite in Pixel
- Created, Updated: Datum der ersten und letzten Speicherung des Bildes
- die Binärdaten des Bildes

Dem Bild werden nun eine oder mehrere ROIs (regions of interest) zugeordnet. Eine ROI ist dabei ein Rahmen um ein oder mehrere Notensysteme, wobei jedes Notensystem jeweils von einem Schreiber erstellt wurde.

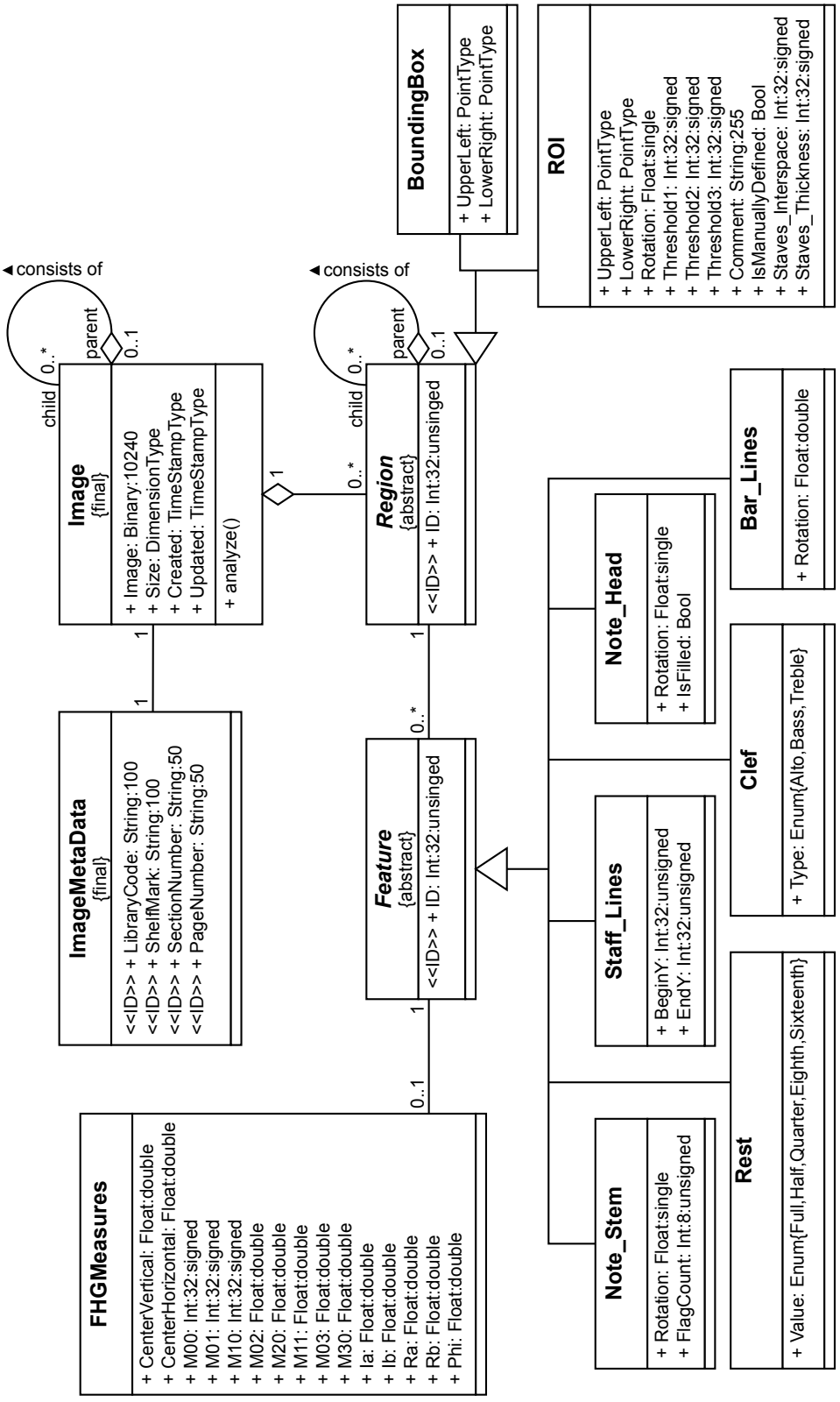
Eine ROI ist eine rechteckige Region, bei der linke obere und rechte untere Ecke gespeichert werden, sowie ein Winkel, der die Drehung dieses Rechtecks angibt. Folgende Eigenschaften werden dieser Region in Form von Attributen zugeordnet:

- Threshold 1/2/3: Schwellwerte für die Bildverarbeitung
- ein Kommentar
- die Angabe, ob es sich bei dieser ROI um eine manuell erstellte Region handelt
- Informationen über den durchschnittlichen Abstand und die Dicke der enthaltenen Linien des Notensystems

Jeder ROI sind weitere Regionen untergeordnet. Diese Regionen umgeben musikalische Objekte, wie beispielsweise Noten oder Taktstriche. Genauer werden folgende Objekte betrachtet, die die Features darstellen:

- Staff Lines: Notensysteme
- Bar Lines: Taktstriche
- Clef (Bass,Alto,Treble): Notenschlüssel
- Rest (Whole,Half,Quarter,Eighth,Sixteenth): Pausen
- Note (Whole,Half,Quarter,Eighth,Sixteenth): Noten

Abbildung 4.8: Das generische Datenmodell für das eNoteHistory-Projekt



In Abbildung 4.8 ist das generische Datenmodell für das eNoteHistory-Projekt abgebildet. Es werden außer einigen vordefinierten Datentypen (siehe Anhang B.2) keine weiteren eigenen Datentypen verwendet.

Nachdem alle Klassen modelliert sind, fehlt nun noch eine Operation zur Extraktion der Features. Der Name dieser Operation ist `analyze()`, und sie wird an die Klasse `Image` gebunden. Der Grund dafür ist, dass nach Einfügen eines Bildes in die Datenbank die Operation `analyze()`, ausgehend von den digitalen Bilddaten, Regionen und Features erzeugt. Die Operation hat weder Parameter noch einen Rückgabetyp, denn das Bild, dem diese Operation zugeordnet ist, ist der einzige, implizite Eingabeparameter.

Die Implementierung dieser Methode ist der bereits in Abschnitt 2.3.2 beschriebene Algorithmus, der automatisch Notensysteme und musikalische Objekte (Notenköpfe, Notenhälse und Taktstriche) erkennt.

Kapitel 5

Transformation des generischen Modells

5.1 Abbildung des GDM in das ORDBMS DB2

5.1.1 Objektrelationale Möglichkeiten

Im Vergleich zum relationalen Datenmodell bieten objektrelationale Systeme verschiedene Vorteile. Der erste, und gravierendste, ist der Wegfall der ersten Normalform (1NF). Die erste Normalform besagt, dass keine komplexen (kollektionswertigen) Attributwerte innerhalb von Tupeln auftreten. Im objektrelationalen Standard wird diese Normalform nicht gefordert, wodurch es möglich wird, komplexe Datenstrukturen zu speichern. Leider ist in SQL/99 der einzige komplexe Datentyp, neben dem `TYPE`-/`TABLE`-Konstruktor für Strukturen und dem nicht OO-kompatiblen Tupelkonstruktor `ROW`, der Typ `ARRAY`, der einen Vektor variabler Größe darstellt. Komplexe Datentypen können vorab definiert werden, wobei strenge Typisierung, Einkapselung, Polymorphismus und andere objektorientierte Konzepte zumindest in einer einfachen Form berücksichtigt werden. Die Typdefinition in SQL/99 kann also mit der Strukturdefinition, d.h. der Definition der Attribute, verglichen werden. Eine typisierte Tabelle in der Datenbank kann dann als Extension einer Klasse angesehen werden.

Die Objektidentität solcher Objekte in Tupelform wird ebenfalls vom System verwaltet, und kann vom Nutzer nicht geändert werden. Bei der Vererbung übernehmen Untertypen Struktur und Verhalten des Obertyps, bei Tabellen werden ebenfalls Trigger, Bedingungen (constraints) und Methoden vererbt. Beziehungen zwischen Objekten können über die OIDs hergestellt werden, da diese Referenz-Charakter haben.

Eine Übersicht über die objektrelationalen Fähigkeiten von SQL/99 ist in [Tue03] zu finden.

5.1.2 Exkurs: SQL/MM

Bevor die Transformation nach SQL/99 vorgestellt wird, betrachten wir kurz den SQL/MM-Standard. Dieser bietet spezielle Möglichkeiten an, um Features und Methoden zu definieren

und zu speichern. Mit Hilfe dieses Standards wäre eine Integration einer vollständigen Retrieval-Anwendung, bei der neben der Speicherung auch die Extraktion der Features eine wichtige Rolle spielt, gut möglich.

Der SQL/MM-Standard [ISO03] bietet einen Datentyp `SI_StillImage` an, der zur Speicherung von Bildern dient. Dieser hat folgende Struktur:

- eine digitale Repräsentation eines Bildes, oder ein Verweis darauf
- Formatangaben
- physische Charakteristika

Der Standard definiert bereits einige Methoden auf diesem Datentyp, nämlich:

- einen Konstruktor
- eine Methode um das Bild oder eine Referenz darauf sowie enthaltene Bildcharakteristika zu erhalten
- Methoden zur Skalierung, Änderung der Bildgröße und Drehung des Bildes
- Methoden zur Ableitung metrischer Werte zur Charaktierisierung des Bildinhaltes

Außerdem stellt der Standard bereits Datentypen zur Speicherung von Features zur Verfügung, zu denen ebenfalls Methoden zur Herleitung vorgesehen sind. Folgende Features sind durch den Standard vorgesehen:

- durchschnittlicher Farbwert (average color)
- Farbhistogramme
- lokal dominierende Farbe
- Texturinformationen

Aufbauend auf diesen Vorgaben ist es möglich, eigene Features zu definieren, und ähnlich wie bei der nachfolgenden Transformation in die Datenbank zu integrieren. Extraktionsprozesse können mittels Methoden an die Features gebunden werden.

5.1.3 Abbildung von Klassen

Da in SQL/99 die Analogie von Klassen zu Typen/Tabellen vorgesehen ist, werden Klassen in Tabellen gespeichert. Der Name der Klasse wird Name der Tabelle. Aus den Attributen der Klasse wird ein komplexer Typ erstellt, welcher die Abbildung der Klasse darstellt. Ist die Klasse instanziierbar, so wird eine typisierte Tabelle erstellt, welche die Extension der Klasse bildet.

Da es in DB2 aus Sicht der Performance günstig ist, Bilder, sofern sie als BLOB gespeichert werden, in einer eigenen Tabelle und in einem eigenen Schema zu speichern, werden bei der Abbildung von Binären Daten (`binary`) die eigentlichen Daten getrennt von den restlichen Attributen gespeichert.

5.1.4 Abbildung von Attributen

Attribute der Klasse werden auf Attribute der Relation abgebildet. Dabei wird für jedes Attribut der jeweilige Datentyp in eine SQL/99-konforme Typdefinition umgewandelt und auf das jeweilige Attribut angewendet. Tabelle 5.2 zeigt die Abbildung der Datentypen des GDM auf Typen aus SQL/99 (siehe auch [Tue03]).

Da Klassenattribute und Multiplizitäten bei Attributen nicht zugelassen sind (siehe 4.3.8) ist es lediglich notwendig „einfache“ Attribute zu betrachten. Die Sichtbarkeit der Attribute wird bei der Transformation derzeit noch nicht berücksichtigt. Es wäre jedoch möglich, nicht sichtbare Attribute in einer eigenen Tabelle zu speichern, und mit entsprechenden Zugriffsrechten vor dem Nutzer zu isolieren. Jedoch wäre dann jedes Objekt im GDM in mehrere Tabellen verteilt gespeichert. Das Zusammensetzen der Objekte müsste durch spezielle Methoden vorgenommen werden, die erst noch implementiert werden müssen.

