Berücksichtigung von Berechtigungskonzepten bei der Spezifikation von Fachkomponenten

Holger Jaekel, Thorsten Teschke

OFFIS, Escherweg 2, 26121 Oldenburg, Deutschland, Tel.: (04 41) 97 22 - 1 25, Fax: - 1 02, E-Mail: {holger.jaekel | thorsten.teschke}@offis.de. URL: http://www.offis.de

Zusammenfassung. Eine umfassende Beschreibung einer Softwarekomponente ist eine entscheidende Voraussetzung für ihre erfolgreiche Wiederverwendung. Dabei ist auch die Spezifikation von Berechtigungskonzepten unter Berücksichtigung organisatorischer Aspekte für die Bewertung und Auswahl von Fachkomponenten interessant. In diesem Beitrag stellen wir eine Erweiterung des vom GI-Arbeitskreis 5.10.3 erarbeiteten Spezifikationsrahmens um ein Berechtigungskonzept vor und beschreiben Erfahrungen, die damit im Rahmen einer Fallstudie gemacht worden sind.

Schlüsselworte: Komponente, Fachkomponente, betriebliches Anwendungssystem, Spezifikation, Berechtigungskonzept, Fallstudie

1 Einleitung

Effiziente Wiederverwendungsprozesse im Rahmen komponentenbasierter Softwareentwicklung setzen die Existenz einheitlicher, umfassender und eindeutiger Beschreibungen von (Fach-)Komponenten voraus. Erst diese Beschreibungen ermöglichen es, geeignete (Fach-)Komponenten in einem Komponenten-Repository finden und bewerten zu können.

Einen umfassenden Ansatz zur Spezifikation von Fachkomponenten stellt das Memorandum des Arbeitskreises 5.10.3 „Komponentenorientierte betriebliche Anwendungssysteme" [ABC+02] dar. Dieses Memorandum schlägt die Spezifikation von Fachkomponenten auf sieben Beschreibungsebenen vor, die sich durch ihre inhaltliche Ausrichtung und formale Ausgestaltung an verschiedene Adressaten wie z. B. Softwareentwickler, Fachexperten und Vertriebsmitarbeiter richten. Die Beschreibungsebenen spezifizieren im Einzelnen die Schnittstelle einer Komponente, das Verhalten der von ihr angebotenen Dienste, die Abstimmung zwischen Diensten dieser bzw. dieser und anderen Komponenten, Qualitätsmerkmale, betriebliche Aufgaben, für die die Komponente Lösungen anbietet, sowie Marketing-Aspekte. Um die Komponentenbeschreibung semantisch zu fundieren, wird die in der Beschreibung verwendete Terminologie auf einer speziellen Beschreibungsebene definiert.

Aktuelle Komponentenmodelle wie Enterprise JavaBeans (EJB) [DYK01] und das CORBA Component Model (CCM) [OMG01] berücksichtigen neben den durch das Memorandum bereits abgedeckten Aspekten auch organisatorische Fragestellungen des Einsatzes von Komponenten: sie gestatten die Deklaration von Rollen, die Benutzern einer Komponente bei deren Einsatz zugewiesen werden können, und die Zuordnung dieser Rollen zu den Diensten einer Komponente im Sinne eines Berechtigungskonzepts. So lässt sich beispielsweise festlegen, dass gewisse Methoden von einem beliebigen Benutzer, andere aber nur von einem Systemadministrator verwendet werden dürfen.

In diesem Beitrag schlagen wir die Berücksichtigung organisatorischer Aspekte durch die Spezifikation von Rollen einerseits und rollenbezogenen Berechtigungen zur Benutzung von Diensten andererseits als Ergänzung des Memorandums zur Spezifikation von Fachkomponenten vor. Diese Ergänzung basiert auf einem normsprachlichen Ansatz und lässt sich in die bereits definierten Terminologie- und Aufgabenebenen integrieren. Die Anwendung des Ansatzes wird in einer kleinen Fallstudie skizziert.

2 Spezifikation organisatorischer Aspekte von Fachkomponenten

Bislang wird bei der auf fachliche Aspekte ausgerichteten Spezifikation von Komponenten insbesondere auf folgende Fragestellungen eingegangen:

- Welche Dienste werden von der Komponente angeboten?
- Welche Vorbedingungen müssen erfüllt sein, damit ein Dienst erfolgreich ausgeführt werden kann?
- Was sind die fachlichen Leistungen eines Dienstes?
- Welche Input-Faktoren erwartet ein Dienst?
- Welche kausalen und temporalen Abhängigkeiten bestehen zu anderen Diensten?

Eine wichtige Frage, die bislang nicht gestellt wird, ist die nach der Berechtigung, einen Dienst zu nutzen: *Wer darf den Dienst nutzen?* Diese Frage wird von aktuellen Komponentenmodellen wie EJB und CCM beantwortet, indem den Methoden einer Komponente in ihrer Komponentenbeschreibung (*Deployment Descriptor* bzw. *CORBA Component Descriptor*) Rollenbezeichner zugeordnet werden können. Die Benutzung einer Methode setzt dann voraus, dass der Benutzer eine entsprechende Rolle innehat. Auf Grundlage dieses Berechtigungskonzepts kann ein Application Server den Zugriff nicht autorisierter Benutzer auf zugriffsbeschränkte Methoden (z.B. zur Einrichtung eines Kreditlimits bei einer Komponente aus dem Bankwesen) unterbinden.

