Watzke, Tom-Steve

Entwicklung einer Datenbankschnittstelle als Grundlage für Shop-Systeme unter dem Betriebssystem Askemos®

Diplomarbeit

Hochschule Mittweida

University of Applied Sciences

Fachbereich Mathematik/Physik/Informatik

Mittweida, 2009

Watzke, Tom-Steve

Entwicklung einer Datenbankschnittstelle als Grundlage für Shop-Systeme unter dem Betriebssystem Askemos®

eingereicht als

Diplomarbeit

an der

Hochschule Mittweida

University of Applied Sciences

Fachbereich Mathematik/Physik/Informatik

Mittweida, 2009

Erstprüfer: Prof. Dr. rer. nat. Konrad Schulz
Zweitprüfer: Dipl.-Ing. Eberhard Richter

vorgelegte Arbeit wurde verteidigt am: 16.10.2009

Bibliographische Beschreibung:

Watzke, Tom-Steve:
Entwicklung einer Datenbankschnittstelle als Grundlage für Shop-Systeme unter dem Betriebssystem Askemos® - 2009 - 59 S.
Mittweida, Hochschule Mittweida, Fachbereich Mathematik / Physik / Informatik, Diplomarbeit, 2009

Referat:

Ziel der Diplomarbeit war es, eine Datenbankschnittstelle und eine Komponente des Web-Portals MyHomeNotar zu entwickeln. Das Web-Portal MyHomeNotar soll es juristischen Personen ermöglichen, Willenserklärungen elektronisch durchzuführen und zu verwalten. Der Kern von Shop-Systemen wird im Web-Portal MyHomeNotar als ein Onlineshop für Dienstleistung verwendet. Darin sollen Dienstleistungen angeboten werden, die zu der ausgewählten Willenserklärung passen. Eine Grundvoraussetzung ist die Schnittstelle zu SQLite, die im Rahmen der Diplomarbeit zu erweitern war. Schließlich erfolgt die Beschreibung und Auswertung der Entwicklung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingsverzeichnis

Tabellenverzeichnis

Glossar

API

Programmierschnittstelle (engl. Application Programming Interface)

BLOB

große binäre Objekte (engl. Binary Large Object)

Condition-Variable

Eine Condition-Variable ist eine Bedingungsvariable vom Typ Boolean wie ein Semaphor und wird zur Synchronisation von Bedingungen in Threads verwendet.

Mutex

Ein Mutex ist eine Steuer-Variable vom Typ Boolean wie ein Semaphor und wird zur Synchronisation von Threads verwendet.

Shop-System

Die Softwaregrundlage von Onlineshops ist ein Shop-System. Es basiert auf Datenbanken und Webanwendungen oder auf statischen Internetseiten. Ein Shop-System beinhaltet beispielsweise eine Warenkorbfunktionalität und ein Bezahlsystem.

Thread

"Ein **Thread** ist ein *Aktivitätsträger* (sequenzieller "*Ausführungsfaden*") mit minimalem Kontext (Stack und Register) innerhalb einer *Ausführungsumgebung* (Prozess). Jeder Prozess besitzt in diesem Fall mindestens einen (initialen) Thread. Alle Threads, die zu ein und demselben Prozess gehören, benutzen denselben Adressraum sowie weitere Betriebsmittel dieses Prozesses gemeinsam." (in [15], S.181)

WebDAV

WebDAV ist eine Erweiterung des HTTP/1.1 Protokolls und ermöglicht es, mehrere Dateien hochzuladen. Vorteilhaft dabei ist, dass keine weiteren Ports wie bei dem FTP-Protokoll notwendig sind. Im Standard RFC4918 [2] ist auch eine Versionierung für WebDAV vorgesehen. Datenübertragungen mittels HTTPS werden außerdem verschlüsselt, so dass die übertragenen Informationen vor unberechtigtem Zugriff geschützt werden.

1 Einführung

1.1 Hintergrund der Diplomaufgabe

Die vorliegende Diplomarbeit wurde im Auftrag der Firma pitcom PROJECT GmbH entwickelt. Sie gehört zur pitcom Unternehmensgruppe, die unter anderem im Bereich Forschung und Entwicklung tätig ist. Ein Entwicklungsfeld des Bereiches befasst sich mit Software für das Betriebssystem Askemos®, welches im Verlauf der Einführung vorgestellt wird.

Die Diplomarbeit ergab sich als eine Aufgabe aus dem folgenden Projekt: Im Rahmen des Projektes MyHomeNotar entwickelt die pitcom PROJECT GmbH eine softwaregestützte Lösung zur Abwicklung von Willenserklärungen. Das zu entwickelnde Internet-Portal MyHomeNotar soll es juristischen Personen ermöglichen, Willenserklärungen als elektronische Dokumente zu erstellen und zu verwalten. Durch Eigenschaften einer qualifizierten elektronischen Signatur sollen die elektronischen Dokumente für eine gerichtsfeste Beweisführung zur Verfügung stehen. Als ein Bestandteil des Projektes MyHomeNotar ist ein Shop-System geplant, in dem zur ausgewählten Willenserklärung passende Dienstleistungen angeboten werden.

Die Aufgabe der Diplomarbeit ist die Entwicklung eines allgemeingültigen Kerns von Shop-Systemen unter Verwendung des speziellen Betriebssystems Askemos®. Für die dazu notwendige Datenbank soll das Datenbank-Management-System SQLite verwendet werden. Ein Teil der Diplomaufgabe besteht in der Entwicklung einer Datenbank-Schnittstelle zwischen Askemos® und SQLite zur parallelen Verarbeitung von SQL-Anweisungen.

1.2 Kern von Shop-Systemen

Es ist ein Kern von Shop-Systemen zu erstellen, in dem grundlegende Funktionen von Shop-Systemen allgemeingültig implementiert sind, so dass dieser Kern für eine Vielzahl von konkreten Möglichkeiten angepasst und verwendet werden kann.

In einem Shop-System ermöglicht eine Benutzerverwaltung die Unterscheidung von drei grundlegenden Benutzergruppen: Käufer, Verkäufer und Kundendienst. Käufer können Bestellungen verwalten und Angebote des Shop-Systems annehmen. Bestellungen werden direkt an den Verkäufer übermittelt, wobei das Shop-System automatisch die Rechnung erstellt. Verkäufer verwalten ihre Angebote und führen die Bestellung bis hin zur Lieferung aus.

Der Kern von Shop-Systemen ist so abstrakt zu halten, dass er als eigenständiges Produkt für viele Einsatzzwecke benutzerspezifisch eingestellt und genutzt werden kann. Geeignete Tabellen sind zu definieren, die je nach Anforderung des Zweckes erweiterbar sind, beispielsweise zum Anbieten von Dienstleistungen oder Medien wie Bücher, CDs, DVDs und andere.

Basierend auf Askemos® können einem Shop-System im Vergleich zu anderen Shop-Systemen folgende Eigenschaften garantiert werden:

Zurechenbarkeit in digitaler Identität: eine Aktion kann ihrem Akteur eindeutig und nachweisbar zugerechnet werden

Integrität: eine gespeicherte Information zu einer Aktion kann nicht unerkannt verfälscht werden (nicht korrumpierbar)

Vertraulichkeit: Informationen, Daten und Dokumente können von einem Benutzer und von ihm zugelassenen Beteiligten, mit Rechten behaftet, verfügbar gemacht werden (formwahrende Transaktion, im Streitfall beweistauglich)

Verfügbarkeit: alle Daten, Dokumente und Informationen werden redundant gespeichert (redundante Serverarchitektur)

Rechtsverbindlichkeit: für alle Aktionen ist eine Kontrolle sichergestellt (Umsetzung des Signaturgesetzes mit jeder einzelnen Aktion).

1.3 Askemos® – ein virtuelles Betriebssystem

Die Informationen über Askemos® wurden den Dokumenten [1],[3] und [4] entnommen. Seine Eigenschaften werden in folgenden Abschnitten genauer beschrieben.

Das Betriebssystem Askemos® ist grundsätzlich fair, störsicher und nicht korrumpierbar. Es existieren keine zentralen Verwaltungsmittel, die übergeordnete oder besondere Rechte besitzen: Im Gegensatz zu anderen Betriebssystemen existiert in Askemos® kein zentraler Benutzer wie "Administrator" oder "root", sondern die "Öffentlichkeit" (engl. public) als Basis jeder Verarbeitung.

Folgendes Merkmal ist dabei zu berücksichtigen: Niemand darf der Öffentlichkeit (engl. public) Rechte entziehen oder sie manipulieren. Sie wird als Wurzel jeder Verarbeitung verwendet und bildet die Grundlage von Benutzerkonten. Folglich bildet jeder Benutzer mit seinen Strukturen eine eigene Hierarchie und erlaubt anderen seine Daten zu manipulieren.

Askemos® schafft damit eine flexible und sichere Basis zur fälschungssicheren Verarbeitung von Informationen: gewöhnliche Dokumente, Dateien oder Nachrichten zwischen Prozessen, Threads oder "Plätzen". Dabei hat jede Information aktionsbezogene Eigenschaften, Eigentumsrechte und Befähigungsrechte.

Askemos® repräsentiert somit ein autonomes und verteiltes Betriebssystem, das jede Informationsverarbeitung selbständig oder unter Benutzung von Hilfsmitteln verarbeitet und beantwortet. Hilfsmittel sind beispielsweise Konvertierungsprogramme, externe Datenbankanwendungen, Komprimierungsprogramme oder technische Dienste, wie E-Mail- oder SMS-Versand.

Ein Ziel von Askemos® ist es, Schutzmechanismen unseres Rechtssystems mit rechtsbindender Wirkung zu garantieren. Alle Informationen und Informationsverarbeitungen sind während der Übertragung im Netzwerk und Internet gesichert und werden mehrfach geprüft. Verschiedene Askemos®-Server führen dieselbe Verarbeitung gleichzeitig durch und verhalten sich somit wie eine Menge von zusammenarbeitenden Agenten. Dabei wird der Wahrheitswert einer Informationsverarbeitung nach dem Mehrheitsprinzip bestimmt und so Inkonsistenzen vorgebeugt.

Die Störsicherheit wird höher, je mehr Askemos®-Server an der Informationsverar-

beitung als Beobachter teilnehmen. Dabei hat jeder Server den gleichen Informationsgehalt. Diese Redundanz ist vorteilhaft, wenn ein oder mehrere Server ausfallen. Das Betriebssystem Askemos® ist generell fair, indem es sich während der Informationsverarbeitung nicht irreführen lässt. Somit ist Askemos® besonders geschaffen für die Verwaltung hochsensibler Daten und intellektuellen Eigentums wie z.B. im Rechtswesen, Gesundheitswesen und in der Verwaltung.

Es werden ausschließlich öffentliche Standards wie SMTP oder HTTP verwendet. Das universelles Datenaustauschformat XML wird verwendet, um Datenverluste durch veraltete - besonders binäre - Datenformate zu verhindern. Viele Binärformate können durch Hilfsprogramme in XML konvertiert werden. Aufgrund der öffentlicher Standards ist es auch möglich, mit anderen externen Softwaresystemen oder technischen Diensten Informationen auszutauschen.

1.4 Ableitung der konkreten Aufgabe

Der Kern von Shop-Systemen ist auf eine Datenbank angewiesen. Dabei soll eine geeignete Datenbank-Struktur entwickelt werden, die für jede Art von Shop-Systemen eingesetzt werden kann. Auch die Verarbeitung muss so gestaltet werden, dass jeder Bestandteil dynamisch anpassbar ist.

Bisherige Lösungen für Datenbanksysteme in Askemos® sind nicht zufriedenstellend. Datenbank-Operationen werden auf der Anwendungsebene (durch Plätze) von Askemos® transformiert, wodurch Abfragestrukturen zu umständlich werden, als dass sie die Komplexität von SQL abbilden können. Nachteilig sind die Strukturen zur Datenhaltung der berechneten Informationen, wobei Verarbeitungen mit mehr Informationen langsamer werden und Zeitüberschreitungen (engl. timeout) auftreten.

Es existieren jedoch Möglichkeiten, eine Datenbank durch eine externe Software zu verwalten. SQLite ist kein Datenbank-Management-System im Sinne einer Anwendung, sondern eine Bibliothek zur Ausführung von SQL-Anweisungen. Das Problem ist, dass SQLite eine Datenbank in einer Datei des Dateisystemes verwaltet. In Askemos® steht eine Schnittstelle zum Arbeiten mit SQLite bereit. Sie ist so weiterzuentwickeln, dass die Speicherung der SQLite-Datei innerhalb von Askemos® stattfindet.

2 Grundlagen

2.1 Wichtige technische Details von Askemos®

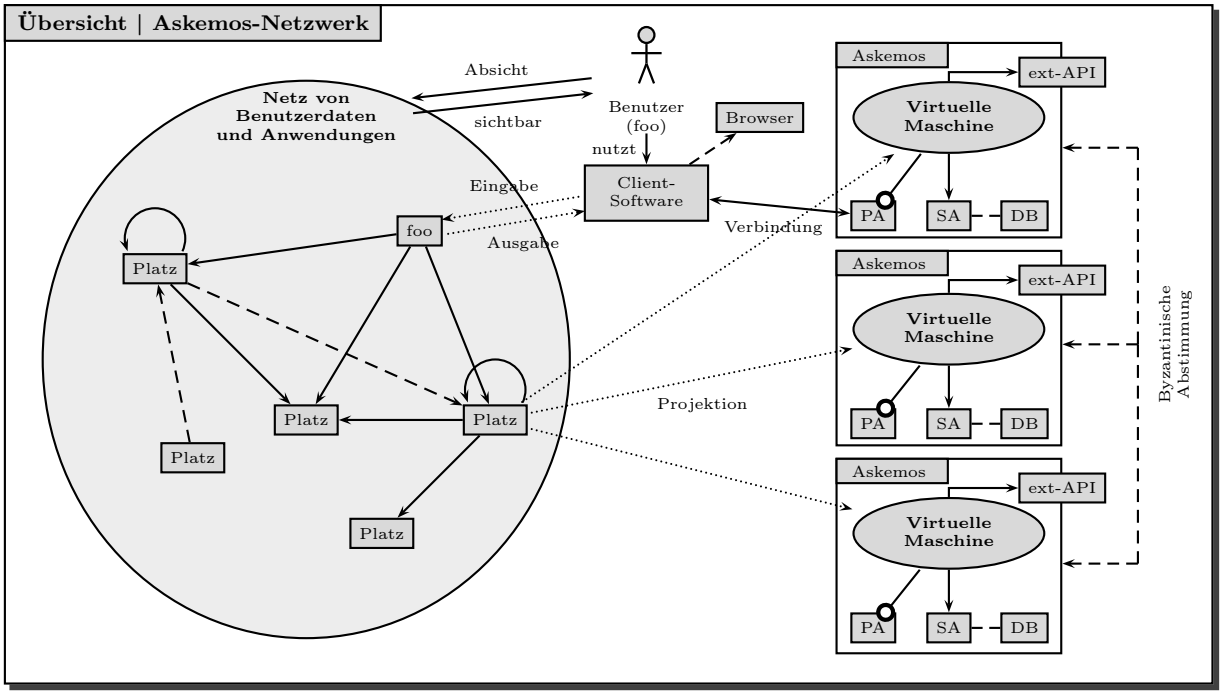

Abbildung 2.1: Übersicht über das Askemos®-Netzwerk

Abbildung 2.1 verdeutlicht die Struktur von Askemos®. Ein Benutzer kann mithilfe einer Client-Software Verbindung mit der Verarbeitungseinheit "PA" (Protokoll-Adapter) eines Askemos®-Servers herstellen. Exemplarisch kann als Client-Software ein Internet-Browser verwendet werden. Der Protokoll-Adapter unterstützt dazu die Protokolle HTTP, HTTPS und WebDAV. Auf diese Weise erhält der Benutzer Informationen in Form von Webseiten auf seinen Internet-Browser.

Der Storage-Adapter (SA) unterstützt Verarbeitungen auf dem Dateisystem und mit Datenbanken. Die erweiterbare API (ext-API) von Askemos® dient zur Anbindung von Hilfsprogramme wie Konvertierungsprogramme oder SQLite. Askemos® basiert auf der Scheme-Implementierung RScheme, das auch eine C-Schnittstelle bereitstellt.

Jeder Askemos®-Server hat einen Informationsraum - die virtuelle Maschine -, in dem eine Menge von "Plätzen" vorhanden sind. Vergleichbar ist das mit dem Internet, in dem Knotenpunkte vorhanden sind und Informationen ausgetauscht werden. Die Basiseinheit Askemos® wird "Platz" genannt, ähnlich dem Konzept "Frame" in der Informatik.

Plätze haben die Eigenschaften von Objekten, Prozessen oder Anwendern in anderen Betriebssystemen. Ein statischer Platz kann beispielsweise ein PDF-Dokument oder ein Bild enthalten. Ein dynamischer Platz kann einer Anwendung gleich gesetzt werden und kann Verbindungen zu anderen Plätzen enthalten.

