DIE ORACLE-SCHNITTSTELLE DES BERLINER 3D-STADTMODELLS

Claus Nagel
Alexandra Stadler

Technische Universität Berlin
Institut für Geodäsie
und Geoinformationstechnik

Kurzfassung: *Als Teil der seiner 3D-Geodateninfrastruktur (GDI), haben wir für die Stadt Berlin eine 3D-Geodatenbank entworfen und umgesetzt. Sie soll für die Speicherung, Verwaltung und den Austausch von räumlichen Informationen über verschiedenste Aspekte der Stadt verwendet werden. Die Datenbank beruht auf dem kommenden internationalen Standard CityGML, der Geoobjekte hinsichtlich ihrer Geometrie, Topologie, Semantik und Erscheinung modelliert. Um einen Zugriff zur Datenbank zu ermöglichen, haben wir ein Administrationstool entwickelt, welches den Import und Export von CityGML Instanzdokumenten beliebiger Größe erlaubt. Durch die Multithreading-Architektur ist das Werkzeug zudem den Möglichkeiten aktueller Mehrfachprozessoren angepasst.*

In diesem Beitrag soll ein Überblick vermittelt werden, welche Arbeitsschritte notwendig sind, um eine Geodatenbank dieses Formats zu entwickeln - von der Vereinfachung des zu bis hin zur Entwicklung des Administrationswerkzeugs. Wichtige Entwurfsentscheidungen werden dabei besonders hervorgehoben.

1 Einleitung

Wie viele andere deutsche Bundesländer, ist auch Berlin gerade damit beschäftigt, eine Geodateninfrastruktur aufzubauen. Neben klassischen 2D-Karteninformationen, ist für viele Anwendungen im öffentlichen und privaten Sektor auch die 3. Dimension von besonderer Bedeutung. Als Beispiele seien hier Stadt- und Regionalplanung, Architektur, Tourismus, 3D Kataster, Umweltsimulationen, Telekommunikation, Katastrophenmanagement und Navigation genannt. Für die effiziente Verwaltung von 3D-Geoinformationen einer Stadt wie Berlin muss eine gemeinsame Sammelstelle geschaffen werden, welche das Zusammentragen, Vergleichen, Anpassen, Fortführen und Austauschen von Daten beliebiger Herkunft ermöglicht. Voraussetzung dafür ist ein grundlegendes Datenmodell, welches eine einheitliche Strukturierung der Daten sicherstellt.

Entwicklerforum Geoinformationstechnik 2008. Shaker Verlag ISBN: 978-3-8322-7393-4

Diese Situation hat zu der Entwicklung eines standardisierten Datenmodells und Austauschformats für 3D-Stadtmodelle geführt: CityGML. Basierend auf dem internationalen Standard GML 3.1.1 (4) des Open Geospatial Consortiums (OGC) definiert es Klassen und Relationen für die grundlegenden Objekte einer Stadt bezüglich ihrer Geometrie, Topologie, Semantik und Erscheinung [11]. Durch die Modellierung von thematischen Informationen geht CityGML weit über die Möglichkeiten üblicher Austauschformate für 3D-Stadtmodelle hinaus: so wird die Exploration virtueller 3D-Stadtmodelle zu Kinderspiel - von einfachen thematischen Abfragen bis zu anspruchsvollen Analysen wie man sie beispielsweise für Simulationen durchzuführen hat.

Nimmt man CityGML als einheitliches Format für die Speicherung von relevanten 3D-Geoinformationen, muss eine strukturierte Sammelstelle geschaffen werden, welche eine optimale Bearbeitung aller zur Verfügung stehender Daten ermöglicht. Für Berlin wurde diese Aufgabe im Rahmen des Projektes *„Geodatenmanagement in der Berliner Verwaltung - Amtliches 3D-Stadtmodell für Berlin“* gelöst. Die darin gesammelten Daten dienen als Grundlage für städtebauliche Studien, den Vergleich von Planungsvarianten, die Berechnung von Sichtverbindungen, die Simulation von Grünraumveränderungen und ganz allgemein für semantische Datenexploration. Die resultierenden Stadtmodellierungen können in digitaler Form, aber auch als 3-dimensionale Präsentationsmodelle Investoren, Politikern sowie der breiten Öffentlichkeit zur Verfügung gestellt werden.

2 Eine 3D-Geodatenbank für Berlin

Die 3D-Geodatenbank wurde in Form einer räumlichen relationalen Datenbank (Oracle 10g_R2 Spatial) umgesetzt. In einer ersten Projektphase hat das Institut für Geodäsie und Geoinformationstechnik der Universität Bonn einen Datenbank-Prototyp entwickelt, der nur die grundlegendsten Konzepte von CityGML implementierte [1]. Unter anderem wurden Gebäude nur bis Level of Detail 3 (LOD3) umgesetzt. In der zweiten Projektphase wurde das existierende Datenbankschema an die aktuelle Version von CityGML (0.4.0, Version des angenommenen OGC Best Practices Paper, [11]) angepasst - diesmal in vollem Umfang. So werden jetzt Gebäude inklusive Innenraummodellierung und Adressierung sowie diverse andere thematische Module (Oberflächeneigenschaften, Gewässer, Verkehrsnetz, etc.) unterstützt.

Im Detail erbt die Datenbank von CityGML folgende wichtige Eigenschaften:

- **Fünf verschiedene Detaillierungsstufen (Levels of Detail, LODs)**

  Nach der Idee der Multirepräsentation kann jedes Geoobjekt (auch Digitale Geländemodelle und Luftbilder) in fünf verschiedenen Detaillierungsstufen gespeichert werden. Mit steigendem LOD erhalten Objekte eine genauere und feiner aufgelöste Geometrie und Semantik.

- **Oberflächendaten**

  Zusätzlich zu Geometrie und Semantik können Objekte auch Oberflächeneigenschaften (= Information über die beobachtbaren Eigenschaften der Oberfläche eines Objektes) speichern. Entgegen allgemeiner Vermutungen, werden hierunter nicht nur Farbinformationen im sichtbaren Bereich verstanden. Im Gegenteil, sollen mit diesem Werkzeug beliebige oberflächenbasierte Themen wie Infrarotintensität als Maß für die Wärmeabstrahlung, Lärmemission, Feuchtigkeit, etc. abgebildet werden. Die Speicherung geschieht in Form von Texturen, welche mit den Elementen der Objektoberfläche verknüpft werden.

- **Komplexe digitale Geländemodelle (DGMs)**

  DGMs könne in der 3D-Geodatenbank in vier verschiedenen Varianten dargestellt werden: als reguläre Raster, Dreiecksvermaschungen (TINs), 3D-Massenpunkte oder 3D-Bruchkanten. Für jeden LOD kann ein komplexes DGM als Kombination einer beliebigen Anzahl an DGM-Einzelkomponenten unterschiedlichen Typs gebildet werden.

- **Komplexe thematische Modellierung**

  CityGML modelliert alle wichtigen Objekte einer Stadt in einer hierarchischen Struktur, die Zerlegungen in Unterobjekte und dadurch unterschiedliche Detaillierungsstufen erlaubt. Ein Gebäude beispielsweise kann seine Zerlegung in einzelne Räume speichern - sowohl auf Seite der Semantik als auch auf Seite der Geometrie. Zusätzlich hat jedes Objekt die Möglichkeit eine Vielzahl von Attributen vorzuhalten. Diese reichhaltige thematische Information ermöglicht die Durchführung komplexer Abfragen, Analysen und Simulationen.

- **Abbildung von generischen und prototypischen 3D-Objekten**

  Generische Objekte (Prototypen) werden für die speichereffiziente Repräsentation von Elementen einer Stadt verwendet, welche in gleicher Form an verschiedenen Orten vorkommen. Beispiele sind Stadtmöbel wie Laternen, Straßenschilder oder Bänke. Jede Instanz eines solchen Objektes kann in jedem LODs auf einen bestimmten Prototyp verweisen.

- **(Rekursive) Gruppierung von Geoobjekten**

  Geoobjekte können beliebig gruppiert werden. Die gruppierten Objekte stellen dann wiederum eigenständige Geoobjekt dar. Und können beliebige Namen und Attribute enthalten. Auch Objektgruppen können wieder gruppiert werden, was zu einer rekursiven Gruppierungshierarchie beliebiger Tiefe führt.

- **Referenzierung von externen Datenquellen**

  Geoobjekte besitzen oft korrespondierende Objekte in anderen Datensätzen (Originalgeometrie, Sachinformationen, etc.). Um diese Zusammenhänge weiter sichtbar zu halten, ist die Speicherung von Referenzen zu externen Datenquellen möglich. Jede Referenz besteht aus der Adresse der entsprechenden Datenquelle und der ID des externen Objektes im referenzierten Datensatz.

- **Flexible 3D-Geometrien**

  Die Geometrie von 3D-Objekten basiert hier auf der Boundary-Representation [8]. Sie ermöglicht die Organisation von Objekten als Kombination von Solids und Surfaces, sowie (rekursiven) Aggregationen dieser Objekte.

Die Vorgängerversion der Datenbank hat dem grundlegenden Datenmodell zwei Aspekte hinzugefügt, welche über die Fähigkeiten von CityGML hinausgehen. Diese Aspekte wurden in der aktualisierten Datenbankversion aufrechterhalten:

- **Verwaltung von DGMs und großen Luftbildern**

  In der Datenbank können Luftbilder beliebiger Größe gespeichert werden. Die Oracle 10g_R2 Spatial GeoRaster-Funktionalität bietet die Möglichkeit, gekachelte, homogene Luftbilder zu großen Gesamtbildern zu aggregieren. Um Luftbilder sowie DGMs zu importieren und exportieren steht ein spezielles Tool zur Verfügung, welches auf der WebService-Technologie beruht. Es erlaubt die beidseitige Konvertierung zwischen georeferenzierten TIFF-Bildern und dem datenbankinternen Format.

- **Versionsmanagement und Verlaufsgeschichte**

  Die Aufgabe eines Versionsmanagements und der Speicherung einer Verlaufsgeschichte wird vom Oracle Workspace Manager übernommen. So können auf einfache, transparente Art und Weise verschiedene Arbeitsstände verwaltet werden und sind von Applikationen, die auf der Datenbank laufen zugreifbar [13].

Die Weiterentwicklung des Datenbankschemas in der zweiten Projektphase war stark von den Erfahrungen der ersten Projektphase geprägt. Die Teile des Datenbankprototyps, auf denen bereits WebServices oder andere Applikationen aufsetzten wurden rückwärtskompatibel gestaltet.(beispielsweise wurde die Modellierung von Rasterdaten nicht verändert, da dort ein Import- und Exporttool existiert).