In den folgenden Abschnitten wird beschrieben, wie derartige organisatorische Aspekte in fachliche Komponentenbeschreibungen auf Grundlage eines normsprachlichen Ansatzes [Ort97] eingeführt werden können (vgl. hierzu auch [Tes02]).

2.1 Definition eines Rollenmodells auf der Terminologieebene

In [ABC+02] wird vorgeschlagen, auf der Terminologieebene ein Lexikon aller Begriffe, die für die Spezifikation der Komponente von Nutzen sind, mitsamt ihrer Definitionen zu sammeln. Als Notationsvorschlag wird (basierend auf einer Normsprache) u. a. die explizite Definition von Begriffen („eine Bilanz ist eine Gegenüberstellung der Aktiva und Passiva einer Unternehmung zu einem bestimmten Zeitpunkt“) sowie die Verwendung von Prädikatorenregeln („$x \in$ Bilanz $\Rightarrow x \in$ Jahresabschluss“) vorgeschlagen.

Die beispielhaft genannte Prädikatorenregel nutzt eine Form der Abstraktion, bei der ein Gegenstand x einem Begriff „Bilanz“ untergeordnet wird (Subsumtion) [Ort97]. Neben diesem

Griff zur Abstraktion auf der Terminologieebene sollen abstraktive Beziehungen laut Memorandum auch im Rahmen der Aufgabenebene genutzt werden. Wir schlagen vor, die Spezifikation abstraktiver (und kompositiver) Beziehungen auf die Terminologieebene zu beschränken, weil durch sie keine Aufgaben beschrieben sondern die dafür verwendeten Begriffe genauer geklärt werden. (Zweistellige) abstraktive Beziehungen können durch den folgenden Satzbauplan ausgedrückt werden:

$$[\ \mathtt{N}_1 \mid \sqsubset \mid \mathtt{N}_2\ ]$$

Dabei stellen $\mathtt{N}_1$ und $\mathtt{N}_2$ Nominatoren und $\sqsubset$ die Kopula „ist" dar. Ein Beispiel für eine solche Abstraktionsaussage ist der Satz „Bilanz $\sqsubset$ Jahresabschluss" (gesprochen „[eine] Bilanz ist [ein] Jahresabschluss").

Dieser elementare Satzbauplan kann auch für die Abbildung eines hierarchischen Rollenmodells genutzt werden. Dabei werden Rollen, deren genaue Bedeutungen – wie im Memorandum gefordert – durch explizite Definition spezifiziert werden, mittels Abstraktionsbeziehungen in einer Generalisierungs-/Spezialisierungshierarchie angeordnet. Soll beispielsweise modelliert werden, dass Buchhalter und Sachbearbeiter Büroangestellte sind, und dass darüber hinaus Bilanzbuchhalter spezialisierte Buchhalter sind, so können die folgenden drei Aussagen genutzt werden:

$$\begin{array}{c}\mathtt{Buchhalter} \sqsubset \mathtt{Büroangestellter}\\ \mathtt{Sachbearbeiter} \sqsubset \mathtt{Büroangestellter}\\ \mathtt{Bilanzbuchhalter} \sqsubset \mathtt{Buchhalter}\end{array}$$

Für den Aufbau eines Lexikons gilt es also, neben expliziten Definition von Begriffen auch abstraktive Beziehungen zwischen diesen Begriffen zu pflegen. Hier bietet sich die Nutzung von Taxonomien für Konzepte (Handlungsträger bzw. Rollen und Gegenstände betrieblichen Handelns) und Aktivitäten (betriebliche Handlungen) zu pflegen. Um Rollen innerhalb der Taxonomie für Konzepte von den Geschäftsobjekten deutlicher zu unterscheiden, haben wir alle Rollen unter dem abstrakten Begriff „Organisationseinheit" subsumiert. Abbildung 1 deutet Aufbau und Inhalt solcher Taxonomien an. Dabei bedeutet eine Vater-Sohn-Beziehung zwischen zwei Knoten x und y im Baum, dass $y \sqsubset x$ gilt.

2.2 Vergabe von Berechtigungen auf der Aufgabenebene

Für die fachliche Beschreibung der betrieblichen Aufgaben, die durch eine Fachkomponente unterstützt werden, sieht das Memorandum normsprachliche Aussagen auf der Aufgabenebene vor. Diese normsprachlichen Aussagen sind durch „Instanziierung" geeigneter Satzbaupläne mit Hilfe der auf der Terminologieebene definierten Begriffen aufzubauen. Ein generischer Satzbauplan zur Beschreibung der durch einen Dienst einer Komponente unterstützen Aufgabe könnte wie folgt aussehen:

$$[\ \mathtt{N} \mid \pi \mid \mathtt{P} \mid \mathtt{O}_1 \mid \mathtt{O}_2 \mid \ldots\ ]$$

Dabei bezeichnet $\mathtt{N}$ einen Nominator in der Stellung des Subjekts, π repräsentiert die Kopula „tut", $\mathtt{P}$ ist das Prädikat, und $\mathtt{O}_\mathtt{i}$ bezeichnen Nominatoren in der Stellung des direkten Objekts $(\mathtt{i} = 1)$ bzw. indirekter Objekte $(\mathtt{i} > 1)$.