Die Kommunikation mit anderen Plätzen erfolgt über Nachrichten. Dabei reagiert ein Platz auf Nachrichten mit Änderung seiner Eigenschaften in einer atomaren Operation. Plätze sind autonom: Nur auf diese Weise ist es möglich, die Eigenschaften eines Platzes zu ändern. Jeder Platz des Informationsraumes hat eine einzigartige Objekt-Identifikationsnummer, die sogenannte "OID". Die Arbeitsweise mit Askemos®, dynamischen Plätzen und Nachrichten wird im Abschnitt 2.2 erklärt.

Weitere wichtige Eigenschaften sind die Rechte und die Protection. In der Einführung zu Askemos® (Abschnitt 1.3) wurde bereits erwähnt, dass Askemos® auf der Öffentlichkeit als Grundlage beruht. Der Entwickler von Askemos® hat sich dabei von der Rousseau'schen Ethik leiten lassen.

Jedem Platz wird eine "Protection" zugewiesen, um die Menge der erlaubten Zugriffe durch andere Plätze zu verringern. Gewöhnliche Sicherheitsmechanismen besitzen übergeordnete Rechte, die es erlauben alle anderen Rechte außer Kraft zu setzen. Askemos® basiert auf der logischen Umkehrung einer Supermacht, dem öffentlichem Recht, das nicht außer Kraft gesetzt werden darf. Eine Protection wird als Pfad abgebildet, z.B. `/A7c4309ba601ecb05f93b50fe14d89b41/A0570f5adda03620a82c9ae870ead3726`. Beide OIDs `A7c4309ba601ecb05f93b50fe14d89b41` und `A0570f5adda03620a82c9ae870ead3726` des Beispiels sind jeweils ein Startplatz eines Benutzers und stehen als Be-

zeichner für einen Benutzer. Das Beispiel zeigt, dass sich eine Protection auf einen Platz `A7c4309ba601ecb05f93b50fe14d89b41` bezieht und einem weiteren Platz erlaubt, auf den Platz `A0570f5adda03620a82c9ae870ead3726` zuzugreifen.

Der Protection-Schutzmechanismus reicht nicht aus, um Zugriffe von Aktionen und den Nachrichtenverkehr zu beschreiben. Aufgrund dieser Tatsache existiert eine Erweiterung des Protection-Schutzmechanismus: die sogenannten "Capabilities". Die Capabilities sind besondere Rechte, die es einem Objekt, das sie besitzt, erlauben, andere Objekte zu verwenden. Die Verwendung beinhaltet das Senden von Nachrichten, um den Platz zu ändern oder um Informationen des Platzes abzurufen. Capabilities können erteilt (engl. grant) und entzogen (engl. revoke) werden.

Abbildung 2.1 weist eine Verbindung der Askemos®-Server auf. Das besondere Protokoll ist vergleichbar mit einer antiken "byzantinischen Abstimmung": Sobald eine Aktion, z.B. Absenden eines Formulares, ausgeführt wurde, wird die Informationsverarbeitung der Plätze auf jedem beteiligten Server durchgeführt. Das entstandene Ergebnis wird geprüft und bei einer Zweidrittel-Mehrheit anerkannt. Beteiligte Server werden in einer Liste (sog. "Quorum") des Platzes eingetragen. Tritt während der Verarbeitung ein Fehler auf, wird der Benutzer darüber informiert.

2.2 Funktionsweise dynamischer Plätze

Dynamische Plätze beinhalten ein XSL-Stylesheet, das mithilfe des XML-Prozessors von Askemos® und XML-Daten in einen anderen Zustand transformiert werden kann. Nach einer Transformation kann ein Platz ein anderes XSL-Stylesheet enthalten. Die Grundlage der XSL-Stylesheets bildet die vom W3C vorgeschlagene Struktur und Dokumentation [10].

Das vorliegende XSL-Stylesheet besteht aus einer Menge von Templates und Variablen. Variablen beinhalten Informationen zur globalen Benutzung im entsprechenden Bereich, beispielsweise innerhalb von Templates oder innerhalb des globalen XML-Knotens `<xsl:stylesheet [...]>`. Der Vorteil der Variablen besteht in der Zwischenspeicherung des berechneten Ergebnisses zur Mehrfachbenutzung. Dadurch muss das Ergebnis nicht bei jeder Benutzung neu berechnet werden.

Templates sind Programmgerüste, die unter bestimmten Bedingungen ausgeführt werden. Zwei Templates geben eine allgemeine Struktur vor. Das erste Template führt alle ankommenden Anfragen lesend aus. Das betrifft auch Formulare, die mit der Formular-Methode `GET` abgeschickt werden. Formulare, die eine Änderung des Platzes notwendig machen, werden mit der Formular-Methode `POST` gesendet. Das dafür zuständige zweite Template führt damit schreibende Anfragen (engl. request) aus.

Beide Templates bilden das allgemeine Programmgerüst eines XSL-Stylesheetes für dynamische Plätze. Im folgenden Listing 2.1 wurden alle Namensräume außer `xsl` und `d` zur besseren Übersicht entfernt:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:d="http://www.askemos.org/2005/NameSpaceDSSSL/"
     <!-- [...] -->
     version="1.0">
  <xsl:template match="*[@type=&quot;read&quot;]">
    <!-- [...] -->
  </xsl:template>
  <xsl:template match="request[@type=&quot;write&quot;]">
    <!-- [...] -->
  </xsl:template>
</xsl:stylesheet>
```

Listing 2.1: Allgemeiner Aufbau eines Platzes

Innerhalb von Templates wird die Verarbeitung durch weitere Bedingungen gesteuert. Tag-Namen der notwendigen XML-Knoten beinhalten die Schlüsselwörter `when`, `if` oder `otherwise`. Die auszuwertende Bedingung wird dabei in das Attribut `test` geschrieben und wird in Scheme formuliert, beispielsweise `(> a b)`. Der Körper der genannten XML-Knoten ist dabei die Ausgabe, in die transformiert werden soll.

Das folgende Listing zeigt eine klassiche Anwendung von “if” und “else”. Jedoch finden sich in XSL andere Begriffe: `when` hat die Bedeutung von “if” und `otherwise` die von “else”. XML-Knoten mit Schlüsselwörtern `when` und `otherwise` müssen innerhalb eines XML-Knotens mit dem Schlüsselwort `choose` stehen.

Im Vergleich dazu kommt `if` ohne `choose` aus und bildet damit eine einfache “if-then” Bedingung. Folglich bilden `choose` und dessen Kind-Knoten eine Auswahl aus mehreren Bedingungen. Alle genannten Varianten soll das folgende Listing 2.2 verdeutlichen:

```
<xsl:template match="*[@type=&quot;read&quot;]">
  <d:if test="[...]">
    <!-- [...] -->
  </d:if>
  <xsl:choose>
    <d:when test="[...]">
     <!-- [...] -->
    </d:when>
   <xsl:otherwise>
    <html>
     <!-- [...] -->
    </html>
   </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Listing 2.2: Steuerung der Verarbeitung in Templates

Im letzten Listing 2.2 ist neben dem Namensraum `xsl` auch der Namensraum `d` angegeben. Der Namensraum `d` ermöglicht die Auswertung von Scheme-Quelltext, beispielsweise innerhalb des Attributes `test=""` oder innerhalb eines XML-Knotens `<d:copy-of [...]>`.

Im folgenden Listing 2.3 werden einige Transformationsregeln vorgestellt, die einen Platz verändern. Die Transformationsregeln befinden sich in einem `<core:reply>` XML-Knoten.

```
<xsl:template match="request[@type=&quot;write&quot;]">
  <!-- [...] -->
  <d:when test="[...]">
    <core:reply>

      <core:continue>
        <d:copy-of select="(grove-root (current-node))"/>
      </core:continue>

      <core:link name="public">
        <id>A00000000000000000000000000000001</id>
      </core:link>

      <core:link name="kopie">
        <d:copy-of select="#CONTENT">
         (make element gi: 'new ns: [...]
               attributes:
                 `((action "A00000000000000000000000000000004")
```

```
                    (protection ,(right->string (me 'protection))))
                (grove-root (current-node)))
        </d:copy-of>
      </core:link>

      <core:send xmlns:env="[...]" type="write">
        <to><d:copy-of select="(oid->string (me 'get 'id))"/></to>
        <env:body>
          <form>
            <feld1>inhalt1</feld1>
            <feld2>inhalt2</feld2>
          </form>
        </env:body>
      </core:send>

    </core:reply>
  </d:when>
  <!-- [...] -->
</xsl:template>
```

Listing 2.3: core:update – Schreibende Aufrufe an einen Platz

Die Transformationsregel des XML-Knotens `<core:continue>` gibt an, welchen Inhalt der Platz nach der Schreib-Transformation haben soll. Innerhalb des letzteren XML-Knotens gibt der Scheme-Befehl `(grove-root (current-node))` den kompletten Inhalt des Platzes zurück. Folglich wird der Platz mit bestehendem XSL-Stylesheet geschrieben.

Die Transformationsregel des XML-Knotens `<core:link>` gibt an, dass eine Verknüpfung zu einem bestehenden Platz hergestellt werden soll. Dazu muss ein Name der Verknüpfung im Attribut `name` angegeben und die OID des Platzes in dem XML-Knoten `id` notiert werden.

Die Transformationsregel des XML-Knotens `<core:link>` und `<core:new>` gibt an, dass eine Verknüfpfung zu einem neu zu erzeugenden Platz hergestellt werden soll. Der Inhalt des neuen Platzes wird dabei im Körper des XML-Knotens `<core:new>` notiert. Im Listing 2.3 wird der `<core:new>` XML-Knoten mittels Scheme-Quelltext erzeugt und der Inhalt des Platzes als Inhalt des neuen Platzes zurückgegeben.

Die Transformationsregel des XML-Knotens `<core:send>` gibt an, dass eine Nachricht an den Platz mit der OID aus dem XML-Knoten `<to>` geschickt werden soll.

Das Attribut `type` gibt die Art der Anfrage an, in diesem Fall ist es eine schreibende Anfrage. Im XML-Knoten `<env:body>` wird ein XML-Formular angegeben, welches als Körper der Nachricht an den Ziel-Platz geschickt wird.

Mit einer speziellen Funktionalität im XSL-Stylesheet ist es auch möglich, an einem Platz eine WebDAV-Funktionalität anzubieten. Dabei wird das WebDAV über die URL des Askemos®-Servers und die OID des Platzes angesprochen (ein öffentliches WebDAV ist z.B.: `http://asknet1.tzv.de/Ad796d626b79a727f828946e2d78a4d68`). Startplätze sind mit dem jeweiligen Benutzername verbunden. Damit lässt sich die OID durch den Benutzername austauschen: `http://asknet1.tzv.de/tsw-dav`. WebDAV ist in Askemos® eine Sammlung von Plätzen, wobei diese miteinander verlinkt sind. So sind Verzeichnisse in Askemos® Plätze mit Links auf die Verzeichniseinträge.

2.3 Grundlagen der Programmiersprache Scheme

Scheme ist ein LISP-Dialekt. Der Sprachumfang der Programmiersprache Scheme ist in Askemos® begrenzt und ermöglicht die Verarbeitung von Transaktionen. Eine kurze Einführung in Scheme soll die wichtigsten Merkmale darstellen. Das Script aus [13] wird dabei zusammengefasst.

Die Grundlage der Syntax von Scheme ist eine vollständig geklammerte Präfix-Notation. Jeder Ausdruck und jede Definition, eine Zusammensetzung von Ausdrücken, hat die folgende Form: `(operator operant-1 operant-2 ... operant-n)`.

Ausdrücke können global oder lokal als Funktion oder Variable definiert werden. Folgendes Listing 2.4 zeigt 3 Varianten:

```
(define (plus a b) (+ a b))
(define pi 3.14159265)
(let ((alpha 0.5)
      (mult (lambda (n) (* pi n))))
     (mult alpha))
(let* ((a (data (form-field 'a (current-node))))
       (b 10)
       (a (if (equal? a b) b a)))
      (> a 100))
```

Listing 2.4: Scheme: define let let*

Die Variante `define` bindet einen Ausdruck global an einen Wert oder Ausdruck. Das erste Beispiel zeigt die Bindung einer Funktion `plus` mit Parametern `a` und `b` an die Additionsfunktion. Zweites Beispiel hingegen zeigt die Bindung des Symboles `pi` an den Wert `3.14159265`.

Die Variante `let` erzeugt eine lokale Umgebung und bindet lokal Ausdrücke, die in einer Liste übergeben werden. Im ersten Listenelement wird `alpha` an den Wert `0.5` gebunden. Im zweiten Listenelement wird ein Symbol `mult` an eine anonyme Funktion `lambda` mit einem Parameter `n` gebunden. Innerhalb der `lambda` Funktion werden die Inhalte der Symbole `pi` und `n` miteinander multipliziert. Der zweite Parameter der `let`-Umgebung beinhaltet den Quelltext, der ausgeführt werden soll.

Die Variante `let*` verhält sich analog zur Variante `let`, allerdings können Ausdrücke in der lokalen Variablendefinition überschrieben werden. Im Listing 2.4 wird das Symbol `a` überschrieben, nachdem es zuvor aus einem Formular-Feld geholt wurde.

Schließlich stellen folgende Abschnitte und das Listing 2.5 Schleifen und Listen dar:

```
; Rekursion
(define (fak n)
        (if (= n 0) 1
            (* n (fak (- n 1)))))

; Schleife mit named let
(let loop ((zahlen (list 1 2 3 4))
           (summe 0))
     (if (pair? zahlen)
         (loop (cdr zahlen)
               (+ summe (car zahlen)))
         summe))

; Listenverarbeitung
(fold + 0 '(1 2 3 4 5))
```

Listing 2.5: Scheme: Schleifen und Listen

Das erste Beispiel stellt eine Rekursion dar, welche die Fakultät von `n` berechnen soll. Solange `n` nicht 0 ist, wird die Funktion `fak` aufgerufen.

Das zweite Beispiel zeigt eine benannte `let`-Umgebung. Eine derartige `let`-Umgebung ermöglicht es sich selbst aufzurufen. Das Beispiel beinhaltet auch die Verarbeitung von Listen. Eine Liste wird beispielsweise erstellt mit `'()` oder `(list ...)`. Listen sind

in Scheme zusammenhängende Paare, bei denen der erste Teil "`car`" den Wert enthält und der zweite Teil "`cdr`" die Referenz auf das nächste Paar.

Das dritte Beispiel führt dieselbe Verarbeitung durch wie im zweiten Beispiel. Die Listen-verarbeitende Funktion `fold` verlangt dabei folgende Parameter: eine verarbeitende Funktion, einen Start-Wert und die zu verarbeitende Liste.

Abschließend soll darauf hingewiesen werden, dass es "Scheme Requests for Implementation" (SRFI) gibt. Die Internetseite `http://srfi.schemers.org/final-srfis.html` beinhaltet eine Auflistung von SRFI-Beschreibungen. RScheme enthält SRFI-Implementierungen, die als Module eingebunden sind. Allerdings muss nicht jede Funktion umgesetzt worden sein. Spezielle Funktionen werden im Laufe der Diplomarbeit erklärt.

2.4 Merkmale der SQLite-Programmbibliothek

Die Vorteile von SQLite werden in folgenden Abschnitten (nach [6] und [8]) dargestellt; weiterführende Informationen dazu finden sich in der SQLite Dokumentation [5].

SQLite ist eine eingebettete SQL-Datenbankanwendung und verwendet die Datenbanksprache SQL, wobei der größte Teil des SQL-92-Standards implementiert ist und bei der alle Funktionen vollständig getestet sind (siehe [7]).

Im Gegensatz zu anderen Datenbank-Management-Systemen verwendet SQLite **keinen eigenen Serverprozess** und bedarf **keiner Vorkonfiguration nach der Installation**. SQLite verwaltet eine **komplette Datenbank in einer Datei** und greift während der Verarbeitung nur auf diese Datei zurück. Des Weiteren werden Terrabyte-große Datenbankdateien und Gigabyte-große Zeichenketten und BLOBs unterstützt.

Der Inhalt einer solchen Datenbank-Datei ist unabhängig vom Datei- und Betriebssystem. Nachteilig ist die fehlende Rechteverwaltung innerhalb der Datenbank-Datei. Folglich muss die Rechteverwaltung auf dem Hostsystem oder dem Dateisystem stattfinden. Das hat natürlich eine besondere Bedeutung für Askemos®. Nur der Benutzer, dem die Datenbank gehört oder der Verarbeitungsrechte hat, darf die Datenbank verarbeiten.

SQLite ist eine Programmbibliothek, die auf **Leistung und niedrigen Ressour-**

cenverbrauch optimiert ist. Ohne zusätzliche Funktionen kann die Größe der Programmbibliothek auf unter 180KB reduziert werden und eignet sich somit auch für kleine Geräte wie PDAs oder MP3 Player.