Eigene Transformationstabellen

Wie bereits in Abschnitt 4.3.1 erwähnt wurde, soll es für den Fall, dass der Nutzer sich bereits auf eine Zielplattform festgelegt hat, möglich sein die Datentypen der Zielplattform zu nutzen. Dadurch wird der impedance mismatch bezüglich der Typsysteme, der durch das GDM-eigene Typsystem zwangsweise entsteht, vermieden. Allerdings ist die Transformation in diesem Fall bereits auf eine konkrete Zielplattform festgelegt. Das Typsystem des GDM und Transformationstabellen können parallel verwendet werden, da sie unabhängig voneinander exisitieren und dementsprechend in der Transformation auch getrennt behandelt werden.

Um plattformspezifische Typen zu erstellen, erstellt der Nutzer eine Liste, die den UML-Typnamen enthält, den Namen des Typs auf der Zielplattform, die Anzahl der Parameter des Typs, und eine optionale Beschreibung.

Ein Eintrag für den DB2-Datentyp `DECIMAL` könnte dann wie in Tabelle 5.1 dargestellt aussehen. Das bedeutet, dass ein im UML-Diagramm vorkommender Typ mit Namen „DB2_Decimal“, wobei der Präfix „DB2_“ der Kennzeichnung als externen, plattformspezifischen Typ dient, auf

Tabelle 5.1: Beispiel für den Eintrag einer eigenen Transformationstabelle

UML-Name	**ZP-Typ**	**Parameter**	**Beschreibung**
⋮			
DB2_Decimal	DECIMAL	0,1,2	packed decimal number, DECIMAL(precision,scale), parameter defaults are (5,0) if left out
⋮			

den Typ `DECIMAL` abgebildet wird, wobei keiner, ein oder zwei Parameter übergeben werden müssen.

Im Anhang C sind die vollständigen Transformationstabellen für DB2 und ICM zu finden.

Schlüsselattribute

Falls im Modell Attribute mit dem Stereotyp `<<ID>>` gekennzeichnet sind, so können diese Attribute direkt als Schlüssel übernommen werden. Da alle Objekte mit einer Objekt-ID (OID) versehen werden, ist es technisch nicht notwendig überhaupt Schlüssel zu definieren. Allerdings wird dann die Suche nach Objekten ineffektiv, da das Datenbanksystem Attribute mit Schlüsseleigenschaft durch Indizierung besser finden kann. Es bleibt dennoch Aufgabe des Nutzers, identifizierende Attribute selbst zu kennzeichnen, da dies aufgrund fehlender Daten nicht automatisch, z.B. mit Hilfe von Statistiken über den Datenbankinhalt, erfolgen kann.

Abbildung der Typen auf SQL/99

Zunächst betrachten wir die Abbildung des Typsystems auf Datentypen von SQL/99. Es wird schnell deutlich, dass das vollständige Typsystem nicht abbildbar ist, da in SQL/99 einige Typen nicht oder nur auf Umwegen darstellbar sind.

So gibt es beispielsweise keinen 64-bit Integer-Typen, auch fehlt die Möglichkeit, Integer mit Vorzeichen zu behaften. Eine `CHECK`-Klausel kann die Einhaltung des Zahlenbereichs zusichern, so dass vorzeichenlose Integer auf den nächsthöheren, vorzeichenbehafteten Zahlenbereich abgebildet werden können.

Vektoren können ohne weiteres auf `ARRAY` abgebildet werden, da dieses bereits eine dynamische Liste implementiert. Vektoren mit fester Länge können alternativ mit Tabellen realisiert werden. Dazu wird eine Relation mit einer ID-Spalte und den Feldern des Vektors angelegt. Eine Fremdschlüsselreferenz erledigt die Zuordnung zum entsprechenden Objekt.

Problematisch sind jedoch die komplexen Datentypen Menge, Multimenge und Matrix. Mengen und Multimengen können in `ARRAY`s gespeichert werden, jedoch muss dann die Integrität der

Typen überprüft werden, z.B. bei Mengen, damit doppelte Elemente nicht vorkommen. Es ist außerdem notwendig Operationen zu definieren, da sonst (Multi-)Mengen zwar gespeichert, aber nur von einem speziellen Client auch genutzt werden können.

Die Tabelle 5.2 zeigt eine Übersicht über alle GDM-Datentypen und deren Umsetzung in SQL/99.

Besonderheiten von DB2

Als ganzzahligen Typ unterstützt DB2 zusätzlich `BIGINT`, und kann damit sowohl vorzeichenbehaftete 64-bit Integer, als auch vorzeichenlose 32-bit Integer als Attributtypen enthalten. Außerdem unterstützt DB2 den Typ `DATALINK`, der im SQL/MED-Standard definiert ist und externe Dateireferenzen erlaubt, [Mel02].

Der Datentyp `ARRAY` ist nicht implementiert. Es ist möglich diesen Datentyp mit dem Typ LOB und den entsprechenden Operationen (`get`, `set`, `size`, ...) zu simulieren. Diese Einschränkung betrifft Mengen, Multimengen, Vektoren und Matrizen. Eine Umsetzung als LOB würde allerdings bedeuten, dass die Daten nicht ohne weiteres indiziert werden können.

Für Vektoren und Matrizen mit fester Länge kann auch eine Tabelle zur Umsetzung verwendet werden, indem eine Spalte je Feld angelegt wird. Dies kann bei Matrizen jedoch sehr schnell unübersichtlich werden, da die mehrdimensionale Struktur der Matrix auf eine Dimension herunter gebrochen werden muss. Da in dieser Form eine Indizierung wenig Sinn macht, ist eine Umsetzung der Matrix als LOB besser.

Mengen und Multimengen können alternativ mit Tabellen simuliert werden. Dazu wird für jede Menge bzw. Multimenge einer Klasse eine Tabelle angelegt, die eine Spalte enthält, in der die Elemente enthalten sind. Für Multimengen wird außerdem eine Spalte angelegt, die die Kardinalität eines Elements enthält. Allerdings können so keine Fremdschlüssel zu Referenzierung verwendet werden, da pro Spalte nur auf eine Tabelle referenziert werden kann. Also müssen alle Mengen bzw. Multimengen in je einer Tabelle gespeichert werden. Zusätzlich wäre eine ID notwendig, die dann zusammen mit dem Element Schlüssel der Tabelle wird. Dieses Attribut kann dann als Fremdschlüssel verwendet werden.

Auch Aufzählungen werden von DB2 nicht unterstützt. Da die `CREATE DOMAIN`-Anweisung ebenfalls nicht implementiert ist, muss eine Simulation mittels Tabelle erfolgen. Dafür wird eine Tabelle mit den Werten der Aufzählung angelegt. Diese Tabelle kann mit Zugriffsrechten gegen Veränderung durch den Nutzer geschützt werden. Da die Einträge eindeutig sind, sind sie gleichzeitig Schlüssel. Das transformierte Attribut wird mittels einer Fremdschlüsselbeziehung mit den Werten der Aufzählung in Verbindung gebracht.

Außerdem ist kein Datentyp `BOOLEAN` in DB2 vorhanden, dieser kann jedoch mit `SMALLINT` und einer entsprechenden `CHECK`-Klausel simuliert werden.

Tabelle 5.3 zeigt die Veränderungen gegenüber Tabelle 5.2, die durch DB2-Besonderheiten entstehen.

Tabelle 5.2: Transformation der Datentypen vom GDM nach SQL/99

GDM-Datentyp	SQL/99-Datentyp
Ganzzahlige Typen (`Int`)	`SMALLINT` (16bit), `INTEGER` (32bit) Da diese Datentypen stets vorzeichenbehaftet sind, ist es notwendig die Nichtnegativitätseigenschaft mittels einer `CHECK`-Klausel zu prüfen. 64-bit Integer-Zahlen werden in SQL/99 nicht unterstützt (vgl.[Tue03]), somit ist eine Transformation des Datentyps `Int:64:...` nicht möglich. 8-bit-Integer werden generell auf `SMALLINT` abgebildet.
Fließkommazahlen (`Float`)	`REAL` (single precision), `DOUBLE` (double precision)
Wahrheitswerte (`Bool`)	`BOOLEAN`
Zeichen (`Char`,`String`)	`CHAR` bzw. `VARCHAR`, wobei einzelne Zeichen auf `CHAR(1)` abgebildet werden. Die Entscheidung, ob `CHAR` oder `VARCHAR` verwendet wird, erfolgt mit Hilfe der maximalen Länge: ab 30 Zeichen wird `VARCHAR` verwendet.
Binärdaten (`Binary`)	`BLOB`
Referenz auf Datei (`FileRef`)	In SQL/99 ist dieser Typ nicht umsetzbar, jedoch ist der Datentyp `DATALINK` Teil des SQL/MED-Standards (Management of External Data)
Aufzählung (`Enum`)	mittels `CREATE DOMAIN` kann eine Domäne erstellt werden, die die entsprechenden Elemente beinhaltet: `CREATE DOMAIN` *enumName* `AS` *typ* `CHECK(VALUE IN(`e_1,...,e_n`))`
Menge (`Set`)	(wird nicht unterstützt)
Multimenge (`Bag`)	(wird nicht unterstützt)
Vektor (`Vector`)	`ARRAY` Als Elemente dürfen jedoch keine ARRAYs verwendet werden, erst ab SQL/2003 ist eine Schachtelung möglich.
Matrix (`Matrix`)	(wird nicht unterstützt)
Struktur (`Struct`)	strukturierter Typ, Erzeugung mittels `CREATE TYPE`

Tabelle 5.3: Transformation der Datentypen vom GDM in die DB2-Umsetzung von SQL/99

GDM-Datentyp	SQL/99-Datentyp (DB2)
Ganzzahlige Typen (`Int`)	`SMALLINT` (16bit), `INTEGER` (32bit), `BIGINT` (64bit); Da diese Datentypen stets vorzeichenbehaftet sind, ist es notwendig die Nichtnegativitätseigenschaft mittels einer `CHECK`-Klausel zu prüfen.
Wahrheitswerte (`Bool`)	da DB2 keinen Booleschen Typ enthält wird stattdessen auf `SMALLINT` abgebildet, wobei der Wert 0 false bedeutet, und 1 true. Eine `CHECK`-Klausel wird dafür sorgen das nur 0 und 1 als Werte für dieses Attribut zugelassen sind: `CHECK` *col* `BETWEEN 0 AND 1`
Referenz auf Datei (`FileRef`)	`DATALINK`
Aufzählung (`Enum`)	Zur Umsetzung dieses Typs wird ein `CHAR`-Attribut definiert und mit einer `CHECK`-Klausel die den Wertebereich überprüft.
Menge (`Set`)	Simulation mittels LOB und entsprechenden Methoden
Multimenge (`Bag`)	Simulation mittels LOB und entsprechenden Methoden
Vektor (`Vector`)	bei Vektoren fester Länge erfolgt eine Simulation mittels einer Tabelle und einem Fremdschlüssel, ansonsten als LOB
Matrix (`Matrix`)	Simulation mittels LOB und entsprechenden Methoden

LOBs als Ersatz für komplexe Datentypen

In Tabelle 5.3 wird für die Transformation einiger komplexer Datentypen die Verwendung von LOBs vorgesehen. Wie dies genau in DB2 realisiert werden kann, sei am Beispiel einer Menge von Zeichenketten erläutert, die im GDM mit `Set{String:255}` definiert wird.