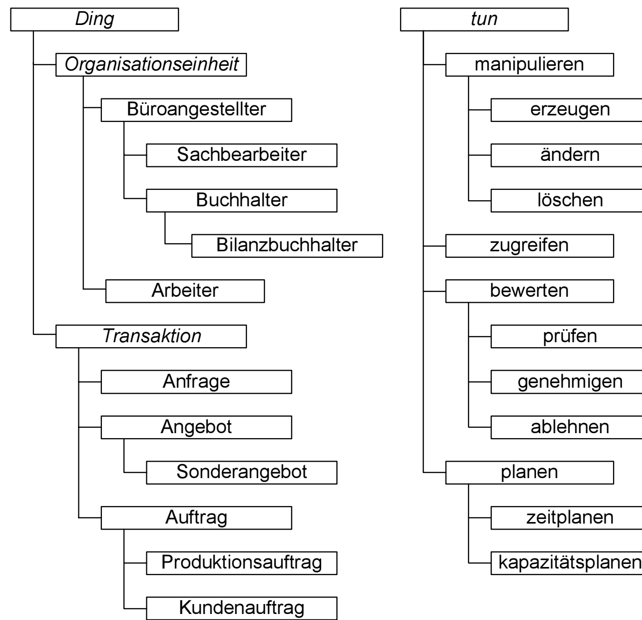

Abbildung 1: Taxonomien von Konzepten und Aktivitäten

Für die inhaltliche Belegung der Position `N` bieten sich verschiedene Alternativen an. So könnte die Komponente, die den Dienst anbietet, als Subjekt betrachtet werden („Buchhaltungskomponente tut buchen Zahlungseingang“). Diese Variante bringt aber nur in beschränktem Maße zusätzliche Informationen, da bereits bekannt ist, welche Komponente den betrachteten Dienst anbietet. Alternativ kann auch der Aufrufer des angesprochenen Dienstes, also z. B. eine andere Komponente oder ein Objekt, als Subjekt angesehen werden („Auftragsabwicklungskomponente tut buchen Forderung“). Allerdings ist dieser im Allgemeinen nicht zur Zeit der Erstellung einer Komponentenbeschreibung bekannt. Wir schlagen daher vor, mit der Subjektstellung Informationen über die Berechtigung, den Dienst zu nutzen, zu verknüpfen. Das Subjekt ist dabei einer der auf der Terminologieebene definierten Rollenbezeichner und bestimmt damit die minimal erforderliche Berechtigungsstufe, die ein Nutzer der Komponente haben muss, um den Dienst nutzen zu können. Als Beispiele für diese Interpretation der Position des Subjekts sollen die folgenden normsprachlichen Aussagen dienen:

```
[Ein] Buchhalter tut buchen [einen] Zahlungseingang.
[Ein] Bilanzbuchhalter tut erstellen [eine] Bilanz.
```

Eine Methode, der als normsprachliche Beschreibung die erste Aussage zugeordnet ist, dient der Buchung eines Zahlungseingangs. Für die Benutzung dieser Methode ist es erforderlich, dass der Nutzer mindestens die Rolle eines Buchhalters bekleidet (gemäß Rollenmodell in Abbildung 1 sind also auch Bilanzbuchhalter zur Nutzung der Methode berechtigt). Es ist Aufgabe der Ausführungsplattform sicherzustellen, dass der Versuch eines Sachbearbeiters, diese Methode zu nutzen, fehlschlägt. Die zweite Aussage erwartet analog zu diesem Fall, dass ein Nutzer die Rolle „Bilanzbuchhalter“ innehat.

3 Fallstudie: The Duke's Bank Application

In dieser Fallstudie soll die zusammen mit [BGH+02, S. 391 ff] veröffentlichte Duke's Bank Application auf den im Memorandum [ABC+02] vorgeschlagenen Ebenen spezifiziert werden. Aufgrund der Ausrichtung dieses Beitrags beschränken wir uns dabei auf die Schnittstellenebene, die Terminologieebene und die Aufgabenebene. Die Duke's Bank Application ist mit drei Entity Beans und drei Session Beans, die von einigen Servlets angesteuert werden, eine recht überschaubare Anwendung, die einen Dienst zum Online-Banking implementiert, für den verschiedene Benutzerrollen festgelegt sind.

Die Enterprise JavaBeans-Spezifikation unterscheidet die deklarative Sicherheit von der programmgesteuerten Sicherheit. Während die programmgesteuerte Sicherheit von der Anwendung selbst überwacht wird, wird die hier interessante deklarative Sicherheit vom Anwendungsentwickler im sogenannten Deployment Descriptor festgelegt und von der Ablaufumgebung der Enterprise Bean überwacht. Im Deployment Descriptor der Bean können *Security Roles* definiert werden, denen anschließend Rechte (auf der Ebene von Methoden) zugeordnet werden können. In Listing 1 ist ein Ausschnitt aus einem Deployment Descriptor dargestellt, der die Security Roles „BankCustomer" und „BankAdmin" definiert und festlegt, dass die Methode `findByPrimaryKey(String)` von jedem Benutzer aufgerufen werden kann, die Methode `withdraw(BigDecimal, String, String)` jedoch nur von Inhabern der Rolle „BankCustomer". Beim Deployment der Bean werden diese Rollen auf Identitäten in der Ablaufumgebung abgebildet.

In den folgenden Abschnitten werden einige Aspekte der Fallstudie diskutiert. Der formale Teil der Spezifikation befindet sich aufgrund seiner Länge im Anhang A dieses Beitrages.