Viele Datenbanksysteme verwenden eine statische Typisierung: Jede Spalte einer Tabelle hat einen bestimmten Datentyp; Daten können nur in diesem Datentyp gespeichert werden. SQLite verwendet eine **manifestierte Typisierung**. Im Gegensatz zur statischen Typisierung ist der Datentyp bei einer manifestierten Typisierung eine Eigenschaft eines Zellenwertes und nicht einer Spalte. Damit können sogar Zeichenketten in eine Spalte mit Datentyp Integer eingefügt werden.

Des Weiteren unterstützt SQLite Einträge variabler Länge. In beiden Fällen wird nur der Speicherbedarf verwendet, der für den Wert einer Zelle notwendig ist. Folglich wird nicht der Speicherbedarf verwendet, der von der Spalte durch den Datentyp vorgegeben wird, wie es bei der statischen Typisierung der Fall ist. Eine Ausnahme dabei ist allerdings, wenn eine Spalte mit Datentyp “INTEGER PRIMARY KEY” definiert wird. In derartige Spalten können nur Primärschlüssel mit Datentyp Integer eingefügt werden.

Ein weiterer wichtiger Vorteil ist, dass innerhalb von SQLite eine **Virtuelle Maschine** existiert. Jede SQL-Anweisung wird in einen Virtuellen-Maschinen-Code umgewandelt. Die Virtuelle Maschine führt den Code aus und verwendet angebundene Schnittstellen zu Dateisystemen, um darauf die Datenbank-Datei zu schreiben.

Letztere Schnittstellen - virtuelle Dateisysteme (VFS) - für die Virtuelle Maschine können sogar während des Betriebes ausgewechselt werden. Die SQLite-Dokumentation enthält eine Beschreibung der Schnittstelle [11]; Funktionen - besonders Lese- und Schreibfunktion - sind dafür nötig. Vorlagen zur Implementierung des Askemos®-VFS sind vorhandene Implementierungen und die C-Quelltext-Datei `test_osinst.c`, die Testfälle auf vorhandenen virtuellen Dateisystemen ausführt.

3 Problem-Analyse

3.1 Analyse vorkommender Anwendungsfälle in Shop-Systemen

Die funktionalen Anforderungen an den Kern von Shop-Systemen werden in den nächsten Abschnitten erfasst. Anwendungsfall-Diagramme geben eine Übersicht, welche allgemeinen Anwendungsfälle existieren und welche Funktionen es in einem speziellen Anwendungsfall gibt. Aufbauend auf den Anwendungsfällen wurden ein Datenbank-Schema und Grundlagen für die Templates entwickelt. Dabei werden die Templates eines XSL-Stylesheets als Funktionen betrachtet.

In jedem Anwendungsfall-Diagramm ist ersichtlich, welche Benutzerrolle welche Aktionen oder Anwendungsfälle verwenden darf. Die Voraussetzung zur Benutzung eines Shop-Systems ist das Vorhandensein eines Zugangs zum Askemos®-Betriebssystem.

Der Kundendienst spielt eine besondere Rolle. Er besteht aus einer Menge von Mitarbeitern, die das Askemos®-Betriebssystem betreuen. Er überwacht Angebote und stellt den ordnungsgemäßen Betrieb sicher und ist daher befugt, alle Anwendungsfälle durchzuführen. Bei Ausführungsfehlern und schwerwiegenden Systemfehlern, ist es die Aufgabe des Kundendiensts, sie zu beheben.

Folgende Abschnitte stellen Produktfunktionen dar, die im Pflichtenheft des Projektes MyHomeNotar genannt wurden. Im Rahmen der Diplomarbeit wurden die Produktfunktionen ausführlich analysiert.

3.1.1 Allgemeine Anwendungsfälle

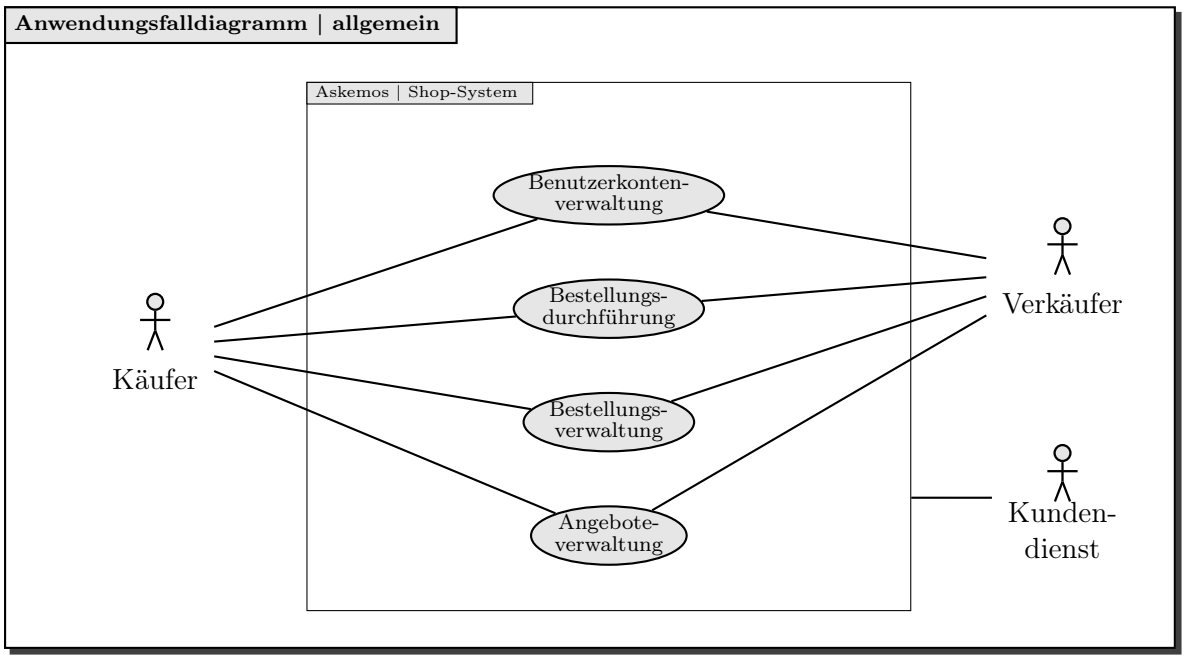

Abbildung 3.1: Anwendungsfalldiagramm - allgemeine Anwendungsfälle

Abbildung 3.1 zeigt eine Übersicht der Module, die zu entwickeln sind. Ein Modul stellt dabei einen Anwendungsfall dar:

- Das Modul **Benutzerkontenverwaltung** beinhaltet Funktionen, die Benutzerkonten verwalten oder sperren.
- Das Modul **Bestellungsdurchführung** enthält Funktionen, die eine Bestellung korrekt durchführbar machen.
- Das Modul **Bestellungsverwaltung** stellt Funktionen bereit, die abgeschlossene Bestellungen auflisten und verwalten lässt.
- Das Modul **Angeboteverwaltung** beinhaltet Funktionen, welche Angebote auflistbar und verwaltbar machen.

Jede Benutzerrolle hat Zugriff auf alle Module, jedoch nicht alle Funktionen der Module. Die Benutzerrollen Käufer, Verkäufer und Kundendienst sind im Anwendungsfall-Diagramm als Akteure gekennzeichnet. Dabei sind die Benutzer reale Personen, die den Internetdienst Shop-System verwenden. Im Nachfolgenden werden Anwendungsfall-Diagramme dargestellt, die auf die einzelnen Module eingehen.

Ein Käufer sucht sich Angebote aus und stellt eine Bestellung aus den Angeboten zusammen. Die Bestellung wird durch das Shop-System ausgeführt und dem Käufer eine Rechnung übermittelt. Alle getätigten Bestellungen werden in seinem Bereich aufgelistet. Zu jeder Bestellung kann er alle erforderlichen Informationen abrufen, exemplarisch den Lieferstatus.

Verkäufern stehen Funktionen zur Verfügung, die es ihnen erlauben, Angebote einzustellen, genau zu beschreiben und sie zu löschen, wenn sie nicht mehr erwünscht oder verfügbar sind. Bestellungen durch Käufer werden ausführlich dargestellt, damit der Verkäufer die Lieferung veranlassen kann. Wenn ein Käufer von seinem Widerrufsrecht gebraucht macht, ist es möglich, die Bestellung zu stornieren.

3.1.2 Spezielle Anwendungsfälle der Benutzerkontenverwaltung

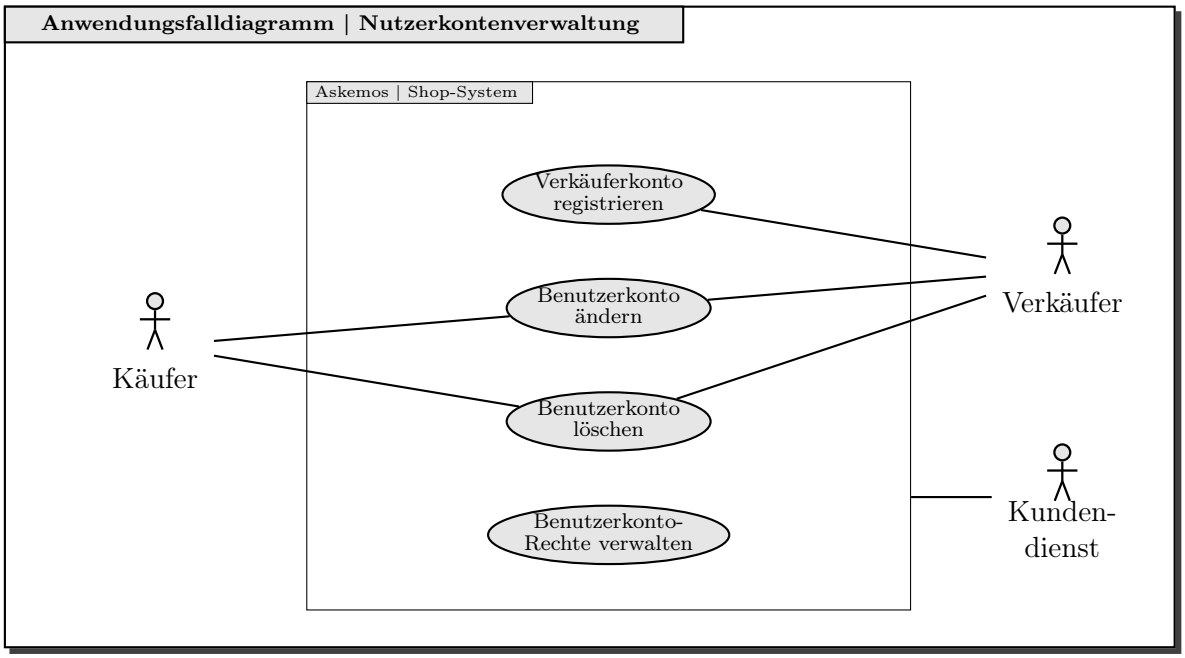

Abbildung 3.2: Anwendungsfall-Diagramm - Benutzerkontenverwaltung

Benutzerkonten-Daten werden in der SQLite-Datenbank gespeichert. Käuferkonten werden erst durch den Kern von Shop-Systemen registriert, wenn es notwendig wird, beispielsweise zum Abschließen einer Bestellung. Käufer werden durch Erteilung eines Rechtes automatisch zur Benutzung freigeschaltet. Folgende Verwaltungsfunktionen werden implementiert:

/F0211/ Ein neuer Verkäufer oder ein Kundendienst-Mitarbeiter kann sich **ein neues Verkäuferkonto registrieren**.

Ein Verkäufer muss ein Formular ausfüllen, in dem er über die zum Verkauf notwendigen Bedingungen belehrt wird und dass ihm weitere Kosten entstehen. Die einzugebenden Daten sind dynamisch einstellbar.

/F0212/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **sein eigenes Benutzerkonto ändern**, wobei Kundendienst-Mitarbeiter auch fremde Benutzerkonten ändern kann.

Das Benutzerkonto eines Benutzers umfasst einerseits die Rechnungsadresse, deren Daten dynamisch einstellbar sind, exemplarisch: Name, Titel, Adresse, Firma, E-Mail-Adresse und Telefonnummer. Andererseits ist es möglich weitere Lieferadressen anzugeben, die während der Bestellung gewählt werden können.

Die Bezahlung von Rechnungen und Gebühren erfolgt über ein weiteres Modul, welches im Rahmen der Diplomarbeit nicht zu implementieren ist.

/F0213/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **sein eigenes Benutzerkonto löschen**, wobei Kundendienst-Mitarbeiter auch fremde Benutzerkonten löschen kann.

Jeder Benutzer kann sein Benutzerkonto zu jeder Zeit löschen, dabei werden alle Daten und Bestellungen des Benutzers gelöscht. Persönliche Informationen wie Visitenkarte oder Lieferadressen werden ebenfalls gelöscht. Die Löschung muss allerdings durch den Kundenservice bestätigt werden.

Die Informationen werden aus der SQLite-Datenbank entfernt und der Benutzer könnte sich somit erneut registrieren. Die Funktion “Benutzerkonto löschen” erfordert dabei bei näherer Betrachtung ein Datenfeld, welches anzeigt, für welche Person alle bisherigen Bestellungen noch ersichtlich sein müssen.

Wenn ein Käufer sein Benutzerkonto löscht, müssen also alle bisherigen Rechnungen noch für den entsprechenden Verkäufer ersichtlich sein. Wenn der Verkäufer eine Lösch-Funktion aufruft, die das Löschen ein oder mehrerer Rechnungen beinhaltet und der Käufer von sich aus auch eine Lösch-Funktion aufgerufen hat, dann

erst darf der komplette Datensatz gelöscht werden. Folglich werden keine Informationen gelöscht, die für mindestens eine Person noch relevant sein könnten.

Ein weiterer Grund, Daten nicht zu löschen, könnten rechtliche Grundsätze sein. Falls beispielsweise ein Käufer eine Rechnung nicht bezahlt hat, kann dies für ihn rechtliche Konsequenzen haben, da er durch Abschluss der Bestellung einen Vertrag eingegangen ist.

/F0214/ Kundendienst-Mitarbeiter können **andere Benutzerkonten sperren oder entsperren**.

Damit ein Benutzer ein Shop-System verwenden kann, darf sein Benutzerkonto nicht gesperrt sein. Das Sperren des Benutzerkontos kann viele Ursachen haben, beispielsweise wenn eine Wartung durchgeführt wird oder ein Benutzerkonto sich nicht an rechtliche Bedingungen hält.

3.1.3 Spezielle Anwendungsfälle der Bestellungsdurchführung

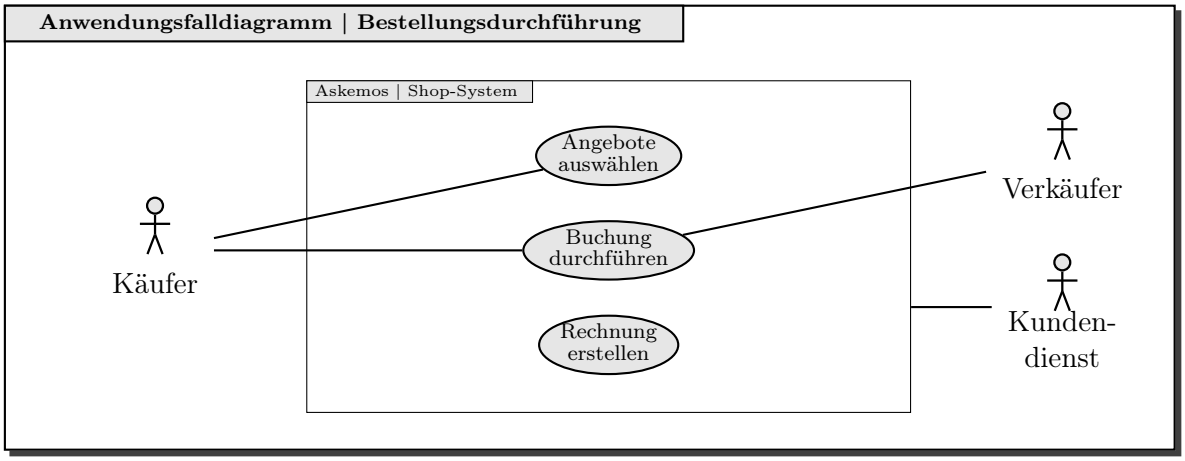

Abbildung 3.3: Anwendungsfall-Diagramm - Bestellungsdurchführung

Bestellungen werden in der SQLite-Datenbank gespeichert, Ausgaben werden im PDF-Format generiert, wenn es erforderlich ist. Folgende Durchführungsfunktionen werden implementiert:

/F0221/ Der angemeldete Käufer, Verkäufer oder ein Kundendienst-Mitarbeiter kann **Angebote auswählen und zu einer Bestellung hinzufügen**.