Zunächst wird ein neuer einzigartiger Typ für Mengen angelegt. Mengenwertige Attribute können diesen Typ anschließend als ihren Attributtyp nutzen:

```
CREATE DISTINCT TYPE GDM_SET AS BLOB;
```

Die eigentlichen Operationen können mit Hilfe von Java erstellt werden. In diesem Beispiel betrachten wir drei Java-Methoden. Die Quellen für die ersten beiden Methoden sind im Anhang D aufgeführt:

- `void newSet()` - erstellt eine leere Menge
- `Blob add(String element, Blob set)` - fügt der Menge `set` ein Element hinzu und gibt eine neue Menge zurück
- `Blob delete(String element, Blob set)` - löscht ein Element aus der Menge `set` und gibt eine neue Menge zurück

Diese Methoden können nun wiefolgt in die Datenbank integriert werden, die Registrierung der anderen Methoden verläuft anschließend analog:

```
CALL sqlj.install_jar('setRoutines.jar','mySetRoutines');
CREATE FUNCTION set_new()
RETURNS GDM_SET CAST FROM BLOB(100k)
LANGUAGE JAVA
PARAMETER STYLE DB2GENERAL
NO SQL
FENCED
DETERMINISTIC
NO EXTERNAL ACTION
EXTERNAL NAME 'mySetRoutines:SetRoutines!newSet';
```

Der Datentyp kann nun genutzt werden. Folgendes Beispiel demonstriert das:

```
CREATE TABLE SetTest (
ID INTEGER NOT NULL PRIMARY KEY,
TheSet GDM_SET);

INSERT INTO SetTest VALUES
(1, set_new()),
(2, set_add('Ein Element',set_new()));
```

5.1.5 Abbildung von Operationen

In DB2 gibt es mehrere Möglichkeiten die Methoden der Objekte zu implementieren. In [IBM02d] werden folgende Möglichkeiten zur Definition von Routinen genannt:

- **Stored Procedures (SP)**
  SPs sind Prozeduren, die in prozeduralem SQL oder einer externen Sprache geschrieben sind und innerhalb der Datenbank ausgeführt werden. SPs müssen explizit aufgerufen werden, können also nicht in Anfragen verwendet werden. Die Ergebnisse der Stored Procedure können demnach auch nicht direkt verwendet werden. Sie dienen der Implementierung der Business-Logik und können beispielsweise komplexe, aber stets identische Update-Aufgaben durchführen.

- **User Defined Functions (UDF)**
  UDFs sind Funktionen, die aus einem oder mehreren Eingabewerten einen Ausgabewert berechnen. Dabei werden zwei verschiedene Typen unterschieden. Zum einen gibt es skalare UDFs, die aus den Eingabeparametern einen Ausgabewert berechnen. Neben den skalaren existieren Table UDFs, deren Rückgabewert eine Tabelle ist. Diese UDFs können nur in der FROM-Klausel einer Anfrage verwendet werden. UDFs können keine Daten mit SQL verändern.

- **Methoden**
  Methoden werden an einen strukturierten Datentyp gebunden, verhalten sich aber ansonsten wie skalare UDFs. Sie eignen sich somit nur zur Berechnung von Werten aus Eingabeparametern oder gespeicherten Daten, können jedoch ebenfalls keine Daten verändern.

5.1.6 Abbildung von Beziehungen

Ähnlich wie bei der Transformierung von ERM in relationale Schemata ([HS00]) gibt es, je nach Art, mehrere Möglichkeiten Beziehungen umzusetzen. Grundsätzlich werden Beziehungen auf relationale Tabellen abgebildet. Die jeweiligen Schlüsselattribute der beteiligten Klassen werden zusammen Schlüssel dieser Relation. Hat eine Beziehung keine Attribute, so kann unter Umständen auch auf diese Zwischentabelle verzichtet werden. Hat eine Seite der Beziehung die Kardinalität 1, so können weitere Vereinfachungen vorgenommen werden. Handelt es sich sogar um eine 1:1-Beziehung, so ist keine eigene Tabelle für diese Beziehung notwendig. So können beispielsweise die Klassen `Image` und `ImageMetadata` des eNoteHistory-Modells (siehe Abbildung 4.8) in der selben Tabelle gespeichert werden.

Assoziationen

Es gibt in UML mehrere Möglichkeiten Assoziationen zu definieren. Um die Transformation zu vereinfachen, sind nicht alle Modellierungskonzepte für Assoziationen erlaubt (siehe 4.3.8). So sind z.B. qualifizierte Assoziationen[1] nicht zugelassen, da die damit möglichen Beziehungen nicht direkt, sondern nur auf Umwegen abbildbar sind. Derartige Beziehung sind zwar in der Anwendungsmodellierung denkbar, jedoch für die Datenmodellierung nicht unbedingt üblich. Deshalb wird zugunsten der Komplexität der Transformation auf dieses Konstrukt verzichtet.

Prinzipiell wird jeder Assoziation eine eigene Tabelle zugewiesen. Dies ist jedoch nicht immer notwendig. Folgende Möglichkeiten für eine Assoziation zwischen zwei Klassen A und B sind hierbei zu unterscheiden:

- 1:1
  Bei dieser Assoziation ist jedem Objekt aus A genau ein Objekt aus B zugeordnet. Zur Abbildung dieser Beziehung ist eine eigene Tabelle nicht notwendig, die Attribute von A, B und gegebenenfalls der zugeordneten Assoziationsklasse, werden in der selben Tabelle gespeichert, die Tabellen wurden verschmolzen.

[1] Mit qualifizierten Assoziationen zwischen zwei Klassen A und B mit einer x:N-Beziehung werden der Klasse A (N-Seite) Attribute zugeordnet, die eine Gruppierung der Zuordnung zur Klasse B (x-Seite) ermöglichen. Diese Attribute gehören eigentlich zur Klasse B, werden aber auf Seite der Klasse A notiert und müssen auch dort gespeichert werden. Würde dieses Konstrukt auf eine Datenbank abgebildet werden, so verliert man entweder die Qualifizierung der Beziehung, oder benötigt neben einer eigenen Tabelle zur Speicherung auch einen Mechanismus, beispielsweise spezielle Anfragen, um qualifizierende Attribute als solche zu nutzen.

- 1:N
  Bei dieser Beziehung kann ein Objekt der Klasse A mehreren Objekten der Klasse B zugeordnet sein. Jedem Objekt aus B ist jedoch nur genau eines aus A zugeordnet. Im UML-Diagramm ist B die Klasse, dessen Seite der Beziehung mit 1 beschriftet ist. Wir benötigen in diesem Fall eine eigene Tabelle für die Objekte der Klasse A, können jedoch die Tabellen für die Beziehung und die Objekte aus B verschmelzen.

- N:N
  Bei dieser Beziehung können keine resultierenden Tabellen verschmolzen werden.

Aggregationen

Eine Aggregation ist als „Teil-Ganzes“-Beziehung zu verstehen, bei der die Teile jedoch auch unabhängig vom Ganzen existieren können. Eine Aggregation zeigt eine Unterordnung der Teile gegenüber dem Ganzen auf der Ebene der Semantik an.

Dies hat jedoch keine Auswirkungen auf die Transformation, so dass Aggregationen wie Assoziationen transformiert werden. In DB2 existieren keine Konzepte um diese Art der Beziehung speziell zu kennzeichnen.

Kompositionen

Eine Komposition ist eine Verschärfung der Aggregation, und zwar in der Hinsicht, dass Teile nicht mehr ohne das Ganze existieren können, somit als abhängig sind. Jedem Teil kann nur genau ein Ganzes zugeordnet werden, die Kardinalität ist hierbei stets 1.

Diese Abhängigkeit kann in der Datenbank umgesetzt werden, indem bei Objekten, die Komponenten besitzen, beim Löschen auch die Komponenten entfernt werden. Eine Fremdschlüsselbeziehung sorgt dafür, dass nur Teile existieren, zu denen auch ein Ganzes existiert.

Vererbung

Bei der Vererbung wird eine Klasse spezialisiert, d.h. es wird eine Kindklasse durch Hinzufügen von Attributen und/oder Methoden erstellt. DB2 unterstützt eine Subtypen-Bildung, es können somit die komplexen Datentypen, die die Attribute der Klasse beschreiben in einer Hierarchie angeordnet werden. Für jede Klasse, also sowohl für die Eltern- als auch dessen Kindklassen, werden eigene Tabellen angelegt, die vom System automatisch verwaltet werden. DB2 unterstützt neben Typhierarchien auch Tabellenhierarchien. So wird mit der `UNDER`-Klausel in der `CREATE TABLE`-Anweisung eine Tabelle als Kindtabelle einer anderen Tabelle definiert.

5.2 Abbildung des GDM in den IBM Content Manager

5.2.1 Das CM-Datenmodell

Dieser Abschnitt stellt kurz das Datenmodell (DM) des IBM Content Manager (ICM) vor. Weitere Informationen hierzu sind in [Dol04], sowie [IBM02c] zu finden.

Der IBM Content Manager unterscheidet zwischen den Metadaten und den eigentlichen Datenobjekten, die z.B. Bilder, Dokumente oder Videos sein können. Die Daten werden getrennt von den Metadaten im *Resource Manager*, einer Komponente des ICM, gespeichert. Die Speicherung der Metadaten und Verwaltungsinformationen erfolgt im *Library Server*, einer anderen Komponente des ICM, welche auf einer DB2-Datenbank basiert.

Die Grundelemente der Speicherung sind Attribute. Attribute können in Gruppen zusammengefasst werden, diese Gruppen haben jedoch keine weiteren Eigenschaften außer ihrem Namen.

Mehrere Attribute werden in *Components* zusammengefasst. Diese können hierarchisch angeordnet werden: Auf der obersten Ebene existiert eine Root Component, welche dann mit Child Components verknüpft werden. Tiefe und Breite dieser Hierarchie sind nicht beschränkt. Bei der Definition der Hierarchie können Kardinalitäten angegeben werden. Mit Hilfe der Components können komplexe und mengenwertige Attribute definiert werden.

Mehrere solcher Components können, inklusive ihrer Hierarchie, in einem *Item Type* zusammengefasst werden. Ein Item Type enthält eine Root Component, der mehrere Child Components zugeordnet sein können. Item Types dienen als Vorlage, nach der Items im ICM erstellt werden. Dabei können Item Types wiefolgt klassifiziert werden:

- als Item, das reine Metadaten enthält,
- als Resource Item, das eine zugeordnete Resource im Resource Manager besitzt,
- als Document, was zeigt, dass dieser Item Type ein vollständiges Dokument repräsentiert,
- oder als Document Part, der Teil eines vollständigen Dokuments ist. Ein Document Part ist von einem bestimmten Typ, der die Art des Inhalts näher angibt.

Die ersten beiden bilden das so genannte *Custom Model*, die letzten beiden das *Document Model*. Einige Clients, darunter beide dem ICM beliegenden Clients, ignorieren die Items des Custom Model. Es ist daher für sehr spezielle Anwendungen vorgesehen.

Das Document Model ist im ICM optimiert, aber leider ist es nicht ausreichend, um das GDM abzubilden. Die Typen der Items können nicht frei definiert werden, und somit können keine Features oder Regionen abgebildet werden. Es ist also notwendig auf das Custom Model (Items und Resource Items) zurückzugreifen, wodurch allerdings auch ein spezieller Client notwendig wird.

Mittels so genannter *Item Type Subsets* können Sichten auf den Item Types definiert werden. Allerdings sind diese Sichten auf einzelne Item Types beschränkt. Verbundsichten sind daher nicht realisierbar.

Instanzen von Item Types werden *Items* genannt. Diese können Referenzen auf Objekte im Resource Manager enthalten. Der ICM unterstützt eine Versionierung sowie das Zuweisen semantischer Informationen zu Items.

Der ICM bietet verschiedene Möglichkeiten an Referenzen zu definieren. Die erste Möglichkeit sind *Links*, die zwischen zwei Root-Elementen eine gerichtete Beziehung definieren. Diese Links werden in separaten Tabellen gespeichert. Auf Links sind keine Einschränkungen definiert, sowohl Quelle als auch Ziel können unabhängig voneinander gelöscht werden.

Ein zweiter Referenztyp heißt *Reference*. Dieser definiert eine gerichtete 1:1-Beziehung zwischen einer Root- oder Child-Component und einer Root-Component. Anders als bei Links kann hierbei das Verhalten des Ziels beim Löschen der Quellseite spezifiziert werden.

Als dritten Referenztyp bietet der ICM *Foreign Keys* an. Diese werden durch die darunterliegende Datenbank bereitgestellt und entsprechen in ihrer Funktionsweise Fremdschlüsseln. Da Components als Tabellen gespeichert werden und die Attribute den Attributen der Tabelle entsprechen, können Foreign Keys zwischen allen Component-Typen definiert werden. Es können sogar externe Tabellen referenziert werden.

5.2.2 Abbildung von Klassen

Jede Klasse wird auf eine Item Type abgebildet. Dabei wird das Bild selbst als Resource im Resource Manager gespeichert, und ist somit als Resource Item modelliert. Dies trifft auch auf alle Klassen zu, in denen der Datentyp `Binary` vorkommt.