Im Folgenden wird ein Überblick über die Aufgaben bei der Entwicklung der 3D-Geodatenbank für Berlin gegeben. Wichtige Designentscheidungen werden im Detail beschrieben. Entsprechend der Reihenfolge an Arbeitsschritten, ist der Beitrag in folgende Abschnitte unterteilt (gekennzeichnet als (a), (b) und (c) in Abb.1):

(a) **Vereinfachung des Datenmodells von CityGML** (Abschnitt 3)

  Um die Geschwindigkeit von Datenimport und -export zu erhöhen, wurde das komplexe Datenmodell von CityGML an einigen kritischen Stellen merkbar vereinfacht.

(b) **Ableitung des relationalen Datenbankschemas** (Abschnitt 4)

  Das vereinfachte objektorientierte Datenmodell wurde dann in einem weiteren Schritt auf ein relationales Datenbankschema abgebildet. Um den Datenzugriff zu optimieren, wurde die notwendige Anzahl von Tabellen-Joins durch entsprechende Tabellenstrukturierung minimiert. Für die Modellierung des Datenbankschemas und die anschließende automatische Ableitung von SQL-Skripten für die Datenbankerzeugung wurde das an Oracle gekoppelte Programm JDeveloper verwendet.

(c) **Entwicklung eines Import- und Exporttools** (Abschnitt 5)

Für den Import und Export von CityGML-Instanzdokumenten wurde ein zusätzlich ein Datenbankadministrationstool entwickelt. Es unterstützt die Prozessierung von sehr großen Dateien (> 4GB - also nicht mehr vollständig im Hauptspeicher ablegbar). Die Multithreading-Architektur kommt Multiprozessorensystemen und Multi-Core-CPUs zu Gute.

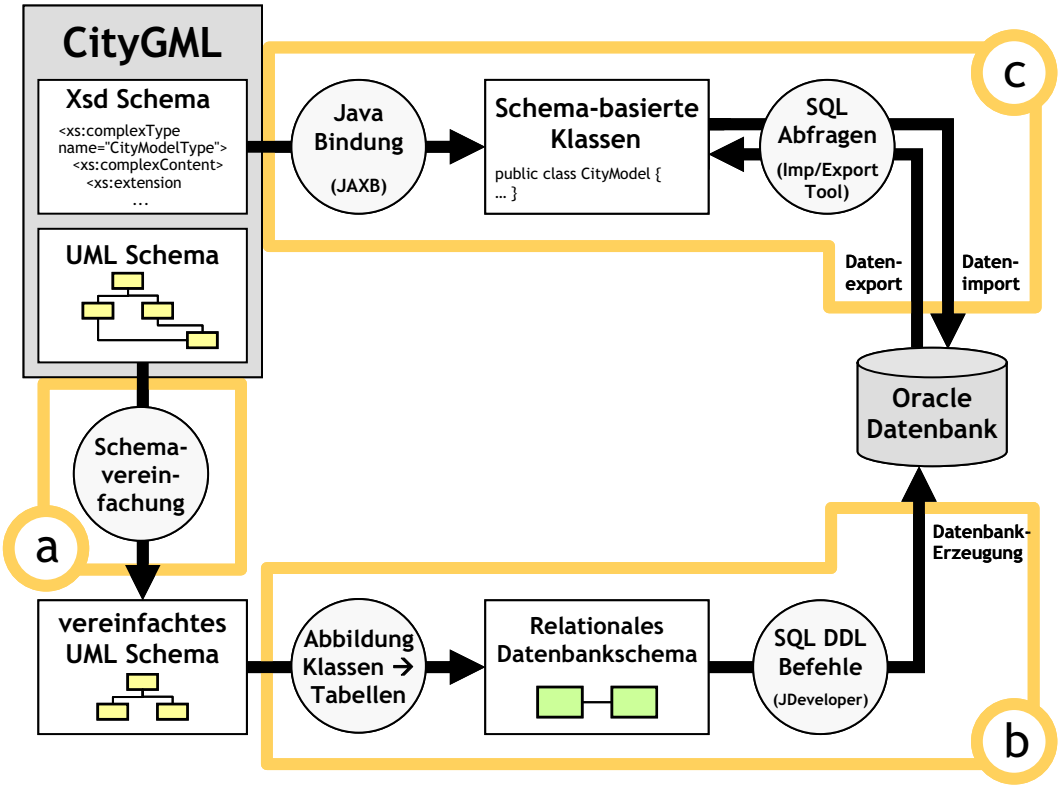

Abbildung 1: Aufgaben bei der Entwicklung der Berliner 3D-Geodatenbank:
(a) Vereinfachung des Datenmodells von CityGML,
(b) Ableitung des relationalen Datenbankschemas und
(c) Entwicklung eines Import/Exporttools für CityGML-Instanzdokumente.

3 Vereinfachung des Datenmodells von CityGML

Um die interoperable Verwaltung von Geodaten sicherzustellen, müssen ein Sammelplatz mit dem Ziel der konsistenten Datenhaltung sowie eine standardisierte Austauschmöglichkeit existieren. Für Berlin wurde daher eben vorgestellte 3D-Geodatenbank entwickelt, welche auf der aktuellen Version von CityGML (0.4.0, Version des angenommenen OGC Best Practices Paper, [11]) basiert. Die von CityGML verwendeten Konzepte werden von Anwendern und Softwareherstellern als ziemlich komplex angesehen - sie dienen allerdings der flexiblen Datenmodellierung.

Verschiedene Aspekte könnten dabei als komplex interpretiert werden:

- Das **allgemeine thematische Modell** hinter CityGML deckt ein breites Spektrum an Anwendungsfeldern ab. Um CityGML voll zu unterstützen, musste bisher das gesamte Datenmodell abgebildet werden, was unter Umständen zu übertriebenem Implementationsaufwand führen kann. Vermeiden lässt sich das durch eine Modularisierung von CityGML. In der neuen Version (1.0.0, Version der zur Standardisierung eingereichten Implementation Specification, [12]) wird die Aufteilung des Datenmodells in ein Kernmodul und mehrere Erweiterungsmodule vorgeschlagen. Implementationen müssen dann nicht mehr das gesamte Datenmodell abbilden, sondern können sich auf ausgewählte Teilmodule festlegen. Sie bleiben damit trotzdem CityGML-konform. Die Modularisierung hat auch einen weiteren Vorteil: durch die transparente Modulfestlegung ist schon vor dem Austausch klar, welche Informationen ein Datensatz enthält.

- Auf der **Ebene einzelner thematischer Module** trifft man auf komplexe Relationen zwischen Objekten. Sie ermöglichen den variablen Detaillierungsgrad der Modellierung. Erstens basiert die semantische Komponente von CityGML auf einer hierarchischen Struktur. Geoobjekte können unterschiedlich feingliedrig dargestellt werden und die feineren Strukturen ergeben sich aus Unterteilungen gröberer Strukturen. Ein Gebäude beispielsweise kann in Gebäudeteile, Räume, Wände, Fenster und Türen zerlegt werden. Zweitens existieren einige zyklische Relationen, die zu beliebig verschachtelten Objektstrukturen führen können. Ein Beispiel hier ist die rekursive Zerlegung von Gebäudeteilen in weitere Gebäudeteile in weitere Gebäudeteile... Ein anderes wichtiges Beispiel ist die Generalisierungsbeziehung („generalizesTo“) die für jedes Stadtobjekt (CityObject) gilt. Sie erlaubt die Verlinkung von korrespondierenden Objekten in unterschiedlichen LODs. Drittens existieren Mehrfachaggregationen zwischen thematischen Klassen und ihren geometrischen Eigenschaften. Für jedes Objekt können auf diese Weise gleichzeitig verschiedene geometrische Repräsentationen vorgehalten werden (Multirepräsentationsgedanke, [3]).

- Auf der **Ebene der Geometrie** finden sich vergleichbare Hierarchien wie jene seitens der Thematik. CityGML unterstützt eine Teilmenge des in GML definierten Geometriemodells. Die Einschränkung liegt in der Unterstützung von lediglich polygonalen Geometrien, die auf unterschiedlichste Arten aggregiert und verschachtelt werden können. Im besten Fall entspricht die Zerlegung auf geometrischer Seite jener auf thematischer Seite.

Eine zentrale Eigenschaft von CityGML ist die Möglichkeit komplexe Sachverhalte zu modellieren. Wenn es jedoch um die Datensammlung in großem Rahmen (z.B. für eine große Stadt) geht, bieten sich einige Schemavereinfachungen an um Datenbankanfragen zu beschleunigen. Für Berlin wurde daher ein vereinfachtes Datenmodell gebildet, welches später die Grundlage für die Ableitung eines relationalen Datenbankschemas darstellte. Folgende Modifikationen wurden durchgeführt:

- **Vereinfachte Behandlung von Rekursionen**

  Rekursive Datenbankabfragen sind sehr zeitaufwändig - speziell wenn a priori nicht bekannt ist, mit wie vielen Rekursionen zu rechnen ist. Um dennoch hohe Performanz zu gewährleisten, wird für die Abbildung von Rekursionen eine einfache aber dennoch leistungsfähige Methode angewandt: Jedes Objekt speichert Referenzen auf sein direktes Eltern- und Wurzelelement. Letzteres ist von besonderem Interesse wenn es um ganz grundlegende Abfragen geht, wie z.B. jene nach den Teilelementen eines bestimmten Gebäudes. Dann müssen lediglich alle Elemente ausgegeben werden, welche die ID des Gebäudes als Wurzelelement speichern. Die Operation lässt sich sowohl auf thematischer als auch auf geometrischer Seite durchführen. Durch die explizite Verknüpfung mit den direkten Elternelementen, kann die gesamte Aggregationshierarchie ohne Informationsverlust wieder nachgebaut werden.