Es ist eine Warenkorb-Funktionalität zu implementieren, die das Hinzufügen von Artikeln zu einer Bestellung erlaubt. Ebenso sind Funktionen zum Bearbeiten (z.B. Menge) und Entfernen der Positionen einer Bestellung einzurichten. Es ist möglich eine offene Bestellung als Warenkorb zu verwenden, solange die Bestellung nicht abgeschlossen ist.

Wenn im Warenkorb keine Artikel vorhanden sind, ist die "offene" Bestellung zu entfernen. Wenn Artikel für die Bestellung nicht mehr verfügbar sind, ist das sofort auszugeben. Wenn der Käufer die Bestellung erst nach einer Woche fortsetzt, kann es sein, dass ein Angebot nicht mehr verfügbar ist.

Bei näherer Betrachtung ist zu erkennen, dass eine offene Bestellung bestehen bleibt, wenn noch Artikel vorhanden sind und der Käufer die Bestellung nicht abgeschlossen hat. Die Datensätze der offenen Bestellung stellen damit eine Aufblähung der Datenbank dar. Das bedeutet im Weiteren, dass dem Käufer eine Frist gewährt wird, in der die offene Bestellung gültig bleibt. Wenn also beispielsweise der Verkäufer die Bestellung nicht innerhalb von 3 Tagen abschließt, werden die Datensätze der offenen Bestellung entfernt.

/F0222/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **die Buchung einer Bestellung durchführen**, dabei schließt der Käufer die Bestellung ab, so dass der Verkäufer die Bestellung durchführen kann.

Alle Bedingungen, die zur Buchung einer Bestellung führen, werden in einer letzten Übersicht angezeigt, exemplarisch die Allgemeinen Geschäftsbedingungen, die jeder gewerbliche Verkäufer angeben muss.

Mit der Buchung der Bestellung geht der Käufer einen Kaufvertrag ein. Treten dabei Probleme auf, muss der Käufer den Verkäufer kontaktieren. Handelt der Verkäufer nicht gemäß seiner rechtlichen Vorgaben, wird er entsprechend verwarnt, von der Rolle des Verkäufers entbunden oder sein Benutzerkonto gesperrt.

Die Buchung einer Bestellung führt dazu, dass die komplette Bestellung als abgeschlossen in die Datenbank eingetragen wird. Käufer wie auch Verkäufer können sich danach zu jeder Zeit die Rechnung anzeigen lassen.

/F0223/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann die

Rechnung einer Bestellung im PDF-Format jederzeit aufrufen, die durch das Shop-System **erstellt** wird.

Rechnungen werden grundsätzlich aus den vorhanden Informationen der Datenbank generiert. Die Speicherung in der Datenbank ist vorteilhaft für eine leichte Änderung von Informationen der Rechnung. Folglich beruht die Speicherung der Informationen vorrangig auf der Datenbank.

3.1.4 Spezielle Anwendungsfälle der Bestellungsverwaltung

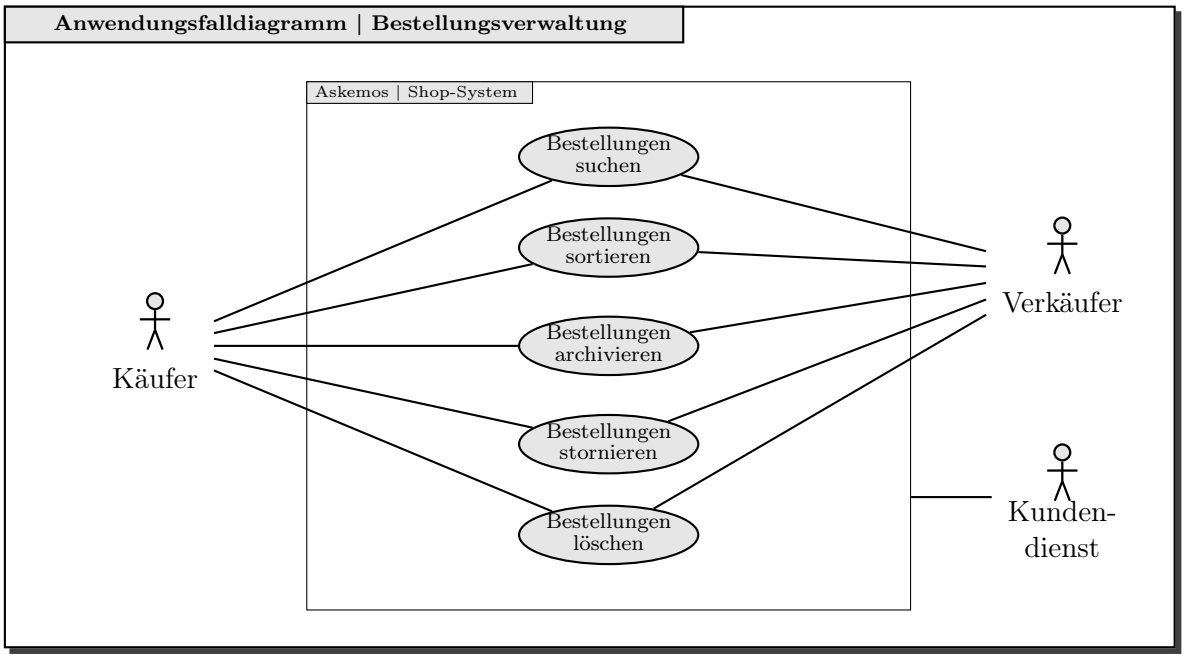

Abbildung 3.4: Anwendungsfall-Diagramm - Bestellungsverwaltung

Bestellungen werden in der SQLite-Datenbank gespeichert. Folgende Verwaltungsfunktionen werden implementiert:

/F0231/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **in der Liste der Bestellungen suchen**.

Die Liste der Bestellungen wird aus den Informationen der Datenbank zusammengestellt. Bei der Abfrage der Liste der Bestellungen ist es möglich, nach dem eingegebenen Text suchen zu lassen. Wenn keine Bestellungen gefunden wurden, ist eine entsprechende Meldung auszugeben.

/F0232/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **die Liste der Bestellungen sortieren**.

Die Liste der Bestellungen wird aus den Informationen der Datenbank zusammengestellt. Dabei lässt sich beliebig nach jeder Spalte sortieren, auch mehrstufig, nach mehreren Spalten. Wenn die Liste der Bestellungen keine Elemente enthält, dann ist eine entsprechende Meldung auszugeben.

/F0233/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **eine Anzahl von Bestellungen archivieren**.

Im Rahmen der Implementierungsphase soll entschieden werden, welche Variante besser realisiert werden kann: Alle ausgewählten Bestellungen werden in getrennte PDF-Dateien generiert und zu einer komprimierten Datei zusammengefasst oder per E-Mail versendet. Folglich steht dem Benutzer eine Menge von PDF-Dateien zur Verfügung, die er auf einem seiner Datenträger speichern kann.

/F0234/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **eine Bestellung stornieren**. Ein Käufer widerruft dabei die Bestellung und der Verkäufer storniert sie.

Im Rahmen der gesetzlichen Richtlinien ist dem Käufer ein Widerrufsrecht einzuräumen, solange es sich bei dem Verkäufer um einen geschäftlichen Verkäufer handelt. Eine Widerrufs- oder Rücknahmebelehrung muss vor und nach der Buchung der Bestellung durchgeführt werden.

Ist keine Belehrung angegeben, dann hat der Käufer ein fristloses Widerrufsrecht. Innerhalb einer Frist hat der Käufer das Recht, die Bestellung unter bestimmten Bedingungen zu widerrufen. Sobald der Verkäufer bestellte Angebote storniert hat, wird dies dem Käufer ersichtlich, z.B. wenn er die Rechnung neu generieren lässt.

/F0235/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **eine Anzahl von Bestellungen löschen**.

In der Produktfunktion /F0213/ wurde bereits beschrieben, unter welcher Bedingung Bestellungen gelöscht werden können. Die Informationen der Bestellungen werden beim Löschen aus der Datenbank entfernt.

3.1.5 Spezielle Anwendungsfälle der Angebotsverwaltung

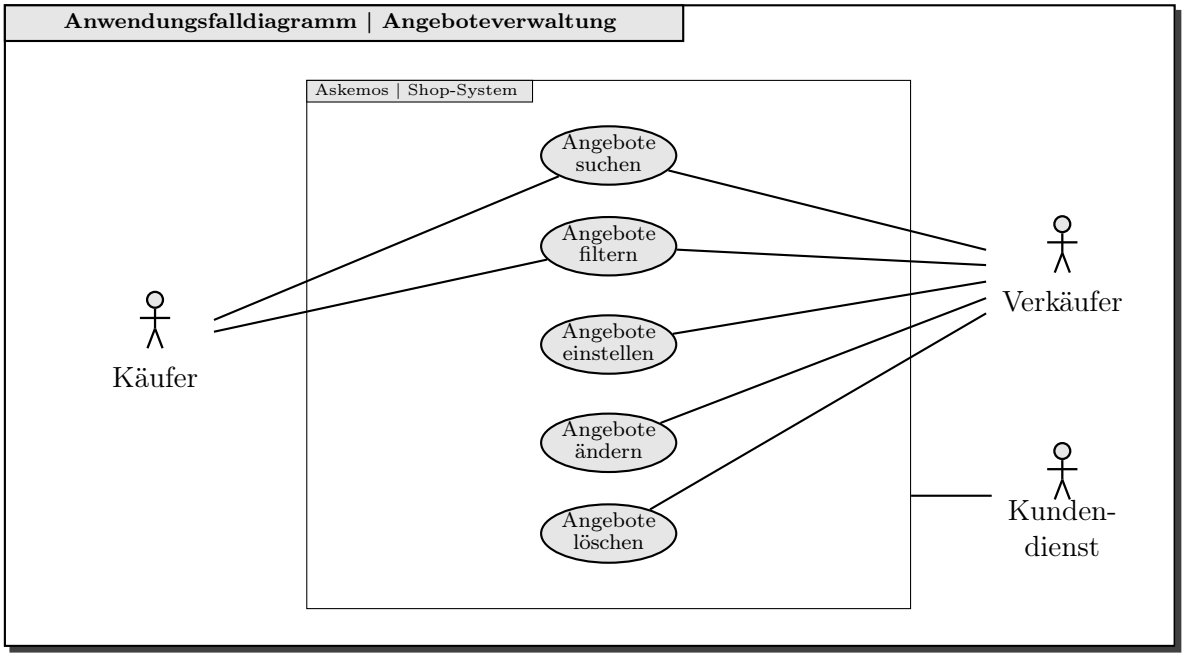

Abbildung 3.5: Anwendungsfall-Diagramm - Angebotsverwaltung

Angebote werden in der SQLite-Datenbank gespeichert. Folgende Verwaltungsfunktionen werden implementiert:

/F0241/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **in der Liste der Angebote suchen**.

Suchen von Angeboten ist analog der Produktfunktion /F0231/.

/F0242/ Der angemeldete Käufer, Verkäufer oder Kundendienst-Mitarbeiter kann **die Liste der Angebote filtern**.

Das Filtern beinhaltet eine Sortierung und Verfeinerung (Ausschluss von Angeboten) von Datensätzen. Der Ablauf des Sortierens von Angeboten ist analog der Produktfunktion /F0231/. Die Verfeinerung macht es damit notwendig, dass zu Angeboten eine Menge von Kriterien mit angegeben werden können, anhand welcher die Verfeinerung stattfinden kann.

Allerdings ist es auch möglich die Such-Funktion so zu erweitern, dass Kriterien ausgewählt werden können, die die Ergebnismenge zusätzlich filtern. Wenn keine Angebote gefunden wurden, dann ist eine entsprechende Meldung auszugeben.

/F0243/ Der angemeldete Verkäufer oder Kundendienst-Mitarbeiter kann **neue Angebote einstellen**.

Das Einstellen von Angeboten verursacht Kosten für Verkäufer. Für Angebote soll es eine einstellbare Menge an Eingabefeldern geben. Alle Eingabefelder müssen vor der Benutzung eines Shop-Systems festgelegt werden.

Weiterhin soll für Angebote auch die Möglichkeit, bestehen Abbildungen hinzuzufügen.

/F0244/ Der angemeldete Verkäufer oder Kundendienst-Mitarbeiter kann **eigene Angebote ändern**. Der Kundendienst-Mitarbeiter kann dabei auch fremde Angebote ändern.

Angebote können bearbeitet werden, solange kein Käufer das Angebot bestellt. Wenn ein Käufer ein Angebot bestellt hat und die Bestellung nicht komplett abgeschlossen ist, dann muss der Käufer gefragt werden, ob er mit der Änderung einverstanden ist. Die Bearbeitung erfolgt analog zu vorhergehender Produktfunktion /F0243/, mit der Ausnahme, dass keine Kosten entstehen.

/F0245/ Der angemeldete Verkäufer oder Kundendienst-Mitarbeiter kann **eigene Angebote löschen**.

Analog zur vorhergehender Produktfunktion /F0244/ können Angebote nur gelöscht werden, wenn sie in keiner Bestellung auftauchen. Jedoch sollte es möglich sein, Angebote für neue Bestellungen nicht sichtbar zu machen.

3.1.6 Datenbank-Schema – Ergebnis der Problem-Analyse

Aus der Analyse wird das folgende Datenbank-Schema (Abb. 3.6) abgeleitet:

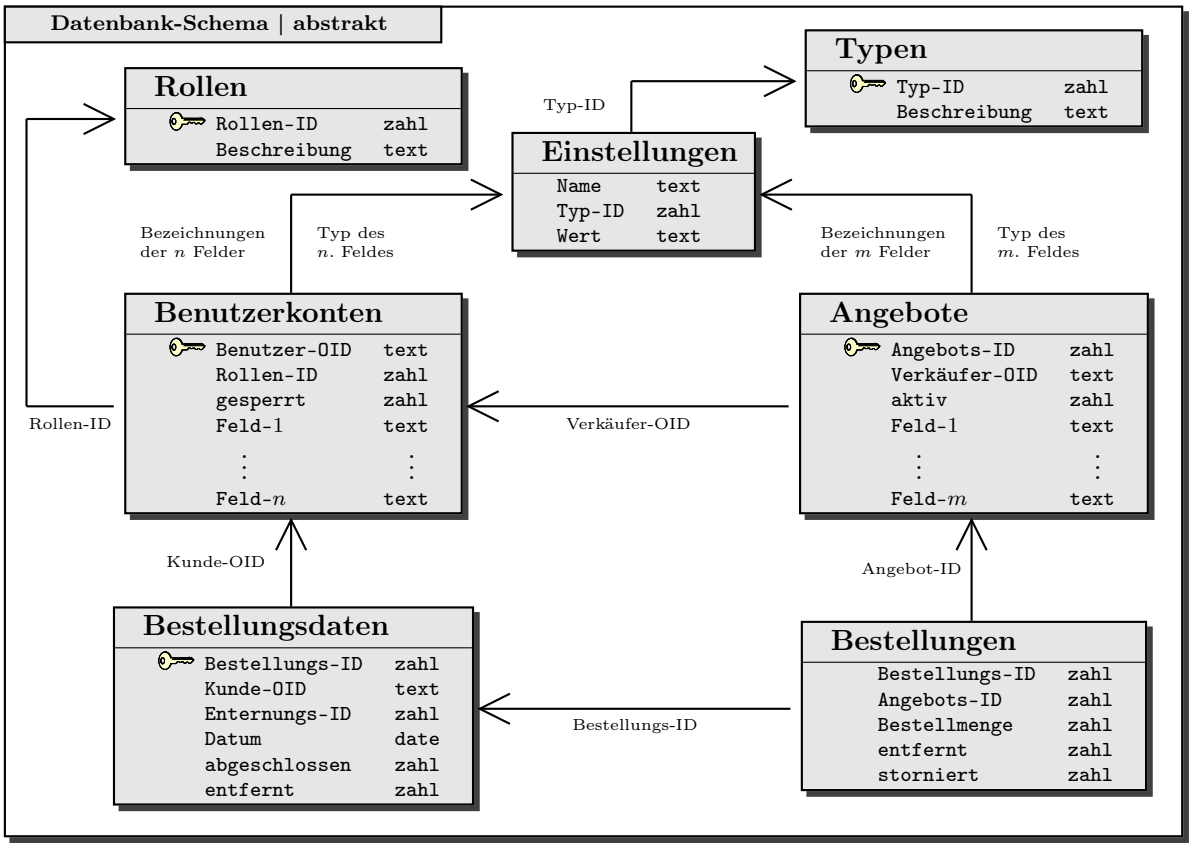

Abbildung 3.6: Datenbank-Schema - Kern von Shop-Systemen

Vor der ersten Benutzung der Datenbank sind folgende Bedingungen zu erfüllen:

- Es sind alle Anforderungen für die Module einzustellen, die eine genaue Struktur benötigen. Die einstellbaren Module sind: Benutzerverwaltung, Angebotsverwaltung, Bestellungsverwaltung.
- Der Kundenservice legt dabei weitere Plätze fest, die an der Verarbeitung beteiligt sind, beispielsweise den SMS-Platz und den E-Mail-Platz.
- Schließlich ist die Datenbank-Struktur zu generieren, damit das Shop-System benutzbar wird. Die Generierung soll durch die Betätigung eines Formular-Knopfes auslösbar sein.

