Studienarbeit

Implementierung einer Datenbankanbindung zur Speicherung und Abfrage von Mikroprozessereignissen

Julia Schenk – schenk@inf.fu-berlin.de

betreut von Sebastian Jekutsch
und Prof. Dr. Lutz Prechelt

Fachbereich Mathematik und Informatik
AG Software Engineering

1 Kontext 1

2 Aufgabenstellung 3

3 Anforderungsbeschreibung 4

3.1 Auswahl eines geeigneten Datenbanksystems 4

3.2 Zu entwickelnde ECG Module 4

3.3 Umgebung aufbauen und evaluieren 5

4 Entwurfsentscheidungen 7

4.1 Auswahl des Datenbanksystems 7

4.1.1 Kategorisierung von XML-Dokumenten 9

4.1.2 Speicherung von XML in relationalen Datenbanken 9

4.1.3 Speicherung von XML in nativen XML Datenbanken 11

4.1.4 Einordnung der XML-Dokumente und Auswahl eines Datenbankmodells 12

4.2 Betrachtete Frameworks 13

4.2.1 JaxMe 2 13

4.2.2 Hibernate 14

4.2.3 Castor 14

4.2.4 XMLBeans 15

4.2.5 Abwägungen zum Einsatz eines Frameworks 16

4.3 Abbildung der XML Schemata auf das Datenbankschema 16

4.4 Struktur für die Abbildung eines Events auf die Datenbank 19

4.5 Anforderungen an die Schemata für das automatischen Mapping 23

4.6 Beispiel: Abbildung eines Schemas in entsprechende Tabellen 24

5 Feinentwurf 26

5.1 Das DatabaseTargetModule 26

5.1.1 Klassen im DatabaseTargetModule 26

5.1.2 Starten des Moduls 27

5.1.3 Einfügen eines Ereignisses in die Datenbank 28

5.2 Das DatabaseSourceModule 29

5.2.1 Klassen im DatabaseSourceModule 29

5.2.2 Starten des Moduls 30

5.2.3 Unmarshalling der Mikroprozessereignisse 30

5.2.4 Die DBSource-GUI 31

6 Abschließende Betrachtungen 32

7 Quellen 34

Anhang I Tests i

1 Kontext

Die Arbeitsgruppe Software Engineering des Fachbereichs Mathematik und Informatik an der Freien Universität Berlin beschäftigt sich mit der Problematik Fehler bei der Softwareentwicklung frühzeitig zu erkennen und gegebenenfalls zu vermeiden. „Frühzeitig“ heißt in diesem Zusammenhang, dass schon während des Programmierens die Tätigkeiten des Programmierers betrachtet werden sollen, um somit Fehler des Programmierers, welche zu Defekten der entwickelten Software führen können, zu erkennen und durch automatisierte Erkennungsmechanismen darauf aufmerksam zu machen.

Dazu wurde in der Diplomarbeit von Frank Schlesinger [1] eine Software entwickelt (Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung), welche zum einen aus einem Sensor und zum anderen aus einem Server besteht.

Der Sensor ist ein Plug-In, welcher in die Entwicklungsumgebung des Programmierers integriert wird. Dort schneidet er die Programmiertätigkeiten des Programmierers in Form von Mikroprozessereignissen mit, ohne diesen jedoch in irgendeiner Weise in seiner Arbeit zu beeinträchtigen.

Ein Mikroprozessereigniss ist eine für einen Mikroprozess relevante Aktivität des Programmierers, wobei ein**e** Mikroprozess durch Sebastian Jekutsch wie folgt definiert wird: „A microprocess is the detailed, every-day process of programming (low-level designing, coding, comprehending, testing, understanding) described as a sequence of activities, i.e. events.” Beim Mikroprozess des Implementierens beispielsweise sind relevante Aktivitäten des Programmierers das Ändern eines Methoden- oder Parameternamens oder das Ausführen des Programms. Solche relevanten Aktivitäten des Programmierers werden als Ereignisse bezeichnet.

Die Daten der mitgeschnittenen Ereignisse sowie Informationen über die Programmierumgebung (Benutzername, Projektname) werden als XML-Dokumente ‚verpackt’ und vom Sensor als ein als Mikroprozessereignis (MPE) bezeichnetes Datenpaket, welches mit einem Zeitstempel und der Art des Ereignisses versehen wird, an den Server geschickt.

Auf dem Server läuft die Software ECG-Lab, welche die oben genannten MPE entgegen nimmt und die Daten entsprechend verarbeitet. Das ECG-Lab ist modular aufgebaut, d.h. alle Funktionen sind in Modulen implementiert, die in das ECG-Lab eingebunden werden.

Dabei wird unterschieden zwischen Modulen, welche MPE entgegennehmen und verarbeiten (TargetModule) und solchen, welche wiederum MPE an andere Module weitergeben können (SourceModule).

Für den Fall der nachträglichen Analyse und für Forschungszwecke müssen die vom Server empfangenen Daten persistent gemacht werden. Werden Programmierer lange beobachtet, so werden viele MPE aufgezeichnet und es kommt zu erheblichen Datenmengen.

Ziel dieser Studienarbeit war es einerseits, ein Modul zu entwickeln, welches die Daten der zu empfangenen Ereignisse extrahiert und in geeigneter Form auf eine Datenbank abbildet, damit Abfragen zur Auswertung der Ereignisdaten möglich sind.

Andererseits sollte es aber auch möglich sein, die gespeicherten Daten wieder aus der Datenbank lesen zu können und in MPE zusammenzufassen sowie über eine grafische Oberfläche die Anzeige der Daten aus der Datenbank zu ermöglichen.

2 Aufgabenstellung

Die vorgegebene Aufgabenstellung, welche auch unter der URL: http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisDB4ECG zu finden ist, war folgende:

Derzeit werden die Ereignisse in einem XML-Format auf Dateien gespeichert. Diese können naturgemäß sehr groß werden, so dass - auch aus Gründen der Datensicherheit - es vorzuziehen ist, dafür ein Datenbankmanagementsystem einzusetzen. Dies bietet auch die Chance, Sichten auf die Daten zu bilden. Der besondere Aspekt ist, dass die Ereignisdaten einen Zeitstempel haben, und dass daher die Anfragen häufig einen Zeitbezug haben. Das Ergebnis einer Datenbankanfrage ist also immer eine Ereignismenge.

Ziel dieser Arbeit ist es idealer Weise

- Ein passendes Datenbankschema auf Basis eines passenden Datenbankmodells zu entwickeln, dessen Eignung anhand der Einfachheit/Verständlichkeit der Anfragen zu beurteilen ist.
- Ein "DatabaseTargetModule" für den ECG-Server auf Basis dessen Modulkonzepts zu bauen.
- Ein "DatabaseSourceModule" zu bauen, der die Daten für den ECG-Server auch wieder lesbar macht, wobei nicht alle Ereignisse, sondern nur die von Anfrageergebnissen gesendet werden, und zwar in Echtzeit oder am Stück. Idealerweise kann die Anfrage über eine GUI eingegeben werden.
- Eine entsprechende Umgebung am Institut aufzubauen und zu evaluieren, wobei auch ein passendes DBMS ausgewählt werden muss.

Wie üblich müssen vorweg die Anforderungen verstanden und die Arbeit am Ende mit einer Ausarbeitung und einem Vortrag präsentiert werden. Es kann im Falle einer umfangreicheren Evaluation des passenden Datenbankschemas die Implementierung gekürzt werden.

3 Anforderungsbeschreibung

3.1 Auswahl eines geeigneten Datenbanksystems

Es muss ausgewählt werden, in welcher Art und Struktur die zu speichernden Daten abgelegt werden. Dabei müssen Anfragen zur Analyse der Daten leicht zu formulieren sein.

Wie unter http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisDB4ECGRequirements definiert, sind Beispielanfragen immer zeitlicher Natur und können wie folgt aussehen:

- Alle Ereignisse des Programmierers Hugo am 24.12.2001.
- Alle Ereignisse zwischen Ereignis Run1 und Ereignis Run2.
- Alle Ereignisse zwischen zwei weniger als 30 Sekunden auseinander liegenden Run-Ereignissen, gruppiert nach zeitlichem Zusammenhang.
- Alle Codeänderungsereignisse eines Programmierers, die ohne zwischenzeitlichen Run geschahen.

Die MPE liegen in Form von EventPacket - Objekten vor und enthalten Standardattribute und je nach Ereignistyp spezifische Attribute.

Bei der Wahl des Datenmodells ist die Skalierbarkeit ein entscheidender Faktor, denn es können nachträglich Ereignistypen mit eigenen ereignisspezifischen Attributen hinzukommen. Auch die Standardattribute der Ereignisse können möglicher Weise erweitert werden.

Es wird keine Notwendigkeit bestehen, Daten nachträglich zu ändern, außer dass sie evtl. gelöscht werden.

Aufgrund der Betrachtungen für das Datenmodell ist zu entscheiden, welches Datenbankmanagementsystem geeignet ist und zum Einsatz kommen soll.

3.2 Zu entwickelnde ECG Module

Die Abspeicherung der Daten soll durch ein DatabaseTargetModule geschehen, welches für das Marshalling der Ereignisse, also das Konvertieren eines Ereignisses in die Datenbank, zuständig ist.

Damit diese Daten analysiert und auch wieder rekonstruiert werden können, soll das DatabaseSourceModule entwickelt werden. Dieses ist im Sinne des ECG-Lab ein Quellmodul, wodurch Anfragen an die Datenbank bezüglich der gespeicherten Daten möglich sein sollen. Ebenso soll von diesem Modul das Unmarshalling der Ereignisse, also das Wiederherstellen der EventPacket Datenstruktur der Ereignisse aus der Datenbank realisierbar sein.

Zu Analysezwecken soll es idealer Weise möglich sein, Datenbankabfragen über eine GUI einzugeben und sich die Ergebnisse anzeigen zu lassen. Dabei würde die GUI als DBSource-GUI realisiert werden und kein eigenes ECG-Modul darstellen.

3.3 Umgebung aufbauen und evaluieren

Die entwickelten Module sollten nicht nur Prototypen, sondern produktiv einsetzbar sein. Kriterien zur Evaluierung der Lösung sind folgende:

- Korrektheit
- Portabilität
- Geschwindigkeit
- Skalierbarkeit

Die folgende Abbildung 1 zeigt eine Übersicht über das ECG-Lab sowie den Datenfluss eines von einem Sensor empfangenen EventPackets. Integriert in diesen Überblick sind auch die im Rahmen dieser Studienarbeit zu entwickelnden Module.