  Alternative Abbildung des GML-Geometriemodells

  In CityGML werden räumliche Eigenschaften von Elementen als Objekte des GML3 Geometriemodells repräsentiert. Es besteht aus Geometrieprimitiven, welche kombiniert werden können zu so genannten Complexes, Composites und Aggregates. CityGML beschränkt sich auf eine Teilmenge des GML3 Geometriepaketes, welche nur die Repräsentation polygonaler Geometrien behandelt. Im Speziellen handelt es sich dabei um Punkte (Points), Linien (LineStrings), Flächen (Polygons), Körpern (Solids) und allen gültigen Kombinationen wie z.B. CompositeSurfaces oder MultiSolids. Um Topologie und Oberflächeneigenschaften dieser Objekte abbilden zu können, muss es möglich sein, alle Geometrieteile einzeln zu identifizieren - auch wenn sie Teil einer kombinierten Geometrie sind. Räumliche Datenbanken stellen typischerweise Datentype für alle vorher erwähnten Geometrietypen zu Verfügung (inklusive kombinierter Geometrien). Die Verwendung dieser Typen erlaubt dann auch die Verwendung von inhärenten räumlichen Abfragen. Unglücklicherweise erlauben übliche Datenbankimplementierungen kombinierter Geometrien keine Zuweisung von Namen oder Referenzen für einzelne Teilgeometrien. Sie sind also nicht mehr einzeln ansprechbar und dadurch unzulänglich für die Abbildung entscheidender Relationen in CityGML. Dieses Problem wurde durch eine alternative Geometriemodellierung gelöst (Abb.2), welche alle polygonalen 2D und 3D-Geometrien abbildet, die in GML verfügbar sind. Alle Körper und Flächen sind aus einzeln ansprechbaren Polygonen aufgebaut, denen explizite Koordinaten zugeordnet sind. Sie sind vom Typ Polygon - eine nativer Datentyp, der die Ausführung von räumlichen Abfragen ermöglicht. Polygone können auf verschiedene Weisen zu BRepAggregates kombiniert werden. Auch diese Aggregate sind eindeutig identifizierbar. Die eigens eingeführte Attribute geben Auskunft darüber, von welchem Typ eine Aggregation ist: isTriangulated bezeichnet triangulierte Oberflächen, isSolid unterscheidet zwischen 2D und 3D-Geometrien (Surfaces bzw. Solids) und isComposite bestimmt ob es sich um eine Aggregation

im eigentlichen Sinn (z.B. MultiSolid) oder eine Komposition (z.B. CompositeSolid) handelt. Die rekursive Relation bRepMember ermöglicht die mehrfache Verschachtelung von kombinierten Geometrien.

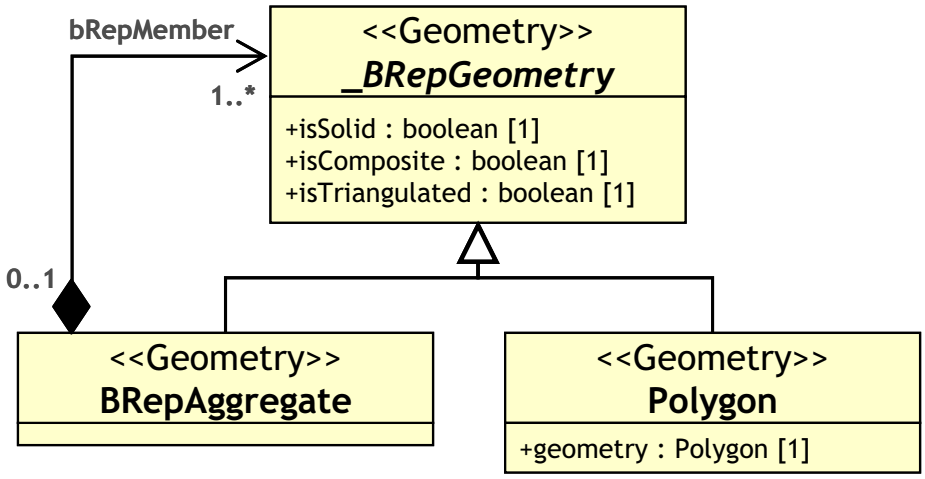

Abbildung 2: Vereinfachte Modellierung der GML Geometrieklassen.

- **Projektspezifische Klassen und Klassenattribute**

  In der Berliner Datenbank werden zusätzlich/alternativ zu den in CityGML vorgesehenen Klassen noch Orthofotos, spezifische Metadaten, Versionskontrollen und vereinfachte Attribute für die Speicherung von Adressen benötigt. Diese wurden dem Datenmodell hinzugefügt.

- **Anpassung der Datentypen** Um eine effizientere Darstellung in der Datenbank zu erzielen, wurden einige der in CityGML spezifizierten komplexen Datentypen durch einfachere Datentypen ersetzt. Codelisten, Farbvektoren und Matrizen beispielsweise wurden durch Strings ersetzt - für die Abgrenzung einzelner Werte wurden eindeutige Trennzeichen definiert; beliebige GML Geometrietypen werden wie vorher erläutert durch einfache Polygone abgebildet (unter Verwendung des Oracle-spezifischen Datentyps SDO_GEOMETRY).

- **Einschränkung der Multiplizität von Klassenattributen**

  Attribute mit einer uneingeschränkten Anzahl von Instanzen (*) werden in der Datenbank entweder als ein Feld mit vordefinierter Anzahl von Einträgen oder mittels eines Datentyps der die Speicherung von beliebig vielen Werten innerhalb eines Objektes erlaubt (z.B. String mit Definition von eindeutigen Trennzeichen). Nur dann können die entsprechenden Attribute in einer einzigen Datenbankspalte abgebildet werden.

- **Kardinalitäten und Typen von Relationen**

  Um Relationen der Kardinalität n:m in einer Datenbank darzustellen, ist eine zusätzliche Tabelle notwendig, die Paare zugeordneter Objekt-IDs enthält. Mit der

Vereinfachung zu 1:n oder n:1-Relationen könnte das Anlegen dieser zusätzlichen Tabellen vermieden werden. Daher wurden alle n:m-Relationen in CityGML auf eine mögliche restriktivere Definition untersucht - mit dem Ergebnis, dass einige Vereinfachungen in der Kardinalität aber auch im Typ der Relationen durchgeführt werden konnten. Ein Beispiel ist die ehemalige n:m-Aggregation zwischen Räumen (Rooms) und Einrichtung (Furniture). Sie wurde in eine 1:n-Komposition umgewandelt, da man verlangen kann, dass Einrichtungsgegenstände immer einem bestimmten Raum zugeordnet werden.

4 Ableitung des relationalen Datenbankschemas

Um aus einem objekt-orientiertes Datenmodell wie CityGML eine relationale Datenbankstruktur anzuleiten, müssen alle Objektklassen des Datenmodells (Features und Geometry) auf entsprechende Datenbanktabellen abgebildet werden. Realisiert wird das durch ein relationales Datenbankschema mit speziellen Datentypen für die Speicherung von Geometrien. Bei der Verbindung von objekt-orientierten und relationalen Techniken kommt es oft zu Reibungsverlusten, die auch als object-relational impedance mismatch [2] bezeichnet werden. Das objekt-orientierte Paradigma beruht auf Prinzipien des Software Engineerings bei denen Objekte entlang ihrer Relationen durchlaufen werden. Das relationale Paradigma hingegen baut auf mathematischen Operationen auf, die kombinatorische Vereinigungen von Datensätzen aus unterschiedlichen Tabellen erlauben. Man kann sich vorstellen, dass es oft schwierig oder gar unmöglich ist, eine 1:1-Abbildung zwischen diesen so unterschiedlichen Konzepten herzustellen. Die folgenden Beispiele sollen ein Gefühl dafür vermitteln, wo die Knackpunkte in der Ableitung eines relationalen Datenbankschemas liegen.

Abb.2 zeigt das bereits in Abschnitt 3 beschriebene alternative Design der GML Geometrieklassen. Die abstrakte Klasse _BRepGeometry stellt die Basisklasse für alle Geometrieobjekte innerhalb der Datenbank dar. Sie besitzt zwei abgeleitete Klassen: Polygon enthält explizite Koordinaten und BRepAggregate sorgt für die Darstellbarkeit kombinierter Geometrien. Um so eine Vererbungshierarchie im Datenmodell auf explizite Datenbanktabellen abzubilden, kann man zwischen drei grundlegend verschiedenen Ansätzen wählen [16]:

- **Abbildung der gesamten Klassenhierarchie auf eine einzige Tabelle:**

  Vererbungshierarchien beliebiger Tiefe lassen sich auf eine einzige Tabelle abbilden, indem man die Attribute aller beteiligten Klassen in dieser einen Tabelle vereint. Standardmäßig wird der Name der Basisklasse als Tabellenname weiterverwendet (Abb.3). Die Tabelle muss mit zwei zusätzlichen Attributen ausgestattet werden: ID und Typ. Die ID definiert den Primärschlüssel der Tabelle, während der Typ angibt, welcher Klasse im Datenmodell ein Eintrag entspricht. In unserem Beispiel könnten die Typen Polygon oder BrepAggregate angenommen werden - der Datensatz würde also entweder Koordinaten oder Aggregationsinformationen enthalten.

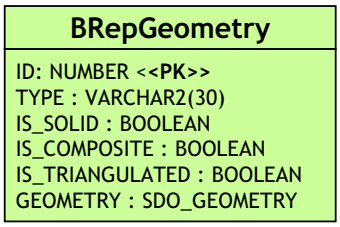

Abbildung 3: Abbildung der Klassenhierarchie aus Abb.2 auf eine einzige Tabelle. Die Relation bRepMember wurde in diesem Beispiel außer Acht gelassen.

- **Abbildung jeder konkreten Klasse auf eine eigene Tabelle**

  Ein alternativer Ansatz ist die Erzeugung von Tabellen für jede konkrete Klasse. Die Tabellen enthalten dann die Attribute der korrespondierenden Klasse und aller abstrakten Elternklassen. In unserem Beispiel werden lediglich die Klassen Polygon und BrepAggregate auf Tabellen abgebildet. Beide übernehmen die Attribute isSolid, isComposite und isTriangulated der gemeinsamen abstrakten Elternklasse _BRepGeometry (Abb.4). Wie im vorherigen Ansatz müssen auch hier ID-Attribute als Primärschlüssel eingeführt werden.

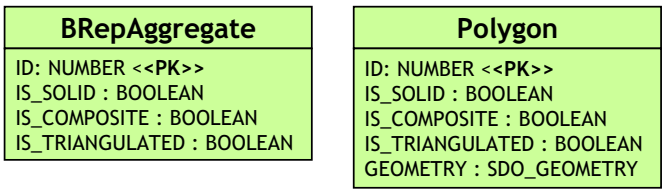

Abbildung 4: Jede konkrete Klasse aus Abb.2 wird auf eine eigene Tabelle abgebildet. Die Relation bRepMember wurde in diesem Beispiel außer Acht gelassen.