Die Verarbeitung der Bedingungen macht es notwendig, ein Modul zu entwickeln, welches die Einstellungen verwaltet und die Datenbank-Struktur generiert.

3.2 Analyse der vorhandenen Datenbankschnittstelle

Askemos® bietet eine Grundlage zur Verarbeitung von XML-Informationen und Strukturen, um diese abzulegen. Dazu existiert neben WebDAV auch die Datenstruktur WT-Tree, ein ausbalancierter Baum. Basierend auf beiden Strukturen wurden Datenbank-Lösungen entwickelt, untersucht und schließlich als ungeeignet und zu aufwendig befunden: komplexe Anfragen für wenige Datensätze oder Datenstruktur mit unzureichendem Speicherplatz.

Folglich wird eine externe Datenbank-Lösung notwendig, die bereits alle notwendigen Operationen zufriedenstellend ausführt und auch ausreichend getestet wurde. SQLite bietet dafür eine geeignete Lösung; es speichert die Datenbank in einer Datei, was sehr gut zur Übertragung der Datei in Fragmenten in eine WebDAV-Struktur geeignet ist.

Ein besonderer Vorteil von SQLite ist die Verarbeitung von SQL-Anweisungen. Um die Informationen abzufragen oder zu senden, ist lediglich eine Zeichenkette notwendig, die die passenden Informationen enthält. Das macht SQLite zu einer sehr leicht zu nutzenden Datenbank-Bibliothek für Askemos® und verringert die Möglichkeit, dass eine Zeitüberschreitung stattfindet, um ein Vielfaches.

Auf Anwendungsebene von Askemos® wird zugelassen, dass ein Benutzer eine SQL-Anweisung ausführt, die im Kern von Askemos® vorbereitet und im Bereich von RScheme ausgeführt wird. Dabei wird aus einem WebDAV von Askemos® gelesen, die SQL-Anweisung verarbeitet und in dasselbe WebDAV geschrieben. Das WebDAV ist in dieser Hinsicht eine “Datenbank-Datei” und einfacher im Netzwerk verteilbar.

Um die Datenbank-Datei in ein WebDAV speichern zu können, wird eine effiziente Teilung der Datenbank-Datei in kleine Fragmente, beispielsweise mit einer Größe von 512KB, realisiert, die sich sehr einfach im Netzwerk verteilen lassen. Schließlich ist es erforderlich, die bestehende Schnittstelle zu analysieren und die Stelle zu finden, an der die Erweiterung für SQLite und Askemos® - die Dateisystem-Schnittstelle - erfolgen muss.

Folgende Abschnitte zeigen Analysen des Askemos® Quelltextes, wobei Funktionen aus verschiedenen Abschnitten verwendet wurden. Die gesamte Auflistung zeigt somit den kompletten Ablauf, der analysiert wurde und stellt das Gesamtverständnis her.

3.2.1 Beispiel zur Verwendung von SQLite

Ein Entwicklungs-WebDAV enthält ein Anwendungsbeispiel [9] zu SQLite. Das Beispiel stellt eine Buchhaltung dar und gibt einen Überblick, wie man SQLite aufruft und verwendet.

Folgende Abschnitte stellen eine Operation und zwei Aufruf-Strukturen vor. Die Lese-Operation `(me "SQL-ANWEISUNG" 'sql-query)` führt einen lesenden SQLite-Aufruf durch. Eine SQL-Anweisung (engl. SQL statement) ist beispielsweise `"SELECT * FROM tabelle1"`. An jeder Stelle im XML-Stylesheet ist es möglich die Lese-Operation aufzurufen und die zurückgegebenen Daten zur Weiterverarbeitung zu verwenden. Schreibende SQL-Anweisungen sind beispielsweise `"INSERT INTO [...]"`, die Daten einfügt oder `"CREATE TABLE [...]"`, die eine neue Tabelle anlegt.

Die zwei Aufruf-Strukturen sind XML-Knoten, die eine Verarbeitung realisieren. Die erste Aufruf-Struktur stellt eine Grundlage bereit, um Informationen mit lesenden Aufrufen formatiert auszugeben. Die Struktur besteht aus einem XML-Knoten mit dem Schlüsselwort `query`, Attributen und der SQL-Anweisung im Körper des XML-Knotens.

```
<table>
  <thead> <!-- [...] --> </thead>
  <xsql:query id-attribute="id" id-attribute-column="rowid"
              max-rows="10000"  row-element="tr"
              rowset-element="tbody">
   SELECT nummer as td, name as td, name2 as td, telefon as td, fax as td, handy as td,
          email as td, strasse as td, plz as td, ort as td, url as td
     FROM adresse
     ORDER BY name ASC
  </xsql:query>
</table>
```

Listing 3.1: xsql:query – Abfrage einer SQLite-Datenbank

Wie im Beispiel-Listing 3.1 gezeigt, erleichtert die Aufruf-Struktur `xsql:query` die Formatierung von Informationen. Im XML-Knoten `<xsql:query [...]>` sind gezeigte Attribute möglich, um das Ergebnis zu formatieren und einzuschränken, beispielsweise die Begrenzung der maximal anzuzeigenden Zeilen. Das Transformationsergebnis des genannten XML-Knotens ist eine XML-Struktur, die in diesem Fall ein HTML-Tabellen-Körper ergibt.

Das Wichtigste ist das Aufrufen von schreibenden SQLite-Anfragen. Die zweite Aufruf-Struktur führt schreibende SQLite-Aufrufe ohne Einschränkungen durch. Allerdings wird die Aufruf-Struktur nur im Schreib-Bereich eines Platzes transformiert.

Das folgende Listing 3.2 zeigt, wie eine Antwort auf eine Anfrage formuliert wird. Dabei werden zwei Mittel gezeigt, um Aktionen durchzuführen:

```
<xsl:template match="request[@type=&quot;write&quot;]">
  <!-- [...] -->
  <d:when test="[...]">
    <core:reply>
      <core:continue>
        <d:copy-of select="(grove-root (current-node))"/>
      </core:continue>
      <core:update>
        <d:copy-of select="#CONTENT">
         (string-append
          "INSERT OR REPLACE INTO
           cursor(user, focus)
           values('"
          (oid->string (msg &apos;dc-creator)) "','"
          (sql-quote (data (form-field &apos;name (current-node)))) "')")
        </d:copy-of>
      </core:update>
    </core:reply>
  </d:when>
  <!-- [...] -->
</xsl:template>
```

Listing 3.2: core:update – Schreibende Aufrufe an eine SQLite-Datenbank

Der XML-Knoten `<core:update>` ist eine Platz-Erweiterung, die die angegebene Zeichenkette an die SQLite-Datenbank schickt. Im Scheme-Quelltext, der durch den XML-Knoten `<d:copy-of [...]>` (Listing 3.2) ausgeführt wird, sollten Inhalte von Formular-Feldern mit `sql-quote` überprüft werden. Der Scheme-Befehl `sql-quote` ersetzt Zeichen der übergebenen Zeichenkette, die Fehler in der Ausführung zur Folge haben.

3.2.2 Ablauf von lesenden und schreibenden SQLite-Aufrufen

In der Scheme-Datei `mechanism/place.scm` sind Funktionen zu finden, die den Zugriff für einen Platz steuern: `make_reader_access` für Lesezugriffe und `make_access` für Schreibzugriffe. In beiden Funktionen findet eine ähnliche Verarbeitung für SQLite-Aufrufe statt.

Zuerst wird ein Mutex namens `sqlite-read-connection` definiert, damit kein anderer Thread gleichzeitig die Datenbank-Datei verwendet. Im Laufe der Verarbeitung ist der Mutex eine Variable, die den Dateizeiger auf die geöffnete Datenbank-Datei enthält.

Der Dateizeiger wird dabei wie folgt initialisiert. Zunächst wird der Dateiname ermittelt (Listing 3.3):

```
(sqlite3-open-restricted-ro ((xsql-user-database-file)
                             (aggregate-entity frame)))
```

Listing 3.3: Funktion - Öffnen der DB-Datei

Der Dateiname besteht aus der OID des Platzes. Dateiname und Pfad der Datei werden durch die Funktion `(aggregate-entity frame)` zurückgegeben. Der Aufruf von `(xsql-user-database-file)` speichert Dateiname und Pfad in einer globalen Variablen ab, die während der Verarbeitung zur Verfügung steht.

Im folgenden Verlauf der Verarbeitung wird die Art des Zugriffes (durch `(me 'slot 'system)`) einem Aufruf zugeordnet. Um einen SQLite-Aufruf zu starten, ist die Angabe einer SQL-Anweisung notwendig, z.B. `(me "SELECT * FROM table1" 'sql-query)`.

Es wird die Funktion `sql-exec-protected` (aus `mechanism/function/xsql.scm`) mit folgenden Parametern ausgeführt: die Datenbank-Datei und der auszuführenden SQL-Anweisung. In der Definition der Funktion ist der lesende Zugriff auf die SQLite-Datenbank-Datei formuliert.

Weiterverfolgt lässt sich erkennen, dass die Funktion `sqlite3-exec` aufgerufen wird (aus `rscheme/library/dev/rs/db/sqlite3/api.scm`). Dies ist der Bereich, in den die SQLite-Schnittstelle eingebunden wurde.

Schreibende SQLite-Aufrufe werden nur im schreibenden Template gestartet. Der

XML-Knoten `<core:update>` wurde bereits im Listing 3.2 vorgestellt. In der Scheme-Datei `mechanism/nunu.scm` befindet sich die Definition der Transformation:

```
(define (is-sql-extension? x r n)
        (is-mind-element? x 'update))
(define-transformer do-sql-extension
  (let ((update (xml-walk-down sosofos: (children nl))))
    (place (if (and (pair? update) (symbol? (car update)))
               (sql-write update)
               (data update))
           'sql)))
(define sql-extension
        (cons is-sql-extension? do-sql-extension))
```

Listing 3.4: nunu.scm: SQLite-Erweiterung

Die lokale Variable `update` im Listing 3.4 beinhaltet die SQL-Anweisung, die zuvor in einem `<core:update>` XML-Knoten angegeben wurde. Die Ausführung der Funktion `place` führt dazu, dass in der laufenden Anfrage für den Platz, der die Transformation ausführt, die SQL-Anweisung notiert wird.

Nach Ausführen der Funktion `make-access` kommt es zum Aufruf der Funktion `(commit-cache commit-version)`. Dabei wird schließlich die Funktion `commit-sql!` aufgerufen, die eine Datenbankverbindung mit uneingeschränkten Zugriffsrechten öffnet und mittels der Funktion `sqlite3-exec` die SQL-Anweisung ausführen lässt.

3.2.3 Verwalten der Daten in einem WebDAV

Im letzten Schritt wurde festgestellt, dass die SQLite-Bibliothek im RScheme-Bereich von Askemos® `rscheme/library/dev/rs/db/sqlite3/` angebunden wurde. Eine Analyse des `sqlite3`-Ordners zeigt, dass es eine Möglichkeit gibt, Quelltext der Programmiersprache C direkt mit RScheme zu verbinden.

In der Scheme-Datei `glue.scm` ist ein Makro definiert, das die Verbindung der Funktionen von SQLite erleichtert. Im Makro wird eine Funktion `define-safe-glue` aufgerufen, der ein Funktionsname, ein oder mehrere Parameter und ein auszuführender C-Quelltext übergeben werden. Die angegebenen Parameter sind dabei auch im C-Quelltext erreichbar.

Im weiteren Verlauf der Funktionsdefinition werden Dateityp-Behandlungsroutinen angegeben, die im Hintergrund die Dateitypen zwischen den beiden Programmiersprachen verbinden. Letztlich werden Dateien angegeben, die während der Übersetzung von Askemos® in das Binärformat, alle nötigen Funktionsdefinitionen und Funktionen bereitstellen. Zu den als letztes einzubindenden Dateien zählt auch eine C-Quelltext-Datei `sqglue.c`, in der sich Konvertierungsfunktionen für RScheme und SQLite befinden.

In der Scheme-Datei `api.scm` wurden alle notwendigen SQLite Aufrufe formuliert, die an RScheme angebunden werden. Weiterhin befinden sich darin auch Fehlermeldungen und Scheme-Funktionen, die die Abarbeitung der SQLite-Anweisungen erleichtern.

Folglich ist eine weitere C-Quellcode-Datei anzulegen, in der sich die Implementierung des neuen VFS-Objektes befindet. Diese Implementierung muss stets vor dem Aufruf einer `sqlite3_open` Funktion registriert werden, damit keine Verarbeitungsprobleme auftreten.

Bereits bekannt ist, wie eine SQL-Anweisung bis in den C-Quelltext gelangt. Die Analyse der SQLite-VFS-Implementierungen hat nun folgende Vorgehensweise von SQLite beim Schreiben in und Lesen aus einer Datenbank-Datei ergeben:

- Anforderung eines Bereiches der Datenbank-Datei (ein oder mehrere Blöcke)
- Verarbeitung des Bereiches
- Freigabe des angeforderten Bereiches.

Eine Anfrage an den Entwickler von Askemos® konnte klären, wie Daten aus einer C-Funktion in den RScheme-Bereich geschickt werden können. Die C-Schnittstelle von RScheme bietet verschiedene Möglichkeiten, Unterbrechungsanforderungen (eng. interrupt call, interrupt request) abzusenden. Innerhalb der Unterbrechung verarbeitet RScheme die Daten, die von einer C-Funktion gesendet werden. Eine Unterbrechung ist möglichst schnell zu beenden, damit RScheme auf andere Anfragen reagieren kann.

Die Scheme-Funktion `signal-subject!`, die vom Entwickler vorgeschlagen wurde, verarbeitet folgende drei Parameter und schickt die Nachricht an den angegebenen Platz: die OID des Ziel-Platzes, die Anfrage-Art (`'read` oder `'write`) und eine Nachricht mit Inhalt. Eine Nachricht mit der Anfrage-Art `'write` löst eine schreibende Transformation eines Platzes aus.

Letztere Funktion hat den Vorteil, dass sie eine gleichzeitige Verarbeitung auf jedem eingetragenen Askemos®-Server verursacht, die aus dem "Quorum" eines Platzes bezogen werden (die Liste der beteiligten Askemos®-Server).

Schließlich muss der Ziel-Platz entsprechende Transformationsregeln beinhalten, damit dieser auf schreibende und lesende Anfragen der Datenbank-Datei reagieren kann.

Analog zu schreibenden Anfragen wird der C-Quelltext die Verarbeitung einer lesenden Anfrage durchführen, wobei jedoch als Anfrage-Art das Symbol `'read` an `signal-subject!` übergeben wird. Der Rückgabe-Wert von `signal-subject!` enthält Daten, die von einem Platz ausgegeben werden, beispielsweise wie folgt (Listing 3.5):

```
<!-- [...] -->
<xsl:template match="*[@type=&quot;read&quot;]">
  <!-- [...] -->
  <d:when
   test='(equal? (data (form-field &apos;action (current-node))) "sqlite")'>
    <d:copy-of select="#CONTENT">
      (fetch (list "sqlite-dav" (data (form-field &apos;block_nr (current-node)))))
    </d:copy-of>
  </d:when>
  <!-- [...] -->
</xsl:template>
<!-- [...] -->
```

Listing 3.5: Rückgabe-Wert einer lesenden Anfrage

Listing 3.5 stellt dar, wie man die Ausgabe eines Datei-Blockes realisieren könnte. Wie der Datei-Block zu SQLite weitergereicht wird, ist Aufgabe der Implementierung. Der beschriebene Ablauf wiederholt sich bis SQLite die komplette Datei in den Speicher geladen hat.

Eine Erweiterung [12] von SQLite ermöglicht, dass alle Schreib- und Lese-Anfragen asynchron durch einen weiteren Thread verarbeitet werden. Auf diese Weise wären komplexe SQLite-Verarbeitungen unterbrechbar. Folglich kann Askemos® währenddessen auf andere Verarbeitungen reagieren.

Aufgabe der Implementierung ist die Entwicklung einer Hülle (engl. wrapper) um eine bestehende VFS-Struktur und die Erweiterung durch spezielle Askemos®-Aufrufe.