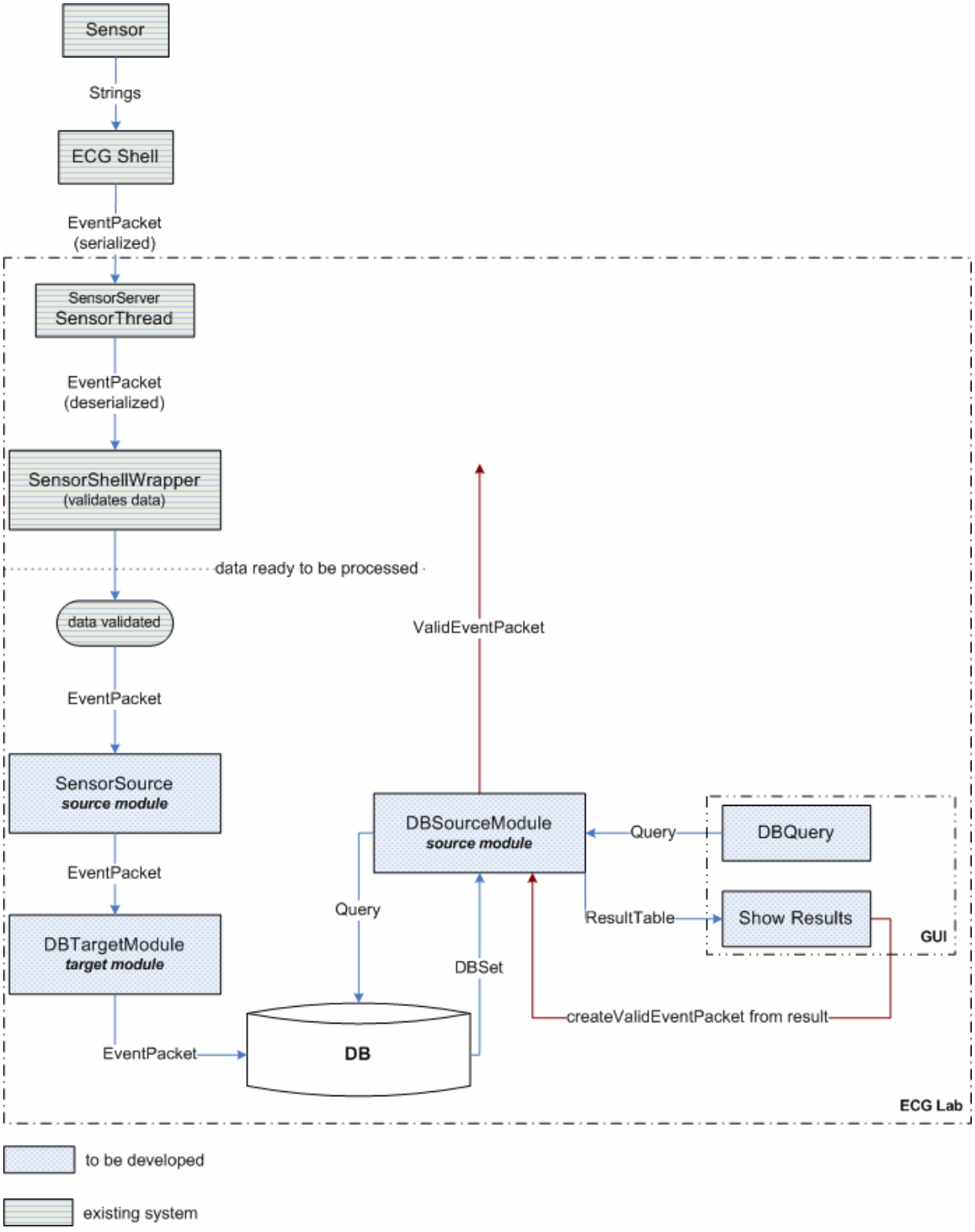

Abbildung 1 - Übersicht der vorhandenen und zu entwickelnden Module

4 Entwurfsentscheidungen

Unter Betrachtung bestimmter Faktoren wie z.B. die Art, Struktur und Verwendung der zu speichernden Daten habe ich zunächst evaluiert, welches Datenbanksystem zur Speicherung der MPE zum Einsatz kommen soll. Dabei war zunächst prinzipiell die Frage über die Verwendung eines relationalen Datenbankmodells oder einer nativen XML-Datenbank zu klären.

Um zu entscheiden, welches Datenbanksystem sinnvoll ist, habe ich mir zum einen angeschaut, welche Art von XML-Dokumenten vorliegen und zum anderen, welches Datenbanksystem zur Speicherung welcher Art von XML-Daten am besten geeignet ist. Des Weiteren betrachtete ich die verschiedenen Möglichkeiten, XML in eine Datenbank abzubilden.

Danach erarbeitete ich anhand verschiedener Ansätze einen möglichen Lösungsansatz. Auf dem Weg zu diesem Lösungsansatz untersuchte ich verschiedene Frameworks, da der Einsatz von Frameworks viel Zeit und neu zu implementierenden Code ersparen und somit Fehler vermeiden kann. Ich recherchierte nach Frameworks, welche mir bei der Abbildung der MPE in die Datenbank hilfreich sein könnten. Dabei fand ich vier Frameworks, in deren Anwendung ich mich hinein arbeitete, um so evaluieren zu können, wie gut und unter welchem Aufwand das Framework für die gewünschten Funktionen geeignet ist.

4.1 Auswahl des Datenbanksystems

Nach [3] ist der wohl wichtigste Faktor bei der Auswahl eines Datenbanksystems, ob man die Datenbank zur Speicherung von in XML-Dokumenten stehenden Daten oder XML-Dokumenten selbst nutzen möchte.

Daher muss man sich zunächst ein Bild über die Beschaffenheit der zu speichernden Daten insbesondere hinsichtlich ihrer Größe und Strukturkomplexität machen:

Jedes zu speichernde Ereignis enthält Informationen, welche zum einen Teil in einem im Ereignis enthaltenen XML-Dokument und zum anderen Teil separat von diesem XML Dokument als Attribute im Ereignis stehen. Die im Ereignis enthaltenen Informationen kann man kategorisieren in Standardinformationen, welche in jeder Art von Ereignis zu finden sind und Informationen, welche ereignisspezifisch sind, also bei jeder Art von Ereignis differieren. Abbildung 2 zeigt diese Kategorisierung und die Anordnung der Attribute in einem Ereignis.

Die separat vom XML-Dokument stehenden Attribute des Ereignisses fassen keine ereignisspezifischen Daten, sondern sind Teil der in jedem Ereignis obligatorisch vorhandenen Standardattribute. Der andere Teil der Standardattribute steht im XML-

Dokument unterhalb des *commonData* Elements, diese Standardattribute innerhalb des *commonData* Elements sind für jeden Ereignistyp fakultativ und es können auch weitere hinzukommen.

Ereignisspezifische Informationen stehen ausschließlich im XML-Dokument.

Das *commonData* Element steht in dem XML-Dokument unterhalb des *microActivity* Elements auf gleicher Ebene mit dem Element, welches die eventspezifischen Attribute enthält.

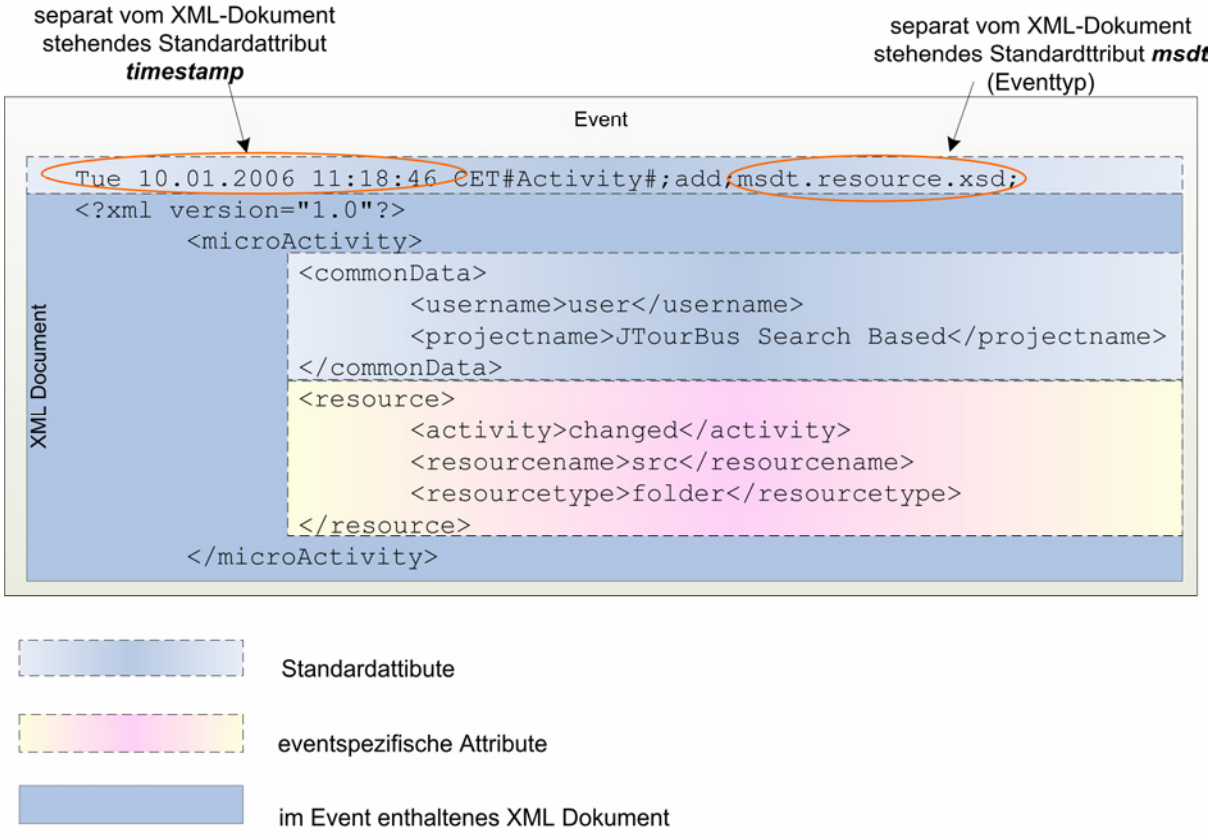

Abbildung 2 - Stringrepräsentation eines Ereignisses

4.1.1 Kategorisierung von XML-Dokumenten

Wie in [2] und [3] gezeigt, kann man XML-Dokumente je nach Beschaffenheit und Informationshaltung in folgende drei Kategorien unterteilen, wobei die Grenzen jedoch fließend sein können.

a) **Datenzentriertes XML**
   - beinhaltet Daten mit flachen Hierarchien
   - Informationen werden in flachen Strukturen mit einem festen Schema abgebildet
   - für Maschinenlesbarkeit optimiert
   - sehr regelmäßige Struktur
   - Bsp.: Bestellungen, Flugpläne
   - ideal für relationale oder objektorientierte Datenbanken

b) **Dokumentenzentriertes XML**
   - XML hat Dokumentencharakter
   - es steht meist nicht fest, in welcher Form Informationen abgelegt sind
   - es existiert meist kein festes Schema
   - Informationen werden durch Markup-Elemente beschrieben und ergeben somit möglicherweise ein tiefverschachteltes Dokument
   - ist für den menschlichen Gebrauch entworfen
   - haben weniger reguläre Struktur und verwenden gemischten Inhalt
   - Bsp.: Bücher, Konfigurationsdatei
   - geeignet für native XML Datenbanken

c) **Semistrukturiertes XML**
   - enthält sowohl strukturierte (datenzentrierte) als auch unstrukturierte (dokumentenzentrierte) Bereiche
   - Bsp.: HTML Seiten
   - geeignet für native XML Datenbanken

4.1.2 Speicherung von XML in relationalen Datenbanken

Die Speicherung von XML-Dokumenten in relationalen Datenbanksystemen bietet die Möglichkeit von der Einfachheit und Erweiterbarkeit sowie von den vielfältigen Einsatzmöglichkeiten von XML zu profitieren, ohne dabei auf die in vielen Bereichen eingesetzten relationalen Datenbanksysteme mit ihrer Robustheit, Effizienz und guten Skalierbarkeit verzichten zu müssen [10].

Es gibt verschiedene Möglichkeiten XML in relationalen Datenbanken zu speichern, wobei man abwägen muss, inwiefern man im Kontext seiner Anwendung und der gegebenen Daten noch von den Vorteilen der relationalen Datenbanken profitieren kann, d.h. die

Anwendung der Daten und die Dokumentenstruktur der XML-Dokumente ist entscheidend für die Auswahl der Speichermethode.

Im Folgenden werde ich kurz die in [2] und [5] vorgestellten Speichermethoden von XML in relationalen Datenbanken darlegen:

Vollspeicherung

Dies ist zunächst die einfachste Methode, ein XML-Dokument in einer Datenbank abzuspeichern. Dabei wird das gesamte Dokument in einem Attribut vom Typ CLOB, Text etc. abgelegt, was dazu führt, dass die eigentlichen Vorteile einer relationalen Datenbank nicht genutzt werden.

Das Dokument in seiner Ordnung bleibt erhalten und es muss keinerlei Schema zu den gespeicherten Dokumenten vorhanden sein.

Anfragen auf einzelne Elemente des XML-Dokuments sind nicht möglich, da Anfragen immer nur das gesamte Dokument als Ergebnis liefern können.

Da das Dokument als Ganzes in einem einzigen Attribut gespeichert ist, können Werteindizes nicht greifen und es müssen Volltextindizes genutzt werden, was eine aufwändige Indizierung bei der Speicherung des Dokuments und bei Suchanfragen deutlich reduzierte Antwortzeiten zur Folge hat.

Die Methode der Vollspeicherung eignet sich also dann am besten, wenn das Interesse primär auf dem XML-Dokument als Ganzes und nicht auf den darin enthaltenen Daten liegt.