- Jede Klasse wird auf eine eigene Tabelle abgebildet Der einfachste, aber leider auch ineffizienteste Ansatz, ist die Abbildung jeder (auch abstrakten) Klasse auf eine eigene Tabelle. Die Beziehungen zwischen Kindklassen und ihren Elternklassen werden über Fremdschlüssel gespeichert: Die ID der Elternklasse fungiert als Fremdschlüssel für die Kindklasse(n). In unserem Beispiel zeigen die ID-Attribute der Tabellen Polygon und BrepAggregate auf das ID-Attribut der Tabelle *_BRepGeometry* (Abb.5).

  Da Kindklassen die Attribute ihrer Elternklassen erben, erfordert dieser Ansatz die Vereinigung (Join) zahlreicher Tabellen um alle Informationen über ein Objekt zu sammeln. Tabellenjoins sollen aber generell vermieden werden, da sie sehr zeitaufwändig sind.

Während die meisten generischen Applikationen wie beispielsweise Snowflake's GO-Loader (Snowflake Software 2008) einen der letzten beiden Ansätze verfolgen, ähnelt die Umsetzung der Geometrie in der Berlin Datenbank sehr dem ersten Ansatz (Abb.6).

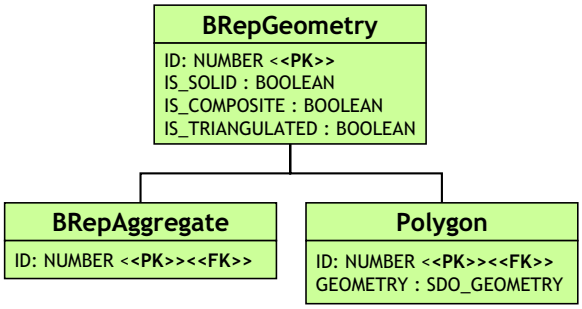

Abbildung 5: Jede Klasse aus Abb.2 wird auf eine eigene Tabelle abgebildet. Die Relation bRepMember wurde in diesem Beispiel außer Acht gelassen.

Durch die Vermeidung von überflüssigen Tabellen werden ineffiziente Joins umgangen und damit die Performance der Datenbank gesteigert. Wie in Abschnitt 3 (Vereinfachte Behandlung von Rekursionen) beschrieben, wird auch hier die rekursive Relation bRepMember (Abb.2) durch die Attribute PARENT_ID und ROOT_ID ersetzt.

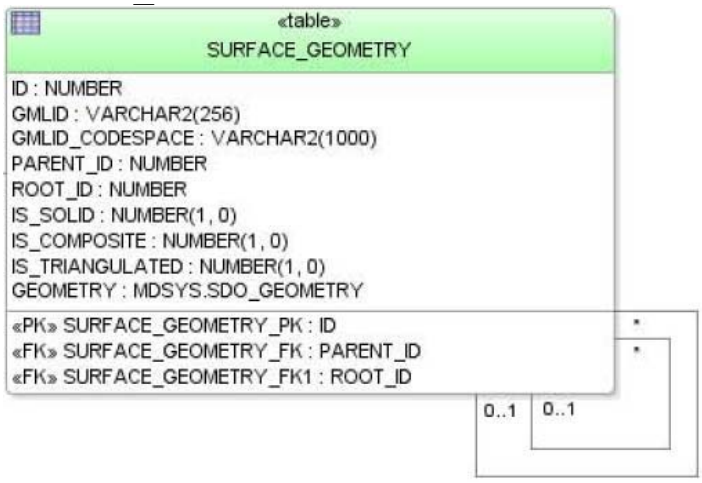

Abbildung 6: Implementierung von BRep-Geometrie in der Berliner 3D-Geodatenbank.

Die Abbildung von CityGML's semantischer Klassenstruktur auf Datenbanktabellen greift hingegen auf alle drei Ansätze zurück. Die Analyse von zu erwartenden Datenbankanfragen hat zur Identifizierung von zentralen Klassen geführt, welche als eigene Tabellen umgesetzt wurden. Die wichtigsten Vertreter sind CityObject, CityModel, SurfaceData und große thematische Klassen wie PlantCover oder AbstractBuilding. Die verbleibenden Klassen wurden entweder aufgrund ihrer ähnlichen Struktur (z.B. BuildingInstallation and IntBuildingInstallation) oder aufgrund eben beschriebener Vererbungshierarchie zu gemeinsamen Tabellen zusammengefasst (z.B. Road, Track, Railway und Square zu TransportationComplex).

Die übergeordnete Basisklasse Feature hat keine eigens assoziierte Tabelle. Anstelle dessen wurden die Attribute GMLID, GML_NAME und NAME_CODESPACE in

die direkten Kindtabellen übernommen. Zusätzlich zu GMLID wurde das Attribut GMLID_CODESPACE eingeführt. Es enthält entweder einen benutzerdefinierten Wert oder (standardmäßig) den vollständigen Pfad zur Quelle des originalen CityGML-Instanzdokumentes. Der Codespace wird benötigt da GMLIDs nur innerhalb eines Instanzdokumentes eindeutig sind. Durch die Kombination von ID und Codespace erreicht man die gewünschte Eindeutigkeit innerhalb der gesamten Datenbank. Im Fall der Klasse CityObject wurden die Attribute GML_NAME und NAME_CODESPACE eine Ebene entlang der Vererbungshierarchie hinunter geschoben. Der Grund dafür war die Untersuchung von typischen Abfragen - sucht man beispielsweise ein Objekt der Klasse Building namens „Brandenburger Tor“, würde die Speicherung des GML-Namens (GML_NAME) direkt in CityObject einen zusätzlichen Tabellen-Join erzwingen.

Zu guter Letzt wurde noch das Attribut CLASS_ID zur Tabelle CityObject hinzugefügt. Es erleichtert die weitere Verarbeitung von Abfragen, die sich direkt auf CityObject beziehen. Das betrifft z.B. die Suche nach bestimmten GML_IDs, die Einschränkung der räumlichen Ausdehnung (durch die Definition einer Bounding Box) oder die Abfrage von Metainformation. Durch das Attribut CLASS_ID weiß jedes selektierte CityObject, welcher Objektklasse es angehört und damit kann ein direkter Zugriff zur entsprechenden Tabelle hergestellt werden. Tabellen-Joins werden dadurch aber leider nicht vermieden.

5 Entwicklung eines Import- und Exporttools

Die Anwendung für den Import und Export von CityGML-Objekten stellt die grundlegende Funktionalität zum Befüllen der Datenbank mit den räumlichen Daten eines 3D-Stadtmodells zur Verfügung. Die Unterstützung von CityGML-Instanzdokumenten beliebiger Dateigröße sowie die Nebenläufigkeit der Datenverarbeitung durch softwareseitiges Multithreading stellen die wichtigsten Eigenschaften der Anwendungsentwicklung dar. In Abb.7 ist eine Übersicht der Umsetzung des Import/Export-Werkzeugs dargestellt. Die Abbildung ist in drei horizontale Bereiche gegliedert. Der mittlere Bereich zeigt den Prozess der Datenbindung des CityGML-XSD-Schemas an ein Java-Objektmodell, auf welchen in Abschnitt 5.1 eingegangen wird. Der obere und untere Bereich skizzieren den Import- bzw. Exportprozess von CityGML-Datensätzen, die in den Abschnitten 5.2 und 5.3 erläutert werden.

Ein besonderes Augenmerk wird während des gesamten Datenimports und -exports auf die Verwaltung von Objekt-IDs gelegt, da nur eine systematische und sorgfältige Verwendung weltweit eindeutiger IDs die konsistente Fortführung der Geo-Objekte in der Datenbank ermöglicht. GML-IDs erfüllen dieses Kriterium nur bedingt, da sie optional und auch nur im Kontext eines einzelnen Dokumentes eindeutig sind. Um dieses Problem zu lösen werden zwei alternative Strategien während des Imports von Objekten verfolgt: entweder werden allen Objekten neue und weltweit eindeutige IDs (UUIDs) zugewiesen, oder aber nur solchen Objekten, für welche keine GML-ID modelliert wurde. In beiden Fällen wird für jedes Feature zusätzlich das Attribut GMLID_CODESPACE, welches in Abschnitt 4 beschrieben wurde, geführt.

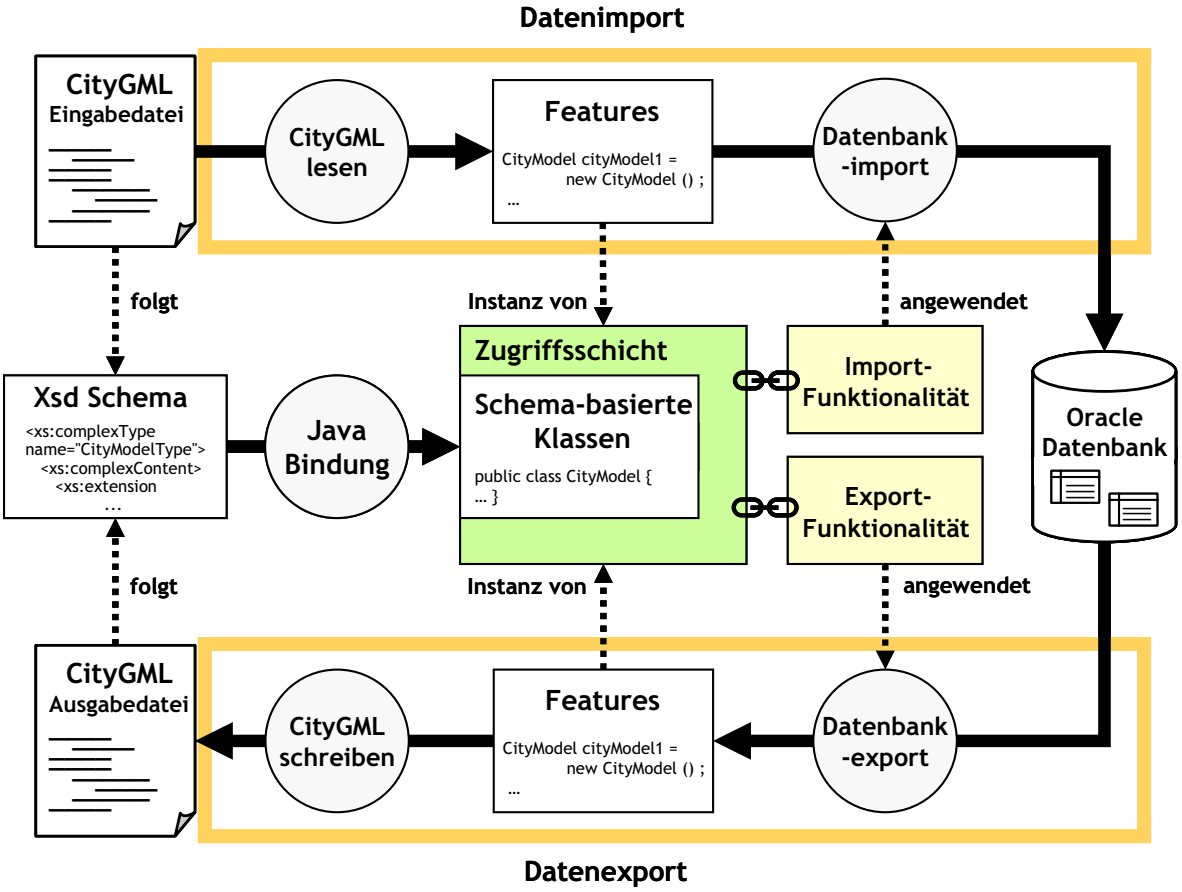

Abbildung 7: Datenimport und -export im Überblick.