3.1 Schnittstellenebene

Auf der Schnittstellenebene sollen auf einer technischen Ebene die von der Komponente angebotenen Dienste beschrieben werden. Als primäre Notation ist hierfür in [ABC+02] die Interface Definition Language (IDL) vorgesehen. In diesem Fallbeispiel werden betriebliche Dienste zur Verwaltung von Bankkonten von Session und Entity Beans angeboten. Jede Bean bietet dabei zwei Schnittstellen an: Das Home-Interface enthält die Lebenszyklusmethoden, das Object-Interface die eigentlichen Geschäftsmethoden der Komponente. Um die Zusammengehörigkeit dieser beiden Interfaces auszudrücken, wurden diese gemeinsam in einem `module` gruppiert. In einem weiteren Modul `Extern` wurden extern benötigte Dienste (Hilfsklassen, die nicht Teil der Komponenten sind und verwendete Dienste der Java-Klassenbibliothek) zusammengefasst.

Weiterhin soll auf der Schnittstellenebene die Angabe korrespondierender Begriffe und Typen bzw. Aufgaben und Dienste erfolgen. Diese Aufstellung ist von uns um die Angabe von Berechtigungen erweitert worden (siehe Abschnitt 2.2 sowie Tabellen 2, 3, 4 und 5). Da die in diesen Aufstellungen zusammengefassten Informationen einen starken Bezug zu den betrieblichen Aufgaben einer Fachkomponente aufweisen, ordnen wir sie der Aufgabenebene anstelle der Schnittstellenebene zu.

3.2 Terminologieebene

Auf der Terminologieebene wird neben den Erklärungen der wichtigsten Begriffe der Fachkomponenten eine Taxonomie für diese Begriffe aufgebaut. In dieser Taxonomie werden, wie

Listing 1: Auszug aus dem Deployment Descriptor von TxEJB

```
<assembly−descriptor>
  <security−role>
    <role−name>BankCustomer</role−name>
  </security−role>
  <security−role>
    <role−name>BankAdmin</role−name>
  </security−role>
  <method−permission>
    <unchecked />
    <method>
      <ejb−name>TxEJB</ejb−name>
      <method−intf>Home</method−intf>
      <method−name>findByPrimaryKey</method−name>
      <method−params>
        <method−param>java.lang.String</method−param>
      </method−params>
    </method>
  </method−permission>
    [. . . ]
    <method−permission>
      <role−name>BankCustomer</role−name>
      <method>
        <ejb−name>TxControllerEJB</ejb−name>
        <method−intf>Remote</method−intf>
        <method−name>withdraw</method−name>
        <method−params>
          <method−param>java.math.BigDecimal</method−param>
          <method−param>java.lang.String</method−param>
          <method−param>java.lang.String</method−param>
        </method−params>
      </method>
    </method−permission>
    [. . . ]
  </assembly−descriptor>
```

in Abschnitt 2.1 vorgeschlagen, nicht nur Gegenstände betrieblichen Handelns, sondern auch Handlungsträger bzw. Rollen, eingeordnet. Die Duke's Bank Application definiert die Rollen „Bankadministrator“ und „Kunde“, die unter dem Begriff „Organisationseinheit“ subsumiert werden (vgl. Abbildung 2). Bei der anschließend angedeuteten Definition der Begriffe kann hier auf die Angabe von Prädikatoren verzichtet werden, da diese bereits implizit durch die Taxonomie gegeben sind.

3.3 Aufgabenebene

Die Spezifikation auf der Aufgabenebene soll alle betrieblichen Aufgaben, die von der Komponente angeboten werden, auflisten. Wie bereits in Abschnitt 3.1 begründet, verwenden wir dafür die vom Memorandum eigentlich für die Schnittstellenebene vorgesehene tabellenartige Zuordnung von angebotenen Diensten der Komponente zu den betrieblichen Aufgaben (siehe Tabellen 2, 3, 4 und 5). Dabei erweitern wir die Beschreibung der betrieblichen Aufgaben um die in Abschnitt 2.2 vorgestellte Vergabe von Berechtigungen. Falls der betriebliche Dienst von

jedem verwendet werden kann, ist das Subjekt in der entsprechenden normsprachlichen Aussage „Organisationseinheit“, andernfalls wird dort die entsprechende Rolle eingesetzt.

4 Zusammenfassung

Weit verbreitete Komponentenmodelle wie CCM oder EJB erlauben die Definition von Berechtigungskonzepten über ein Rollenkonzept. Beim betrieblichen Einsatz der Komponente („Deployment“) werden diesen Rollen dann den entsprechenden Organisationseinheiten im Unternehmen zugeordnet. In dem vom GI-Arbeitskreis 5.10.3 vorgeschlagenen Spezifikationsrahmen werden solche Berechtigungskonzepte bislang noch nicht berücksichtigt. In diesem Beitrag haben wir einen Vorschlag zur Repräsentation von hierarchisch strukturierten Rollenmodellen auf der Terminologieebene und einen Ansatz zur Repräsentation von Berechtigungen auf der Aufgabenebene mittels normsprachlicher Aussagen als Erweiterung des bisherigen Spezifikationsrahmens vorgestellt.