Abbildung der Dokumentstruktur

Bei dieser Art der Speicherung wird die Struktur des XML-Dokuments in einem festen Datenbankschema gespeichert, wobei dann die Dokumentenordnung weitgehend erhalten bleibt.

Dabei wird für die Element- und für die Attributknoten je eine Tabelle erstellt. Die Elementstruktur wird durch eine Vater-Kind-Beziehung (selbstreferenzierender Baum) erhalten und die Attribute werden den Elementen über eine Schlüsselbeziehung zugeordnet.

Bei komplexen Anfragen müssen die Tabellen zur Auflösung der Vater-Kind-Beziehungen mit sich selbst verknüpft werden, was eine schlechte Performance zur Folge hat.

Durch diese Verknüpfung der Tabellen mit sich selbst entstehen außerdem Rekursionen, die im relationalen Datenbankenwurf jedoch vermieden werden sollen.

Eine ebenso schlechte Performance hat man bei der Methode, die Dokumentenstruktur eines Dokuments auf eine Tabelle abzubilden, also den XML-Baum innerhalb einer Tabelle

darzustellen. Dabei gibt es noch mehr rekursive Beziehungen, da alle Knoten in einer Vater-Kind-Beziehung innerhalb einer Tabelle stehen. Lediglich der Vorteil von leicht zu formulierenden Anfragen auf alle in einer Tabelle stehenden Knotentypen steht dem gegenüber.

Übersetzung der Dokumentenstruktur

Hierbei wird die hierarchische Struktur eines XML-Dokuments in eine flache Struktur von Datenbanktabellen überführt, indem die Knoten des XML-Dokuments in Attribute der Tabellen übersetzt werden.

Dafür muss ein Schema der XML Struktur existieren, damit die Datenbankstruktur entsprechend gestaltet werden kann. Das hat wiederum zur Folge, dass Änderungen am XML-Schema im Datenbankschema nachvollzogen werden müssen.

Das so genannte Mapping, also die Abbildung der Dokumentenstruktur auf das Datenbankschema oder auch die Übersetzung der Dokumentenstruktur in eine flache Struktur, verwendet man hauptsächlich bei datenzentriertem XML, also wenn das Interesse auf den im XML-Dokument enthaltenen Daten liegt.

4.1.3 Speicherung von XML in nativen XML Datenbanken

Speichert man ein XML-Dokument in einer nativen XML Datenbank, so wird das Dokument in keine andere Struktur konvertiert, sondern die Struktur des XML-Dokuments wird in ein für XML ausgelegtes Speicherformat übertragen.

Bei einer nativen XML-Datenbank liegt ein XML-Dokument als fundamentale Einheit der Speicherung zugrunde, so wie bei einer relationalen Datenbank die Zeile in einer Tabelle die fundamentale Einheit der Speicherung ist [7].

Native XML Datenbanken definieren intern ein logisches Modell für ein XML-Dokument und erfordern dabei kein spezifisches zugrunde liegendes physisches Speichermodell, da sie beliebig auf einer relationalen, objektorientierten oder hierarchischen Datenbank oder auch auf einem proprietären Speicherformat aufgebaut werden können [7].

Vorteile von nativen XML Datenbanken bestehen darin, dass eine Bearbeitung des XML-Dokuments auf Elementebene möglich ist und dass eine direkte Unterstützung für XML Standards vorhanden ist. Jedoch sind native XML Datenbanken heute noch nicht so ausgereift wie beispielsweise relationale Datenbanksysteme

XML mit Dokumentencharakter oder Semistrukturen lässt sich in nativen XML Datenbanken besser verarbeiten als in relationalen oder objektorientierten Datenbanksystemen [5].

Abbildung 3 zeigt dazu eine Übersicht über die Speichermethoden von XML und deren jeweiligen Eigenschaften bezüglich der zu speichernden Daten.

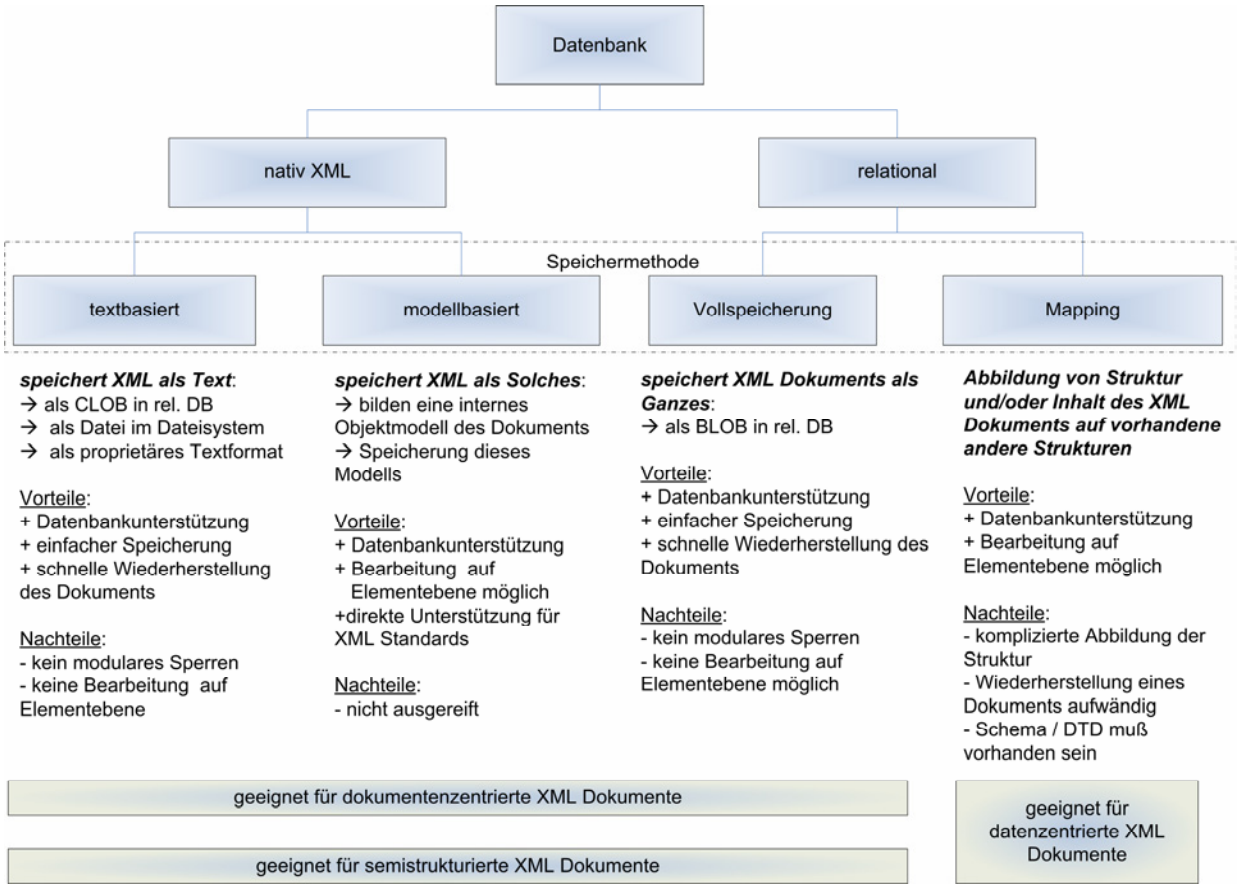

Abbildung 3 - Speichermethoden von XML

4.1.4 Einordnung der XML-Dokumente und Auswahl eines Datenbankmodells

In der Anwendung ECG-Lab besteht kein Interesse an den XML-Dokumenten an sich, sondern nur an den darin enthaltenen Informationen. Die XML-Dokumente sind von sehr einfacher Struktur und werden ausschließlich zum Datentransport genutzt. Die in den MPE enthaltenen XML-Dokumente folgen jeweils einem festen, wenn auch modifizierbaren, Schema. Von daher ordne ich die XML-Dokumente der MPE in die Kategorie der datenzentrierten XML-Dokumente ein.

Des Weiteren stehen die zu speichernden Informationen nicht ausschließlich im XML-Dokument, sondern zum Teil sind es separat vom XML-Dokument stehende Attribute im Ereignis.

Aufgrund der oben aufgeführten Argumente habe ich mich für den Einsatz eines relationalen Datenbanksystems entschieden.

MySQL [8] ist ein frei verfügbares relationales Datenbankmanagementsystem, bei dem ich im Sinne dieser Anwendung keine Einschränkungen bzgl. der benötigten Funktionalität erkennen konnte und es daher einsetzte.

4.2 Betrachtete Frameworks

Da ich mich für den Einsatz eines relationalen Datenbanksystems entschlossen habe, wollte ich nun ein Framework finden, welches mir bei der persistenten Speicherung der Eventdaten sowie der Rekonstruktion von Events aus der Datenbank hilfreich sein kann.

Integraler Bestandteil der Betrachtungen waren dabei folgende Anforderungen:

a) Überführung der hierarchischen XML Struktur der XML Schemata in flache Relationen des Datenbankschemas.
b) Extrahieren der Informationen aus den (verschiedenartigen) XML-Dokumenten der Events.
c) Einfügen der extrahierten Daten in die Datenbank.
d) Extrahieren der Informationen aus den Standardattribute welche separat vom XML-Dokument im Event stehen.
e) Konsolidiertes Speichern der Daten aus d) und der Standardattribute aus dem *commondata* Element im XML-Dokument.
f) Wiederherstellen der XML-Dokumente entsprechend ihrer Schemata.
g) Rekonstruieren von Events mit dem enthaltenen XML-Dokument und den weiteren Attributen.
h) Hinzukommende Schemata automatisch erfassen und in das Datenbankschema abbilden.
i) Änderungen an existierenden Schemata erkennen und entsprechend das Datenbankschema modifizieren.
j) Schemaänderungen müssen rückwärtskompatibel sein.

4.2.1 JaxMe 2

JaxMe 2 [20] ist eine Opensource Implementierung von JAXB [21], der Spezifikation für Java/XML Binding. Das Framework transformiert also XML Dokumente in ein Java-Objektmodell und umgekehrt. Es stellt Methoden zum Zugriff und zur Manipulation der Attribut- und Elementwerte eines XML-Dokuments bereit.

Der Java/XML Binding Compiler generiert aus einem Schema Java-Bean Klassen, welche das Objektmodell des Schemas repräsentieren. Dazu werden ein zugehöriger Marshaller und Unmarshaller generiert, um die Instanzen des XML Schemas, also die XML-Dokumente, und die Java-Beans ineinander transformieren zu können.

JaxMe erfordert keine Konfiguration des Mappings, sondern der Binding Compiler benötigt lediglich ein Schema zu Generierung der Java-Bean Klassen.

Die Zusammenhänge beim XML Data-Binding von Schema, Schema-Instanz, Klassen und Objekten ist durch die Grafik in Abbildung 4 - XML Data-Binding-Struktur dargestellt.

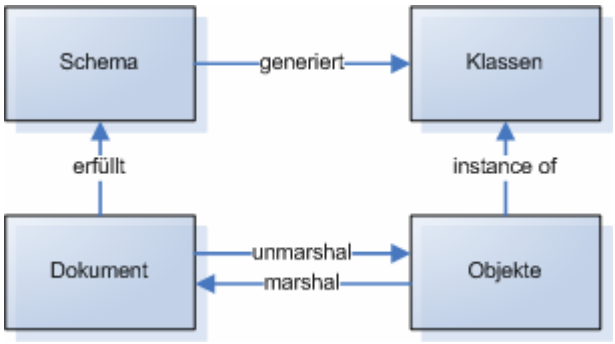

Abbildung 4 - XML Data-Binding-Struktur

4.2.2 Hibernate

Hibernate [15] ist ein Java-basiertes Opensource Framework für Objekt-Relationales Mapping. Man übergibt Hibernate die persistent zu speichernden Objekte, ohne sich dabei um SQL Anweisungen oder – Abfragen kümmern zu müssen. Dadurch bleibt die Anwendung unabhängig vom eingesetzten Datenbanksystem und der Zugriff auf das Datenbanksystem aus der Programmiersprache heraus wird erheblich vereinfacht.

Ebenso kann Hibernate aus den Datensätzen der Datenbank wieder Objekte erzeugen.