5.1 Software-Entwurf

Das Import/Export-Werkzeug ist als Java-Anwendung umgesetzt. Der Software-Entwurf orientiert sich an zwei maßgeblichen Entwicklungszielen: an der Unterstützung auch größter CityGML Instanzdokumente einerseits, sowie an einer hochperformanten Datenverarbeitung andererseits. Zu diesem Zweck wird eine Strategie zur partiellen Verarbeitung von XML-Dokumenten verfolgt. Darüber hinaus bildet das Konzept des softwareseitigen Multithreadings die Basis der gesamten Softwarearchitektur.

Unterstützung auch größter CityGML-Instanzdokumente

Die Unterstützung auch größter XML-Dokumente (> 4GB) wird durch ein Datenstrom-basiertes, partielles Verarbeiten von XML-Daten sowie deren Bindung an Java-Klassen realisert. Dieser Ansatz beruht auf zwei bereits existierenden Java-Frameworks für das Einlesen und Schreiben von XML. Zum einen kommt die Java Architecture for XML Binding (JAXB) zum Einsatz, welche eine objekt-orientierte Sicht auf XML-Daten ermöglicht. Zum anderen erfolgt das Datenstrom-basierte und ereignisorientierte Parsen von XML-Dokumenten unter Verwendung der Simple API for XML (SAX). Beide Me-

thoden verfolgen grundsätzlich unterschiedliche Konzepte der XML-Verarbeitung, die jeweilige Vor- und Nachteile mit sich bringen.

Das JAXB Framework bietet eine einfach zu handhabende Programmierschnittstelle zur Validierung und Prozessierung von XML. Es beruht auf der Abbildung einer XML-Schema-Instanz auf ein Java-Objektmodell. Diese Bindung erlaubt es, XML-Inhalte durch entsprechende Instanzen der erzeugten Klassenstruktur zu repräsentieren. Die Schema-Bindung erfolgt anhand vorgegebener Regeln, auf welche durch Binding Customizations Einfluss genommen werden kann. Das resultierende Objektmodell bildet die Struktur von Instanzdokumenten daher besser ab als universelle Ansätze wie das XML Document Object Model (DOM) oder auch SAX. Für die Kommunikation mit dem Objektmodell stellt JAXB zwei wesentliche Schnittstellen bereit: das Marshalling überführt einen Java-Objekt-Baum gemäß den Bindungsregeln in ein XML Dokument, während das Unmarshalling den umgekehrten Vorgang umsetzt. Von den zugrundeliegenden Operationen auf Dateiebene wird hierbei abstrahiert. JAXB bietet somit eine High-level-API für die XML-Verarbeitung, die es ermöglicht, im Rahmen der Softwareentwicklung auf die eigentliche Geschäftslogik und deren objekt-orientierten Entwurf zu fokussieren. Um ein XML-Dokument mit JAXB verarbeiten zu können, muss dieses jedoch zuvor komplett in den Hauptspeicher eingelesen werden. Damit begrenzt im Wesentlichen die Größe des Hauptspeichers die maximale Dateigröße der XML-Dokumente.

Im Gegensatz hierzu ermöglichen Datenstrom-basierte Parser wie SAX den sequentiellen Zugriff auf XML-Dokumente. Die einzelnen XML-Elemente werden in der Reihenfolge ihres Auftretens im Dokument an die Anwendung weitergeleitet. Hierbei arbeitet SAX im Gegensatz zu JAXB oder DOM ereignisorientiert. Für jedes XML-Element werden benutzerdefinierte Ereignisse ausgelöst, indem entsprechend vordefinierte Callback-Methoden aufgerufen werden. Der Speicherverbrauch eines SAX-Parsers basiert somit in erster Linie auf dem Speicherbedarf der einzelnen XML-Elemente, und entspricht demnach nur einem Bruchteil der gesamten Dateigröße. Allerdings sind SAX-Ereignisse zustandslos. Vorhergegangene Ereignisse werden weder referenziert noch stehen sie in einem sonstigen Verhältnis. Das Erkennen einer syntaktischen Struktur oder gar deren Überführung in eine objekt-orientierte Repräsentation wird durch den Low-level-Ansatz von SAX stark erschwert.

Die Import/Export-Anwendung setzt ein zweistufiges Verfahren ein, um die Vorteile beider Ansätze auszunutzen, sowie ihre jeweiligen Nachteile auszugleichen. In einer ersten Stufe werden die Eingangsdaten mittels eines SAX-Parsers eingelesen und in Einzelstücke (XML-Chunks) zerteilt. Alle SAX-Ereignisse, die zu einem einzelnen XML-Chunk gehören, werden in einem Zwischenspeicher aufgezeichnet. Die Inhalte der einzelnen Zwischenspeicher werden in der zweiten Stufe mittels der JAXB-API in Java-Objekte umgesetzt. Für das Schreiben von XML-Dokumenten wird das Verfahren umgekehrt.

Der vorgestellte Ansatz erlaubt es, die Zwischenspeicher nach Verarbeitung der XML-Chunks freizugeben. Dementsprechend ist der Speicherverbrauch nicht mehr abhängig von der Gesamtgröße des XML-Dokuments, sondern nur noch von der Datenmenge, die zum gleichen Zeitpunkt in verschiedenen Zwischenspeichern vorghehalten wird. Letzteres kann jedoch abhängig von Systemgegebenheiten beeinflusst werden. Darüber hinaus

ermöglicht JAXB die objekt-orientierte Sicht auf die XML-Chunks und somit deren effiziente Weiterverarbeitung. Die sinnvolle Aufteilung eines CityGML Instanzdokuments in einzelne XML-Chunks kann anhand von <cityObjectMember>-Elementen erfolgen, die Toplevel-Feature innerhalb eines Stadtmodells repräsentieren.

Die Geschäftlogik der Import/Export-Anwendung bedient sich schließlich einer weiteren Schicht, um auf die generierten JAXB-Klassen zuzugreifen. Diese Wrapper-Schicht abstrahiert von Datenbindungsdetails der JAXB-Klassen, um neben CityGML Version 0.4.0 auch zukünftig Version 1.0.0 unterstüzen zu können. Beiden Versionen liegen unterschiedliche XML-Schema-Definitionen und somit unterschiedliche JAXB-Datenbindungen zugrunde.

Nebenläufigkeit der Datenverarbeitung

Die Softwarearchitektur des Import/Export-Werkzeugs basiert auf Nebenläufigkeit, d.h. auf der (quasi-)parallelen Ausführung mehrerer interagierender Prozesse. Grundsätzlich werden einzelne Programmteile des Import/Export-Prozesses, wie etwa die partielle Verarbeitung von XML-Dokumenten, das (Un-)Marshalling von JAXB-Objekten, ihre Weiterverarbeitung sowie die Datenbankkommunikation durch separate Threads (dt. Ausführungsstränge) verarbeitet. Die hierdurch vorgenommene Entkopplung von rechenintensiven und I/O-intensiven Prozessen sowie ihre nebenläufige, nicht - blockierende Ausführung führt grundsätzlich zu einer erhöhten Geschwindigkeit der Programmausführung. Während Einprozessorsysteme die Nebenläufigkeit durch ein schnelles Umschalten der Prozesse umsetzen, erlauben Mehrprozessor- und Mehrkernsysteme ihre tatsächliche Parallelisierung.

Die Ausführungsgeschwindigkeit lässt sich allerdings nicht beliebig durch nebenläufige Prozesse steigern. Im Gegenteil erfordert eine zu hohe Anzahl gleichzeitig aktiver Threads ein Übermaß an Lebenszyklus- und Ressourcenverwaltung aufgrund von Kontextwechseln. So kann die Erzeugung zu vieler Threads innerhalb einer JVM beispielsweise zu einem hohen Arbeitsspeicherverbrauch und damit zu Thrashing führen, wodurch die Systemleistung dramatisch sinkt. Aus diesem Grund werden Threads im Rahmen des Import/Export-Werkzeugs für gleichartige Aufgaben nicht jeweils neu generiert, sondern in einem Threadpool vorgehalten und wiederverwendet. Dieses Threadpool-Modell erlaubt es, den Aufwand der Thread-Erzeugung über viele gleichartige Aufgaben hinweg zu verteilen. Verzögerungen aufgrund der Erzeugung von Threads entfallen, da diese im Threadpool bereits zur Verfügung stehen, sobald die zu bearbeitende Aufgabe anfällt. Dies ermöglicht kurze Antwortzeiten und eine hohe Performanz der Anwendung.

Das Threadpool-Modell ist umgesetzt als Warteschlange (engl. queue) mit einer festen Gruppe assoziierter Arbeiter-Threads (engl. worker threads, vgl. Abb.8). Die Implementierung der Warteschlange folgt hierbei dem Entwurfsmuster der „blocking queue“, welches typischerweise im Rahmen nebenläufiger Programmierung zum Einsatz kommt: ein Thread erzeugt Objekte (die einzelnen abzuarbeitenden Aufgaben entsprechen) und stellt diese in die Warteschlange ein. Von dort werden die Objekte von den mit der Warteschlange assoziierten Arbeiter-Threads entnommen und abgearbeitet. Um Verklemmungen (engl. deadlocks) zwischen den Threads während des konkurrierenden

Zugriffs auf die gemeinsame Ressource der Wartschlange zu vermeiden, ist der Zugriff gemäß dem Erzeuger-Verbraucher-Entwurfsmuster synchronisiert und somit blockierend [9]. Die Abarbeitung der einzelnen Aufgaben hingegen erfolgt asynchron.