Alle anderen Klassen werden auf „reguläre“ Item Types abgebildet, d.h. als Item klassifizierte Item Types.

Um diesen Item Type zu bilden, werden alle Attribute in die Root Component aufgenommen, bzw. durch Child Components modelliert.

5.2.3 Abbildung von Attributen

Da der ICM auf DB2 basiert sind viele Datentypen aus DB2 auch im ICM verfügbar:

- Zeichenketten: fixe und variable Länge ([VAR]CHAR in DB2)
- Ganzzahlige Typen: 16- und 32-bit Integer
- Fix- und Fließkommazahlen (vgl. `DECIMAL` und `DOUBLE` in DB2)
- Datum, Zeit und Zeitstempel (`DATE`, `TIME`, `TIMESTAMP`)

- BLOB und CLOB, jeweils bis zu einer Größe von 5 Megabytes

Die Umsetzung erfolgt hierbei im Wesentlichen wie bei DB2. Tabelle 5.4 zeigt die Zuordnung der Datentypen.

5.2.4 Abbildung von Operationen

Wie in [Dol04] beschrieben, bietet ICM keine Mechanismen an, um das Verhalten von Item Types zu definieren. Es kann jedoch auf Umwegen eine Integration von Methoden erreicht werden:

- Definition eigener *Media Object Classes*: Media Object Classes dienen dazu, Objekte im Resource Manager zu klassifizieren. Neben einigen vordefinierten Typen ist es möglich, eigene Typen zu erstellen, und spezielle Methoden zur Behandlung dieser Typen zu definieren. Da diese Möglichkeit nicht für „reguläre Items“, d.h. Items vom Typ Item Type, anwendbar ist, ist sie für die Transformation des GDM nicht brauchbar.

- Integration in der Client-Anwendung: Da ohnehin ein spezieller Client notwendig ist um das entstehende Modell zu nutzen, können Methoden auch im Client realisiert werden. In diesem Fall wären Operationen bei der Transformation nicht zu berücksichtigen, sondern müssen nur an den Client weitergeleitet werden.

- Stored Procedures/UDFs: Da der ICM auf einer DB2-Datenbank basiert, können Methoden direkt auf der darunterliegenden Datenbank definiert werden. Diese werden dann serverseitig verfügbar sein, können jedoch nur über die Schnittstellen der Datenbank, nicht des ICM aufgerufen werden.

Da sowohl die Client-Integration, als auch die Verwendung von Stored Procedures nicht generisch, sondern nur angepasst an spezielle Anwendungen umsetzbar sind, und somit in Implementierungsaufwand und funktionellen Möglichkeiten mit externen Operationen identisch sind, werden Operationen nicht in den ICM abgebildet.

5.2.5 Abbildung von Beziehungen

Assoziationen

Allgemein werden Assoziationen zwischen zwei oder mehreren Klassen hergestellt. Um dies im ICM umzusetzen, können *Links* verwendet werden. Diese werden in separaten Tabellen gespeichert, und verbinden zwei Root Components miteinander. Attribute der Beziehung können durch ein zugeordnetes Item gespeichert werden. N-äre Beziehungen, d.h. Beziehungen zwischen mehr als zwei Klassen, werden allerdings nicht unterstützt.

Tabelle 5.4: Transformation der Datentypen vom GDM für den ICM

GDM-Datentyp	ICM-Datentyp
Ganzzahlige Typen (`Int`)	Small Integer (16bit), Long Integer (32bit) unsigned ist nicht transformierbar, 64-bit-Integer ebenfalls nicht. Beide werden vom ICM nicht unterstützt.
Fließkommazahlen (`Float`)	Double ICM hat keinen Single Precision Fließkomma-Typ, dieser wird daher ebenfalls auf Double abgebildet.
Wahrheitswerte (`Bool`)	Small Integer ICM unterstützt keine Wahrheitswerte, eine Speicherung als Integer mittels 0 und 1 simuliert dies.
Zeichen (`Char`,`String`)	(Variable) Zeichenkette
Binärdaten (`Binary`)	Binärdaten werden mittels Resource Items umgesetzt.
Referenz auf Datei (`FileRef`)	Dateireferenzen werden vom ICM nicht unterstützt, es erfolgt eine Abbildung auf eine Zeichenkette, die die URI der Resource aufnehmen kann. Ein Client kann diese Angabe dann interpretieren.
Aufzählung (`Enum`)	Aufzählungen werden zunächst auf eine Zeichenkette abgebildet, die lang genug für jedes Element ist. Es wird dann eine Tabelle angelegt, die die Elemente beinhaltet. Eine Fremdschlüsselbedingung wird anschließend beim Attribut im Item Type definiert, wodurch der Wertebereich dieses Attributs eingeschränkt wird auf die Elemente der Aufzählung.
Menge (`Set`)	wird mit einer Child-Component simuliert
Multimenge (`Bag`)	wird mit einer Child-Component simuliert
Vektor (`Vector`)	wird mit einer Child-Component simuliert
Matrix (`Matrix`)	Eine Simulation mittels Child Components ist zu aufwändig, da Components eindimensional sind. Es wäre jedoch möglich diesen Datentyp als LOB zu modellieren. Es ist dann notwendig den Client anzupassen, damit Matrizen interpretiert werden können.
Struktur (`Struct`)	Child-Component

Aggregationen

Wie bereits bei DB2 werden auch im ICM Aggregationen wie Assoziationen behandelt, denn der ICM stellt keine speziellen Konzepte zur Verfügung, um Aggregationen entsprechend zu unterstützen. Für den Spezialfall einer 1:1-Aggregation können auch Referenzen anstelle von Links genutzt werden. Diese benötigen keine eigene Tabelle und sind unidirektional.

Kompositionen

Der ICM bietet auch keine speziellen Konzepte an, um jede Form der Komposition zu unterstützen. Jedoch können im Falle einer 1:1-Komposition anstelle von Links ebenfalls Referenzen verwendet werden. Dies hat den Vorteil, dass Komponenten beim Löschen des Ganzen automatisch mit gelöscht werden können. Falls jedoch eine Komponente optional ist oder mehrfach vorkommen kann, muss die Referenzierung mittels Links erfolgen.

Vererbung

Das Konzept der Vererbung wird durch den ICM nicht unterstützt und muss daher simuliert werden. Die Funktionalität dafür muss durch einen speziellen Client realisiert werden.

In der Transformation werden für jede Klasse, auch für abstrakte Klassen, alle lokal definierten Attribute gespeichert. Es wird außerdem eine Referenz zwischen Kind- und Vater-Item gesetzt.

Eine detaillierte Beschreibung dieser Umsetzung der Vererbung ist im nachfolgenden Abschnitt beschrieben.

5.2.6 Anforderungen an den Client

Wie in den vorangegangen Abschnitten deutlich wurde, ist die Verwendung eines eigenen ICM-Client notwendig, da mitgelieferte Clients nicht in der Lage sind, die für das GDM erforderlichen Konstrukte abzubilden.

Dazu werden einige vom Client verwaltete Tabellen zur Speicherung von Verwaltungsinformationen benötigt. Diese können im Content Manager oder in der darunterliegenden Datenbank gespeichert werden.

Unterstützung des Typsystems

Der ICM unterstützt leider nicht alle Datentypen des GDM-Typsystems, es ist daher notwendig einige Typen zu simulieren. Für kollektionswertige Typen (Menge, Multimenge, etc.) werden Child Components verwendet. Der ICM bietet auch keine Möglichkeiten an, die Integrität der Daten zur Laufzeit zu prüfen. Es ist daher notwendig dies durch den Client zuzusichern.

In einer separaten Tabelle werden die Datentypen für jeden Item Type gespeichert, wodurch der Client die Attribute korrekt behandeln kann, denn im Item Type kann nicht kenntlich gemacht werden, dass eine Child Component z.B. eine Multimenge enthält.

Weiterhin müssen Methoden zur Speicherung, Abfrage und Änderung der Daten integriert werden, die die Semantik der Datentypen berücksichtigen. Beispielsweise kann beim Einfügen in eine Menge nur der Client für Duplikatfreiheit sorgen.

Anfrage- und Updateschnittstelle, API

Da einige Konzepte des GDM, wie beispielsweise Vererbung, vom ICM nicht unterstützt werden, ist eine Anfrageschnittstelle vom Client bereitzustellen, um es dem Nutzer oder einer Applikation zu ermöglichen, auf die Daten entsprechend zuzugreifen. Dabei kann auf die ICM-API zurückgegriffen werden.

Der Client muss außerdem Methoden bereitstellen, mit der die Daten modifiziert werden können. Auch hier können entsprechende Methoden basierend auf der ICM-API erstellt werden, um weitere GDM-spezifische Konstrukte, wie beispielsweise spezielle Datentypen, nutzbar zu machen.

Nur so können Operationen auf das Modell zugreifen, und die Semantik spezieller Konzepte beim Zugriff von außen zugesichert werden, wie beispielsweise die Duplikatfreiheit in Mengen, oder die korrekte Anordnung von Item Types innerhalb der Vererbungshierarchie.

Da im Transformationsschritt für den ICM lediglich ein Datendefinitions-Skript in Form einer XML-Datei erzeugt wird, der ICM dafür jedoch keine Import-Möglichkeit wie DB2 besitzt, muss der Client in der Lage sein solche Import-Skripte zu interpretieren und aus den darin enthaltenen Daten entsprechende Strukturen im ICM anlegen.

Integration der Vererbung

Die Integration des Vererbungskonzeptes sei anhand folgenden Beispiels erläutert: Wir betrachten zwei Klassen A und B, wobei B die Kindklasse (Spezialisierung) von A ist, also die Attribute und Operationen von A übernimmt bzw. überlädt und eigene hinzufügt. Es gibt mehrere Ansätze zur Speicherung der Attribute (vgl. [SH99]), z.B. Speicherung aller Attribute in jeder Klasse (Repeat Class Model) oder Speicherung der lokalen Attribute in jeder Klasse (Split Instance Model). Die Verfahren unterscheiden sich in Speicherbedarf und Aufwand zur Rekonstruktion des Objektes. Diese spezielle Speicherung erfordert darauf abgestimmte Update- und Retrieval-Methoden, die im Client implementiert werden müssen.

Aufgrund des geringeren Speicherbedarfs und der relativ selten vorkommenden Vererbung innerhalb des GDM, wird in der Transformation das Split Instance Modell benutzt. Es werden für jede Klasse nur diejenigen Attribute gespeichert, die auch direkt für diese Klasse definiert sind. So werden zur Klasse B aus dem obigen Beispiel nur Attribute gespeichert, die in B, aber nicht in A definiert sind. Um nun Objekte rekonstruieren zu können, muss eine Referenz vom Kindobjekt auf das Vaterobjekt gesetzt werden. Im Beispiel referenziert jedes Objekt aus der Extension von

B sein Vaterobjekt aus der Extension der Klasse A. Der Client kann diese Referenz verfolgen und somit die fehlenden Attribute ergänzen.

Um dieses Modell umzusetzen, ist es notwendig auch Objekte abstrakter Klassen zu speichern. Diese Klassen müssen jedoch speziell gekennzeichnet sein, damit der Client sie als abstrakt erkennt und den direkten Zugriff auf die gespeicherten Objekte verbietet, da diese Objekte lediglich der Speicherung der Attribute dienen.

Integration von Operationen

Um die Operationen des Modells zu integrieren kann eine Schnittstelle zum Aufruf externer Module im Client umgesetzt werden. Dazu muss der Client eine Möglichkeit anbieten, Operationen an Item Types zu binden sowie eine Möglichkeit bereitstellen die Implementierung anzubinden und aufzurufen.

Eine Möglichkeit dies umzusetzen besteht darin, in die Retrieval-Schnittstelle den Methodenaufruf zu integrieren. Nachdem die Zugehörigkeit der Methode zum entsprechenden Item verifiziert wurde, kann eine zur Laufzeit angebundene Implementierung aufgerufen werden.