Hibernate benötigt eine Vielzahl von Konfigurationsparametern, welche in einer hibernate.properties Datei abgelegt werden, sowie XML-Konfigurationsdateien zur Abbildung der Objekte auf die Datenbank [16].

Ein mit der Version 3 hinzugekommenes Feature von Hibernate ist die Unterstützung von XML-Relationaler Persistenz. Hibernate bekommt XML-Dokumente anstatt POJOs übergeben und bildet diese direkt in die Datenbank ab [17].

Dieses Feature ist im Rahmen dieser Studienarbeit sehr interessant, da dadurch dass für die Anwendung im ECG Lab überflüssige XML-Data-Binding umgangen wird. Genauso wie für das Objekt-Relationale Mapping werden auch für das XML-Relationale Mapping in Hibernate Konfigurationsdateien benötigt.

Das XML-Relationale Mapping ist laut der Hibernate Dokumentation jedoch ein experimentelles Feature, welches sich noch im Entwicklungsstadium befindet.

4.2.3 Castor

Castor [24] ist ein sehr mächtiges Hilfsmittel, welches zu den Opensource Data-Binding Frameworks zählt. Das XML Feature [25] von Castor ist ein XML-Data-Binding Tool, welches das Marshalling Java Objekten in XML Dokumente und umgekehrt umsetzt. Es kann jedes Java Objekt, welches Java-Bean - Konventionen entspricht, in ein XML-Dokument konvertieren und umgekehrt werden aus XML-Dokumenten Klassen nach Java-Bean Konventionen erzeugt. Das XML-Data-Binding muss nicht manuell gesteuert werden, [28]

dann geht das Framework nach seinen statischen Methoden vor. Es besteht jedoch die Möglichkeit, eigene Transformationsregeln in einer Mapping-Datei zu hinterlegen.

Des Weiteren verfügt Castor über einen XML Code Generator, welcher zu einem XML Schema eine Menge von Java Klassen generiert, die das Objektmodell des Schemas repräsentieren. Die so generierten Klassen verfügen über Marshalling- und Unmarshalling Methoden, um so zwischen den Instanzen dieser Klassen und deren entsprechenden XML-Dokumenten hin- und her wechseln zu können.

Zur persistenten Speicherung von Java Objekten verfügt Castor über das JDO (Java Data Object) Feature, welches Java Objekte in SQL Tabellen abbildet und umgekehrt.

4.2.4 XMLBeans

XMLBeans [22] ist eine vom Apache XML Project entwickelte Technologie, welche das Lesen und Manipulieren von XML-Dokumenten erheblich vereinfacht, indem es XML an Java Typen bindet. Laut [18] ist Castor neben XMLBeans das bekannteste XML-Data-Binding Framework in der Opensource Welt.

XMLBeans kompiliert ein XML Schema und generiert dabei Java Typen, welche die Schema Typen repräsentieren. Dadurch kann auf Instanzen des Schemas durch JavaBean - artige getter- und setter- Methoden zugegriffen werden. Das Kompilieren eines Schemas liefert eine JAR Datei, welche die aus dem Schema generierten Interfaces enthält.

XMLBeans besteht zum Hauptteil aus drei APIs [23]:

- *XmlObject*
  Alle durch Kompilieren eines XML Schemas generierten Klassen sind von XmlObject abgeleitet. Diese Klassen bieten streng typisierte getter- und setter- Methoden.
- *XmlCursor*
  Es ist möglich von jedem XmlObject einen Cursor zu erhalten, welcher einen effizienten Zugriff auf das XML Infoset bietet. Ein Cursor repräsentiert eine Position in einer XML Instanz.
- *SchemaType*
  Diese API bietet ein Objekt-Modell eines XML Schemas, mit dessen Hilfe man Informationen aus dem XML Schema selbst ziehen kann. Somit ist es möglich die Metainformationen der Elemente eines XML-Dokuments zu erhalten.

XMLBeans bietet also nicht nur eine API für das XML data-binding, sondern zusätzlich eine SchemaType API, welche nicht Klassen generiert, die das Objektmodell eines Schemas repräsentieren, sondern mit der direkt ein Objekt-Modell eines Schema erstellt wird, durch das dann Informationen über die Struktur eines Schemas und der darin definierten Typen und deklarierten Elemente bezogen werden können.

4.2.5 Abwägungen zum Einsatz eines Frameworks

Für den Einsatz im DatabaseTargetModule sowie im DatabaseSourceModule würde die Generierung von Objekten aus den im Event enthaltenen XML-Dokumenten lediglich dem Zweck der Objekt–Relationalen Abbildung dienen, ansonsten würden die Objekte keinen weiteren Nutzen für die Applikation erbringen.

Aufgrund der Tatsache, dass im ECG Lab Schemata hinzukommen sowie Änderungen an vorhandenen Schemata vorgenommen werden können, kann dem Nutzen der getter- und setter- Methoden der Objektmodelle der XML-Dokumente keine Rechnung getragen werden, da der Anwendung nicht statisch bekannt ist welche Attribute die XML-Dokumente haben. Abgesehen davon besteht kein Bedarf an der Bearbeitung der XML-Dokumente.

Es sind nicht nur Daten aus dem XML-Dokument, welche in der Datenbank gespeichert werden müssen, sondern auch die Informationen aus den separat vom XML-Dokument stehenden Standardattributen des Events (Anforderung e). Damit müssten aus den XML Dokumenten generierte Objekte mit den (gesondert extrahierten) Informationen aus den separat stehenden Standardattributen konsolidiert werden.

Werden für das Data-Binding statische Abbildungsvorschriften in Form von Konfigurationsdateien zu Entwicklungszeit benötigt, so steht das im Konflikt mit der geforderten Dynamik der Anforderungen h) und i).

Aufgrund dieser Betrachtungen und nachdem ich viel Zeit in die Evaluierung der Frameworks und deren Verwendungs- und Anpassungsmöglichkeiten für den Einsatz in den Modulen investiert hatte, hielt ich den Einsatzes eines XML Data-Binding Frameworks in Kombination mit Objekt-Relationaler Abbildung für nicht sinnvoll und das Verhältnis von Zeitinvestition und Aufwand gegenüber dem vielleicht möglichen Nutzen für nicht mehr gerechtfertigt. Daher habe ich mich dazu entschieden, das Marshalling und Unmarshalling der MPE und auch die automatische Abbildung der Schemata in die Datenbank selbst zu implementieren.

Die Abbildung XML Schemata in das Datenbankschema ist ein recht komplexer Vorgang, der genaue Informationen über die Struktur und Eigenschaften der in dem XML Schema deklarierten und definierten Elemente erfordert. Daher habe ich die SchemaType API des XMLBeans Frameworks benutzt, um allgemein gültig und dynamisch aus allen Schemata die Metainformationen der enthaltenen Elemente zu erhalten und dann nach einer festgesetzten Methode jedes Element und Attribut auf die Datenbank abzubilden.

4.3 Abbildung der XML Schemata auf das Datenbankschema

Ich habe mich für eine tabellenbasierte Abbildung der XML Schemata in die Datenbank [6] entschieden, wobei ich ein Schema auf mehrere Relationen, je nach Typ und Inhalt der Elemente, abbilde.

Für Anfragen ist die Dokumentenordnung des ursprünglichen XML-Dokuments nicht relevant und da ich die spätere Rekonstruktion der XML-Dokumente anhand der vorhandenen Schemata vornehmen werde, bilde ich keinerlei Strukturen und Metainformationen der XML-Dokumente in die Datenbank ab. Ich erzeuge die Relationen unter dem Aspekt, die Werte und Attribute der Elemente in möglichst einfacher Form abzuspeichern, so dass Anfragen an die Datenbank auch möglichst einfach zu formulieren sein werden.

Ich lege keine Tabelle mit Metainformationen der gespeicherten Dokumente an und ich benutze auch keine Lookup Tabelle zur Zusammenführung der Datensätze eines Dokuments. Zusammengehörige Datensätze verschiedener Tabellen werden lediglich über eine Fremdschlüsselbeziehung einander zugeordnet. Dadurch werden die gebildeten Datenbankrelationen einfach gehalten, was immer vorteilhaft bzgl. der Datenintegrität ist.

Aufgrund der geforderten Dynamik bzgl. der Veränderung von vorhandenen Schemata, welche eine Modifizierung des Datenbankschemas nach sich zieht, habe ich auch darauf verzichtet, die Reihenfolge der Elemente eines XML-Dokuments durch die Reihenfolge der Elemente in der Tabelle auszudrücken. So ist es mir möglich für hinzugekommene Elemente ohne Betrachtung jeglicher Reihenfolge eine neue Spalte an eine bestehende Tabelle anzufügen. Dadurch, dass Spalten zu Tabellen zwar hinzugefügt, jedoch nicht gelöscht werden können, wird die Anforderung j) aus 4.2 erfüllt.

Um so etwas wie einen Namensraum für die Attributnamen in den Tabellen zu konstruieren, setzen sich die Tabellennamen aus dem Typ des Ereignisses und dem Namen des komplexen Elements, für das die Tabelle erstellt wird, zusammen. Die Spaltennamen in den Tabellen entsprechen exakt dem Namen des durch diese Spalte repräsentierten Elements oder Attributs.

Ein Element in einem XML-Dokument wird durch einen Elementtyp definiert [4]. Wie in Abbildung 5 dargestellt, kann man diese Abhängigkeit von Element und Typ ins Analogon einer objektorientierten Programmiersprache übertragen, wobei dann der Elementtyp der Klasse und das Element im XML-Dokument der Klasseninstanz, dem Objekt entspricht.

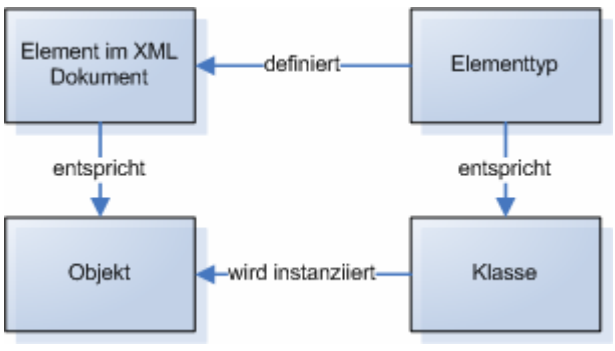

Abbildung 5 - Analogon von Elementtyp und Klasse

Für die Abbildung eines Schemas unterscheide ich die Elemente des Schemas primär entsprechend ihrem Typ und möglichem Inhalt [9] :

- **Complex Types**
  Zur Abbildung komplexer Typen auf das Datenbankschema wird grundsätzlich eine neue Relation in das Datenbankschema eingefügt.
  Komplexe Typen können Attribute enthalten und sie werden entsprechend ihres Inhaltstyps kategorisiert:
  - *Empty Content*
    Das Element hat keinen Inhalt. Die gegebenenfalls vorhandenen Attribute des Elements werden als Spalten in die angelegte Tabelle abgebildet.
  - *Simple Content*
    Das Element hat einfachen Inhalt (Wert eines einfachen Datentyps). Neben den vorhandenen Attributen wird eine Spalte mit dem Namen des Elements und dem entsprechenden Datentyp in die neu angelegte Tabelle eingefügt.
  - *Element Content*
    Das Inhaltsmodell eines solchen komplexen Elements besteht aus weiteren Elementen. Die direkten Kindelemente einfachen Typs sowie die Attribute des Elements werden als Attribute in der angelegten Relation abgebildet. Für komplexe Kindelemente wird eine eigene Relation in das Datenbankschema eingefügt.
  - *Mixed Content*
    Das Element kann sowohl Text als auch weitere Elemente enthalten. Gemischter Inhalt wird nur insofern unterstützt, dass der gesamte Inhalt des Elements in eine Textspalte der angelegten Tabelle abgebildet wird.