In der anschließenden Fallstudie haben wir festgestellt, dass sich die vorgeschlagene Darstellung von Berechtigungskonzepten für die gewählte Beispielanwendung umsetzen lässt. Dabei ist jedoch aufgefallen, dass einzelne Zuordnungen von zu spezifizierenden Sachverhalten zu den Beschreibungsebenen noch einmal überdacht werden sollten. So werden abstraktive Beziehungen zwischen Begriffen sowohl auf der Terminologieebene als auch auf der Aufagabenebene spezifiziert. Die Angabe korrespondierender Begriffe und Typen bzw. Aufgaben und Dienste könnte auch auf der Aufgabenebene statt auf der Schnittstellenebene stattfinden.

Literatur

[ABC+02] J. Ackermann, F. Brinkop, S. Conrad, P. Fettke, A. Frick, E. Glistau, H. Jaekel, O. Kotlar, P. Loos, H. Mrech, E. Ortner, U. Raape, S. Overhage, S. Sahm, A. Schmietendorf, T. Teschke, and K. Turowski. Vereinheitlichte spezifikation von fachkomponenten, Februar 2002. Memorandum des Arbeitskreises 5.10.3 Komponentenorientierte betriebliche Anwendungssysteme.

[BGH+02] Stephanie Bodoff, Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, and Beth Stearns. *The J2EE™ Tutorial.* Addison-Wesley, 2002.

[DYK01] L. G. DeMichiel, L. Ü. Yalçinalp, and S. Krishnan. *Enterprise JavaBeans Specification, Version 2.0.* Sun Microsystems, 2001.

[OMG01] Object Management Group. *CORBA 3.0 New Components Chapters, Proposed Available Specification 1.0 of the CORBA Component Model, OMG document ptc/01-11-03*, 2001.

[Ort97] Erich Ortner. *Methodenneutraler Fachentwurf.* Wirtschaftsinformatik. B. G. Teubner Verlag, Stuttgart, Leipzig, 1997.

[Tes02] Thorsten Teschke. Ontological intermediation between business process models and software components. In H.-M. Haav and A. Kalja, editors, *Databases and Information Systems, Proceedings of Fifth International Baltic Conference Baltic DB&IS'2002*, volume 1, Tallinn / Estland, 2002.

A Spezifikation der Duke's Bank Application

A.1 Schnittstellenebene

A.1.1 Session Beans

```
module CustomerControllerBean {
    interface CustomerControllerHome {
        exception CreateException ();
        CustomerController create () raises ( CreateException );
    }

    interface CustomerController {
        exception InvalidParameterException ();
        exception CustomerNotFoundException();

        string createCustomer(in string lastName, in string firstName , in string middleInitial ,
                              in string street , in string city , in string state , in string zip,
                              in string phone, in string email ) raises ( InvalidParameterException );
        void removeCustomer(in string customerId)
            raises ( InvalidParameterException , CustomerNotFoundException);
        CustomerBean:CustomerList getCustomersOfAccount(in string accountId )
            raises ( InvalidParameterException , CustomerNotFoundException);
        CustomerDetails getDetails (in string customerId)
            raises ( InvalidParameterException , CustomerNotFoundException);
        CustomerBean:CustomerList getCustomersOfLastName(in string lastName)
            raises ( InvalidParameterException );
        void setName(in string lastName, in string firstName , in string middleInitial ,
            in string customerId ) raises ( InvalidParameterException , CustomerNotFoundException);
        void setAddress(in string street , in string city , in string state , in string zip ,
                        in string phone, in string email , in string customerId)
            raises ( InvalidParameterException , CustomerNotFoundException);
    }
}

module AccountControllerBean {
    interface AccountControllerHome {
        exception CreateException ();
        AccountController create () raises ( CreateException );
    }

    interface AccountController {
        exception IllegalAccountTypeException ();
        exception CustomerNotFoundException();
        exception InvalidParameterException ();
        exception AccountNotFoundException();
        string createAccount(in string customerId , in string type , in string description ,
                             in BigDecimal balance , in BigDecimal creditLine ,
                             in BigDecimal beginBalance, in Date beginBalanceTimeStamp)
            raises ( IllegalAccountTypeException , CustomerNotFoundException,
                     InvalidParameterException );
        void removeAccount(in string accountId )
            raises ( AccountNotFoundException, InvalidParameterException );
        void addCustomerToAccount(in string customerId , in string accountId )
            raises ( AccountNotFoundException, CustomerNotFoundException,
```

```
                CustomerInAccountException, InvalidParameterException );
        void removeCustomerFromAccount(in string customerId, in string accountId)
            raises ( AccountNotFoundException, CustomerNotFoundException,
                CustomerInAccountException, InvalidParameterException );
        AccountBean::AccountList getAccountsOfCustomer(in string customerId)
            raises ( CustomerRequiredException, CustomerNotInAccountException,
                InvalidParameterException );
        AccountDetails getDetails (in string accountId)
            raises ( AccountNotFoundException, InvalidParameterException );
        void setType(in string type , in string accountId)
            raises ( AccountNotFoundException, IllegalAccountTypeException,
                InvalidParameterException );
        void setDescription (in string description , in string accountId)
            raises ( AccountNotFoundException, IllegalAccountTypeException,
                InvalidParameterException );
        void setBalance(in BigDecimal balance , in string accountId)
            raises ( AccountNotFoundException, IllegalAccountTypeException,
                InvalidParameterException );
        void setCreditLine (in BigDecimal creditLine , in string accountId)
            raises ( AccountNotFoundException, IllegalAccountTypeException,
                InvalidParameterException );
        void setBeginBalance(in BigDecimal beginBalance, in string accountId)
            raises ( AccountNotFoundException, IllegalAccountTypeException,
                InvalidParameterException );
        void setBeginBalanceTimeStamp(in Date beginBalanceTimeStamp, in string accountId)
            raises ( AccountNotFoundException, IllegalAccountTypeException,
                InvalidParameterException );
    }
}

module TxControllerBean {
    interface TxControllerHome {
        exception CreateException ();
        TxController create () raises ( CreateException );
    }

    interface TxController {
        exception InvalidParameterException ();
        exception AccountNotFoundException();
        exception IllegalAccountTypeException ();
        exception InsufficientFundsException ();
        TxList getTxsOfAccount(in Date startDate , in Date endDate, in string accountId)
            raises ( InvalidParameterException );
        TxDetails getDetails (in string txId)
            raises ( TxNotFoundException, InvalidParameterException );
        void withdraw(in BigDecimal amount, in string description , in string accountId)
            raises ( InvalidParameterException , AccountNotFoundException,
                IllegalAccountTypeException , InsufficientFundsException );
        void deposit (in BigDecimal amount, in string description , in string accountId)
            raises ( InvalidParameterException , AccountNotFoundException,
              IllegalAccountTypeException );
        void transferFunds (in BigDecimal amount, in string description , in string fromAccountId,
                    in string toAccountId)
            raises ( InvalidParameterException , AccountNotFoundException,
                InsufficientFundsException , InsufficientCreditException );
        void makeCharge(in BigDecimal amount, in string description , in string accountId)
```

```
        raises ( InvalidParameterException , AccountNotFoundException,
                 IllegalAccountTypeException , InsufficientCreditException );
    void makePayment(in BigDecimal amount, in string description , in string accountId )
        raises ( InvalidParameterException , AccountNotFoundException,
                 IllegalAccountTypeException );
  }
}
```