Die Methoden könnten über die API auf das Modell zugreifen und somit beispielsweise bei der Implementierung einer Feature-Extraktion die Feature-Objekte erzeugen. Die Anbindung der Methoden könnte bei einem Java-Client mittels des Aufrufs von `Class.forName(...)`, bei anderen Programmiersprachen mittels Anbindung von Bibliotheken zur Laufzeit erfolgen.

Die Zuordnung der Methoden zu Item Types kann tabellarisch festgehalten werden. Diese Tabelle, die entweder in der Datenbank oder auch im ICM gespeichert wird, enthält den jeweiligen Item Type und die zugeordneten Methoden. Eine Auswertung der Vererbungsinformationen des jeweiligen Item Types liefert die Operationen des Obertyps. Falls Overloading und Overriding unterstützt werden soll, sind entsprechende Vergleiche der Signatur zu den gegebenen Parametern zu implementieren.

Kapitel 6

Entwurf eines Prototyps

6.1 Entwurf eines Algorithmus für die Transformation

Dieser Abschnitt stellt einen Algorithmus vor, der die Aufgabe der Transformation durchführt. Der Algorithmus ist umgangssprachlich formuliert, eine Implementierung erfolgt im Prototypen. Zugunsten der Übersicht ist der gesamte Algorithmus in Module aufgeteilt, die jeweils nacheinander vorgestellt werden.

Die Eingabe für den Algorithmus ist das anwendungsspezifische Datenmodell, das als UML-Klassendiagramm vorliegt.

6.1.1 Hauptprogramm

Eingabeparameter:

- `Model` - ein Modell in UML
- (optional) `ExtTypes` - eine Transformationstabelle für externe Typen (siehe 5.1.4)

Ablauf:

1. Überprüfe, ob im Modell keine verbotenen Konstrukte enthalten sind (siehe Abschnitt 4.3.8). Falls solche Konstrukte enthalten sind breche mit einer Fehlermeldung ab. Gebe eventuelle Warnungen aus (z.B. falls bei der Transformation nach DB2 in der Klasse `Image` keine Schlüssel definiert sind).
2. Erstelle eine Liste aller in `Model` vorkommenden Klassen. Diese Liste sei mit `Classes` bezeichnet. Informationen zur Vererbung sind hier gespeichert.

3. Erstelle eine Liste `IntTypes`, die alle im Modell benutzten Datentypen enthält, und außerdem jedem Datentyp einen entsprechenden Typen auf der Zielplattform zugeordnet. (→6.1.2)
4. Für jedes Element `cl` ∈ `Classes` transformiere die Attribute von `cl` in einen komplexen Datentyp. (→6.1.3)
5. Erstelle eine Liste aller Beziehungen in `Model`. Diese Liste sei `Relations`. Vererbungshierarchien werden hierbei nicht als Beziehungen betrachtet.
6. Transformiere die für jedes jedes Element `rel` ∈ `Relations` die Attribute der Assoziationsklasse in einen komplexen Datentyp. Ist keine Assoziationsklassen vorhanden, so wird ein leerer komplexer Typ erzeugt.
7. Für jede Beziehung `rel` suche Möglichkeiten zur Verschmelzung. (→6.1.4)
8. Erstelle Referenzen zwischen komplexen Datentypen. Dazu werden alle an einer Beziehung `rel` beteiligten Klassen mittels eines entsprechenden Konstruktes referenziert. Dabei sind leere Assoziationsklassen zu beachten, da eventuell andere Konstrukte zur Referenzierung verwendet werden können.
9. Erzeuge die Ausgabe.

6.1.2 Erstellung der Internen Typ-Transformationstabelle

Eingabeparameter:

- `Classes` - die Liste aller Klassen
- `PredefTypes` - eine Liste vordefinierter Datentypen
- (optional) `ExtTypes` - eine Transformationstabelle für externe Typen

Ablauf:

1. Erstelle die leere Liste `IntTypes`.
2. Erstelle eine Menge `UsedTypes`, die alle in `Classes` vorkommenden Datentypen enthält.
3. Für jeden Datentyp `type` ∈ `UsedTypes` erstelle einen transformierten Typ:
   - Suche dafür zunächst einen vordefinierten Datentyp und versuche ihn zu transformieren.
   - Existiert kein vordefinierter Typ, so suche einen externen Typ.
   - Existiert auch kein externer Typ namens `type`, so versuche den Typ zu parsen, ergibt sich ein GDM-Typ, so benutze diesen um einen Typ auf der Zielplattform zu erstellen.

- Konnte bis hier keine Transformation des Typs erfolgen, dann gib eine Fehlermeldung aus. In diesem Falle handelt es sich um einen nicht definierten Typ.

4. Speichere den transformierten Typ zusammen mit der Definition des Typs in der Liste `IntTypes`. Erfordert ein Typ eine Deklaration auf der Zielplattform (z.B. ein `CREATE TYPE` bei DB2), so wird diese Deklaration bereits in die Ausgabe geschrieben, damit dieser Typ bei der Klassedefinition zur Verfügung steht.

6.1.3 Erstellen von komplexen Datentypen für eine Klasse

Eingabeparameter:

- `class` - eine Klasse im UML-Diagramm
- `IntTypes` - die interne Typ-Transformationstabelle
- `Classes` - die Liste aller Klassen

Ablauf:

1. Falls die Zielplattform Methoden unterstützt, sammle alle abgeleiteten Attribute der Klasse `class` in einer Liste `DerivedAtts`, und alle nicht abgeleiteten in `Atts`. Werden keine Methoden unterstützt, so speichere alle Attribute in `Atts`.
2. Ordne jedem Attribut `att` $\in$ `Atts` einen internen Typ `intType` $\in$ `IntTypes` zu.
3. Ordne jedem Attribut `dAtt` $\in$ `DerivedAtts` einen Methoden-Namen zu.
4. Erstelle einen komplexen Typ für diese Klasse, indem:
   - für alle Attribute eine transformierte Form generiert wird,
   - für alle Methoden eine Signatur generiert wird,
   - eine Klausel generiert wird, die den Typ in der Typhierarchie korrekt platziert,
   - und ein plattformspezifisches Statement zur Deklaration des komplexen Typs auf der Zielplattform erzeugt wird.
5. Ordne der Klasse `class` die erzeugten Statements zu.

6.1.4 Verschmelzen von Beziehungen und beteiligten Klassen

Eingabeparameter:

- `AbstractTargetTypes` - die Menge der zu erstellenden komplexen Datentypen
- `rel` - eine Beziehung aus `Relations`

Ablauf:

1. Falls keine der an der Beziehung `rel` beteiligten Klassen die Kardinalität 1 hat kann an dieser Stelle mit der nächsten Beziehung fortgesetzt werden, da keine Verschmelzung möglich ist.
2. Der komplexe Typ `relType` sei der der Beziehung zugeordnete, komplexe Datentyp. Hatte die Beziehung keine Attribute, so ist `relType` leer.
3. Für jede an der Beziehung beteiligte Klasse mit einer Kardinalität von 1 wird der komplexe Typ der Klasse mit `relType` verschmolzen: die Attribute der jeweiligen Typen werden zu einem Typ zusammengefasst. Anschließend wird die Beziehung, die zwischen den nun vereinigten Typen bestand gelöscht. Beim Veschmelzen ist zu berücksichtigen, dass in rekursiven Beziehungen der komplexe Typ der Klasse nicht mehrfach im verschmolzenen Typ enthalten sein darf.

6.2 Durchführung am Beispiel von eNoteHistory

Das Datenmodell für eNoteHistory ist Abbildung 4.8 auf Seite 50 zu entnehmen. Ausgehend von diesem Modell wird der Algorithmus für die Transformation nach DB2 durchgeführt:

Als erstes wird eine Liste aller Klassen erstellt, wobei die Vererbungshierarchie hierbei mit angegeben ist:

```
Classes:= {ImageMetaData, Image, FHGMeasures,
           Region, Feature,
           BoundingBox:Region, ROI:Region,
           Note_Stem:Feature, Staff_Lines:Feature,
           Note_Head:Feature, Rest:Feature,
           Clef:Feature, Bar_Lines:Feature}
```

Als zweites wird die interne Typ-Transformations-Tabelle erzeugt, indem alle vorkommende Typen zunächst gesammelt werden, und jedem Typ ein DB2-Typ zugeordnet wird. Dabei werden falls nötig bereits DDL-Statements in die Ausgabe geschrieben, um beispielsweise komplexe Datentypen zu erzeugen:

```
IntTypes:= {Binary:10240 -> BLOB(10240K),
            Bool -> SMALLINT (CHECK <col> BETWEEN 0 AND 1),
            ...
            PointType -> GDM_PointType,
            ...}

out('CREATE TYPE GDM_PointType AS (...)');
```

Ausgehend von dieser Liste werden komplexe Datentypen für die Klassen erstellt. Dabei werden Informationen über die Typen, z.B. ob es sich um Primärschlüssel handelt, mit gespeichert. Ausgehend von diesen Informationen werden später entsprechende Typen mit `CREATE TYPE` erzeugt sowie Extensionen mittels `CREATE TABLE` angelegt:

```
ImageMetaData -> Type{LibraryCode=VARCHAR(100) (PK),
                      ShelfMark=VARCHAR(100) (PK),
                      SectionNumber=VARCHAR(50) (PK),
                      PageNumber=VARCHAR(50) (PK)},
Image -> Type{Image=BLOB(10240K),
              Size=GDM_DimensionType,
              Created=GDM_TimeStampType,
              Updated=GDM_TimeStampType},
FHGMeasures -> Type{CenterVertical=DOUBLE,
                    CenterHorizontal=DOUBLE,
                    ...},
Feature -> Type{ID=BIGINT (PK)},
Region -> Type{ID=BITINT (PK)},
...
```

Jetzt wird die Liste der Beziehungen erstellt, wobei die Notation in der Form (Seite1[:Rolle], Seite2[:Rolle], ...)=(Kardinalität,Art) erfolgt:

```
Relations:= {
  (ImageMetadata,Image)=(1:1,regular),
  (Image:parent,Image:child)=(0..1:0..*,aggregation),
  (Image,Region)=(1:0..*,aggregation),
  (Region:parent,Region:child)=(0..1:0..*,aggregation),
  (Region,Feature)=(1:0..*,regular),
  (Feature,FHGMeasures)=(1:0..1,regular)
}
```

Nun wird nach Möglichkeiten zur Verschmelzung von komplexen Typen gesucht. Dafür wird jede Beziehung aus `Relations` untersucht, wobei folgende Beziehungen für die Verschmelzung in Frage kommen:

```
(ImageMetadata,Image)=(1:1,regular),
(Image,Region)=(1:0..*,aggregation),
(Region,Feature)=(1:0..*,regular),
(Feature,FHGMeasures)=(1:0..1,regular)
```

Lediglich die erste Beziehung führt zu einer Veschmelzung. Bei allen anderen Beziehungen würde zwar die Assoziationsklasse mit verschmolzen, da jedoch keine dieser Beziehungen Attribute enthält werden auch keine Typen verschmolzen. Als Resultat der Verschmelzung werden die komplexen Datentypen für `Image` und `ImageMetaData` zu einem neuen Typ `ImageMetaData_Image` verschmolzen.

Nun werden alle komplexen Typen in DDL-Statements umgewandelt und ausgegeben:

```
out('CREATE TYPE T_ImageMetaData_Image AS (
     LibraryCode VARCHAR(100) PRIMARY KEY NOT NULL,
     ShelfMark VARCHAR(100) PRIMARY KEY NOT NULL,
     ...
     )');
...
out('CREATE TABLE ImageMetaData_Image
     OF T_ImageMetaData_Image ...');
...
```

6.3 Anbindung an den UML-Entwurf

Der Entwurf des UML-Diagramms erfolgt mit Hilfe der Rational Rose Enterprise Edition, Version 2002.05.20. Dieses Programm bietet neben seinen guten Möglichkeiten zum Entwurf auch die Möglichkeit, mit dem *Rose Extensibility Interface* (REI) auf das Modell mittels einer Java-API zuzugreifen. Außerdem können eigene Menüpunkte erstellt werden, die den Aufruf verschiedener Komponenten, wie beispielsweise der Transformation, direkt aus Rational Rose heraus ermöglichen.