- **Simple Types**
  Einfache Typen enthalten keine Subelemente oder Attribute, sondern nur einen primitiven Datenwert und werden daher als Attribut in der Relation des übergeordneten komplexen Elements abgebildet. Eine Abweichung ergibt sich jedoch dann, wenn ein Element einfachen Typs mehr als einmal im XML-Dokument stehen darf, dann wird für dieses Element eine eigene Tabelle angelegt.
  Einfache Typen unterscheiden sich in
  - *Atomic Types*
    Es wird eine Spalte mit dem Namen und dem entsprechenden Datentyp des Elements in der Tabelle erstellt.
  - *List Types*
    Enthalten eine Sequenz atomarer Datentypen, diese Sequenz der atomaren Datentypen wird nicht weiter abgebildet, sondern als Attribut vom Typ Tinytext in die Datenbank gespeichert.

- *Union Types*
  Vereinigungstypen sind Datentypen, welche sich aus der Vereinigung mehrerer einfacher Datentypen zusammensetzen. Damit lässt sich für ein Element nicht eindeutig der Datentyp und damit der Spaltentyp für die Abbildung ins das Datenbankschema bestimmen. Deshalb werden Vereinigungstypen ebenfalls als Spalte vom Typ Tinytext abgebildet.

Beim Abbilden der Schemata auf die Datenbank habe ich einen so genannten Basic-Inlining Ansatz aus [11] verfolgt, d.h. ich habe versucht für ein Element so viele Subelemente wie möglich in einer einzelnen Relation zusammenzufassen, also ein Inlining der Kindelemente zu betreiben.

Damit erreicht man eine möglichst geringe Fragmentierung eines XML-Dokuments bei der Abbildung in die Datenbank, was beim Zugriff auf alle Elemente eines Dokuments zu einer möglichst geringen Anzahl von Joins bzw. Tabellen, welche angefragt werden müssen, führt.

Abbildung 6 zeigt das Vorgehen des Algorithmus zur Generierung der Datenbanktabellen zu einem Schema.

4.4 Struktur für die Abbildung eines Events auf die Datenbank

Wie bereits erwähnt stehen die Informationen eines MPE nicht ausschließlich in dem enthaltenen XML-Dokument, sondern auch in Form separat vom XML-Dokument stehender Attribute im MPE.

Da die meisten Abfragen an die Datenbank voraussichtlich Abfragen sein werden, welche sich auf die Standardinformationen der MPE beziehen, sollen alle Standardattribute eines MPE konsolidiert in einer Tabelle abgebildet werden.

Aufgrund der separat stehenden Standardattribute kann die Tabelle zur Speicherung der Standardattribute nicht rein dynamisch erzeugt werden. Die Tabelle *commondata* zur Speicherung der Standardattribute, beinhaltet die separat vom XML-Dokument stehenden Attribute des MPE als Spalten sowie eine Spalte *linkid*, welche einen vom Datenbanksystem automatisch generierten Primärschlüssel für jedes MPE enthält. Diese Spalten werden explizit erzeugt und dynamisch durch die vom Mapping erstellten Spalten ergänzt.

Alle durch die automatische Abbildung der Schemata erzeugten Tabellen zur Haltung der ereignisspezifischen Daten erhalten ebenso eine Spalte *linkid*, welche den generierten Primärschlüssel der *commondata* Relation als Fremdschlüssel enthält.

Die Datensätze eines MPE sind somit immer auf die *commondata* Tabelle und die automatisch erzeugten ereignisspezifischen Tabellen verteilt.

Verbunden sind alle Datensätze eines MPE durch die Fremdschlüssel- Beziehungen in der *linkid* Spalte einer jeden Tabelle.

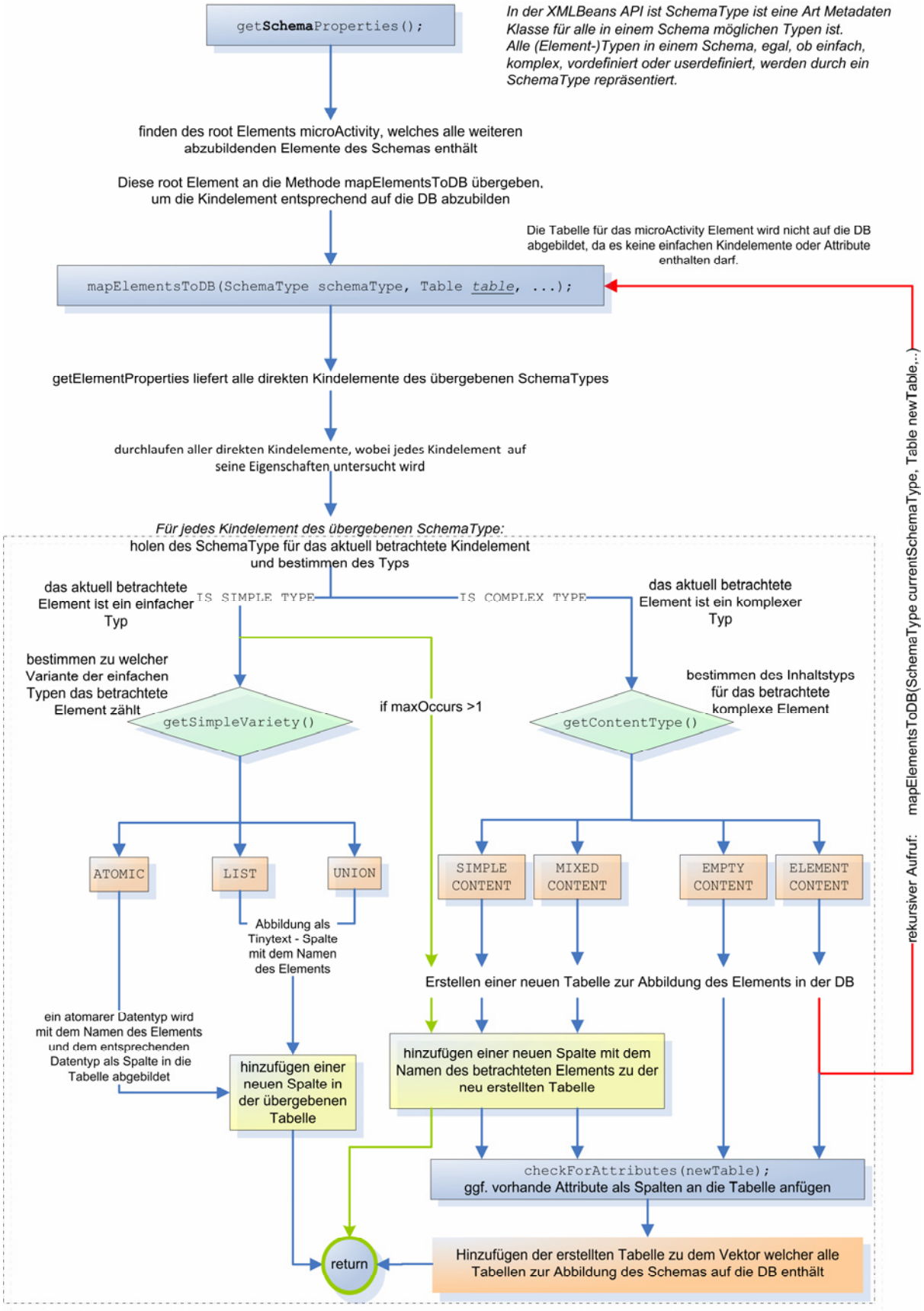

Abbildung 6 - Algorithmus zur Abbildung der Elemente auf das Datenbankschema

Wie die jeweiligen XML Datentypen der Simple Type Elemente in entsprechende SQL Datentypen für die Spalten abgebildet werden, zeigt die folgende Tabelle [12]

XML Typ	*SQL Typ*
XmlID	VARCHAR(255)
SimpleValue	VARCHAR(255)
XmlAnySimpleType	TEXT
XmlAnyURI	VARCHAR(255)
XmlBase64Binary	BLOB(255)
XmlBoolean	TINYINT(1)
XmlByte	TINYINT
XmlDate	DATE
XmlDateTime	DATETIME
XmlDecimal	DECIMAL
XmlDouble	DOUBLE
XmlDuration	TIMESTAMP
XmlENTITIES	TEXT
XmlENTITY	TEXT
XmlFloat	FLOAT
XmlGDay	TIMESTAMP(8)
XmlGMonth	TIMESTAMP(8)
XmlGMonthDay	TIMESTAMP(8)
XmlGYear	YEAR(4)
XmlGYearMonth	TIMESTAMP(8)
XmlHexBinary	BLOB(255)
XmlIDREF	VARCHAR(255)
XmlIDREFS	TEXT
XmlInt	INT
XmlInteger	INTEGER
XmlLanguage	VARCHAR(255)
XmlLong	BIGINT
XmlName	VARCHAR(255)
XmlNCName	VARCHAR(255)
XmlNegativeInteger	INTEGER
XmlNMTOKEN	VARCHAR(255)
XmlNMTOKENS	TEXT
XmlNonNegativeInteger	INTEGER
XmlNonPositiveInteger	INTEGER
XmlNormalizedString	TEXT
XmlNOTATION	TEXT
XmlPositiveInteger	INTEGER
XmlQName	VARCHAR(255)
XmlShort	SMALLINT
XmlString	TEXT
XmlTime	TIME
XmlToken	VARCHAR(255)
XmlUnsignedByte	TINYINT
XmlUnsignedInt	INT
XmlUnsignedLong	BIGINT
XmlUnsignedShort	SMALLINT

4.5 Anforderungen an die Schemata für das automatischen Mapping

a) Das *microActivity* Element eines Schemas muss immer das root Element sein.
b) In *microActivity* muss es immer genau zwei globale Elemente geben: *commonData* und das ereignisspezifische Element, welches alle weiteren Attribute des Ereignistyps enthält.
c) Das microActivity Element enthält keine direkten Simple Type Elemente.
d) Bei Simple Type Elementen muss der Datentyp durch
<xs:element name="xxx "type="xs:…" > oder durch
<xs:element name="xxx"><xs:simpleType><xs:restriction base="xs:string">
angegeben werden.
e) Elemente ohne Typangabe werden als komplexe Elemente mit gemischtem Inhalt interpretiert.
f) Elemente welche CDATA Felder enthalten sollen müssen vom Typ xs:string sein.
g) Für Elemente unterhalb von *commonData* muss immer maxOccurs <=1 gelten.
h) UNION Datentypen werden nicht unterstützt.

*Generelle Informationen **zum** XML Schema:*

Ein XML Schema ist eine Menge von Komponenten. Es gibt insgesamt 13 Arten von Komponenten, welche man in drei Gruppen aufteilen kann [14]:

1. Komponenten, welche einen Namen haben können (Typ Definitionen) oder Namen haben müssen (Element und Attribut Deklarationen)
    a. SimpleType Definitionen (Name optional)
    b. ComplexType Definitionen (Name optional)
    c. Attribut Deklarationen (Name zwingend)
    d. Element Deklarationen (Name zwingend)
2. Komponenten welche einen Namen haben müssen:
    a. Attribute Goup Definitionen
    b. identity-constraint Definitionen
    c. Model Group Definitionen
    d. Notation Deklarationen
3. „Helfer“ Komponenten, welche kleine Teile anderer Komponenten darstellen, sie sind also nicht kontextunabhängig
    a. Kommentare
    b. Model Groups
    c. Particles
    d. Wildcards
    e. Attribute uses

Deklarationskomponenten sind durch einen (qualifizierenden) Namen mit Informationseinheiten verknüpft, welche validiert werden [13].

Definitionskomponenten definieren interne Schemakomponenten, welche in anderen Schemakomponenten genutzt werden können.