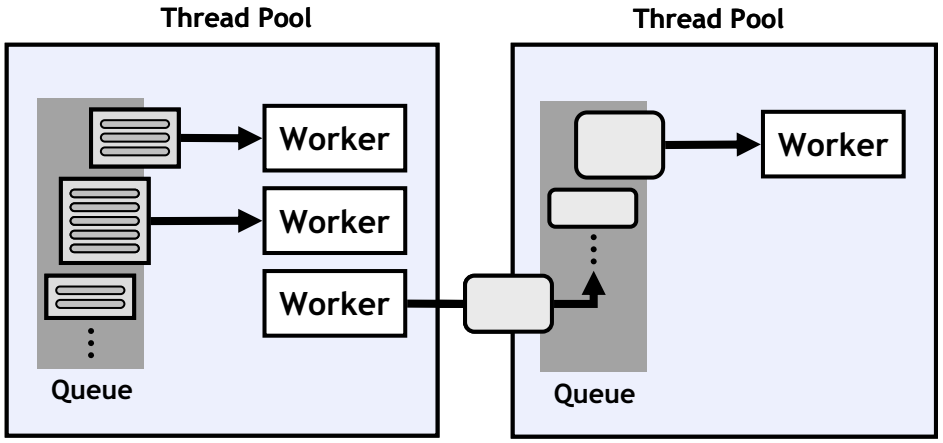

Abbildung 8: Threadpools sind umgesetzt als Warteschlange (engl. queue) mit einer festen Gruppe assoziierter Arbeiter-Threads (engl. worker threads).

Mehrere Threadpools können auf einfache Weise zu gesamten Prozessketten kombiniert werden. Jeder einzelne Schritt einer solchen Prozesskette wird hierbei durch einen separaten Threadpool abgebildet. Die Arbeiter innerhalb eines Threadpools führen parallel diejenigen Aufgaben aus, welche auf die ihnen zugewiesene Warteschlange gestellt werden. Das Resultat ihrer Datenverarbeitung stellen sie dem nachfolgenden Threadpool innerhalb der Prozesskette zur weiteren Bearbeitung zur Verfügung, indem sie es in der entsprechenden Warteschlange einstellen. Somit wird eine Entkopplung der einzelnen Prozessschritte anhand einer wohldefinierten Schnittstelle erreicht. Zusätzliche Prozessschritte können aufgrund dieser einheitlichen Schnittstelle auf einfache Weise hinzugefügt werden. Die Interprozesskommunikation zwischen den einzelnen Prozessschritten wird innerhalb des Import/Export-Werkzeugs durch einen eigenen Nachrichtendienst ermöglicht. Dieser ist als eigenständiger Threadpool mit nur einem einzelnen Arbeiter-Thread umgesetzt, und erlaubt sowohl den synchronen wie asynchronen Nachrichtentransfer entsprechend dem Publisher-Subscriber-Entwurfsmuster [9].

Die optimale Anzahl an Arbeiter-Threads innerhalb eines Threadpools hängt in erster Linie von der Anzahl verfügbarer Prozessoren bzw. Prozessorkerne sowie von der Art der zu bearbeitenden Aufgabe, etwa rechenintensiv oder I/O-intensiv, ab und variiert daher je nach System. Aus diesem Grund ermöglicht das Import/Export-Werkzeug zum einen die Angabe von benutzerdefinierten Grenzwerten für jeden Threadpool. Des Weiteren unterstützt es darüber hinaus die Anpassung der Anzahl aktiver Arbeiter-Threads durch die Threadpools selbst, beispielsweise auf Basis ihrer aktuellen Auslastung. Schließlich ist die Größe der jeweiligen Warteschlangen ein entscheidender Faktor für den Hauptspeicherverbrauch und ist dementsprechend auch durch den Benutzer anpassbar.

5.2 Importprozess

Der Ablauf des Imports eines CityGML-Datensatzes in die Datenbank ist in Abb. 9 dargestellt. Die entsprechende Prozesskette ist softwareseitig durch die Verkettung von Threadpools umgesetzt, die einzelne Schritte des Importprozesses abdecken. Dies umfasst Threadpools für das SAX-basierte Einlesen des Eingangsdatensatzes in XML-Chunks, die Umwandlung der XML-Chunks in JAXB-Objekte, welche Toplevel-Feature in CityGML repräsentieren, die Verarbeitung dieser Objekte sowie ihre Speicherung in der Datenbank. Die mit den Threadpools verbundenen Warteschlangen sind ebenfalls in der Abb. 9 dargestellt

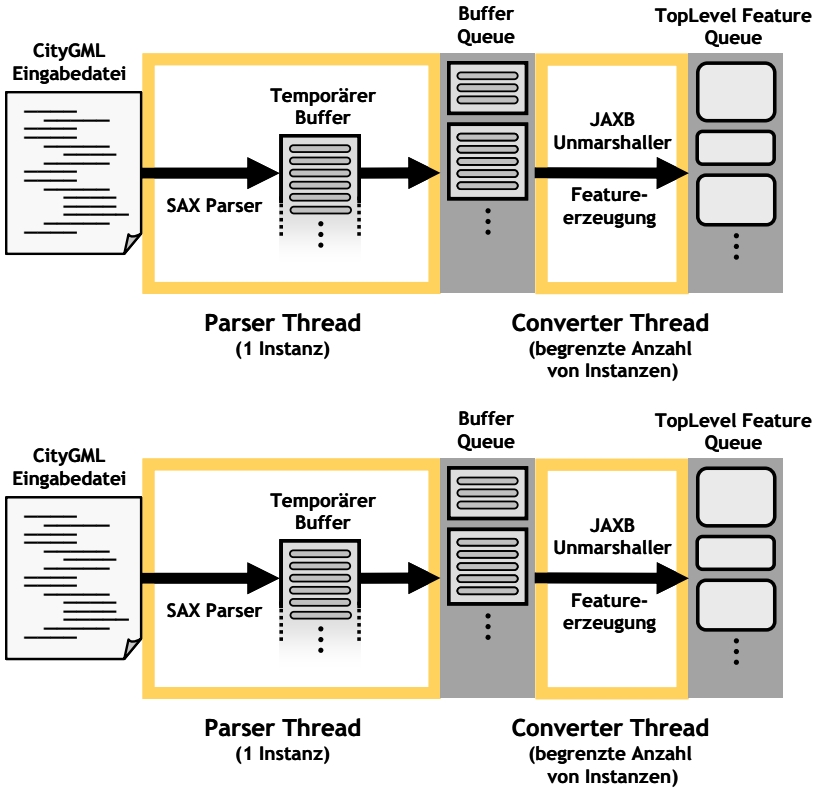

Abbildung 9: Import Werkzeug (Stufe 1) – Detaillierter Ablauf des Importprozesses vom Einlesen des CityGML Datensatzes bis hin zur Speicherung in der Datenbank (ohne die Auflösung von XLinks).

Die ersten beiden Schritte der Prozesskette (Verarbeitung des Eingangsdatensatzes anhand von XML-Chunks sowie deren Überführung in JAXB-Objekte) wurden bereits in Absatz 5.1 dargestellt. Durch die Analyse der resultierenden JAXB-Objekte werden durch den „Importer-Thread“ im anschließenden Prozessschritt diejenigen SQL-Befehle abgeleitet, welche für die Abbildung der CityGML-Objekte auf entsprechende Tabellen des relationalen Datenbankschemas erforderlich sind. Dieser Vorgang kann durch benutzerdefinierte Importfilter beeinflusst werden. So lässt sich der Import etwa auf bestimmte CityGML-Featureklassen oder auf Feature innerhalb einer geographischen

Region beschränken. Die gefilterten Objekte werden ohne Weiteres übergangen und der durch sie belegte Hauptspeicher freigegeben.

Um die Antwortzeiten der Datenbank zu optimieren, werden die generierten SQL-Befehle vorkompiliert und in entsprechenden Java-Objekten (sog. PreparedStatements) innerhalb des „Importer-Threads“ vorgehalten. Vorkompilierte SQL-Befehle ermöglichen die mehrmalige Ausführung desselben SQL-Befehls mit unterschiedlichen Werten. Datenbankseitig entfällt hierbei die Übersetzung jedes einzelnen SQL-Befehls, wodurch die unterschiedlichen Wertbelegungen ohne Verzögerung verarbeitet werden können. Darüber hinaus werden die vorkompilierten SQL-Befehle nur dann an die Datenbank weitergeleitet, wenn eine entsprechende Anzahl an SQL-Befehlen erreicht wurde. Dies erlaubt ihre performante Stapelverarbeitung (engl. batch processing) durch die Datenbank. Die optimale Anzahl an SQL-Befehlen variiert hierbei je nach Komplexität der Elemente des CityGML-Instanzdokuments und ist folglich benutzerdefiniert einstellbar.

Aufgrund des nebenläufigen Entwurfs des Import-Werkzeugs werden die dargestellten Prozessschritte stets parallel ausgeführt. Dementsprechend blockieren etwa Threads, die auf eine Antwort der Datenbank warten, nicht das Parsen des XML-Dokuments, und mehrere JAXB-Objekte können zur gleichen Zeit verarbeitet werden. Allerdings erhöht das Chunk-basierte Parsen der Eingangsdaten auch die Komplexität des Importprozesses. In CityGML-Instanzdokumenten können bestimmte Eigenschaften einzelner Feature, wie etwa Relationen zu anderen Feature- oder Geometrieobjekten, unter Verwendung des XLink-Konzepts von GML3 angegeben werden. Demzufolge enthält das XML-Element, welches das Feature repräsentiert, die entsprechende Eigenschaft nicht als Kindelement, sondern verweist hierfür per Referenz auf ein entferntes Objekt, das über seine GML-ID identifiziert wird. Die korrekte Abbildung des CityGML-Elements in der Datenbank erfordert folglich auch die Beachtung des referenzierten Objektes. Neben Vorwärtsreferenzen sind hierbei auch Rückwärtsreferenzen innerhalb eines Dokuments erlaubt. Ein einstufiger Ansatz zur Auflösung von Rückwärtsreferenzen erfordert, dass das gesamte XML-Dokument im Hauptspeicher verfügbar ist. Dies ist jedoch durch die partielle Verarbeitung der Eingangsdaten nicht gegeben.