Um die Vorgaben des GDM in Rose zu integrieren, wurde ein Rose-Framework entworfen und von eNoteHistory-Projekt bereitgestellt, das die Klassen des GDM wie in Abbildung B.1 im Anhang auf Seite 90 enthält. Der Nutzer hat somit ein Basismodel zur Verfügung, das alle Klassen und Beziehungen enthält, die in dieser Arbeit definiert wurden. Ausgehend von diesen kann das problemspezifische Modell erstellt werden.

Die Transformationskomponente greift mittels REI direkt auf das Modell zu, und erzeugt daraus eine plattformspezifische Ausgabe.

Ausführliche Informationen zur Erweiterung der Funktionalität von Rational Rose durch Verwendung von Add-Ins sind in [Nel05] und [Rat01] zu finden.

6.4 Anbindung an die Zielplattform

Prinzipiell wäre es möglich, mittels JDBC direkt auf DB2 zuzugreifen, um so das transformierte Modell direkt in der Datenbank anzulegen. Für den ICM existiert eine vergleichbare API ebenfalls. Um das Modell direkt anzulegen werden jedoch verschiedene Informationen benötigt, bei DB2 beispielsweise Name der Datenbank, JDBC-Treiber, etc.

Da der Prototyp keine grafische oder textuelle Oberfläche enthält, die mit dem Nutzer in Interaktion tritt und solche Parameter erfragen kann, werden lediglich Skripte erzeugt, mit deren Hilfe das Modell auf den jeweiligen Zielplattformen angelegt werden kann. Diese müssen anschließend manuell ausgeführt werden.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Rückblick

Nachdem in Kapitel 3 verschiedene Datenmodelle zur Beschreibung von Bilddaten erläutert wurden, speziell waren dies das Entity-Relationship-Modell (ERM), der Standard MPEG-7 und die Unified Modeling Language (UML), sind aus den Schwächen der einzelnen Modelle Ideen für ein neues konzeptuelles Modell entstanden. Dabei wurde aus Ausgangspunkt für das Modell UML ausgewählt, da es sich als gute Basis für den Entwurf präsentierte. UML bietet vielfältige Modellierungskonzepte, unterstützt Operationen und ist außerdem weit verbreitet.

Das generische Datenmodell (GDM) ermöglichte es also, den Inhalt von digitalen Bildern mit Hilfe von UML zu beschreiben. Es wurde dafür ein Framework aus verschiedenen Klassen bereitgestellt, von dem ausgehend der Anwender eigene Bildinhalte beschreiben kann. Es wurden verschiedene Aspekte der Bildmodellierung betrachtet, z.B. die Aufteilung des Bildes, aus denen Ideen und Ansätze für das GDM entwickelt wurden. Einige Konzepte der im vorangegangenen Kapitel betrachteten Modelle, wie beispielsweise die Aufteilung eines Bildes in ein gleichmäßiges Grid, was in MPEG-7 enthalten ist, finden sich ebenfalls im GDM wieder. Außerdem wurde im Kapitel 4 das entwickelte generische Datenmodell genutzt, um die Inhalte von Bildern von Notenblättern des eNoteHistory-Projektes zu beschreiben.

Kapitel 5 widmete sich der Beschreibung der Transformation des UML-Modells für eine spezifische Zielplattform. Speziell waren dies IBM DB2 und der IBM Content Manager. Dabei wurden Probleme wie die Abbildung von Datentypen, Klassen und Beziehungen diskutiert und Möglichkeiten zur Lösung dieser Probleme genannt. Es wurde außerdem festgestellt, dass nicht jedes UML-Konstrukt, das im GDM verwendet werden kann, auch abbildbar ist. Es gibt noch Schwierigkeiten bei Datentypen sowie einige Probleme bei der Umsetzung von Operationen, speziell auf dem Content Manager. Auch bei DB2 gibt es einige Einschränkungen und Abweichungen vom SQL/99-Standard, so dass die Transformation auf DB2 speziell abgestimmt werden musste.

Anschließend wurden in Kapitel 6 konkrete Ansätze für einen Prototypen entwickelt, der in der Lage sein soll die Transformation automatisch durchzuführen. Es wurden der implementierte Algorithmus sowie die Anbindung an den UML-Entwurf und die Zielplattformen beschrieben.

7.2 Ausblick

Die Aufgabe dieser Arbeit war es, unter anderem ein Modell zur Bildbeschreibung zu entwickeln und die Abbildung auf verschiedene Zielplattformen prototypisch umzusetzen. Dies ist erfolgt, jedoch gibt es verschiedene Teile des Prototypen, die weiterentwickelt werden müssen. So wäre beispielweise eine grafische Oberfläche sehr sinnvoll. Zum einen kann dadurch die Komunikation von Transformation und Rational Rose besser gestaltet werden, zum anderen können der Transformation notwendige Parameter zur direkten Ausführung der erzeugten DDL-Befehle auf der Zielplattform mitgeteilt werden. Eine Verbesserung der Rose-Transformation-Kommunikation könnte sich auch in der Nutzung der Log-Funktionalität durch die Transformation ausprägen, denn bisher wird das Log von Rose nicht benutzt.

Eine weitere fehlende Komponente ist bereits in Abschnitt 5.2.6 genannt worden: ein Client für den IBM Content Manager. Das Problem dieser Anwendung ist, das die mitgelieferten Clients nur über eine eingeschränkte Funktionalität verfügen, und somit für das GDM nicht brauchbar sind. Der Prototyp erzeugt ein Skript in Form einer XML-Datei, das anschließend von genau diesem Client ausgewertet werden muss, um die entsprechenden Datenstrukturen im ICM anzulegen.

Eine andere, interessante Möglichkeit ist es, statt der sehr speziellen Implementierung der Transformation eine allgemeinere Form zu entwickeln. So könnte die Transformationskomponente beispielsweise lediglich eine Beschreibung des UML-Modells in Form einer XML-Datei erzeugen, aus der anschließend mit Hilfe XSLT ein plattformspezifisches DDL-Skript erstellt wird. Auf diesem Wege ließen sich Zielplattformen ohne Programmierung einer entsprechenden Transformationskomponente anbinden, da nun nicht mehr in die Implementierung eingegriffen werden muss, sondern lediglich XSLT-Skripte erstellt werden.

Es wäre außerdem interessant zu untersuchen, welche Rolle XML-Schema dabei spielen kann, denn wie bei der Betrachtung von MPEG-7 (siehe Abschnitt 3.2 auf Seite 13) deutlich wurde, eignet sich XML-Schema sehr gut um die Struktur eines XML-Dokuments vorzugeben. Dies betrifft nicht nur die Element-Hierarchie, sondern auch die Datentypen der Attribute. Durch Verwendung von XML-Schema würde sich vor allem die Interpretation der Datentypen vereinfachen, da diese vom Schema vorgegeben werden kann. Gleichzeitig wären sehr verschiedene Datentypen darstellbar. Mit Hilfe von XML-Schema könnte der Aufbau der Modelle, die mit UML erstellt wurden, beschrieben werden, so dass Instanzen des Modells in Form von XML angelegt werden können.

Eine weitere Verallgemeinerung der Transformation könnte durch Einführung von Tabellen erfolgen, mit deren Hilfe Datentypen transformiert werden. Bisher ist die Umwandlung von Datentypen mit Java-Code beschrieben, wodurch auch Änderungen ausprogrammiert werden müssen. Würde die Transformation auf Tabellen basieren, die alle notwendigen Informationen enthalten, so kann die Implementierung bei Änderungen unberührt bleiben. Diese Methode hätte auch den Vorteil, dass für neue Zielplattformen lediglich neue Tabellen zur Datentyp-Transformation angelegt werden müssen („Konfigurieren statt Programmieren“). Um dies umzusetzen bedarf es allerdings einer Möglichkeit, Regeln zu formulieren, mit denen die Informationen aus den Tabellen zu Typdeklarationen etc. der Zielplattform werden. Vorstellbar ist auch eine möglichst

allgemeingültige Tabellenstruktur, die für viele verschiedene Plattformen angewendet werden kann.

In Abschnitt 4.3.8 wurden verschiedene Einschränkungen formuliert, die den verwendbaren UML-Sprachanteil reduzieren. Viele dieser Einschränkungen ergeben sich aus den Fähigkeiten der in dieser Arbeit genutzten Zielplattformen. Da das GDM ein konzeptuelles Entwurfswerkzeug sein soll, ist es sinnvoll in einer Weiterentwicklung des Modells diese Einschränkungen zu beheben. Auch erhebt die Liste der vorgestellten Einschränkungen keinen Anspruch auf Vollständigkeit. Da UML stets weiterentwickelt wird, müssen auch die Einschränkungen stets aktuell gehalten werden.

Ein anderes Problem beim Entwurf mit dem GDM ist die Integration von Operationen. Zwar werden Operationen bei der Transformation berücksichtigt, aber das Binden der Implementierung dieser Methoden an die Zielplattform ist Aufgabe des Nutzers. Es bleibt zu untersuchen, inwiefern eine vor der Transformation bekannte Implementierung einer Methode bereits in der Transformation automatisch an die Zielplattform gebunden werden kann. Dabei gilt es Probleme, wie z.B. Integrierbarkeit der verwendeten Programmiersprache, oder die Einhaltung der Anforderung der Zielplattform an die Signatur der Operation, zu untersuchen.

Anhang A

Abkürzungsverzeichnis

1NF:	Erste Normalform (Eigenschaft einer Relation)
DBIS:	Datenbanken- und Informationssysteme (Lehrstuhl, Universität Rostock)
DDL:	Data Definition Language
EBNF:	Erweiterte Backus-Naur Form, auch Erweiterte Backus-Normalform
ERM:	Entity Relationship Modell
ICM:	IBM Content Manager
ICM-DM:	IBM Content Manager Datenmodell
IGD:	Frauenhofer Institut für Grafische Datenverarbeitung
MPEG:	Motion Picture Experts Group
OID:	Objekt-Identifikator
OMG:	Object Management Group
OO:	Objektorientierung, Objektorientierte(s/n/m) ...
ROI:	Region of Interest
SP:	Stored Procedure
TIF:	Tagged Image File Format
UDF:	User Defined Function
UML:	Unified Modeling Language
URI:	Unified Resource Identifier
XML:	Extensible Markup Language