4.6 Beispiel: Abbildung eines Schemas in entsprechende Tabellen

Schema msdt.editor.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:annotation>
        <xs:documentation>
                This XML schema defines the structure of the "Editor" MicroSensorDataType.
                The instance documents, the XML documents using this schema, are actual
                "Editor" MicroActivities.
        </xs:documentation>
        </xs:annotation>

        <xs:annotation>
        <xs:documentation>
                        This includes the data fields that are common to all MicroSensorDataTypes.
        </xs:documentation>
        </xs:annotation>

        <xs:include schemaLocation="msdt.common.xsd"/>

        <xs:element name="microActivity">

        <xs:complexType>
        <xs:sequence>
                <xs:element name="commonData" type="commonDataType"/>
                        xs:element name="editor">
                        <xs:complexType>
                                <xs:sequence>
                        ------> <xs:element name="activity">
                                        <xs:simpleType>
                                                <xs:restriction base="xs:string">

                                                <xs:annotation>
                                                <xs:documentation>
                                                The "activity" element of the "editor"
                                                MicroActivity can have one of the following values.
                                                </xs:documentation>
                                                </xs:annotation>

                                                <xs:enumeration value="opened"/>
                                                        <xs:enumeration value="closed"/>
                                                        <xs:enumeration value="activated"/>
                                                        <xs:enumeration value="deactivated"/>
                                                </xs:restriction>
                                        </xs:simpleType>
                                </xs:element>
                        ------> <xs:element name="editorname" type="xs:token"/>
                        </xs:sequence>
                </xs:complexType>
                </xs:element>

        </xs:sequence>
        </xs:complexType>
        </xs:element>
</xs:schema>
```

msdt.common.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:annotation>
		<xs:documentation>
			This XML schema defines the structure of the "Common" complexType
			which is used in every MicroSensorDataType XML Schmema definition:
			*version - (optional) denotes the version of the msdt datatype starting with 1 on 1.1.3 release of ECG
			* creator - (optional) a short string identifying the name and version  of the event's creator, i.e. a sensor or module
			* username - name of the programmer, derived from the operating sy	* projectname - (optional) name of the project which the programmer has been working on
			* id - (optional) an identificator based on the resource (e.g. file, window) for which this event has been sent.
			should be used in corresponding events as well, e.g. open of id3 and close of id3
		</xs:documentation>
	</xs:annotation>

	<xs:complexType name="commonDataType">
		<xs:sequence>
			<xs:element name="version" type="xs:int" minOccurs="0"/>
			<xs:element name="creator" type="xs:token" minOccurs="0"/>
			<xs:element name="username" type="xs:token"/>
			<xs:element name="projectname" type="xs:token" minOccurs="0"/>
			<xs:element name="id" type="xs:token" minOccurs="0"/>
		</xs:sequence>
	</xs:complexType>
</xs:schema>
```

Wird in folgende Tabellenstruktur abgebildet:

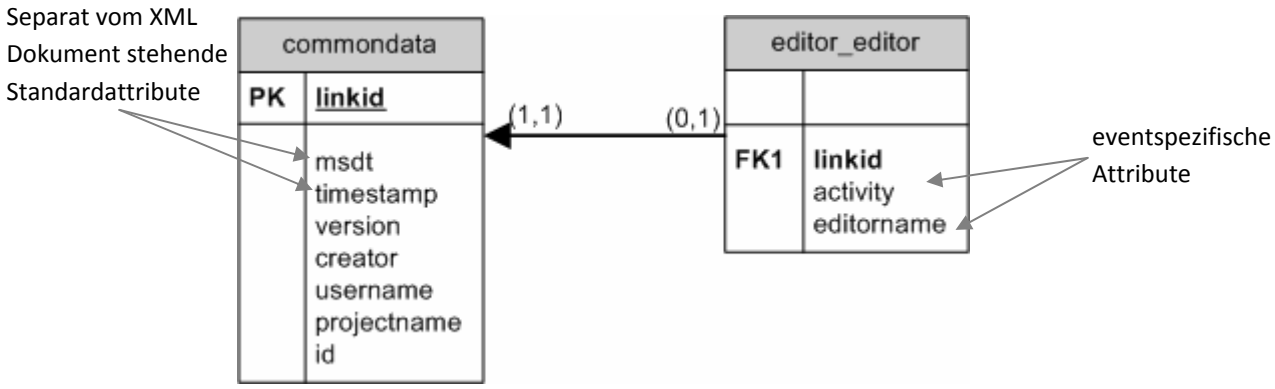

Abbildung 7 - Resultierende Tabellen des Schemas msdt.editor.xsd

5 Feinentwurf

5.1 Das DatabaseTargetModule

5.1.1 Klassen im DatabaseTargetModule

- *DBCommunicator*
  Die direkte Kommunikation mit der Datenbank erfolgt ausschließlich über diese Klasse. *DBCommunicator* öffnet und schließt die Datenbankverbindung und stellt Methoden bereit, um Anweisungen oder Anfragen an die Datenbank zu senden.

- *DBTargetModuleConstants*
  Enthält die Konstanten des DatabaseTargetModuls.

- *XMLSchemaProxy*
  Diese Klasse stellt eine Art Proxy für ein XML Schema dar. Sie bekommt im Konstruktor ein Schema übergeben und stellt dann Methoden zum Zugriff auf das Schema sowie zur Verarbeitung des Schemas bereit. In dieser Klasse ist die automatische Abbildung des Schemas auf das Datenbankschema implementiert. Die automatische Abbildung liefert einen Vektor mit Elementen vom Typ *Table*, der Informationen über alle Tabellen enthält, welche zur Abbildung des Schemas in der Datenbank benötigt werden.

- *Table*
  Repräsentiert eine Tabelle der Datenbank. Die Klasse enthält den Namen der Tabelle sowie die Spalten dieser Tabelle in Form eines *ColumnElement* – Vektors.

- *ColumnElement*
  Repräsentiert eine Spalte einer Tabelle und enthält Informationen über den Spaltennamen, den SQL Datentyp der Spalte sowie den entsprechenden Datentyp des Elements aus dem XML Schema, welches durch diese Spalte repräsentiert wird.

- *SQLDatatypes*
  Diese Klasse enthält die Information darüber, welcher XML Datentyp auf welchen SQL Datentyp abgebildet wird und stellt entsprechende Methoden bereit, um für ein gegebenes *ColumnElement* mit XML Datentyp den entsprechenden SQL Datentyp zu setzen.

- *TableInformation*
  Während der automatischen Abbildung eines Schemas wird sozusagen auch mitprotokolliert, welche Tabellen bei der Abbildung des Schemas auf die Datenbank

involviert sind. Die Klasse *TableInformation* ist als Singleton realisiert und enthält zu jedem Schema-Namen, und damit Ereignistyp, die Information, welche Tabellen für das Mapping des XML-Dokuments diesen Ereignistyps in die Datenbank relevant sind.

- *DBTablesMetadataPool*
  Diese Klasse ist ebenfalls nach dem Singleton-Muster umgesetzt und hält während der Laufzeit die Metadaten aller Tabellen in der Datenbank, also Spaltennamen und deren Datentyp, bereit.

- *ValidEventPacketProxy*
  Ein ValidEventPacketProxy ist eine Proxy für ein zu speicherndes ValidEventPacket, welcher den Zugriff auf die Elemente des Ereignisses erleichtert.

- *CreateSQL*
  In dieser Klasse werden die Strings der auszuführenden SQL-Anweisungen und -Abfragen entsprechend der aus der Anwendung übergebenen Daten generiert.

- *EventBuffer*
  Empfängt das *DatabaseTargetModule* ein Ereignis und es steht momentan keine Datenbankverbindung zur Verfügung, um dieses in der Datenbank zu speichern, so wird das Ereignis, um nicht verworfen werden zu müssen, zunächst gepuffert, um später bei vorhandener Datenbankverbindung in die Datenbank eingefügt werden zu können.

- *DatabaseTargetModule*
  Dies ist die zentrale Klasse des Moduls, welche beim Initialisieren des Moduls geladen wird und welche auch zu speichernde MPE entgegen nimmt und entsprechend weiter delegiert.

5.1.2 Starten des Moduls

Beim Starten des Moduls wird zunächst die *startWriter()* – Methode des *DatabaseTargetModuls* aufgerufen, welche durch Delegation an den *DBCommunicator* dafür sorgt, dass alle vorhandenen Schemata mit der Datenbank synchronisiert werden.

Die für die Synchronisation verantwortliche Methode im *DBCommunicator* holt sich vom *ModulSystem* alle im ECG-Lab registrierten Schemata und synchronisiert jedes dieser Schemata, indem zunächst ein *XMLSchemaProxy* erzeugt wird. Dann wird über die *getSchemaProperties(…)* Methode des *XMLSchemaProxys* die automatische Abbildung des Schemas initialisiert.

Die durch die automatische Abbildung eines Schemas generierten *Table* - Objekte werden in Form eines Vektors festgehalten.

Zur Synchronisierung wird dann jede der in diesem Vektor enthaltenen Tabellen (*Table* - Objekt) mit der Datenbank abgeglichen, d.h. es wird zuerst überprüft, ob diese Tabelle schon in der Datenbank existiert. Ist das nicht der Fall, so wird die Tabelle erstellt. Existiert die Tabelle bereits, so werden gegebenenfalls hinzugekommene Spalten, welche in dem durch das erstellten *Table* – Objekt, aber nicht in der entsprechenden Tabelle der Datenbank vorhanden sind, in das Datenbankschema abgebildet.

Das Singleton *TableInformation* erhält durch die automatische Abbildung eines Schemas den Namen des Schemas mit den Namen der zugehörigen Tabellen in der Datenbank. Ein Ereignis besitzt ein Attribut *msdt*, welches Aufschluss über den Typ des Ereignisses gibt. Der *msdt*-Typ eines Ereignisses wird durch ein XML-Schema definiert und der Wert des msdt Attributs enthält den Namen des definierenden Schemas. Aufgrund dessen kann nun über das *msdt* Attribut und der in der *TableInformation* Instanz enthaltenen Informationen bestimmt werden, welche Tabellen für diesen Ereignistyp beim Mapping des XML-Dokuments betrachtet werden müssen.

5.1.3 Einfügen eines Ereignisses in die Datenbank

Soll ein Event in die Datenbank eingefügt werden, so wird dieser von der write() – Methode im DatabaseTargetModule entgegen genommen und an die insertEvent()- Methode des DBCommunicator weiter gereicht. Dort wird zunächst ein ValidEventPacketProxy des übergebenen Events erstellt und dann übernimmt die Methode Steuerung zum Einfügen der Eventdaten in die Datenbank.

Zuerst werden durch die Klasse CreateSQL die commondata Informationen aus dem Ereignis extrahiert und das entsprechende Statement zum Einfügen dieser Daten generiert. Dieses Statement wird dann durch die insertEvent ()- Methode ausgeführt.

Beim Einfügen der Daten in die commondata Tabelle wird von der Datenbank automatisch ein Primärschlüssel für die Spalte *linkid* erzeugt. Dieser Primärschlüssel wird in einem zweiten Schritt direkt nach dem Einfügen des commondata Datensatzes von der Datenbank an die insertEvent Methode zurückgegeben.

Im dritten Schritt übergibt die insertEvent() – Methode wieder den EventProxy sowie den im zweiten Schritt erhaltenen Schlüssel an eine Methode der Klasse CreateSQL. Diese holt sich zunächst aus der TableInformation Instanz die Information darüber, welche Tabellen in der Datenbank an der Speicherung dieses Ereignistyps beteiligt sind, um dann für jede Spalte dieser beteiligten Tabellen die Daten, falls vorhanden, aus dem Ereignis zu ziehen und damit sowie dem übergebenen Schlüsselwert das entsprechende Statement zum Einfügen der Daten zu generieren.

Diese generierten Einfügestatements werden wieder der Methode insertEvent() zurückgegeben und dort nacheinander ausgeführt.

Die gesamten Einfügeoperationen für ein Ereignis müssen aus Gründen der Datenintegrität als eine Transaktion in der Datenbank ausgeführt werden.

5.2 Das DatabaseSourceModule

5.2.1 Klassen im DatabaseSourceModule

- *DatabaseSourceModule*
  Wie der Name der Klasse schon vermuten lässt, ist diese Klasse das Herzstück des Moduls, welche beim Starten des Moduls geladen wird.

- *CreateEventFromDB*
  Diese Klasse ist dafür verantwortlich, die für die Rekonstruktion eines Ereignisses nötigen Schritte an entsprechende Methoden und Klassen zu delegieren.

- *DBQueries*
  Diese Methoden dieser Klasse führen Datenbankabfragen aus, welche sich mitunter aus mehreren Einzelanfragen zusammensetzen. Sie konsolidieren die Ergebnisse der Einzelanfragen und liefern sie teilweise in transformierter Form an den Aufrufer.

- *Event*
  Die Datensätze eines über seinen Schlüssel aus der Datenbank gezogenen MPE werden zur Weiterverarbeitung zunächst in ein Objekt der Klasse *Event* transformiert. Diese Klasse enthält Objekte von Typ *EventDataEntry* und bietet Zugriffsmöglichkeiten auf die Elemente eines Ereignisses.