Zur Lösung des XLink-Problems setzt das Import-Werkzeug eine zweistufige Strategie um. In einem ersten Durchlauf (Abb. 9) werden die einzelnen CityGML-Feature ohne Beachtung etwaiger XLinks in die Datenbank importiert. Besitzt ein Feature eine XLink-Referenz, so wird diese durch einen separaten Threadpool in einer temporären Datenbanktabelle festgehalten. Ein entsprechender Tabelleneintrag umfasst das referenzierende Element sowie die referenzierte GML-ID. Darüber hinaus verzeichnet das Import-Werkzeug jede gelesene GML-ID und die zugehörige Objektrepräsentation in der Datenbank. In einem zweiten Durchlauf wird nur noch der Inhalt der temporären Datenbanktabelle abgefragt. Da zu diesem Zeitpunk das gesamte XML-Dokument bereits einmal verarbeitet ist, können gültige Referenzen aufgelöst werden, indem anhand der referenzierten GML-ID die entsprechenden Objekte gefunden werden. Diese zweite Stufe des Importprozesses ist in Abb. 10 illustriert.

Durch diesen zweistufigen Importprozess können die Vorteile des Chunk-basierten Parsens der Eingangsdaten weiterhin ausgeschöpft werden, während gleichzeitig die korrekte Abbildung der CityGML-Elemente in der Datenbank gewährleistet ist.

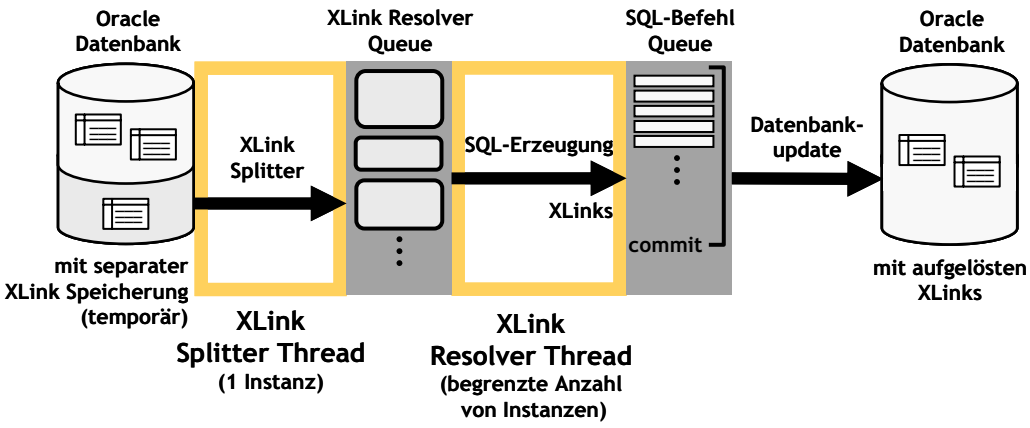

Abbildung 10: Import Werkzeug (Stufe 2) Auflösung von XLink-Referenzen.

5.3 Exportprozess

Der Ablauf des Exports eines CityGML-Datensatzes aus der Datenbank ist in Abb.11 dargestellt. Entsprechend dem Importprozess werden auch die Teilschritte des Datenexports durch separate Threadpools realisiert.

In einem ersten Prozessschritt erfolgt die Abfrage der Datenbank anhand benutzerdefinierter Selektionskriterien. Die Exportfilter sind identisch mit den bereits dargestellten Importfiltern, und erlauben etwa räumliche Abfragen, die durch entsprechende hochperformante Funktionalitäten von Oracle 10g_R2 Spatial realisiert werden. Die Abfragen werden in diesem ersten Schritt nur auf einer einzelnen Datenbanktabelle durchgeführt, welche allgemeine Informationen der gespeicherten Feature enthält. Hierdurch können die Abfragen sehr schnell verarbeitet und die entsprechenden Resultate an den „Export-Thread“ (vgl. Abb.11) weitergeleitet werden.

Sobald die ersten Ergebnisse der Datenbank zurückgeliefert werden, stellt sie der „Splitter-Thread“ in die Warteschlange des sich anschließenden „Export-Thread“ ein. Auf diese Weise können dessen Arbeiter bereits mit der Rekonstruierung der CityGML-Objekte beginnen, während der „Splitter-Thread“ parallel weitere Ergebnisse der anfänglichen Datenbankanfrage verarbeitet.

In Abhängigkeit des zu rekonstruierenden CityGML-Features führt der „Export-Thread“ zusätzliche komplexere Datenbankabfragen durch, welche sich über verschiedene Tabellen erstrecken können, um die erforderlichen Objektdaten zu erhalten. Die Exportfunktionalität stellt den entsprechenden Programmcode für jeden Featuretyp zur Verfügung. Bis zu diesem Stadium erfolgt der Großteil der Datenaufbereitung und -verarbeitung aufgrund effizienter SQL-JOIN-Befehle auf Seiten des Datenbankservers. Weiterhin erlaubt der nebenläufige Entwurf des Export-Werkzeugs, dass einzelne Arbeiter-Threads nicht durch Threads blockiert werden, die auf eine Antwort der Datenbank warten. Sobald alle Objektdaten vorliegen, werden sie auf entsprechende JAXB-Objekte abgebildet. Während dieses Vorgangs verzeichnet auch das Export-Werkzeug alle bereits gelesenen GML-IDs, um XLink-Referenzen wiederherstellen zu können. Das

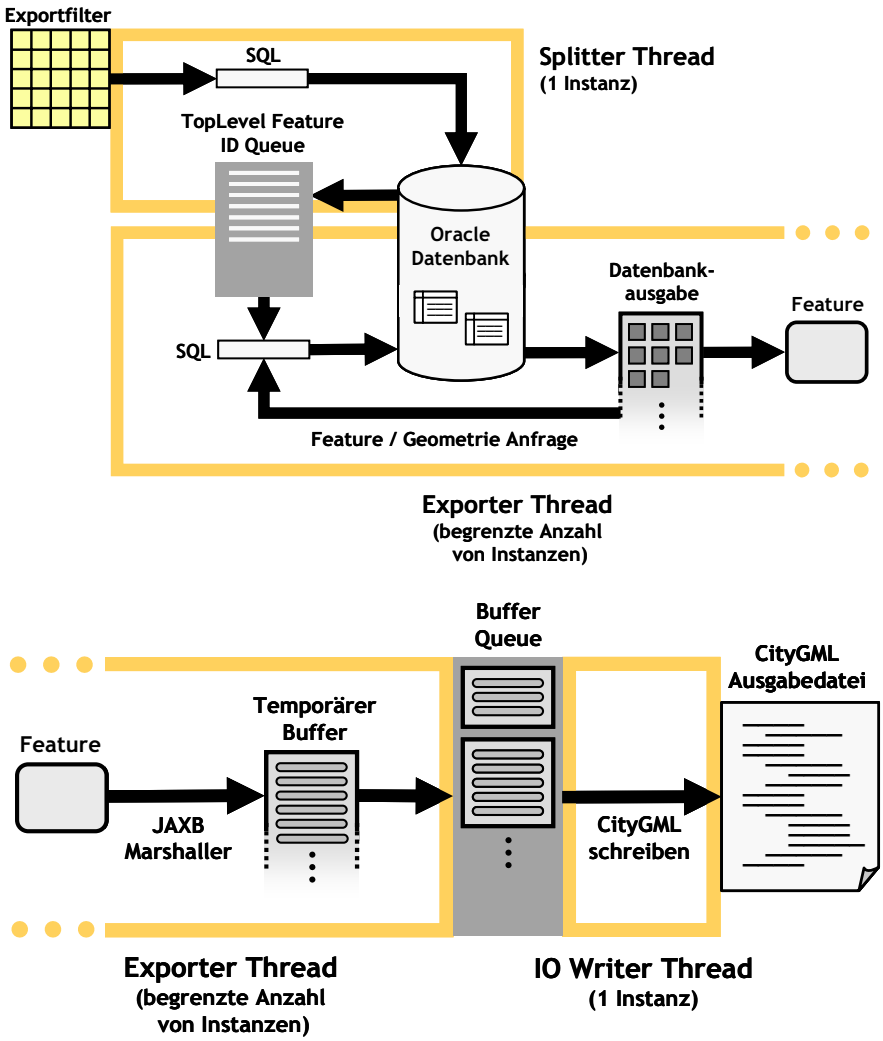

Abbildung 11: Export Werkzeug – Detaillierter Ablauf des Exportprozesses von der Datenbankabfrage bis zum Schreiben des CityGML-Instanzdokuments.

Marshalling der JAXB-Objekte erzeugt schließlich aufgrund der Schema-basierten Datenbindung SAX-Ereignisse, die in Zwischenspeichern aufgezeichnet werden. Die Inhalte dieser Zwischenspeicher werden in der Warteschlange des darauf folgenden Threadpools platziert.

Der letzte Schritt des Exportprozesses besteht im Schreiben aller Feature-Daten in ein CityGML-Instanzdokument und wird durch den „I/O-Writer-Thread“ ausgeführt. Dieser Threadpool entnimmt seiner Warteschlange die zwischengespeicherten SAX-Ereignisse und überführt sie in entsprechende XML-Elemente. Abermals sind die entsprechenden Dateioperationen komplett von allen anderen Schritten der Prozesskette entkoppelt.

5.4 Performanz

Die Performanz der Import/Export-Werkzeugs wurde noch nicht vollständig durch Messungen erfasst und kann daher zu diesem Zeitpunkt nicht adäquat belegt werden. Erste Tests zeigen jedoch einen sehr guten Datendurchsatz: In einer Testumgebung bestehend aus einem Intel DualCore 6750 Prozessor-System für die Import/Export-Anwendung und einem Datenbankserver mit Intel Dual Xeon-Prozessor, benötigte der Import eines 7GB CityGML-Datensatzes mit mehr als 1 Million LOD1 Gebäudemodellen weniger als 90 Minuten. Der Export war nach 10 Minuten beendet.

6 Verwandte Arbeiten

CityGML ist ein allgemeines Informationsmodell für die Repräsentation und den Austausch von urbanen 3D-Objekten bzw. ganzen Stadt- oder Landschaftsmodellen. Es wurde entwickelt als ein System das 3D-Geodaten genügend Freiheit in ihrer geometrischen, topologischen und semantischen Entfaltung gibt. Die Geometrie und Semantik eines Stadtmodells kann also ganz flexibel strukturiert werden - vom reinen geometrischen Datensatz ohne semantische Information bis zu komplexen topologisch korrekten und räumlich-semantisch kohärenten Daten. Ein Datensatz wird als räumlich-semantisch kohärent bezeichnet, wenn 1:1-Beziehungen zwischen semantischen Objekten und den korrespondierenden Geometrieelementen bestehen [18]. CityGML definiert also ein Datenmodel und Austauschformat, welches in diversen Phasen der 3D-Stadtmodellierung Anwendung findet - von der Geometrieerfassung, Prüfung der Datenqualität und Datenverfeinerung bis hin zur Aufbereitung für spezifische Anwendungen. Der Lebensweg von 3D-Geodaten kann also trotz mehrfacher Datenanreicherung mitverfolgt und zu jeder Zeit im Sinne eines Datenaustauschs verlustfrei weitergegeben werden.