4 Lösungskonzeption / Programm-Entwurf

4.1 Lösungsschema des Kerns von Shop-Systemen

In der Abbildung 4.1 ist ein Beispiel für die Tabellen `Benutzerkonten` und `Angebote` des abstrakten Datenbankschemas (Abbildung 3.6) dargestellt; die Bezeichnungen der Felder befinden sich in der Tabelle `Einstellungen`:

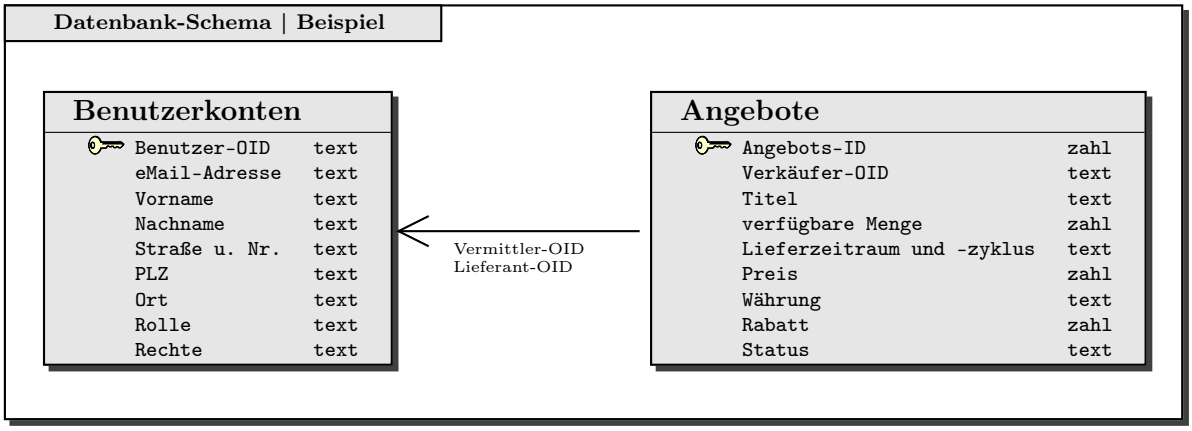

Abbildung 4.1: Exemplarische Felder von Tabellen

Alle Anwendungsfälle der Analysephase werden zweifach als XSL-Templates implementiert: Lesende und schreibende XSL-Templates. Ausnahmen werden in der Tabelle 4.1 durch den Zelleninhalt "*Keine Aktion*" angegeben. Aufgerufen werden die Templates mittels eines Formular-Feldes "action", dabei werden die globalen Anwendungsfälle, wie Benutzerkontenverwaltung, abgekürzt.

Folgende Tabelle (Tabelle 4.1) zeigt alle zu implementierenden Templates entsprechend den Anwendungsfällen mit Name und Zusatz. Ein Beispiel für einen Template-Name ist `bkv:create-read`. Der Zusatz zeigt an, welche Transformationsart durchgeführt wird (`-read` für Lesen und `-write` für Schreiben). In jeder lesenden Transformation werden XML-Daten zurückgegeben.

	`-read`	`-write`	**Prod.-Funk.**
`bkv:create`	Formular: Erstellen eines Verkäuferkontos	Erfassung und Erstellung des Verkäuferkontos	/F0211/
`bkv:edit`	Formular: Bearbeiten eines Benutzerkontos	Schreiben der Änderungen des Benutzerkontos	/F0212/
`bkv:delete`	Formular: Löschen eines Benutzerkonotos	Löschen des Benutzerkontos und eigener Angebote/Bestellungen	/F0213/
`bkv:lock`	Formular: Benutzerkonto sperren oder entsperren	Benutzerkonto sperren oder entsperren	/F0214/
`bd:shopping`	Formular: Übersicht der vorhandenen Artikel im Warenkorb und verfügbare Aktionen	Durchführen der möglichen Aktionen (Hinzufügen, Ändern, Löschen)	/F0221/
`bd:booking`	Formular: Auflistung der Bestellung zum Abschließen der Bestellung	Buchen (Abschließen) der Bestellung	/F0222/
`bd:invoice`	Ausgabe der Rechnung als PDF-Dokument	*Keine Aktion*	/F0223/
`bv:list`	Auflistung der Bestellungen	*Keine Aktion*	/F0231/, /F0232/
`bv:archive`	Formular: Archivierung von Bestellungen	Schreiben der Archivierung von Bestellungen	/F0233/
`bv:cancel`	Formular: Stornierung einer Bestellung	Schreiben der Stornierung einer Bestellung	/F0234/
`bv:delete`	Formular: Löschen einer Bestellung	Löschen einer Bestellung	/F0235/

Fortsetzung auf nächster Seite

	-read	-write	**Prod.-Funk.**
av:list	Auflistung der Angebote	*Keine Aktion*	/F0241/, /F0242/
av:insert	Formular: Eingabe von Informationen zu einem Angebot	Einfügen des Angebotes	/F0243/
av:edit	Formular: Bearbeiten von Informationen eines Angebots	Schreiben der Änderung	/F0244/
av:delete	Formular: Löschen eines Angebotes	Löschen eines Angebotes	/F0245/

Tabelle 4.1: Übersicht der zu implementierenden XSL-Templates

Aus Tabelle 4.1 ist ersichtlich, dass die zwei Arten von Auflistungen (Bestellungen und Angebote) durch ein Shop-System zu formatieren und zu verarbeiten sind. Der Kern von Shop-Systemen gibt eine Auflistung zurück, die auf den Benutzer, der eine Anfrage an das Shop-System gestellt hat, abgestimmt ist.

4.2 Entwurf der Templates des Kerns von Shop-Systemen

Die 27 Templates (Tabelle 4.1) sind entsprechend der Abkürzungen in eigene XSL-Stylesheets zu implementieren, die in ein globales XSL-Stylesheet eingebunden werden.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet
    xmlns:d="http://www.askemos.org/2005/NameSpaceDSSSL/"
    xmlns:editor="urn:editor"
    xmlns:core="http://www.askemos.org/2000/CoreAPI"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

 <settings>
  <action>A00000000000000000000000000000001</action>
  <protection>A00000000000000000000000000000001</protection>
  <mailer>A7ba26ad21d4585f9c44c8199a50325b9</mailer>
 </settings>
```

```
<xsl:template match="request[@type=&quot;read&quot;]">
 <xsl:choose>
  <d:when test="(pair? (msg &apos;destination))"><core:forward/></d:when>
  <d:when test='(pair? (xsl-variable apply: "is-meta-form"))'>
   <d:copy-of select="(message-body (metaview me msg))"/>
  </d:when>
  <xsl:otherwise>
   <d:call-template name='
      (let ((f (form-field &apos;action (current-node))))
           (if (null? f) "html" (literal (data f) "-read")))'/>
  </xsl:otherwise>
 </xsl:choose>
</xsl:template>

<xsl:template match="request[@type=&quot;write&quot;]">
 <xsl:choose>
  <d:when test="(pair? (msg &apos;destination))"><core:forward/></d:when>
  <d:when test='(pair? (xsl-variable apply: "is-meta-form"))'>
   <d:if test="(not (or (service-level (me 'get 'id))
                        (error &quot;Forbidden (only owner)!!&quot;)))"/>
   <d:copy-of select="(message-body (metactrl me msg))"/>
  </d:when>
  <xsl:otherwise>
   <d:call-template name='
      (let ((f (form-field &apos;action (current-node))))
           (if (null? f)
               (error (literal "Unknown action in "
                               (xml-format (current-node))))
               (literal (data f) "-write")))'/>
  </xsl:otherwise>
 </xsl:choose>
</xsl:template>

<xsl:include href="Ad796d626b79a727f828946e2d78a4d68/lib/kss/util.xsl"/>
<xsl:include href="Ad796d626b79a727f828946e2d78a4d68/lib/kss/bkv.xsl"/>
<xsl:include href="Ad796d626b79a727f828946e2d78a4d68/lib/kss/bd.xsl"/>
<xsl:include href="Ad796d626b79a727f828946e2d78a4d68/lib/kss/bv.xsl"/>
<xsl:include href="Ad796d626b79a727f828946e2d78a4d68/lib/kss/av.xsl"/>

</xsl:stylesheet>
```

Listing 4.1: Grundstruktur des XSL-Stylesheets des Kerns von Shop-Systemen

Im Folgenden werden einige Template-Entwürfe (Templates siehe Tabelle 4.1) vorgestellt. Der Entwurf erfolgt durch Ablauf-Definitionen in Kommentaren. Alle Anfor-

derungen von Templates werden in einem Kommentar vor der Definition angegeben.

```
<xsl:stylesheet [...] >
 <!-- [...] -->
 <!-- Benutzer-Konten-Verwaltung : Erzeugen - Lesen
      Ablauf:
        1) aktuelle Benutzerdaten holen
        2) Kopfdaten der dynamische Felder extrahieren
        3) Testen ob Benutzer existiert (ob Daten existieren)
        4) Ausgabe der Kopfdaten und notwendiger Formulardaten
 -->
 <xsl:template name="bkv:create-read">
  <core:output media-type="text/xml">
   <kss>
    <d:copy-of select="#CONTENT">
     (guard (ex (else `(error ,(if (condition? ex)
                                   (condition-message ex) (literal ex)))))
            (error "Zu implementieren!"))
    </d:copy-of>
   </kss>
  </core:output>
 </xsl:template>

 <!-- Benutzer-Konten-Verwaltung : Erzeugen - Schreiben
     Ablauf:
      <core:continue> : Platz neu schreiben mit gleichem Inhalt
      <core:update>   : SQL-Anweisung senden, Einstellungen holen,
                        nur eingestellte Felder abrufen
     Parameter:
       feld0 ... feldn : abzurufende Felder
 -->
 <xsl:template name="bkv:create-write">
  <d:if test='(error "Zu implementieren!")'/>
  <core:reply>
   <core:continue><d:copy-of select="(grove-root (current-node))"/></core:continue>
   <core:update>INSERT INTO users VALUE();</core:update>
   <core:output media-type="text/xml">
    <d:copy-of select="#CONTENT">
     (literal "bkv:create-write")
    </d:copy-of>
   </core:output>
  </core:reply>
 </xsl:template>
 <!-- [...] -->
</xsl:stylesheet>
```

Listing 4.2: XSL-Templates des Kerns von Shop-Systemen

4.3 Schnittstelle zwischen Askemos® und SQLite

Die Schnittstelle zwischen Askemos® und SQLite besteht aus der Implementierung einer `sqlite3_vfs` C-Struktur [11]. Die `sqlite3_vfs` C-Struktur enthält Funktionen, die für ein bestimmtes Dateisystem ausgeführt werden sollen. Folgendes Listing 4.3 zeigt den Lösungsansatz einer derartigen C-Struktur:

```
static sqlite3_vfs askemos_vfs =
{
  1,                    /* iVersion */
  sizeof(askemos_file), /* szOsFile */
  ASKEMOS_MAX_PATHNAME, /* mxPathname */
  0,                    /* pNext */
  "askemos",            /* zName */
  0,                    /* pAppData */

  /* vfs functions */
  askemosOpen,          /* xOpen */
  askemosDelete,        /* xDelete */
  askemosAccess,        /* xAccess */
  askemosFullPathname,  /* xFullPathname */
  askemosDlOpen,        /* xDlOpen */
  askemosDlError,       /* xDlError */
  askemosDlSym,         /* xDlSym */
  askemosDlClose,       /* xDlClose */
  askemosRandomness,    /* xRandomness */
  askemosSleep,         /* xSleep */
  askemosCurrentTime,   /* xCurrentTime */
  askemosGetLastError   /* xGetLastError */
};
```

Listing 4.3: Implementierung der `sqlite3_vfs` C-Struktur

Der erste Abschnitt der C-Struktur (Listing 4.3) enthält fest definierte Werte, die von SQLite vorgegeben sind. Darunter befinden sich die Größe der C-Struktur `askemos_file` und der Name des `sqlite3_vfs` "`askemos`". Die C-Struktur `askemos_file` wird im weiteren Verlauf des Entwurfes vorgestellt. Alle anderen Werte sind festgelegt oder dürfen nicht verändert werden, exemplarisch die `iVersion` oder `pNext`. Die Zeiger-Variable `pNext` wird von SQLite verwendet, um die vorhandenen `sqlite3_vfs` C-Strukture in einer Liste anzuordnen.

Im zweiten Abschnitt der C-Struktur (Listing 4.3) sind Funktionsnamen definiert, die durch SQLite aufgerufen werden. Ein wichtiges Detail ist dabei, dass alle zu implemen-

tierenden `askemos_vfs`-Funktionen zunächst die Funktionen der UNIX-Implementierung aufrufen (vgl. Listing 4.4) und deren Rückgabewert weitergeben.

```
static int askemosSleep(sqlite3_vfs *v, int ms){
  return unixSleep(v, ms);
}
```

Listing 4.4: Wrapping der UNIX-Funktionen

Alle anderen Funktionen außer `askemosOpen`, in welcher die C-Struktur `askemos_file` initialisiert wird, rufen die passende Funktion der UNIX-Implementierung auf. Somit bleibt die Organisation der Datenbank-Datei im Dateisystem bestehen. Allerdings wird der Inhalt der Datei in einem WebDAV verwaltet.

In der `askemos_file` C-Struktur (Listing 4.6) sind Variablen und C-Struktur definiert, die in der Askemos®-Dateisystem-Schnittstelle global verwendet werden sollen.

```
typedef struct askemos_file askemos_file;
struct askemos_file {
  sqlite3_file base;
  sqlite3_file *pReal;
  AskemosVfs *pAskemosVfs;
  int iFileId;              /* File id number */
  int flags;
  obj callback;
};
```

Listing 4.5: C-Struktur `askemos_file`

Einerseits enthält `askemos_file` einen Zeiger auf die `unixFile` C-Struktur (`pReal`), um die Verwaltung der Datenbank-Datei im Dateisystem zu garantieren. Andererseits enhält `askemos_file` eine C-Struktur und zwei Zeiger, die im Askemos®-VFS angewendet werden:

- der Zeiger `pAskemosVfs` verweist auf die C-Struktur `AskemosVfs`
- die C-Struktur `base` ist eine Ableitung der `sqlite3_file` C-Struktur, die eine C-Struktur mit Datei-Verarbeitungsmethoden beinhaltet:

```
static sqlite3_io_methods askemos_io_methods = {
  1,                                /* iVersion */
  askemosClose,                     /* xClose */
  askemosRead,                      /* xRead */
  askemosWrite,                     /* xWrite */
```

```
  askemosTruncate,                /* xTruncate */
  askemosSync,                    /* xSync */
  askemosFileSize,                /* xFileSize */
  askemosLock,                    /* xLock */
  askemosUnlock,                  /* xUnlock */
  askemosCheckReservedLock,       /* xCheckReservedLock */
  askemosFileControl,             /* xFileControl */
  askemosSectorSize,              /* xSectorSize */
  askemosDeviceCharacteristics    /* xDeviceCharacteristics */
};
```

Listing 4.6: Datei-Verarbeitungsmethoden: C-Struktur `askemos_io_methods`

Die Verarbeitungsmethoden sind analog den Funktionen der VFS-Struktur aufgebaut: Es wird jeweils die Verarbeitungsmethode der übergeordneten VFS-Struktur aufgerufen. Besondere Bedeutung haben die Verarbeitungsmethoden `askemosRead` und `askemosWrite`. In beiden findet die Implementierung der WebDAV-Verarbeitung statt.

Im Listing 4.4 wurde vorausgesetzt, dass die Funktionen des UNIX-VFS global exportiert wurden, was anhand des Quelltextes nicht der Fall ist. Folglich kann das einfache Aufrufen einer Funktion nicht verwendet werden, was wie folgt gelöst wurde:

1. Im SQLite ist die UNIX-VFS-Struktur registriert.
2. Einrichten einer VFS-Struktur, welche die C-Struktur `askemos_vfs` und eine übergeordnete VFS-Struktur, beispielsweise die UNIX-VFS-Struktur, enthält.
3. Suchen der UNIX-VFS-Struktur mittels folgender Funktion:
   `sqlite3_vfs *sqlite3_vfs_find(const char *zVfsName);`.
   Der Ergebnis-Wert ist die gesuchte VFS-Struktur oder `NULL`, wenn keine VFS-Struktur mit dem Name `zVfsName` registriert ist.
4. Funktionen der `askemos_vfs` C-Struktur an das neue Aufruf-Muster anpassen:

```
static int askemosSync(sqlite3_file *pFile, int flags){
  askemos_file *p = (askemos_file *)pFile;
  return p->pReal->pMethods->xSync(p->pReal, flags);
}
#define REALVFS(p) (((AskemosVfs *)(p))->pVfs)
static int askemosSleep(sqlite3_vfs *pVfs, int nMicro){
    return REALVFS(pVfs)->xSleep(REALVFS(pVfs), nMicro);
}
```

Listing 4.7: Wrapping der UNIX-Funktionen durch Zeiger

Anhand des definierten Aufbaus einer `sqlite3_vfs` C-Struktur ist es möglich, eine Funktion einer VFS-Struktur mittels globalen Funktionsnamen aufzurufen, exemplarisch `xSleep` oder `xSync`. Hilfreich dabei sind C-Makros zur Vereinfachung der Funktionsaufrufe der übergeordneten VFS-Struktur, z.B.: `#define REALVFS(p)` ersetzt `p` durch einen Zeiger auf die übergeordnete VFS-Struktur. Folglich sind alle Funktionen anzupassen und im Verlauf der Implementierung so zu verändern, dass eine WebDAV-Verarbeitung ermöglicht wird.

Die neue C-Struktur `AskemosVfs` (Listing 4.8) stellt eine Verbindung zwischen dem übergeordneten und dem Askemos®-VFS her. Dabei beinhaltet die C-Struktur `base` die grundlegenden Verarbeitungsmethoden, in denen die jeweiligen Verarbeitungsmethoden der übergeordneten VFS-Struktur (Zeiger `pVfs`) aufgerufen werden.

Zur Vorbereitung der Daten für RScheme und zur Kommunikation mit RScheme wurden RScheme-Objekte (`obj`) angelegt. Zwei dieser Objekte (`callback` und `cb_return`) enthalten anonyme Lambda-Funktionen, die beim Start des `AskemosVfs` übergeben worden und dem Unterbrechungsaufruf von RScheme mitgegeben werden. Damit müssen keine Symbole aufgelöst werden und vereinfacht die Abarbeitung der Funktionen `askemosRead` und `askemosWrite`.