- *EventDataEntry*
  Ein *EventDataEntry* repräsentiert ein Element mit einem oder mehreren Werten. Ist ein Element mehr als einmal in einem XML-Dokument vorhanden, so enthält das Element genau so viele Werte, wie oft das Element in dem Dokument vorkommt.

- *SchemaTree*
  Erstellt ein Modell eines Schemas und bietet die Möglichkeit durch traversieren des XML-Schema-Baums und einem übergebenen *Event*-Objekt ein XML-Dokument zu rekonstruieren.

- *SqlQueryStrings*
  In dieser Klasse werden die Strings der auszuführenden SQL - Abfragen entsprechend der aus der Anwendung übergebenen Werte generiert.

- *TableModelFromResultSet*
  Das Ergebnis einer Datenbankabfrage, eine ReslutSet, wird dieser Klasse übergeben und sie überführt dieses in eine Tabellendarstellung. Die Klasse wird zur Darstellung der Ergebnisse von Datenbankanfragen in der GUI benutzt.

- *QueriesWindow*
  Repräsentiert das Fenster zur Eingabe bzw. Auswahl von Datenbankabfragen.

- *ResultWindow*
  Übernimmt die Darstellung der Anfrageergebnisse.

5.2.2 Starten des Moduls

Wie beim *DatabaseTargetModule* werden auch hier zu Beginn alle Schemata mit der Datenbank synchronisiert, da auch zum Unmarshalling eines Ereignisses, was die Rekonstruktion eines XML-Dokuments impliziert, die Information aus der *TableInformation*-Instanz benötigt wird.

Des Weiteren verfügt das *DatabaseSourceModule* über die Möglichkeit, über einen Konfigurationswert direkt Ereignisse aus der Datenbank zu rekonstruieren. Dazu wird beim Starten des Moduls die *ModuleProperty SQLQuery* überprüft, in der ggf. eine Datenbankabfrage in Form eines Strings vom Benutzer gegeben sein kann. Ist diese Property mit einem String gesetzt, so wird die Abfrage an die Datenbank ausgeführt und aus den Ereignissen welche in der Ergebnismenge stehen, werden gültige MPE erzeugt. Der Abfragestring in der Property SQLQuery kann zur Laufzeit modifiziert werden, wodurch dann auch direkt die Abfrage ausgeführt wird und aus der Ergebnismenge MPE rekonstruiert werden. Zu beachten ist dabei lediglich, *dass die Ergebnismenge der eingegebenen Abfrage die Spalte linkid enthält*, denn nur darüber ist eine Rekonstruktion der Ereignisse möglich.

5.2.3 Unmarshalling der Mikroprozessereignisse

Die für das Rekonstruieren eines MPE verantwortliche Klasse *CreateEventFromDB* benötigt lediglich den Schlüsselwert eines in der Datenbank gespeicherten Ereignisses, um dieses mit allen seinen Datensätzen aus der Datenbank zu holen und ein *ValidEventPacket* zu erstellen.

Im ersten Schritt werden anhand des gegebenen Schlüssels über die Methode *getEventByID* die Daten aus allen an der Speicherung dieses Events betroffenen Tabellen geholt. An genau dieser Stelle wird wieder auf die im Singleton *TableInformation* gespeicherte Information, welche Tabellen an der Speicherung eines Ereignistyps beteiligt sind, zugegriffen.

Die Methode *getEventByID* liefert der *CreateEventFromDB* Klasse die Daten in Form eines *Event*-Objekts, welches dann zur Erstellung des *ValidEventPackets* genutzt wird.

Im zweiten Schritt wird eine Instanz der Klasse *SchemaTree* erzeugt, welche zunächst aus dem *Event*-Objekt das definierende Schema holt und über die definierten Zugriffmethoden

des *Event*-Objekts die Werte für die separat im Ereignis stehenden Attribute msdt, Timestamp und Date aus dem Event extrahiert.

Im dritten Schritt wird durch das erzeugte *SchemaTree*-Objekt das XML-Dokument wiederhergestellt.

Die Wiederherstellung eines Dokuments erfolgt, indem ein Modell des definierenden Schemas erstellt wird und dieser Baum vom Elementknoten des Modells per Tiefensuche bis zu den Blättern traversiert wird.

Für jeden Elementknoten des XML-Baums wird im *Event*-Objekt nachgesehen, ob für diesen Knotennamen ein- oder mehrere Werte vorhanden sind. Ist das der Fall, wird in dem zu rekonstruierenden Dokument ein, bzw. mehrere entsprechend der Anzahl der vorhandenen Werte, Elementknoten angelegt und mit den vorhandenen Werten aus dem *Event*-Objekt gefüllt. Hat der Elementknoten Attribute, so werden Attributknoten erstellt und auch deren Werte aus dem *Event*-Objekt geholt und die erstellten Attributknoten an den Elementknoten angehangen.

Das erstellte XML-Dokument wird zusammen mit den anderen separat extrahierten Attributen an die Methode *createValidEventPacket* der Klasse *org.electrocodeogram.misc.xml.ECGWriter* übergeben und damit erstellt diese Methode dann ein *ValidEventPacket*.

5.2.4 Die DBSource-GUI

Über die GUI ist es möglich, entweder eine eigene Abfrage an das Datenbanksystem zu stellen oder eine vordefinierte Abfrage auszuwählen und diese mit eigenen Parametern zu versehen.

Die Ergebnisse dieser Abfrage werden dann in Form einer Tabelle angezeigt und es besteht die Möglichkeit, bestimmte oder auch alle Ereignisse der Ergebnismenge zu selektieren, um daraus die ursprünglichen MPE zu rekonstruieren.

6 Abschließende Betrachtungen

Bisher werden im ECG-Lab die von den Sensoren gesendeten und vom ECG-Lab empfangenen Ereignisse durch ein *FileSystemTargetModule* in einer Textdatei gespeichert, indem dort nacheinander für jedes MPE die Stringrepräsentation des Ereignisses hineingeschrieben wird. Damit ist eine Auswertung der Ereignisdaten nur sehr schwer und mühselig möglich.

Mit dem im Rahmen dieser Studienarbeit entwickelten *DatabaseTargetModule* wird das ECG-Lab insofern in seiner Funktionalität erweitert, dass die Daten der empfangenen MPE nun strukturiert in einer Datenbank abgespeichert werden und somit durch Datenbankabfragen analysiert und ausgewertet werden können.

Dazu wird im ECG-Lab eine grafische Oberfläche bereitgestellt, durch die vordefinierte Abfragen an die Datenbank ausgewählt oder auch eigene Abfragen eingegeben werden können. Die Ergebnisse dieser Abfragen werden dann in Form von Tabellen in einem Fenster präsentiert.

Ebenso besteht durch das *DatabaseSourceModule* die Möglichkeit, aus den gespeicherten Daten auch wieder Ereignisse in ihrer ursprünglichen Form zu rekonstruieren, um diese beispielsweise zu Simulationszwecken an andere Module weiterzuleiten.

Während der Entwicklung der Module habe ich Unit-Tests durchgeführt, um so die Funktionalität einzelner und zusammenhängender Methoden sicherzustellen. Ein Überblick der Tests befindet sich im Anhang A. Als Testumgebung habe ich eine lokale MySQL Datenbank installiert und von bereitgestellten Testmodulen Ereignisse generieren lassen.

Um die Funktionsweise des automatischen Mappings der XML Schemata auf das Datenbankschema überprüfen zu können, habe ich jedes Schema der für das automatische Mapping verantwortlichen Methode übergeben und neben den Informationsausgaben des Loggers die durch das Mapping erstellten Tabellen in der Datenbank mit ihren Namen, Spalten und Datentypen anhand der Schemata nachvollzogen und überprüft.

Die Unit-Tests bezüglich der Methoden zum Einfügen der Ereignisdaten in die Datenbank bekamen als Argumente die vom Testmodul generierten MPE und ich konnte mir durch Abfrage der Datenbank die Ergebnisse der Einfügeoperationen anzeigen lassen.

Das Unmarshalling der Ereignisse ließ sich testen, indem ich eingefügte Ereignisse wieder aus der Datenbank rekonstruieren ließ und anschließend überprüft habe, ob zum einen ein rekonstruiertes MPE bezüglich seines Schemas validiert wird und zum anderen ob die Stringrepräsentation des rekonstruierten MPE mit der Stringrepräsentation des ursprünglichen MPE übereinstimmt.

Durch die Integration der Module in das Gesamtsystem wurde automatisch das Starten der Module und die damit verbundene Abbildung der Strukturen aller im ECG-Lab registrierten Schemata auf die Datenbank durchgeführt.

Weiterführend ist zu sagen, dass es sicherlich noch einige Dinge an den entwickelten Modulen zu verbessern gibt. Beispielsweise ist die Abfrage und Präsentation der Daten durch die GUI nicht optimal. In Tabellenform werden nur die Standardinformationen der Ereignisse angezeigt, welche in der *commondata* Tabelle stehen. Sollen die kompletten, also auch ereignisspezifischen Informationen eines MPE angezeigt werden, so geschieht das bisher, indem nacheinander die Daten jedes Ereignisses ausgeschrieben werden, was keine strukturierte Gesamtdarstellung ergibt. Aufgrund der Tatsache, dass gleichzeitig die Details verschiedener Ereignistypen angezeigt werden können, ist die konsolidierte Darstellung aller Ereignisdaten in einer Tabelle nicht möglich, da jeder Ereignistyp unterschiedliche Attribute und somit Spalten in der Tabelle hat.

Die Komplexität der Aufgabenstellung war nicht von Beginn der Arbeit an klar zu erkennen, sondern erst während der Analyse und der langen Evaluierungsphase für den Entwurf, in der sich nach und nach die Komplexität der Anforderungsdefinition abzeichnete. Es kam während der Evaluationsphase immer wieder zu neuen oder modifizierten Anforderungen, welche in bisherigen Betrachtungen integriert werden mussten.

Wie meist auch im realen Leben der Softwaretechnik stand den sich verändernden Anforderungen und einer langen Evaluierungsphase ein begrenzter Zeitrahmen gegenüber, weshalb die entwickelten Module auch nicht den Anspruch erheben, ein technisch einwandfreies und detailliert ausgefeiltes Ergebnis zu repräsentieren. Ich gehe jedoch davon aus, dass ich die grundlegenden Funktionen zur Abspeicherung, Analyse und Rekonstruktion von Ereignisdaten im ECG-Lab auf einem geeigneten Weg umgesetzt habe und Dinge wie die Ausgestaltung der GUI oder anderer technischer Details durch den interessierten ECG-Programmierer auf dieser Grundlage fortgesetzt werden können.

7 Quellen

[1] Frank Schlesinger: „Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung“ (Diplomarbeit 15.9.2005).