A.1.2 EntityBeans

```
module CustomerBean {

  typedef sequence<Customer> CustomerList;

  interface CustomerHome {
    exception CreateException ();
    exception MissingPrimaryKeyException();
    exception FinderException ();
    Customer create (in string customerId , in string lastName, in string firstName ,
                     in string middleInitialName , in string street , in string city ,
                     in string state , in Strin phone, in string email)
        raises ( CreateException , MissingPrimaryKeyException);
    Customer findByPrimaryKey(in string customerId ) raises ( FinderException );
    CustomerList findByAccountId(in string accountId ) raises ( FinderExcption );
    CustomerList findByLastName(in string lastName) raises ( FinderException );
  }

  interface Customer {
    CustomerDetails getDetails ();
    void setLastName(in string lastName);
    void setFirstName(in string firstName );
    void setMiddleInitial (in string middleInitial );
    void setStreet (in string street );
    void setCity (in string city );
    void setState (in string state );
    void setZip (in string zip );
    void setPhone(in string phone);
    void setEmail(in string email );
  }
}

module AccountBean {

  typedef sequence<Account> AccountList;
  typedef sequence<string> StringList ;
  interface AccountHome {
    exception CreateException ();
    exception MissingPrimaryKeyException();
    exception FinderException ();
    Account create (in string accountId , in string type , in string description ,
                    in BigDecimal balance , in BigDecimal creditLine ,
                    in BigDecimal beginBalance, in Date beginBalanceTimeStamp,
                    in StringList customerIds)
        raises ( CreateException , MissingPrimaryKeyException);
    Account findByPrimaryKey(in string accountId ) raises ( FinderException );
    AccountList findByCustomerId(in string customerId ) raises ( FinderException );
```

```
    }

    interface Account {
        AccountDetails getDetails ();
        BigDecimal getBalance ();
        string getType ();
        BigDecimal getCreditLine ();
        void setType(in string type);
        void setDescription (in string description );
        void setBalance(in BigDecimal balance);
        void setCreditLine (BigDecimal creditLine );
        void setBeginBalance(BigDecimal beginBalance);
        void setBeginBalanceTimeStamp(Date beginBalanceTimeStamp);
    }
}

module TxBean {

    typedef sequence<Tx> TxList;
    interface TxHome {
        exception CreateException ();
        exception MissingPrimaryKeyException();
        exception FinderException ();
        Tx create (in string txId , in string accountId , in Date timeStamp, in BigDecimal amount,
                in BigDecimal balance , in string description )
            raises ( CreateException , MissingPrimaryKeyException);
        Tx findByPrimaryKey(in string txId ) raises ( FinderException );
        TxList findByAccountId(in Date startDate , in Date endDate, in string accountId)
            raises ( FinderException );
    }

    interface Tx {
        TxDetails getDetails ();
    }
}
```

