Entwurfsspezifikation
Überblick

Projekt Verteilte Informationssysteme
Fachbereich Informatik WS2000 - SS2001

27. Februar 2001

Folgende Personen waren an der Erstellung der Entwurfsspezifikation beteiligt:

Studenten	**Lehrkräfte**
Serge Adamowsky	Prof. Klaus Löhr
Alphonse Bendt	Prof. Heinz Schweppe
Michael Bunge	Natalie Ardet
Liviu Coconu	Torsten Fink
Sascha Deutschkron	Gerald Weber
Paul Hackenberger	
Steven König	
Jan Kretzschmar	
Tina Müller	
Manuel Scholz	
Ina Schulz	
Alexander Steidinger	
Jirk Stolze	

Inhaltsverzeichnis

1 Einführung **3**

1.1 Beschreibung des Systemkonzepts 3

1.2 Beschreibung der Zielumgebung 4

1.3 Verteilte Architektur 4

1.4 Einschränkungen gegenüber Systemspezifikation 5

2 Entwurfsspezifikation **5**

2.1 Paketbildung 5

2.2 Interaktion der Schichten 6

2.3 Datenbankschicht 7

2.3.1 Datenbankschema 7

2.4 Logikschicht 7

2.4.1 das ID Konzept 7

2.4.2 Konzept der L-Objekte und der Operationsklassen 9

2.5 Präsentationsschicht 10

2.6 Ablauf einer Auktion 10

1 Einführung

Dieses Dokument dient dazu, einen Überblick über die restlichen Dokumente der Entwurfsspezifikation zu bieten, und gibt darüber hinaus eine Einführung in die zugrundelegenden Konzepte. Alle Dokumente sind unter [1] verfügbar.

Die Entwurfsspezifikation besteht haupsächlich aus UML-Diagrammen, die mit Hilfe von Together (siehe [6]) erstellt wurden. Aus diesen kann mit Hilfe von Together die Dokumentation in verschiedenen Formaten (HTML, RTF) generiert werden.

1.1 Beschreibung des Systemkonzepts

Das System wird in drei Schichten gegliedert. Diese Schichten übernehmen die Aufgaben der Datenhaltung, der Geschäftslogik und der Präsentation.

Die Systemspezifikation besteht im wesentlichen aus einer Spezifikation persistenter Daten und einer Menge von Operationen die den Zustand des Gesamtsystems verändern. Das Datenmodell wird in der Entwurfsspezifikation auf die Datenhaltungsschicht abgebildet und spiegelt sich dort als eine Menge von persistenten Klassen wider.

Die Logikschicht ist für die Realisierung der Operationen aus der Systemspezifikation zuständig. Um die klare Schichtentrennung aufrecht zu erhalten, existieren hier Stellvertreterklassen, im weiteren L(ogik)-Klassen bzw L-Objekte genannt, für die persistenten Klassen der Datenhaltungsschicht. Dadurch wird ein Durchgriff der Präsentationsschicht auf die Datenhaltungsschicht verhindert. Die Operationen der Systemspezifikation, die eindeutig einzelnen Klassen des Systemmodells zugeordnet werden können, werden in einer L-Klasse als Methode implementiert.

Operationen, die mit mehreren Klassen arbeiten bzw. keiner Klasse eindeutig zugeordnet werden können, werden als statische Methode einer Klasse *Operations* realisiert, die in der Geschäftslogikschicht liegt. So hat z.B. die Anmeldung eines Systemnutzers keinen direkten Bezug zu einer Klasse des Systemmodells und findet sich daher in Operations. Die Operationen der Systemspezifikation sind unter transaktionalem Schutz auszuführen, da sie im Allgemeinen nebenläufig von mehreren Benutzern aufgerufen werden können.

Für jede L-Klasse *Classname* die dem Benutzer gegenüber dargestellt werden kann, existieren in der Präsentationsschicht eine oder mehrere Klassen, *JSPXXClassname* genannt. Falls eine L-Klasse auch verändert werden kann, existiert zusätzlich eine *EditableClassname* genannte Klasse in der Präsentationsschicht.

Die in der Systemspezifikation auftretenden binären Assoziationen werden folgendermaßen konkretisiert[1].