Anhang B

Das Generische Datenmodell

B.1 Grammatik für das GDM-Typsystem

Es bietet sich aus Gründen der Portabilität an, als zugrunde liegenden Zeichensatz Unicode (siehe [Uni03]) zu verwenden. Ein Unicode-Zeichen ist durch einen 2 Bytes langen Code definiert. Auf diese Weise wird auch für das Nicht-Terminal `AnyLetter` definiert. Der Code wird hexadezimal angegeben, und wird durch ein angefügtes „h“ als Hexadezimalzahl gekennzeichnet. Die Implementierung der lexikographischen Analyse sollte eine praxisnähere Möglichkeit zur Beschreibung von Wörtern finden.

```
(* ------------------------------------------------------ *)
(* commonly used terminals                                *)
(* ------------------------------------------------------ *)
(* separating type definition and its parameters *)
TypeDefAssign = ':=';
TypeDefSeparator = ':';
TypeDefTerminator = ';';

(* marking complex type's element types *)
ElementDefStart = '{';
ElementDefEnd = '}';

(* definition of a positive integer *)
PosDigit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Digit = '0' | PosDigit;
AnyNumber = PosDigit, {Digit};
DefDynamic = 'dyn';

(* sign *)
```

```
DefSigned = 'signed' | 'unsigned';

(* floating point precision *)
DefPrecision = 'single' | 'double';

(* Structured Types Element Definitions/Enumeration separator *)
DefElementSep = ',';

(* Names *)
HexDigit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
         | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';
(* AnyLetter is any unicode Letter defined by its code;
   Implementation should use the real characters instead;
   the hex number is followed by 'h' *)
AnyLetter = HexDigit, HexDigit, HexDigit, HexDigit, 'h';
Identifier = AnyLetter, {AnyLetter};

(* ------------------------------------------------------ *)
(* Root                                                   *)
(* ------------------------------------------------------ *)
(* a Type is either simple or complex *)
Type = (SimpleType | ComplexType);

(* Declaration of a type *)
TypeDeclaration = Identifier, TypeDefAssign, Type,
                  TypeDefTerminator;

(* ------------------------------------------------------ *)
(* Simple types                                           *)
(* ------------------------------------------------------ *)
SimpleType = IntegerType | FloatType | CharacterType
           | BooleanType | BinaryType | EnumType;

(* Integer; e.g. "Int:32:Signed" for 4-byte signed
   Integer type *)
IntegerType = 'Int', TypeDefSeparator, AnyNumber,
              TypeDefSeparator, DefSigned;

(* IEEE single/double precision floating point *)
FloatType = 'Float', TypeDefSeparator, DefPrecision;

(* single character, or string with min/max length
   (e.g. "String:20" for String having a maximum of
```

```
    20 characters) *)
CharacterType = SingleCharType | StringType;
SingleCharType = 'Char';
StringType = 'String', TypeDefSeparator, AnyNumber;

BooleanType = 'Bool';

(* Binary or file reference *)
BlobType = 'Binary', TypeDefSeparator, AnyNumber,
BinaryType = BlobType | 'FileRef';

(* Enumeration *)
EnumType = 'Enum', ElementDefStart, EnumElementName,
           {DefElementSep, EnumElementName},
           ElementDefEnd;

(* ------------------------------------------------------ *)
(* Complex types                                          *)
(* ------------------------------------------------------ *)
ComplexType = SetType | BagType | VectorType | MatrixType
              | StructType;

(* Set of Elements *)
SetType = 'Set', ElementDefStart, (Type | Identifier),
          ElementDefEnd;

(* Bag (Set allowing multiple occurrences of the
   same element) *)
BagType = 'Bag', ElementDefStart, (Type | Identifier),
          ElementDefEnd;

(* Vector (List) of Elements *)
VectorType = 'Vector', TypeDefSeparator,
             (AnyNumber | DefDynamic),
             ElementDefStart, (Type | Identifier),
             ElementDefEnd;

(* n-dimensional Matrix of any type; e.g.
   "Matrix:10:10:10{Char};" is a cubic Matrix with 10
   elements in each dimension, each element is a
   single character *)
MatrixType = 'Matrix', TypeDefSeparator,
             (AnyNumber | DefDynamic), {TypeDefSeparator,
```

```
            (AnyNumber | DefDynamic)}, ElementDefStart,
            (Type | Identifier), ElementDefEnd;

(* Structured type, having fields accessed by name *)
StructType = 'Struct', ElementDefStart, Identifier,
            TypeDefSeparator, (Type | Identifier),
            {DefElementSep, Identifier, TypeDefSeparator,
            (Type | TypeName)}, ElementDefEnd;
```

B.2 Vordefinierte Datentypen

Datum

```
MonthType:= Enum{Jan,Feb,Mar,Apr,May,Jun,
                 Jul,Aug,Sep,Oct,Nov,Dec};
DateType:= Struct{Day:Int:8:unsigned,
                  Month:MonthType,
                  Year:Int:8:unsigned};
```

Zeit

```
TimeType:= Struct{Hours:Int:8:unsigned,
                  Minutes:Int:8:unsigned,
                  Seconds:Float:single};
```

Zeitstempel

```
TimeStampType:= Struct{Date:DateType,
                       Time:TimeType};
```

Bildpunkt

```
PointType:= Struct{x:Int:32:unsigned,
                   y:Int:32:unsigned};
```

Ausdehnungen

Der folgende Typ dient der Speicherung von Größenangaben von Objekten, die als Fließkommazahlen angegeben werden.

```
DimensionType:= Struct{x:Float:single,
                       y:Float:single};
```

Dieser Typ dient der Speicherung von Pixelgrößen-Angaben.

```
SizeType:= Struct{x:Int:32:unsigned,
                  y:Int:32:unsigned};
```

Farbwerte

Diese Typen repräsentieren Farbwerte der Systeme RGB, CMYK und HSV.

```
RGBColorType:= Struct{Red:Float:Single,
                      Green:Float:Single,
                      Blue:Float:Single}
CMYKColorType:= Struct{Cyan:Float:Single,
                       Magenta:Float:Single,
                       Yellow:Float:Single,
                       Black:Float:Single}
HSVColorType:= Struct{Hue:Float:Single,
                      Saturation:Float:Single,
                      Value:Float:Single}
```

B.3 Die Klassen des GDM

B.3.1 Gesamtmodell

Abbildung B.1 zeigt das gesamte GDM.

B.3.2 Regionen

Eine grafische Übersicht über alle im generischen Datenmodell vordefinierten Regionen ist in Abb.B.2 zu finden. In der Abbildung ist die Assoziation zwischen den Klassen `Image` und `Grid` nicht dargestellt.

Abbildung B.1: Das gesamte GDM

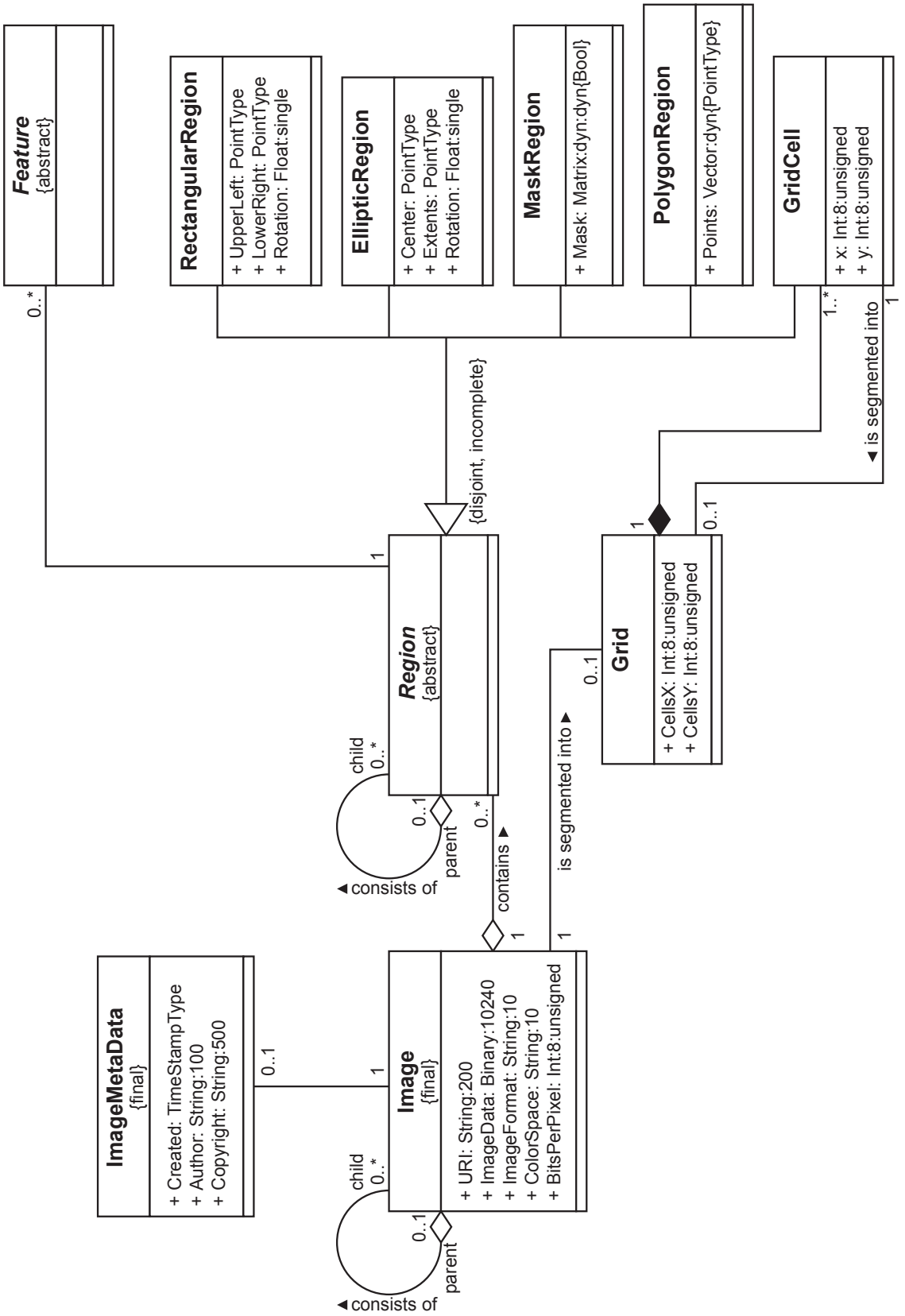
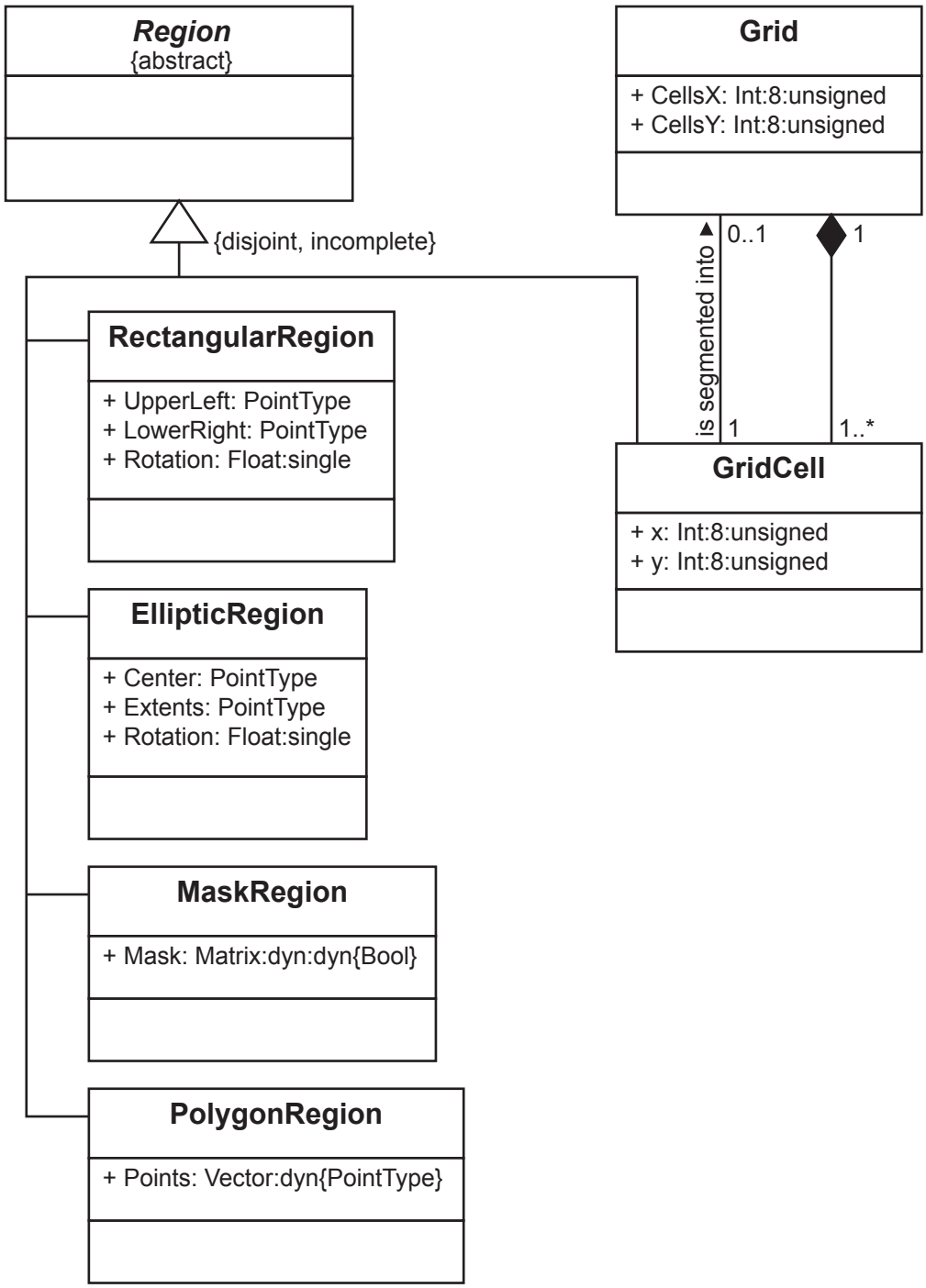


Abbildung B.2: Übersicht über alle im GDM vordefinierten Regionen



Anhang C

Externe Transformationstabellen

Die Erklärung für diese Tabellen ist in Abschnitt 5.1.4 zu finden.