A.1.3 Externe Schnittstellen

```
module Extern {

    typedef sequence<string> StringList ;

    interface CustomerDetails {
        string getCustomerId ();
        string getLastName();
        string getFirstName ();
        string getMiddleInitial ();
        string getStreet ();
        string getCity ();
        string getState ();
        string getZip ();
        string getPhone ();
        string getEmail ();
        void setCustomerId( string customerId );
        void setLastName(string lastName);
        void setFirstName( string firstName );
```

```
    void  setMiddleInitial ( string  middleInitial );
    void  setStreet ( string  street );
    void  setCity ( string  city );
    void  setState ( string  state );
    void  setZip ( string  zip );
    void  setPhone( string  phone);
    void  setEmail( string  email );
}

interface  AccountDetails {
    string  getAccountId ();
    string  getDescription ();
    string  getType ();
    string  getBalance ();
    BigDecimal getCreditLine ();
    BigDecimal getBeginBalance ();
    Date getBeginBalanceTimeStamp();
    StringList  getCustomerIds ();
    void  setAccountId( string  accountId );
    void  setType( string  type );
    void  setDescription ( string  description );
    void  setBalance (BegDecimal balance);
    void  setCreditLine (BegDecimal creditLine );
    void  setBeginBanlance(BigDecimal beginBalance);
    void  setBeginBalanceTimeStamp(Date beginBalanceTimeStamp);
    void  setCustomerIds( StringList  customerIds );
}

interface  TxDetails {
    string  getTxId ();
    string  getAccountId ();
    Date getTimeStamp();
    BigDecimal getAmount();
    BigDecimal getBalance ();
    String  getDescription ();
}

interface  BigDecimal {
    enum RoundingMode {ROUND_CEILING, ROUND_DOWN, ROUND_FLOOR,
                       ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP,
                       ROUND_UNNECESSARY, ROUND_UP};
    BigDecimal add(BigDecimal val);
    BigDecimal divide(BigDecimal val , RoundingMode roundingMode);
    BigDecimal multiply(BigDecimal val);
    BigDecimal subtrace(BigDecimal val);
}

interface  Date {
    int  getYear ();
    int  getMonth();
    int  getDate ();
    int  getHours ();
    int  getMinutes ();
    int  getSeconds ();
    void  setYear ();
    void  setMonth();
```

```
        void  setDate ();
        void  setHours ();
        void  setMinutes ();
        void  setSeconds ();
    }
}
```

A.2 Terminologieebene

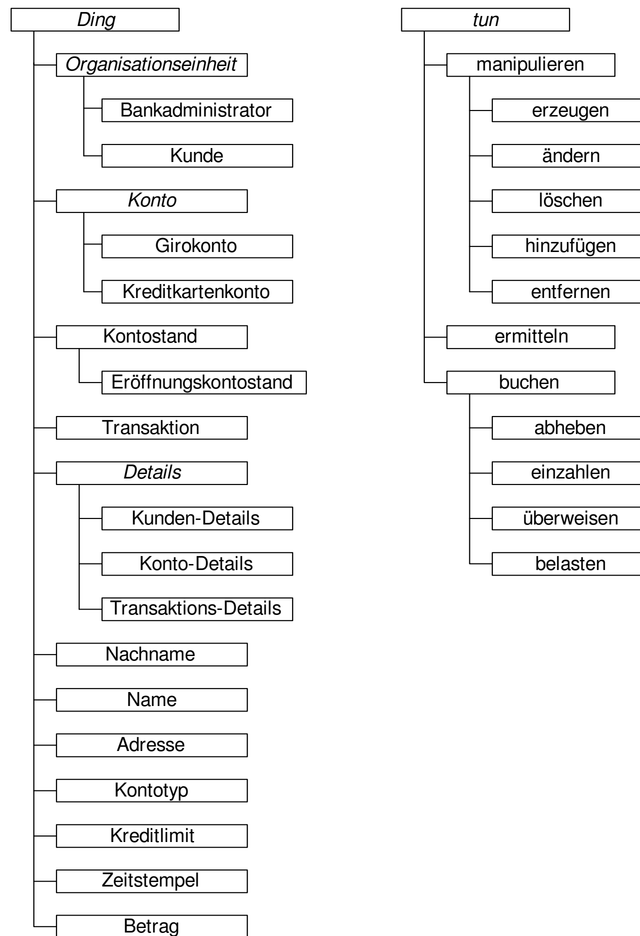

Abbildung 2: Taxonomie der verwendeten Begriffe

Tabelle 1: Spezifikation der Terminologie in einem Lexikon

KONTO **Kurzdefinition:** KONTO $=_{DF}$ die von einer BANK geführte Rechnung über den Zahlungsverkehr ihres Kunden. **Langdefinition:** KONTO $=_{DF}$ Ein KONTO ist die laufende Abrechnung, in der alle Zahlungsvorgänge und daraus resultierende Forderungen und Verbindlichkeiten aus der Geschäftsverbindung zwischen Kreditinstitut und Kunden registriert werden. **Einsatzbeispiele:** SPARKONTO, GIROKONTO, TERMINKONTO, DEPOTKONTO, WÄHRUNGSKONTO
GIROKONTO **Kurzdefinition** GIROKONTO $=_{DF}$ ...
...

Die in Tabelle 1 dargestellte Terminologie stellt nur einen Ausschnitt der tatsächlich auf der Aufgabenebene genutzten Terminologie dar.