[2] Spot Media Multimedia Agentur: „Einsatzgebiete von nativen XML-Datenbanken“. URL: http://www.spot-media.de/index.php/article_product/detail/430 [Stand: 27.11.06]

[3] Ronald Bourret: „XML and Databases“. URL: http://www.rpbourret.com/xml/XMLAndDatabases.htm [Stand: 27.11.06]

[4] Ronald Bourret: “Mapping W3C Schemas to Object Schemas to Relational Schemas” URL: http://www.rpbourret.com/xml/SchemaMap.htm [Stand: 27.11.06]

[5] Jan Baumann (10.06.05): “XML & Datenbanken”. URL: http://www.wwz.unibas.ch/wi/lehre/swi/arbeiten05/xmldb.pdf [Stand: 27.11.06]

[6] Sasaki/Trippel/Milde: “Mapping von XML und Relationalen Datenbanken“ (28.11.2003) URL: http://www.spectrum.uni-bielefeld.de/Courses/Winter03/XQuery/2003-12-12/xquery_rdb_xml.pdf [Stand: 29.11.2006]

[7] Michael Eckhardt: XML-Datenbanken (Vortrag) URL: http://www-1.informatik.fh-wiesbaden.de/~schaefer/Lehrveranstaltungen/DB-Vertiefung/XML%20Datenbanken%20Vortrag.pdf [Stand: 28.11.2006]

[8] MySQL: http://www.mysql.com/ [Stand: 27.11.06]

[9] w3schools: „XML Tutorial“. URL: http://www.w3schools.com/xml/default.asp [Stand: 27.11.06]

[10] Helmut Knappe: „Mapping zwischen XML und Datenbank“. (JavaSPEKTRUM 3/2006)

[11] Thomas Gruber: „XML Abbildung auf relationale Datenbanken“. Universität Passau URL: http://www.fmi.uni-passau.de/~gruber/Hauptseminar/Hauptseminar.pdf [Stand: 27.11.06]

[12] ALTOVA.COM: „Datentypen in DB, die von XML Schema generiert wurden“. URL: http://www.altova.com/de/manual2007/XMLSpy/spyprofessional/index.html?db2xsd_mysql.htm [Stand: 27.11.06]

[13] W3C: “XML Schema Part 1: Structures Second Edition”
URL: http://www.w3.org/TR/xmlschema-1/ [Stand: 28.11.2006]

[14] Andreas Rauch: “XML Schema Komponenten Übersicht”
URL: http://www.devtrain.de/artikel_446.aspx [Stand: 28.11.2006]

[15] Hibernate: “Relational Persitence for Java und .NET”
URL: http://www.hibernate.org/ [Stand: 28.11.2006]

[16] Uwe Hohenstein, Jörg Barthold: “Performante Anwendungen mit Hibernate”. (JavaSPEKTRUM 4/2006)

[17] Edmon Begoli: „Hibernate 3 Adds XML-Relational Persistence“
URL: http://www.devx.com/Java/Article/27896 [Stand: 27.11.06]

[18] Scott Ryan: “XML Processing with Java Object Technology” (05.04.2004)
URL: http://dev2dev.bea.com/pub/a/2004/05/ryan_xml.html [Stand: 27.11.06]

[19] Christian Weber : “XML-Grundlagen” (Seminar)
URL: http://wwwdvs.informatik.uni-kl.de/courses/seminar/WS0203/ausarbeitung1.pdf [Stand: 28.11.2006]

[20] JAXME: Welcome to JaxMe 2
URL: http://ws.apache.org/jaxme/ [Stand: 27.11.06]

[21] Java Sun: “Java Architecture for XML Binding”
URL: http://java.sun.com/webservices/jaxb/ [Stand: 28.11.06]

[22] The Apache XML Project: “Welcome to XMLBeans”
URL: http://xmlbeans.apache.org/ [Stand: 27.11.06]

[23] The Apache XML Project: “XMLBeans Overview”
URL: http://xmlbeans.apache.org/overview.html [Stand: 27.11.06]

[24] Castor: “The Castor Project”
URL: http://castor.org/ [Stand: 27.11.06]

[25] Castor: “Using Castor XML”
URL: http://www.castor.org/xml-framework.html [Stand: 27.11.06]

[26] Castor: “Using the Source Code Generator”
URL: http://www.castor.org/sourcegen.html [Stand: 28.11.06]

[27] Castor: “Castor JDO – An introduction”
URL: http://www.castor.org/jdo-introduction.html [Stand: 28.11.06]

[28] Sebastien Gignoux, Keith Visco: “Castor XML Mapping”
URL: http://www.castor.org/xml-mapping.html [Stand: 27.11.06]

Anhang I Tests

Tests zum DatabaseTargetModule

Getestete Methode	Übergebender Parameter	Beschreibung	Erwartete Ausgabe
CreateSQLTests : testCreateCommonInsertStatement			
CreateSQL: *createCommonDataInsertStmt*	ValidEventPacketProxy eines MPE	Testet die Erzeugung des SQL Strings zum Einfügen der Standardattribute eines MPE in die Datenbank	Gültiger SQL insert – String, welcher die Werte der Standardattribute des übergebenen MPE enthält.
CreateSQLTests : testCreateMSDTInsertStmt			
CreateSQL: *createMSDTInsertStmts*	ValidEventPacketProxy eines MPE	Testet die Erzeugung des SQL Strings zum Einfügen der eventspezifischen Attribute eines MPE in die Datenbank	Gültiger SQL insert - String welcher die Werte der Standardattribute des übergebenen MPE enthält.
CreateSQLTests : testCreateTableStmt			
CreateSQL: *createTableStmt*	Von einem SchemaProxy automatisch generierter Vektor mit Elementen vom Typ Table	Testet die Erzeugung der SQL-Strings zum Erstellen von Tabellen welche durch das automatische Mapping generiert wurden.	Die create Table SQL Strings für jede im übergebenen Vektor vorkommende Tabelle
CreateSQLTests : testCreateCommonDataTable()			
CreateSQL: *createCommonDataTable*	Von einem SchemaProxy des msdt.common.xsd Schemas automatisch generiertes Table-Objekt	Testet die Erzeugung des SQL-Strings zum Erstellen der commondata Tabelle.	Das ‚create table commondata…' Statement mit allen Standardattributen.
DBCommunicatorTest: testGetDBConnection			
DBCommunicator: getDBConnection		Testet das Erstellen der Datenbankverbindung durch den DBCommunicator	Das Connection-Objekt ist nicht null, die Datenbank ist verbunden.
DBCommunicatorTest: testCloseDBConnection			
DBCommunicator: closeDBConnection		Testet das Schließen der Datenbankverbindung durch den DBCommunicator	Die Datenbank ist nicht mehr verbunden
DBCommunicatorTest: testGetMetadataInColumnOrder			

Getestete Methode	Übergebender Parameter	Beschreibung	Erwartete Ausgabe
DBCommunicator:getMetadataInColumnOrder	Name einer existierenden Tabelle in der Datenbank	Testet das Abfragen der Metadaten einer gegebenen Tabelle aus der Datenbank	Ausgabe aller Spaltennamen und -typen der übergebenen Tabellen
DBCommunicatorTest: testInsertStatement			
DBCommunicator: executeStmt	Eine bereits erzeugte SQL-Anweisung	Testet das Ausführen der übergebenen SQL Anweisung an die Datenbank.	Die Anweisung konnte ausgeführt werden, keine SQLException geworfen.
DBCommunicatorTest: testInsertEvent			
DBCommunicator: insertEvent	Ein erzeugtes ValidEventPacket-Objekt	Testet das Einfügen eines kompletten Ereignisses in die Datenbank mit allen beteiligten Methoden.	Das ValidEventPacket-Objekt konnte eingefügt werden, es steht nun in der Datenbank. Keine SQLException
DBCommunicatorTest: testInsertEventDBDown			
DBCommunicator: testInsertEventDBDown	Ein erzeugtes ValidEventPacket-Objekt	Testet das Einfügen eines Ereignisses bei nicht laufender Datenbank.	Das ValidEventPacket-Objekt konnte nicht eingefügt werden, es steht im EventBuffer.
DBCommunicatorTest: testGetInformationAndSyncTables()			
DBCommunicaotr: getInformationAndSyncTables		Alle Schemata sollen mit der Datenbank synchronisiert werden.	Alle in der Datenbank vorhandenen Tabellen werden synchronisiert, hinzugekommene Tabellen werden in die Datenbank abgebildet. Alle Schemata sind komplett in der Datenbank abgebildet.
DBCommunicatorTest: testGetTableNames			
DBCommunicator: getTableNames		Die Namen aller Tabellen in der sollen von der Datenbank abgefragt und in Form eines Vektors geliefert werden.	Ein Vektor der die Namen aller Tabellen in der Datenbank enthält.
MetadataPoolTests: testGetMetadataForAllTables			
DBTablesMetadataPool: getMetaInformationVectors	Ein Objekt der Klasse DBCommunicator, um ggf. eine Datenbankverbindung herstellen zu können und die Metadaten aller Tabellen abfragen zu können	Testet die DBTablesMetadataPool – Instanz, ob diese einen Vektor mit den Metadaten aller Tabellen der Datenbank zurückliefert.	Der zurück gelieferte Vektor mit Metadaten ist nicht leer.

Getestete Methode	Übergebender Parameter	Beschreibung	Erwartete Ausgabe
MetadataPoolTests : testGetMetadataVector()			
DBTablesMetadataPool: testGetMetadataVector	Name einer existierenden Tabelle, von der die Metadaten geliefert werden sollen und ein DBCommunicator-Objekt.	Testet die DBTablesMetadataPool-Instanz, ob diese die Metainformation der gegebenen Tabelle liefern kann.	Die Metadaten der übergebenen Tabelle, falls diese in der Datenbank existiert.
SchemaProxyTest : testGenerateSchemaTypeSystem()			
XMLSchemaProxy: generateSchemaTypeSystem(		Testet die Methode , welche ein SchemaTypeSystem-Objekt des Schemas liefert**,** für das der XML SchemaProxy erstellt wurde.	Ein SchemaTypeSystem-Objekt welches ein Modell des Schemas repräsentiert
SchemaProxyTest : testGetCoreMsdtName			
XMLSchemaProxy: getCoreMsdtName		Testet die Methode , welche den Namen des Schemas liefern soll, für das der XMLSchemaProxy erzeugt wurde. Dabei soll der Name nicht das Präfix ‚msdt.' und das Suffix ‚.xsd' enthalten	Der Name des Schemas ohne Prefix und Suffix.
SchemaProxyTest : testGetCommonProperties			
XMLSchemaProxy: getCommonProperties		Die getestete Methode soll ein Table-Objekt liefern, welches die commondata Tabelle in der Datenbank repräsentiert.	Ein Table-Objekt, welches alle Standardattribute mit entsprechenden Datentypen enthält.
SchemaProxyTest : testGetSchemaProperties			
XMLSchemaProxy: getSchemaProperties		Die getestete Methode soll einen Vektor liefern, der alle Table-Objekte enthält, welche zur kompletten Abbildung des Schemas in die Datenbank nötig sind.	Ein Vektor, der alle Table-Objekte zur Abbildung des kompletten Schemas in die Datenbank enthält.
SchemaProxyTest : testFindElement			
XMLSchemaProxy: findElementDeklaration	Der Name des zu suchenden Elements im XML-Dokument	Die Methode soll das SchemaType-Objekt des zu suchenden Elements liefern.	Das SchemaType-Objekt des gefundenen Objekts, falls das Element nicht existiert, wird null zurückgegeben.
SchemaProxyTest : testSynchronizeTable			

Getestete Methode	Übergebender Parameter	Beschreibung	Erwartete Ausgabe
XMLSchemaProxy: synchronizeSchemaToDatabase		Testet die Methode, die das Schema mit der Datenbank synchronisieren soll, für das der XMLSchemaProxy erzeugt wurde.	Nicht existierende Tabellen des Schemas sollen erstellt werden und vorhandenen Tabellen sollen synchronisiert werden, also ggf. hinzugekommene Spalten im Datenbankschema ergänzt werden.
TestValidEventPacketProxy: testGetMSDTName			
ValidEventPacketProxy: getMSDTName		Soll den msdt-Namen des generierten ValidEventPackets liefern, für den der ValidEventPacketProxy erzeugt wurde.	Der msdt Name des ValidEventPackets.
DatabaseTargetModuleTests : testControlEventToDB			
DatabaseTargetModule: write	Erzeugtes ValidEventPacket-Objekt	Die Methode write soll das übergangene ValidEventPacket-Objekt entsprechend weitergeben und das Ereignis soll komplett in die Datenbank eingefügt werden	Alle Daten des ValidEventPacket-Objekts sind vollständig in den dem Ereignistyp entsprechenden Tabellen der Datenbank enthalten.
DatabaseTargetModuleTests : testStartWriter			
DatabaseTargetModule: startWriter		Die Methode soll ein DBCommunicator-Objekt für das Modul erstellen und das Synchronisieren aller Schema durchführen.	Das DBCommunicator-Objekt konnte erstellt werden, die Datenbank konnte verbunden werden und alle Schemata wurden mit der Datenbank synchronisiert.

Test zum DatabaseSourceModule

Getestete Methode	Übergebender Parameter	Beschreibung	Erwartete Ausgabe
SchemaTreeTests: testGetSchemaElements()			
SchemaTree: getSchemaElements		Die getestete Methode soll für das Ereignis, welches zur Erzeugung des SchemaTree-Objekts übergeben wurde, das entsprechende XML-Dokument erzeugen.	Das XML-Dokument, welches exakt dem XML-Dokument entspricht, das das Ereignis vor seiner Abspeicherung in die Datenbank enthalten hatte.