```
struct AskemosVfs {
  sqlite3_vfs base;
  sqlite3_vfs *pVfs; /* parent */

  int iNextFileId;
  obj callback;
  obj cb_return;
  obj mailbox;
};
typedef struct AskemosVfs AskemosVfs;
```

Listing 4.8: C-Struktur `AskemosVFS`

Zur Registrierung und Entfernung der Askemos®-VFS-Struktur dienen die zwei Funktionen `sqlite3-askemosvfs-create` und `sqlite3-askemosvfs-destroy`. Die Funktion `sqlite3-askemosvfs-create` ist vor dem Öffnen einer Datenbank-Datei aufzurufen und `sqlite3-askemosvfs-destroy` beim Schließen.

Die Integrierung des entworfenen `AskemosVfs` umhüllt das als Standard eingetragene VFS und wird mithilfe letzterer Grundlage entwickelt. Folglich wird das `AskemosVfs`

mit dem Funktionsaufruf `sqlite3_askemosvfs_create();` vor einem öffnenden SQLite3-Aufruf eingerichtet. Dabei ist im Verbindungscode (zwischen RScheme und C) die Funktion zur Deklaration als `extern` anzugeben (Listing 4.9):

```
(define-sqlite-glue (sqlite3-open* (dbn <raw-string>))
  literals: ((& <sqlite3-database>) (& <sqlite3-error>))
{
  extern sqlite3_vfs *sqlite3_askemosvfs_create();

  sqlite3 *cnx;
  int rc;

  sqlite3_askemosvfs_create();

  /* [...] */
  rc = sqlite3_open_v2( dbn, &cnx,
                        SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE, "askemos" );
  /* [...] */
})
```

Listing 4.9: Initialisierung des `AskemosVFS`

Schließlich ist die Funktion `sqlite3_askemosvfs_destroy(sqlite3_vfs*)` nach dem schließenden SQLite3-Aufruf auszuführen, um den zuvor reservierten Speicherbereich freizugeben und das `AskemosVfs` auszuhängen (Listing 4.10):

```
(define-sqlite-glue (sqlite3-close (db <sqlite3-database>))
  literals: ((& <sqlite3-error>))
{
  extern void sqlite3_askemosvfs_destroy(sqlite3_vfs*);

  int rc;
  sqlite3_vfs *aVfs;

  rc = sqlite3_close( db );
  /* [...] */

  aVfs = sqlite3_vfs_find("askemos");
  sqlite3_askemosvfs_destroy(aVfs);

  RETURN0();
})
```

Listing 4.10: Beenden des `AskemosVFS`

5 Realisierung / Implementierung

5.1 Hilfsmittel

Zur Entwicklung wurde der Editor Emacs mit Erweiterungen verwendet, die das Hervorheben von Syntaxkomponenten und bei XML auch eine Überprüfung auf Fehler ermöglichen. Des Weiteren wurden Erweiterungen eingesetzt, die eine Telnet-Verbindung und das Ausführen eines Kommandozeileninterpreters realisieren.

Zur Übertragung von XSL-Stylesheets in ein Askemos®-Entwicklungs-WebDAV wurde das Kommandozeilenprogramm `cadaver` verwendet. Das Ausführen und Testen des Kerns von Shop-Systemen wurde mit dem Browser Firefox durchgeführt. Dabei stellen Firefox und `cadaver` eine Verbindung mit einem laufenden Askemos®-Server her.

Zur Implementierung des Kerns von Shop-Systemen wurden bisherige Programmier-Erfahrungen mit Askemos® eingesetzt, beispielsweise in Form von anderen eigenen Entwicklungen. Des Weiteren wurde zum Testen das Java-Script-Framework MooTools (`http://mootools.net`) verwendet. Als Referenz für Scheme-Befehle (z.B. für Vektoren) wurden SRFIs (`http://srfi.schemers.org/`) zurate gezogen. Entsprechende Anbindungen von Scheme-Befehlen wurden im Askemos®-Quelltext gesucht.

Zur Realisierung der Datenbankschnittstelle wurden folgende Quellen verwendet: SQLite Quelltexte und Dokumentation [5] sowie der Kontakt zum Entwickler von Askemos®. Das Testen der Datenbankschnittstelle erfolgte während der Entwicklung auf einem lokalen Askemos®-Server, der in keinem Netzwerk eingebunden ist.

5.2 Details zum Kern von Shop-Systemen

Nachfolgende grundlegende Regeln sollen bei jedem verarbeitenden Template berücksichtigt werden, außer beim ersten Schreiben von Einstellungen. Zur Vereinfachung ist ein Template anzulegen, welches alle Überprüfungen durchführt und alles Nötige zurückgibt:

- Testen, ob die Datenbank existiert: Ausnahmebehandlung des nächsten Schrittes.
- Prüfen der Datenbank-Struktur: Vorhandensein dynamischer Felder.
- Derzeitigen Benutzer und dessen Daten anhand der Datenbank und der OID (Rückgabe von `(msg 'dc-creator)`) ermitteln.
- Prüfen der Berechtigungen des Benutzers (anhand der Benutzergruppe).

5.2.1 Einrichtung einer Datenbank

Bevor eine Abfrage erfolgreich durchgeführt werden kann, müssen eine Datenbank-Datei (oder ein WebDAV) und alle Tabellen angelegt werden. Eine Datenbank-Datei wird angelegt, sobald eine schreibende Datenbank-Operation ausgeführt wird. Die SQLite-Funktion `sqlite3_open` legt dabei die Datenbank-Datei an.

Ein schreibendes Template für Einstellungen sorgt dafür, dass alle Tabellen entsprechend der Angaben (z.B. dynamische Felder) angelegt werden. Dynamische Felder können nach dem ersten Einstellen auch verwaltet werden: Ändern, Hinzufügen, Löschen. Es wäre dabei hilfreich, auch Beispiele für dynamische Felder anzugeben (eine Weiterentwicklung).

Zur Einrichtung der Datenbank werden Angaben zu den dynamischen Feldern benötigt. Dynamische Felder existieren in den Tabellen Benutzerkonten (users) und Angebote (offers). Die Bezeichnungen der dynamischen Felder werden in einer Einstellungstabelle (settings) abgelegt, die folgende Spalten hat:

name Bezeichnung von Einstellungen (oder dynamisches Feld).

type-id Typ-ID von Einstellungen, deren Beschreibung sich in der Tabelle Typen (types) befindet, in der die Typ-ID-Spalte der Primärschlüssel ist.

value Inhalt der Einstellung (z.B. Titel des dynamischen Feldes).

In den Tabellen Benutzerkonten und Angebote existieren damit Spaltennamen, die einen Platzhalter beinhalten, beispielsweise “feld0” oder “feld1”. Beide Tabellen enthalten dabei noch zwingend notwendige Spalten. Die Tabelle Benutzerkonten enthält eine Spalte Rollen-ID (role-id), deren Bedeutung aus der Tabelle Rollen (roles) bezogen werden kann. Weitere Tabellen werden entsprechend des Datenbankschemas (Abb. 3.6) erstellt. Die Informationen aus den Formular-Feldern werden dabei speziell behandelt:

- Es werden nur Formular-Felder akzeptiert, die einem Schema entsprechen, beispielsweise `0-feld0-val`. Das Schema wird im Folgenden als regulärer Ausdruck dargestellt: `[[:digit:]]-feld[[:digit:]]{1,}-val` (in Worten: eine Ziffer, Zeichenkette `-feld`, mehrere Ziffern und die Zeichenkette `-val`).
- Die Teil-Bezeichnungen der Formular-Felder müssen aufeinanderfolgend sein: `feld0`, `feld1`, ..., `feld`n. Dadurch werden Verarbeitungsfehler und Inkonsistenzen der Datenhaltung verhindert.

Aus allen gesammelten Informationen werden der XML-Transformationsknoten mit folgenden SQL-Anweisungen generiert:

```
CREATE TABLE settings(name TEXT, type_id SMALLINT, value TEXT);
CREATE TABLE orders(order_id INTEGER, offer_id INTEGER, amount INTEGER,
               deleted SMALLINT DEFAULT 0, canceled SMALLINT DEFAULT 0);
CREATE TABLE orderdata(order_id INTEGER PRIMARY KEY, buyer_oid TEXT,
               deleted SMALLINT DEFAULT 0, datetime DATE, closed SMALLINT DEFAULT 0);
CREATE TABLE roles(role_id INTEGER PRIMARY KEY, desc TEXT);
CREATE TABLE types(type_id INTEGER PRIMARY KEY, desc TEXT);
CREATE TABLE users(oid TEXT PRIMARY KEY, role_id SMALLINT,
                  locked SMALLINT DEFAULT 0, feld0 TEXT);
INSERT INTO settings VALUES('feld0', 0, 'Name');
INSERT INTO settings VALUES('typ:feld0', 0, '0');
CREATE TABLE offers(offer_id INTEGER PRIMARY KEY, seller_oid TEXT,
                    activated SMALLINT DEFAULT 1, feld0 TEXT);
INSERT INTO settings VALUES('feld0', 1, 'Titel');
INSERT INTO settings VALUES('typ:feld0', 1, '0');
INSERT INTO types VALUES(0, 'Benutzerkonten, dynamische Felder');
INSERT INTO types VALUES(1, 'Angebote, dynamische Felder');
INSERT INTO roles VALUES(0, 'Kundendienst');
INSERT INTO roles VALUES(1, 'Verkaeufer');
INSERT INTO roles VALUES(2, 'Kaeufer');
INSERT INTO users (oid, role_id) VALUES ('A7c4309ba601ecb05f93b50fe14d89b41', 0);
```

Listing 5.1: Einrichtung der Datenbank

Damit wird die gesamte Datenbank generiert und der Benutzer, der die Erzeugungsanweisung abgeschickt hat, als Kundendienst-Mitarbeiter eingetragen.

Analog dazu erfolgen die Änderungen der dynamischen Felder, wobei Felder hinzugefügt und entfernt werden können: Zur Änderung einer Tabelle ist es in einer SQLite-Datenbank notwendig, eine temporäre neue Tabelle zu erzeugen, bestehende Daten zu kopieren, um danach die alte Tabelle zu löschen und eine komplett neue Tabelle anzulegen. SQLite stellt dazu ein Beispiel bereit (Listing 5.2):

```
BEGIN TRANSACTION;
CREATE TEMPORARY TABLE t1_backup(a,b);
INSERT INTO t1_backup SELECT a,b FROM t1;
DROP TABLE t1;
CREATE TABLE t1(a,b);
INSERT INTO t1 SELECT a,b FROM t1_backup;
DROP TABLE t1_backup;
COMMIT;
```

Listing 5.2: Beispiel zur Änderung von Tabellen

Die Verwendung von Transaktionen ist innerhalb von Askemos® nicht notwendig. Übermittelte SQL-Anweisungen werden intern mittels Transaktionen ausgeführt und bei Fehlschlag zurückgewiesen: Es erfolgt dann kein `COMMIT;` sondern ein `ROLLBACK;`.

Aufbauend auf der Generierung der Tabellen ist es nun möglich, alle weiteren Informationen zu sammeln und zu verarbeiten. In den weiteren Unterkapiteln werden Details der Verarbeitung vorgestellt.

5.2.2 Verarbeitung von Datenbank-Informationen

In der Verarbeitung innerhalb des Kerns von Shop-Systemen werden Informationen von der Datenbank abgefragt. Danach werden empfangene Informationen (Vektoren) zu einer von zwei XML-Strukturen verarbeitet: Informationen für eine Auflistung oder für ein Formular. Sie haben den folgenden Aufbau:

- Die Struktur zur Auflistung von Daten (Listing 5.3) enthält Aufzählungselemente `<item>`, die jedes für sich eine Auflistung darstellen. Eine Auflistung enthält eine Überschrift `<headline>`, Kopfzeilen `<header>` und Zeilen `<row>`, die die Informationen enthalten. Sind keine Informationen vorhanden, gibt es keine Zeilen:

```
<kss>
 <item>
  <headline>*text*</headline>
  <header>
   <entry>*text*</entry>
   <!-- [...] -->
   <entry>*text*</entry>
  </header>
  <row>
   <entry>*text*</entry>
   <!-- [...] -->
   <entry>*text*</entry>
  </row>
  <!-- [...] -->
  <row> <!-- [...] --> </row>
 </item>
 <!-- [...] -->
 <item> <!-- [...] --> </item>
</kss>
```

Listing 5.3: Struktur: Auflistung von Daten

- Die Struktur von Formular-Feldern für Daten (Listing 5.4) beinhaltet einen globalen XML-Knoten `<form>`. Dieser setzt sich aus Formular-Feld-Typen `<type>` zusammen, die eine Überschrift `<headline>` und Formular-Feld-Einträge `entry` (mit vorgeschriebenen Namen `<name>` und Wert `<value>`) enthalten. Dabei existiert immer mindestens ein Formular-Feld-Typ und ein -Eintrag:

```
<kss>
 <form>
  <type>
   <headline>*text*</headline>
   <entry>
    <name>*text*</name>
    <label>*text*</label>
    <value>*text*</value>
    <input_type>*text*</input_type>
   </entry>
   <!-- [...] -->
   <entry> <!-- [...] --> </entry>
  </type>
  <!-- [...] -->
  <type> <!-- [...] --> </type>
 </form>
</kss>
```

Listing 5.4: Struktur: Formular-Felder für Daten

5.2.3 Testen des Kerns von Shop-Systemen

Der Platz, der das XSL-Stylesheet des Kerns von Shop-Systemen enthält ist für eine eigenständige Verarbeitung vorgesehen. Folglich wurde ein Verwaltungsplatz (Admin Panel) entwickelt, welcher nicht nur zur Verwaltung, sondern auch zum Testen des Kerns von Shop-Systemen dient. Des Weiteren erleichtert der Quelltext des Verwaltungsplatzes später die Entwicklung von Shop-Systemen.

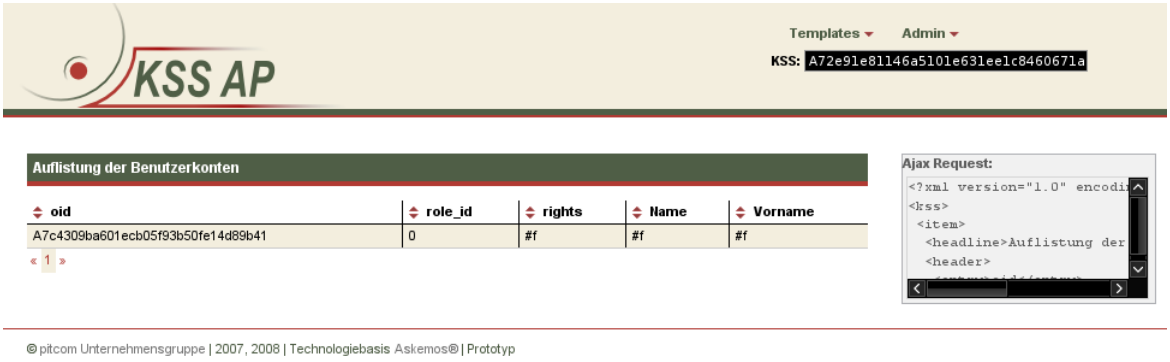

Abbildung 5.1: Kern von Shop-Systemen Verwaltungsoberfläche

Jede Verwaltungsaktion erfolgt asynchron: das Java-Script-Framework MooTools [14] und einige darauf basierende Klassen vereinfachen die Formulierung der Java-Script-Verarbeitung. Während der Verarbeitung einer Verwaltungsaktion wird ein Ajax-Request ausgeführt, entweder ein Abruf (Methode `GET`) oder ein Senden (Methode `POST`) von Informationen. Dabei werden im Bericht-Abschnitt Informationen zum Aktuellen Status des Ajax-Requests ausgegeben.

Nach einer Abfrage enthalten die Variablen des Ajax-Requests eine der beiden Strukturen: eine Auflistung (Listing 5.3) oder ein Formular (Listing 5.4). Anhand der Struktur wird eine passende Funktion aufgerufen und die Struktur entsprechend formatiert in den Hauptteil der Internetseite ausgegeben.

Eine Auflistungsstruktur wird zu einer Tabelle formatiert, die sortierbar und in Seiten eingeteilt ist. Eine Formularstruktur wird zu einem formatierten Formular umgewandelt. Beide Formatierungen basieren auf einer Internetseiten-Formatierung von bereits vorhandenen Projekten auf Basis von Askemos®.

5.3 Details zur Datenbankschnittstelle

In der Phase Programmentwurf (Kapitel 4.3) wurde bereits die `AskemosVfs`-Struktur beschrieben. Sie ist die Grundlage zur Kommunikation zwischen SQLite und Askemos, programmiert in C. Dazu zählt auch die Datei-Struktur `askemos_file`, für die alle vorgeschriebenen Ein- und Ausgabefunktionen definiert sind.

In den folgenden Abschnitten geht es um die Verarbeitung von Datei-Blöcken mithilfe von Askemos®. Dabei wurden die `xRead`- und `xWrite`-Funktionen so erweitert, dass neben der Verarbeitung mit dem Dateisystem auch die Datei-Blöcke mit Askemos® verwaltet werden.

Zu Beginn der Implementierung wurde die Schreib-Funktion erweitert, damit Datei-Blöcke zu einem Askemos®-Platz gelangen und für die Lesen-Funktion bereitstehen. Daher werden in folgenden Abschnitten auf gleiche Weise die Abläufe der Funktionen dargestellt.

5.3.1 Ablauf des Schreibens