CityGML ist implementiert als Anwendungsschema für GML 3.1.1 (4), dem erweiterbaren internationalen Standard für Datenaustausch des Open Geospatial Consortium (OGC) und der ISO TC211. Es basiert außerdem auf einer Reihe von Standards aus der ISO 191xx Familie, des OGC, des W3C Konsortiums, des Web 3D Konsortiums und von OASIS [11].

Das OGC bietet des Weiteren eine Reihe von Web Service Standards bzw. Vorschlägen um Geoapplikationen interoperable zu gestalten, welche größtenteils auf GML basieren. Nachdem auch CityGML auf GML basiert, eignet es sich problemlos als Backend für Web Services wie den Web Feature Service [19], den Web Perspective View Service oder den Web 3D Service [15]. Ein WFS stellt eine standardisierte Schnittstelle zu GML-basierten Geodaten dar. Anwendungen haben dadurch die Möglichkeit, interaktiv auf WFS-konforme Daten zuzugreifen und Selektionen in Form von Filtern anzuwenden. Der WFS stellt dabei nicht nur Daten auf Objektebene zur Verfügung, sondern beschreibt das zu Grunde liegende Schema um saubere Datenevaluierung sicherzustellen. Ein WPVS liefert keine Objekte, sondern gerenderte Ansichten der 3D-Geodaten. Durch den WPVS erlaubt Thin-Clients ohne eigene Renderingfähigkeiten dennoch die Darstellung größter Szenen wie sie in Berlin3D erzeugt werden. Ein W3DS konvertiert

Geodaten in Szenenbeschreibungen wie sie Clients mit Renderingfähigkeiten benötigen. Er erlaubt Symbolisierungsinformation um Szenenbeschreibungen benutzergerecht bzw. anwendungsgerecht zu gestalten.

Die Speicherung von Geodaten in Datenbanken ist ein bekanntes Problem. Neben proprietären und maßgeschneiderten Lösungen, gibt es einige Anwendungen, die aus beliebigen GML-Applikationsschemata automatisch generische Datenbankschemata ableiten. Die resultierenden Datenbanken können dann gleichermaßen für die Speicherung und Beareitung und Weitergabe von Geoobjekten verwendet werden. Zwei solche Produkte sind GO Loader ([17], [14]) und CPA SupportGIS (CPA Geo-Information 2008). Beide ermöglichen manuelles Editieren der abgeleiteten Datenbankschemata und die Verwendung unterschiedlichster Datenbanksysteme. Für die Berlin Datenbank konnten diese generischen Verfahren nicht verwendet werden, da Rückwärtskompatibilität und Schemaanpassung gefordert waren.

Ähnlich zu der in diesem Paper beschriebenen Situation, haben Emgård and Zlatanova [7] die Ableitung einer relationalen Datenbank aus einem 3D-Informationsmodell erläutert. Sie besprechen zwei unterschiedliche Ansätze, von denen einer auf räumliche und der andere auf thematische Abfragen optimiert ist.

7 Fazit und Ausblick

Die vorgestellte 3D-Geodatenbank wurde im Rahmen des Projektes *„Geodatenmanagement in der Berliner Verwaltung - Amtliches 3D-Stadtmodell für Berlin“* [6]. Im Gegensatz zu früheren Versionen, handelt es sich nicht mehr um eine Implementierung, die nur auf projekt-spezifische Bedürfnisse zugeschneidert ist. Die Wahl, ein standardisiertes Format wie CityGML als Basis für die Datenbankentwicklung heranzuziehen, macht es Angehörigen unterschiedlichster Fachbereiche möglich, sich an der Speisung der Datenbank zu beteiligen. Stadtplaner, Architekten, Vermesser, u.a. können so ein gemeinsames Stadtmodell aufbauen, das reich an semantischer Information ist. Dieses Konzept wird noch weiter unterstützt durch den Entschluss, das gesamte Softwarepaket rund um die Datenbank noch 2008 als Open Source zu veröffentlichen. Nach Projektabschluss wird es unter der LPGL Version 2.1 der Öffentlichkeit zur Verfügung stehen. Mehr Information kann der GNU Lesser General Public License entnommen werden [10].

In einem nächsten Schritt wird das Datenbankadministrationstool mit zusätzlicher Matchingfunktionalität ausgestattet. Wenn beispielsweise zwei Gebäude in der Datenbank dasselbe Gebäude in der Wirklichkeit darstellen, soll das automatisch erkannt und die beiden Gebäude verlinkt werden. Über diese Verbindungen können dann thematische Informationen ausgetauscht und automatische Updateprozeduren angestoßen werden. Im Fall äquivalenter Geometrien sollte des Weiteren eines der beiden Objekte gelöscht werden, um Inkonsistenzen in der Datenbank zu vermeiden.

Dank des zu Grunde liegenden Objektmodells, eignet sich die Datenbank hervorragend als Backend für Web Feature Services. Diese können auf existierenden Import- und Exportcode zurückgreifen und stellen dann eine Alternative für die graphische Benutzeroberfläche in Form einer standardisierten computergerechten Schnittstelle dar.

Unabhängig davon könnte die Exportfunktionalität um zusätzliche Formate wie z.B. KML erweitert werden. Abschließend sollte noch erwähnt werden, dass es immer noch Raum für Performanceoptimierungen gibt. Mögliche Ansätze wären hier die Aufteilung in logische Tabellen und Indizes sowie die optimale Verteilung auf physischen Platten.

Danksagung

Die Berliner 3D-Geodatenbank wurde im Auftrag der Berliner Senatsverwaltung für Wirtschaft, Arbeit und Frauen sowie der Berlin Partner GmbH realisiert. Die Entwicklung beruht auf der vorangegangenen Arbeit des Instituts für Geodäsie und Geoinformation (ehemals: Instituts für Kartographie und Geoinformation), Universität Bonn. Das relationale Datenbankschema wurde in Zusammenarbeit mit der Firma 3DGeo GmbH erstellt.

Literatur

[1] 3D city database (2007), www.3dcitydb.org [letzter Zugriff: 2008-06-20].

[2] Ambler SW, The Object-Relational Impedance Mismatch (2008), http://www. agiledata.org/essays/impedanceMismatch.html [letzter Zugriff: 2008-06-20].

[3] Cecconi A (2003), Integration of Cartographic Generalization and Multi-Scale. Databases for Enhanced Web Mapping, Dissertation, Zürich 2003.

[4] Cox S, Daisey P, Lake R, Portele C, Whiteside A (2004), OpenGIS Geography Markup Language (GML) Implementation Specification, Version 3.1.1, OGC Doc. No. OGC 03-105r1, Open Geospatial Consortium 2004.

[5] CPA Geo-Information, SupportGIS Produktseite (2008), www.supportgis.de [letzter Zugriff: 2008-06-20].

[6] Döllner J, Kolbe TH, Liecke F, Sgouros T, Teichmann K (2006) The Virtual 3D City Model of Berlin - Managing, Integrating, and Communicating Complex Urban Information, In: Proceedings of the 25th Urban Data Management Symposium UDMS, Aalborg 2006.

[7] Emgård L, Zlatanova S (2007) Implementation alternatives for an integrated 3D Information Model, In: Advances in 3D Geoinformation Systems, Springer-Verlag, pp. 313-330.

[8] Foley J, van Dam A, Feiner S, Hughes J (1995) Computer Graphics: Principles and Practice. Addison Wesley, 2nd Edition.

[9] Gamma E, Helm R, Johnson RE (1995) Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Amsterdam, 1995.

[10] GNU Lesser General Public License, http://www.gnu.org/copyleft/lgpl.html [letzter Zugriff: 2008-06-20].

[11] Gröger G, Kolbe TH, Czerwinski A (2007), Candidate OpenGIS Implementation Specification (City Geography Markup Language), Version 0.4.0, OGC Doc. No. 07-062, Open Geospatial Consortium 2007.

[12] Gröger G, Kolbe TH, Czerwinski A, Nagel C (2008), OpenGIS OpenGIS City Geography Markup Language (CityGML) Implementation Specification, Version 1.0.0, OGC Doc. No. 08-007, Open Geospatial Consortium 2008.

[13] Gröger G, Kolbe TH, Schmittwilken J, Stroh V, Plümer L (2005) Integrating versions, history and levels-of-detail within a 3D geodatabase, In: Proceedings of International Workshop on Next Generation City Models, Bonn 2005.

[14] Müller H, Curtis E (2005) Extending 2D interoperability frameworks to 3D, In: Proceedings of International Workshop on Next Generation City Models 2005, Bonn.

[15] Quadt U, Kolbe TH (2005) Web 3D Service (W3DS). OGC Discussion Paper, Version 0.3.0, OGC Doc. No. 05-019, Open Geospatial Consortium 2005.

[16] Rumbaugh J, Jacobson I, Booch G (2004) The Unified Modeling Language Reference Manual, 2nd ed., Addison-Wesley Longman, Amsterdam 2004.

[17] Snowflake Software, GO Loader product page (2008), http://www.snowflake software.co.uk/products/goloader/index.htm [letzter Zugriff: 2008-06-20].

[18] Stadler A (2007) Kohärenz von Geometrie und Semantik in der Modellierung von 3D Stadtmodellen, In: Entwicklerforum Geoinformationstechnik 2007, Shaker Verlag, pp. 167-181.

[19] Vretanos PA (2005) Web Feature Service Implementation Specification, Version 1.1.0, OGC Doc. No. 04-094, Open Geospatial Consortium 2005.

[20] Wilson T (2007) OGC KML, Version 2.2.0, OGC Doc. No. 07-147r2, Open Geospatial Consortium 2008.

Kontakt

Claus Nagel | Alexandra Stadler

Technische Universität Berlin
Institut für Geodäsie und Geoinformationstechnik

Strasse des 17. Juni 135
10623 Berlin

e-mail: nagel@igg.tu-berlin.de | stadler@igg.tu-berlin.de

Entwicklerforum Geoinformationstechnik 2008. Shaker Verlag ISBN: 978-3-8322-7393-4