A.3 Aufgabenebene

Tabelle 2: Korrespondierende (Daten-)Typen und (Fach-)Begriffe

(Daten-)Typ	**(Fach-)Begriff**
`Customer`	Kunde
`Account`	Konto
`Tx`	Buchung
`BankCustomer`	Kunde
`BankAdmin`	Bankadministrator
`CustomerDetails`	Kundendaten
`AccountDetails`	Kontodaten
`TxDetails`	Buchungsdaten

Tabelle 3: Korrespondierende Dienste und Aufgaben des `CustomerControllerBean`

Dienst	**Aufgabe**
`string createCustomer(in string lastName, in string firstName, in string middleInitial, in string street, in string city, in string state, in string zip, in string phone, in string email raises (InvalidParameterException);`	Bankadministrator tut erzeugen Kunde.
`void removeCustomer(in string customerId) raises (InvalidParameterException, CustomerNotFoundException);`	Bankadministrator tut löschen Kunde.
`ArrayList getCustomersOfAccount(in string accountId) raises (InvalidParameterException, CustomerNotFoundException;`	Bankadministrator tut ermitteln Kunde mit Konto.
`CustomerDetails getDetails(in string customerId) raises (InvalidParameterException, CustomerNotFoundException);`	Organisationseinheit tut ermitteln Kundendaten mit Kunde.
`ArrayList getCustomersOfLastName(in string lastName) raises (InvalidParameterException);`	Bankadministrator tut ermitteln Kunde mit Nachname.
`void setName(in string lastName, in string firstName, in string middleInitial, in string customerId) raises (InvalidParameterException, CustomerNotFoundException);`	Bankadministrator tut setzen Name mit Kunde.
`void setAddress(in string street, in string city, in string state, in string zip, in string phone, in string email, in string customerId) raises (InvalidParameterException, CustomerNotFoundException);`	Bankadministrator tut setzen Adresse mit Kunde.

Tabelle 4: Korrespondierende Dienste und Aufgaben des `AccountControllerBean`

Dienst	**Aufgabe**
`string createAccount(in string customerId, in string type, in string description, in BigDecimal balance, in BigDecimal creditLine, in BigDecimal beginBalance, in Date beginBalanceTimeStamp) raises (IllegalAccountTypeException, CustomerNotFoundException, InvalidParameterException);`	Bankadministrator tut erzeugen Konto.
`void removeAccount(in string accountId) raises (AccountNotFoundException, InvalidParameterException);`	Bankadministrator tut löschen Konto.
`void addCustomerToAccount(in string customerId, in string accountId) raises (AccountNotFoundException, CustomerNotFoundException, CustomerInAccountException, InvalidParameterException);`	Bankadministrator tut hinzufügen Kunde mit Konto.
`void removeCustomerFromAccount(in string customerId, in string accountId) raises (AccountNotFoundException, CustomerNotFoundException, CustomerInAccountException, InvalidParameterException);`	Bankadministrator tut entfernen Kunde mit Konto.
`ArrayList getAccountsOfCustomer(in string customerId) raises (CustomerRequiredException, CustomerNotInAccountException, InvalidParameterException);`	Organisationseinheit tut ermitteln Konto mit Kunde.
`AccountDetails getDetails(in string accountId) raises (AccountNotFoundException, InvalidParameterException);`	Organisationseinheit tut ermitteln Kontodaten mit Konto.
`void setType(in string type, in string accountId) raises (AccountNotFoundException, IllegalAccountTypeException, InvalidParameterException);`	Bankadministrator tut setzen Kontotyp mit Konto.
`void setBalance(in BigDecimal balance, in string accountId) raises (AccountNotFoundException, IllegalAccountTypeException, InvalidParameterException);`	Bankadministrator tut setzen Kontostand mit Konto.
`void setCreditLine(in BigDecimal creditLine, in string accountId) raises (AccountNotFoundException, IllegalAccountTypeException, InvalidParameterException);`	Bankadministrator tut setzen Kreditlimit mit Konto.
`void setBeginBalance(in BigDecimal beginBalance, in string accountId) raises (AccountNotFoundException, IllegalAccountTypeException, InvalidParameterException);`	Bankadministrator tut setzen Eröffnungskontostand mit Konto.
	...

...	
Dienst	**Aufgabe**
`void setBeginBalanceTimeStamp(in Date beginBalanceTimeStamp, in string accountId) raises (AccountNotFoundException, IllegalAccountTypeException, IvalidParameterException);`	Organistationseinheit tut setzen Zeitstempel für Konto.

Tabelle 5: Korrespondierende Dienste und Aufgaben des `TxControllerBean`

Dienst	**Aufgabe**
`ArrayList getTxsOfAccount(in Date startDate, in Date endDate, in string accountId) raises (InvalidParameterException);`	Kunde tut ermitteln Buchung mit Konto.
`TxDetails getDetails(in string txId) raises (TxNotFoundException, InvalidParameterException);`	Organisationseinheit tut ermitteln Buchungsdaten mit Buchung.
`void withdraw(in BigDecimal amount, in string description, in string accountId) raises (InvalidParameterException, AccountNotFoundException, IllegalAccountTypeException, InsufficientFundsException);`	Kunde tut abheben Betrag mit Girokonto.
`void deposit(in BigDecimal amount, in string description, in string accountId) raises (InvalidParameterException, AccountNotFoundException, IllegalAccountTypeException);`	Kunde tut einzahlen Betrag mit Girokonto.
`void transferFunds(in BigDecimal amount, in string description, in string fromAccountId, in string toAccountId) raises (InvalidParameterException, AccountNotFoundException, InsufficientFundsException, InsufficientCreditException);`	Kunde tut überweisen Betrag mit Konto an Konto.
`void makeCharge(in BigDecimal amount, in string description, in string accountId) raises (InvalidParameterException, AccountNotFoundException, IllegalAccountTypeException, InsufficientCreditException);`	Kunde tut belasten Kreditkartenkonto mit Betrag.
`void makePayment(in BigDecimal amount, in string description, in string accountId) raises (InvalidParameterException, AccountNotFoundException, IllegalAccountTypeException);`	Kunde tut einzahlen Betrag mit Kreditkartenkonto.