```
static int askemosWrite(sqlite3_file *pFile,  const void *zBuf,
                        int iAmt, sqlite_int64 iOfst) {
  AskemosVfs *pA = ((askemos_file *)pFile)->pAskemosVfs;
  sqlite3_send_rs_intr_call2(pA->callback,
                             sqlite3_create_block_list(SQLITE_IO_TYPE_WRITE, zBuf,
                                                       iAmt, iOfst, NIL_OBJ),
                             pA->mb_write);
  return 0;
}
```

Listing 5.5: Funktion askemosWrite

Zur Verarbeitung wurden folgende Parameter übergeben:

- die Datei-Struktur mit allen Verarbeitungsfunktionen und weiteren Variablen
- ein Zeiger auf einen nur lesbaren Speicherbereich, der den Datei-Block enthält
- die Anzahl der zu schreibenden Bytes
- ein Offset: zur Position der Datei, ab der der Datei-Block zu schreiben ist

Bevor eine Liste der Daten und des Datei-Blockes an Askemos® übertragen wird, ist sie auf RScheme anzupassen und in ein Objekt umzuwandeln. Die Anzahl der zu schreibenden Bytes wird mittels eines Makros umgewandelt, wohingegen das Offset eine 64bit-Zahl enthalten kann und damit auf eine andere Weise in ein Objekt RSchemes gebracht wird.

Der Datei-Block muss zuvor in einen anderen Speicherbereich kopiert werden, damit die Daten des Datei-Blockes noch zur Verfügung stehen, wenn RScheme die Daten liest. Denn nachdem die `xWrite`-Funktion abgearbeitet ist, wird der genannte Speicherbereich freigegeben und RScheme könnte keine Daten mehr lesen.

Nachdem alle Vorbereitungen abgeschlossen sind, wird mittels RScheme-Befehlen eine Liste erzeugt. Folglich ensteht ein Objekt mit einer Liste, die alle vorbereiteten Daten enthält und die zusammen mit einem Callback-Objekt über den Unterbrechungsaufruf von RScheme an Askemos® übergeben wird.

Beim Aufrufen von SQLite werden eine Mailbox und ein Callback-Objekt in die `AskemosVfs`-Struktur übertragen. Das Callback-Objekt enthält eine anonyme Lambda-Funktion zum Senden des übergebenen Objektes an eine übergebene Mailbox.

Ein Thread, der zu Beginn des SQLite-Aufrufes gestartet wurde, empfängt die Listen von der Mailbox und nimmt folgende Verarbeitung vor; dabei existiert eine "Schreib-Liste", in der zu kleine Datei-Blöcke gespeichert werden:

1. In einer Schleife empfängt der Thread eine Liste und legt sie in einer "Kontroll-Liste" ab, die im nächsten Schleifendurchlauf verarbeitet wird.
2. Sobald die Kontroll-Liste eine Liste enthält, wird anhand des ersten Listenelements entschieden, ob eine schreibende Verarbeitung oder eine abschließende Sendung des Rest-Blockes stattfindet.

Die schreibende Verarbeitung ermittelt zunächst, ob die Gesamtgröße der Datei-Blöcke einer Mindestgröße (z.B. 512 Byte) entsprechen. Dabei werden Datei-Blöcke aus der Schreib-Liste und aus der gerade übergebenen Liste betrachtet. Wenn die Bedingung erfüllt ist, wird aus den Datei-Blöcken ein Gesamtblock gebildet, entsprechend der Mindestgröße geteilt und zusammen mit einer Block-Nummer in einer Liste gespeichert. Die Blocknummer wird dabei anhand des Offsets bestimmt. Die Liste wird

danach einer Funktion übergeben, die die Datei-Fragmente an den Platz zurückschickt, der den SQLite-Aufruf gestartet hat.

Wenn die Gesamtgröße der Datei-Blöcke zu klein ist, wird der empfangene Datei-Block zur Schreib-Liste hinzugefügt. Am Ende der Verarbeitung müssen übrig gebliebene Daten, die sich noch in der Schreib-Liste befinden, an den Platz geschickt werden. Letzteres wurde in den Funktionen `xSync` und `xClose` implementiert und verhält sich analog zum Schreiben, wobei jedoch nur noch der Rest-Block an den Platz geschickt wird.

5.3.2 Ablauf des Lesens

```
static int askemosRead(sqlite3_file *pFile, void *zBuf,
                       int iAmt, sqlite_int64 iOfst){
  AskemosVfs *pA = ((askemos_file *)pFile)->pAskemosVfs;
  struct askemos_read_bag *arb = malloc(sizeof(struct askemos_read_bag));
  arb->zBuf = zBuf; arb->iAmt = iAmt; arb->mb_read = rs_make_dequeue();
  pthread_cond_init(&arb->cond_read, NULL);
  rscheme_pthread_intr_call(pA->callback,
         sqlite3_create_block_list(SQLITE_IO_TYPE_READ, NULL, iAmt, iOfst,
                                   RAW_PTR_TO_OBJ(&arb->cond_read), arb->mb_read),
                           pA->mailbox);
  start_asynchronous_request(askemos_read_work, arb, pA->cb_return);
  return 0;
}
```

Listing 5.6: Funktion askemosRead

Zur Verarbeitung wurden folgende Parameter übergeben:

- die Datei-Struktur mit allen Verarbeitungsfunktionen und weiteren Variablen
- ein Zeiger auf einen schreibbaren Speicherbereich für den Datei-Block
- die Anzahl der zu lesenden Bytes
- ein Offset: zur Position der Datei, ab der der Datei-Block zu schreiben ist

Beim Aufrufen der Unterbrechungsanforderung von RScheme muss ein globales Mutex gesperrt werden, damit keine anderen Threads RScheme blockieren können. Beide Funktionen `start_asynchronous_request` und `rscheme_pthread_intr_call` sperren das globale Mutex und führen den Unterbrechungsaufruf durch. Deshalb ist es nicht

möglich, die Informationsliste des benötigten Blockes innerhalb einer Funktion zu senden, die an die Funktion `start_asynchronous_request` übergeben wird. Sie sorgt dafür, dass die übergebene Funktion in einem Thread ausgeführt und dessen Rückgabewert an RScheme geliefert wird. Aufgrund dessen werden zuerst die Informationen des geforderten Dateibereiches an RScheme gesendet.

Innerhalb der Verarbeitungsfunktion `askemos_read_work` werden Thread-Synchronisationsmittel eingesetzt, um auf den Block in der Nachrichtenschlange zu warten. Vorerst sind bei jedem Lesen ein Mutex und eine Condition-Variable zu initialisieren, das Mutex zu sperren und auf das Setzen der Condition-Variable zu warten.

Eine Funktion in Askemos® holt dazu Blöcke vom Platz, setzt sie zusammen und kürzt den Puffer entsprechend der Anzahl zu lesender Bytes. Der fertige Puffer wird an eine C-Funktion zurückgegeben, welche den Puffer in die Nachrichtenschlange einfügt und welche die Condition-Variable setzt.

Als Letztes muss der fertige Puffer aus der Nachrichtenschlange gelesen und in den bereitgestellten Speicherbereich kopiert werden. Damit hat SQLite den angeforderten Datei-Bereich im Speicher und kann damit arbeiten.

5.3.3 Weitere Besonderheiten

Des Weiteren wurde eine globale Variable zum Senden der Datei-Blöcke eingeführt, um Fehler während des Übersetzens von Askemos® zu vermeiden. Sie wird beim Start von Askemos® initialisiert, wobei eine anonyme Funktion bereitgestellt wird, die ein einfaches XML-formatiertes Formular annimmt und daraus eine Nachricht generiert und verschickt.

Nach dem Einbinden der Thread-Funktionalität für die Funktion `xRead` ist ein Übersetzungsfehler aufgetaucht. In der Header-Datei `sched.h`, benötigt von `pthread.h`, existiert eine Funktion namens `clone` wie auch in der Header-Datei `rscheme/scheme.h`. Beide zusammen können nicht in derselben C-Datei verwendet werden. Deshalb wurden alle Aufrufe von RScheme-Befehlen in extra Funktionsaufrufe geschrieben, die sich in der C-Datei `sqglue.c` befinden. In `os_askemos.c` sind sie als `extern` angegeben.

6 Zusammenfassung

6.1 Bewertung

Die derzeit implementierte Version des Kerns von Shop-Systemen besitzt alle geforderten Merkmale. Sie ermöglicht Käufern, Bestellungen abzuwickeln und Verkäufern Angebote bereitzustellen. Ein Shop-System ist ein Platz, mit dem alle Aufgaben erledigt werden. Die Oberfläche eines Shop-Systems muss in einem anderen Platz eingebunden werden, wofür die Verwaltungsoberfläche ein Beispiel ist.

Die derzeit implementierte Version der SQLite-VFS-Schnittstelle besitzt alle geforderten Merkmale. Sie ermöglicht lesende und schreibende Verwaltungsvorgänge. Eine Datenbank-Datei kann somit in einem WebDAV verwaltet werden. Die Aufgabe der Diplomarbeit ist damit erfüllt.

6.2 Kritik und Erfahrungen

Trotz umfangreicher Vorkenntnisse mussten einige Abschnitte mehrfach überarbeitet werden. Die geringe Dokumentation von Askemos® hat die Implementierung an vielen Stellen erschwert und es erforderlich gemacht, sich direkt mit dem Quelltext von Askemos® auseinanderzusetzen.

Allerdings hat der Kontakt mit dem Entwickler von Askemos® an vielen Stellen neue Ideen, Implementierungsmöglichkeiten und tiefgreifendere Kenntnisse über die Funktionsweise von Askemos® hervorgebracht.

Anfängliche Schwierigkeiten mit SQLite konnten beigelegt werden, als der Quelltext von SQLite, dessen VFS-Strukturen und Test-Programme untersucht wurden. Auf diese Weise stellte sich heraus, dass in einem Test-Programm die benötigte Verarbeitungsweise, die Ummantelung einer bestehenden VFS-Struktur, vorgestellt wurde.

Des Weiteren war es auch nötig, viele Scheme-Funktionen auf der Anwendungsebene zu testen, weil die Stelle der Implementierung so verschachtelt ist, dass eine komplette Übersetzung von Askemos® notwendig ist und damit die ganze Implementierung verlangsamt wurde. Viele syntaktische Fehler konnten nicht angezeigt werden, da beispielsweise einige Funktionen durch Threads verarbeitet werden.

6.3 Mögliche Weiterentwicklung

Weiterentwicklungen des Kerns von Shop-System

Folgende Weiterentwicklungen sind für Shop-Systeme und deren Kerne denkbar:

- Vorlagen für die dynamischen Felder (Benutzerkonten, Angebote)
- Filterung von Einträgen (Angebote, Bestellungen, u.a.) basierend auf MooTools
- Aufbau von CSS- und HTML-Vorlagen für Shop-Systeme
- E-Mail-Benachrichtigungen bei Aktionen des KSS (z.B. Bestellungsbuchung).

Eine weitere Weiterentwicklung wäre die Komprimierung von Daten durch Askemos® unter Verwendung von Bibliotheken, beispielsweise 7zip.

Weiterentwicklungen der SQLite-Schnittstelle

Eine Weiterentwicklung der SQLite-Schnittstelle wäre mithilfe einer Erweiterung namens “Asynchroner IO-Modus” denkbar. Damit werden alle lesenden und schreibenden Verwaltungsaufrufe durch extra Threads ausgeführt und ermöglichen eine nahezu parallele Ausführung, soweit die beteiligten Komponenten SQLite und Askemos® dies zulassen. Weiterhin sind sehr große Testfälle auszuführen, um mögliche Fehler in der SQLite-Schnittstelle oder Askemos® feststellen zu können und um eine optimale Block-Größe einstellen zu können.

Literaturverzeichnis

Internetquellen

[1] Wittenberger, Jörg F. <Joerg.Wittenberger@softeyes.net>:
Askemos - eine verteilte Umgebung.
URL: <http://www.softeyes.net/A5c54bc5504e7a115d1d9cba3e29cc8b7>,
verfügbar am 01.06.2009

[2] Dusseault, L.M. <ietf-ipr@ietf.org>:
HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV).
URL: <http://www.ietf.org/rfc/rfc4918.txt>,
verfügbar am 25.06.2009

[3] Wittenberger, Jörg F. <Joerg.Wittenberger@softeyes.net>:
Askemos Protection.
URL: <http://www.askemos.org/index.html/AskemosProtection>,
verfügbar am 01.06.2009

[4] Wittenberger, Jörg F. <Joerg.Wittenberger@softeyes.net>:
Askemos Protection 06 (High Level Design).
URL: <http://www.askemos.org/index.html/AskemosProtection06>,
verfügbar am 01.06.2009

[5] Hipp, D. Richard <drh@hwaci.com>:
Available Documentation.
URL: <http://sqlite.org/docs.html>,
verfügbar am 07.06.2009

[6] Hipp, D. Richard <drh@hwaci.com>:

About SQLite.

URL: <http://sqlite.org/about.html>,

verfügbar am 07.06.2009

[7] Hipp, D. Richard <drh@hwaci.com>:

About SQLite.

URL: <http://sqlite.org/testing.html>,

verfügbar am 25.08.2009

[8] Hipp, D. Richard <drh@hwaci.com>:

Distinctive Features of SQLite.

URL: <http://sqlite.org/different.html>,

verfügbar am 07.06.2009

[9] Wittenberger, Jörg F. <Joerg.Wittenberger@softeyes.net>:

SQL-Beispiel Buchhaltung.

URL: <http://login.softeyes.net/jfw/fiXml-dev/SQL-Beispiel%20Buchhaltung.xml>,

verfügbar am 01.06.2009

[10] Clark, James <jjc@jclark.com>:

XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16. Nov. 1999.

URL: <http://www.w3.org/TR/xslt>,

verfügbar am 01.06.2009

[11] Hipp, D. Richard <drh@hwaci.com>:

OS Interface Object.

URL: <http://www.sqlite.org/c3ref/vfs.html>,

verfügbar am 07.06.2009

[12] Hipp, D. Richard <drh@hwaci.com>:

An Asynchronous I/O Module For SQLite.

URL: <http://sqlite.org/asyncvfs.html>,

verfügbar am 07.06.2009

[13] Schwichtenberg, Helmut <schwicht@math.lmu.de>:

Nichtnumerisches Programmieren.

URL: <http://www.mathematik.uni-muenchen.de/~schwicht/lectures/scheme/schemews04>,

verfügbar am 19.07.2009

[14] Proietti Valerio <>:

MooTools – a compact javascript framework.

URL: <http://www.mootools.net>,

verfügbar am 25.08.2009

Bücher

[15] Schneider, Uwe; Werner, Dieter:

Taschenbuch der Informatik. - 5.Aufl. - Leipzig: Fachbuchverlag Leipzig, 2004

Erklärung

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

Bearbeitungsort, Datum	Unterschrift