C.1 DB2

Tabelle C.1: Externe Transformationstabelle für DB2

UML-Name	ZP-Typ	Parameter	Beschreibung
DB2_Datalink	DATALINK	0	
DB2_Time	TIME	0	
DB2_Date	DATE	0	
DB2_Timestamp	TIMESTAMP	0	
DB2_Char	CHAR	1	no of characters
DB2_Varchar	VARCHAR	1	max no of characters
DB2_Clob	CLOB	1	max.size (K,M,G)
DB2_Blob	BLOB	1	max.size (K,M,G)
DB2_Real	REAL	0	single precision
DB2_Double	DOUBLE	0	double precision
DB2_Decimal	DECIMAL	0,1,2	default: (5,0)
DB2_Smallint	SMALLINT	0	16-bit signed
DB2_Integer	INTEGER	0	32-bit signed
DB2_Bigint	BIGINT	0	64-bit signed

C.2 ICM

Tabelle C.2: Externe Transformationstabelle für ICM

UML-Name	**ZP-Typ**	**Parameter**
ICM_Character	CHAR	1
ICM_VarCharacter	VARCHAR	1
ICM_ShortInt	SHORTINT	0
ICM_LongInt	LONGINT	0
ICM_Decimal	DECIMAL	2
ICM_Double	DOUBLE	0
ICM_Date	DATE	0
ICM_Time	TIME	0
ICM_Timestamp	TIMESTAMP	0
ICM_Clob	CLOB	0,1
ICM_Blob	BLOB	0,1

Anhang D

Java-Routinen zur Nutzung von LOBs als komplexe Datentypen

Die folgenden Java-Methoden zeigen, wie in DB2 LOBs zur Speicherung von Mengen genutzt werden können. Die verwendete Klasse `ByteArrayToolbox` stellt hierfür zwei Methoden zur Verfügung, mit denen ein Byte-Array zu einem Array von Strings umgewandelt werden kann, und umgekehrt.

```
// new set
public void newSet(Blob result) throws Exception {
  result = Lob.newBlob();
  byte[] data = new byte[] {0};
  result.getOutputStream().write(data);
  set(1,result);
}

// add an element to set
public void add(String elem, Blob b, Blob result)
throws Exception {
  InputStream inStream = b.getInputStream();
  byte[] data = new byte[(int)b.size()];
  inStream.read(data);

  // decode elements
  String[] elems = ByteArrayToolbox.decode(data);

  // do nothing if element already exists
  int idx = Arrays.binarySearch(elems,elem);
  if (idx>0) {
    set(3,b);
```

```
    return;
  }

  // encode new elements
  String[] newelems = new String[elems.length+1];
  System.arraycopy(elems,0,newelems,1,elems.length);
  newelems[0] = elem;
  data = ByteArrayToolbox.encode(newelems);

  // make a blob
  result = Lob.newBlob();
  OutputStream outStream = result.getOutputStream();
  outStream.write(data);
  set(3,result);
}
```

Anhang E

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 31.Mai 2005

Sebastian W.H. Czymaj

Literaturverzeichnis

[CB02] V.Castelli, L.D.Bergman, u.a.: *„Image Databases – Search and Retrieval of Digital Imagery“*, Verlag John Wiley & Sons, 2002.

[Che76] P.P.-S. Chen: *„The Entity Relationship Model – Toward a Unified View Of Data“*, ACM Transactions on Database Systems, Vol.1, No.1, March 1976, pages 9–36

[Dol04] S.Dolke: *„Umsetzung von Modellen und Methoden bei der Integration spezieller Dokumentenserver in eine IBM Content Manager Umgebung“*, Diplomarbeit, Universität Rostock, Institut für Informatik, 2004;
www.enotehistory.de/db/DA_SebastianDolke_2004.pdf

[FGLX04] J.Fan, Y.Gao, H.Luo, G.Xu: *„Salient Objects: Semantic Building Blocks for Image Concept Interpretation“*, Image and Video Retrieval, 3rd International Conference CIVR, Dublin 2004, Proceedings, Springer-Verlag, 2004

[GV03] R.Göcke, J.Voskamp, E.Krüger, T.Schwinger, I.Bruder, A.Finger, A.Heuer, T.Ignatova: *„Ein System zur Identifikation der Schreiber von historischen Musikhandschriften“*, 4.IuK-Tage Mecklenburg-Vorpommern, Rostock, 18-20.Juni 2003;
www.enotehistory.de/fhg/rgoeckeIuKTage2003.pdf

[Goe03] R.Göcke: *„Building a system for writer identification on handwritten music scores“*, Proceedings of the IASTED International Conference on Signal Processing, Pattern Recognition, and Applications SPPRA 2003, pp.250-255, Rhodes (Griechenland), 30.Juni-2.Juli 2003;
www.enotehistory.de/fhg/rgoeckeSPPRA2003.pdf

[GS00] W.I.Grosky, P.L.Stanchev: *„An image data model“*, Advances in Visual Information Systems, 2000, p.14-25

[Heu97] A.Heuer: *„Objektorientierte Datenbanken – Konzepte, Modelle, Standards und Systeme“*, 2.aktualisierte und erweiterte Auflage, Addison-Wesley, 1997

[HK99] M.Hinz, G.Kappel: *„UML@Work – Von der Analyse zur Realisierung“*, dpunkt.verlag, 1999

[HS00] A.Heuer, G.Saake: *„Datenbanken: Konzepte und Sprachen“*, 2.aktualisierte und erweiterte Auflage, mitp-Verlag, 2000

[HR04] P.Howarth, S.Rüger: *„Evaluation of Texture Features for Content-Based Image Retrieval“*, Image and Video Retrieval, 3rd International Conference CIVR, Dublin 2004, Proceedings, Springer-Verlag, 2004

[IBM02a] IBM Corporation: *„IBM DB2 Universal Database – SQL Reference Volume 1, Version 8“*, WWW, 2002;
www-306.ibm.com/software/data/db2/udb/support/manualsv8.html

[IBM02b] IBM Corporation: *„IBM DB2 Universal Database – SQL Reference Volume 2, Version 8“*, WWW, 2002;
www-306.ibm.com/software/data/db2/udb/support/manualsv8.html

[IBM02c] IBM Corporation: *„IBM Content Manager for Multiplatforms – Modeling Your Data In Content Manager Version 8“*, WWW, September 2002;
www-306.ibm.com/software/data/cm/cmgr/mp/library.html

[IBM02d] IBM Corporation: *„IBM DB2 Universal Database – Application Development Guide: Programming Server Applications“*, WWW, 2002;
www-306.ibm.com/software/data/db2/udb/support/manualsv8.html

[IBM03] IBM Corporation: *„IBM DB2 Content Manager for Multiplatforms – System Administration Guide Version 8 Release 2“*, WWW, Oktober 2003;
www-306.ibm.com/software/data/cm/cmgr/mp/library.html

[IEEE85] IEEE 754-1985: *„IEEE Standard for Binary Floating-Point Arithmetic“*, IEEE Computer Society, 1985

[ISO96] ISO/IEC 14977; 1996(E): *„Information Technology – Syntactic Multilanguage – Extended BNF“*, ISO/IEC International Standard, 1996

[ISO03] ISO/IEC 13249-5:2003: *„SQL Multimedia and Application Packaged (SQL/MM) – Part 5: Still Image“*, ISO/IEC, November 2003

[Mel02] J.Melton, J.-E.Michels, V.Josifivski, K.Kulkarni, P.Schwarz: *„SQL/MED – A Status Report“*, SIGMOD Record, Vol.31, No.3, September 2002

[Mil04] L.Milewski: *„Integration von Clustering/Classification-Techniken in eine objektrelationale Datenbankumgebung“*, Diplomarbeit, Universität Rostock, Institut für Informatik, 2004;
www.enotehistory.de/db/DA_LarsMilewski_2004.pdf

[MSS02] B.S.Manjunath, P.Salembier, T.Sikora, u.a.: *„Introduction to MPEG-7 – Multimedia Content Description Interface“*, Verlag John Wiley & Sons, 2002.

[MVC02] E.Marcos, B.Vela, J.M.Cavero: *„A Metholodical Approach for Object-Relational Database Design using UML“*, Softw Syst Model (2003) 2: 59–72, Springer Verlag, 2003

[Nel05] M.Nelius: *„Entwicklung von Add-Ins für IBM Rational Rose“*, Technische Dokumentation, eNoteHistory Projekt, Mai 2005;
www.enotehistory.de/db/index.html

[OMG03] OMG: *„Unified Modelling Language (UML) Version 2.0“*;
www.omg.org/technology/documents/formal/uml.htm

[Rat00] Rational Software Corporation: *„The UML And The Data Modeling – A Rational Software Whitepaper“*, WWW, 2000;
www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/Tp180.pdf

[Rat01] Rational Software Corporation: *„Using the Rose Extensibility Interface“*, WWW, 2001;
ftp.software.ibm.com/software/rational/docs/v2003/win_solutions/rational_rose/rose_rei_guide.pdf

[SH99] G.Saake, A.Heuer: *„Datenbanken: Implementierungstechniken“*, mitp Verlag, 1999

[Sta99] P.Stanchev: *„General Image Database Model“*, Visual Information and Information Systems, D.Huijmans, A.Smeulders, Lecture Notes in Computer Science 1614, p.29-36, 1999

[Tha00] B.Thalheim: *„ER Modeling with Higher-Order Entity Relationship Model (In a nutshell)“*, Entity-Relationship Modeling - Foundations of Database Technology, Springer, Berlin, 2000;
www.informatik.tu-cottbus.de/˜thalheim/ERinaNutshell.pdf

[Tue03] C.Türker: *„SQL:1999 & SQL:2000, Objektrelationales SQL, SQLJ & SQL/XML“*, dpunkt.verlag, 2003

[Uni03] The Unicode Consortium: *„The Unicode Standard, Version 4.0.1* defined by *The Unicode Standard 4.0“*, Reading, MA, Addison-Wesley, 2003;
www.unicode.org/versions/Unicode4.0.1/

Thesen

1. Für effiziente Speicherung und Retrieval von Digitalen Bildern eignen sich Datenbanken.
2. Das Entity Relationship Model ohne Erweiterungen reicht nicht aus für eine umfassende Bildbeschreibung. MPEG-7 bietet gute Ansätze, ist aber kein Modell sondern eine Speichervorschrift.
3. Eine Modellierung mit der Unified Modeling Language (UML) ermöglicht eine gute Realwelt-Abbildung, Eigenschaften von Bildes können gut spezifiziert werden, da UML auch eine Typisierung sowie Operationen unterstützt.
4. UML eignet sich als Ausgangspunkt für die Modellierung von Bildinhalten. Es bedarf aber einiger Einschränkungen, um ein UML-Diagramm auf Zielplattformen abzubilden.
5. Aufgrund der syntaktischen Komplexität und inkonsequenter Einhaltung von Standards der betrachteten Systeme ist es notwendig, Merkmale digitaler Bilder mit einem generischen Modell zu erfassen.
6. Mit Einschränkungen im verwendeten UML-Sprachumfang ist es möglich das konzeptuelle, generische Modell nahezu automatisch auf das Datenmodell einer Zielplattform abzubilden.
7. Für die Abbildung eines UML-Klassendiagramms auf das Datenmodell des IBM Content Managers ist ein spezieller Client notwendig, der die Nutzung einiger Konstrukte überhaupt erst ermöglicht.
8. Es ist möglich Extraktionsalgorithmen und andere Operationen mittels Stored Procedures oder Methoden in DB2 Server-seitig zu integrieren. Methoden können ebenfalls Clientseitig integriert werden.