- ▷ Gerichtete Assoziationen werden als privates Attribut des Klienten realisiert (siehe Abbildung 1). Typ des Attributs ist die Provider-Klasse oder bei *-Assoziationen eine Java Kollektion (List, Set oder Map).
- ▷ Ungerichtete Assoziationen können als eigene Klasse modelliert werden oder in gerichtete Assoziation umgewandelt werden falls die Operationen dies erlauben. Die konkrete Vorgehensweise hängt vom Einzelfall ab (siehe Abbildung 2).

Die in der Systemspezifikation benutzten einfachen Datentypen werden, soweit möglich auf vorhandene Java - Datentypen abgebildet; so wird z.B. *Datum* mit *java.util.Date*

[1] In Modell der Systemspezifikation treten keine nichtbinären Assoziationen auf.

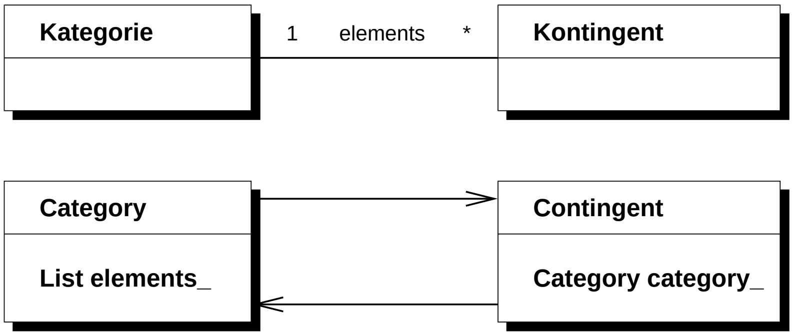

Abbildung 1: Implementierung gerichteter Assoziationen als private Attribute

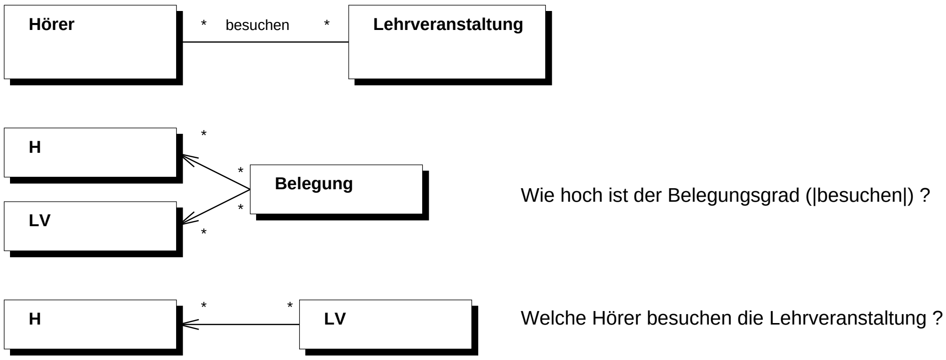

Abbildung 2: Konvertierung ungerichteter Assoziationen zu gerichteten Assoziationen

realisiert. Die weiteren Datentypen werden als Java-Klassen modelliert und sind dem Paket *vis2000.datatypes* zugehörig[2].

1.2 Beschreibung der Zielumgebung

Die Datenhaltung des Systems wird mit Hilfe von Enterprise Java Beans (EJB) realisiert. Die Webschnittstelle des Systems wird mit Hilfe von Servlets und Java Server Pages (JSP) realisiert. Genauere Informationen sind den jeweiligen Spezifikationen [2],[3] zu entnehmen.

Als Servlet-Laufzeitumgebung setzen wir das frei verfügbare Tomcat System ein, das dem Apache Projekt Jakarta entstammt (siehe [4]). Als EJB Container kommt die Open Source Implementierung Jonas zum Einsatz (siehe [5]).

1.3 Verteilte Architektur

Im folgenden wird die verteilte Architektur des Systems beschrieben, die in Abbildung 3 abgebildet ist. Die eingesetzte EJB Technologie ermöglicht einen objektorientierten Zugriff auf die Datenbank und bietet Transaktionsunterstützung. Die Kommunikation zwischen EJB Container und der konkreten Datenbank ist über JDBC realisiert. Wie oben erwähnt, müssen die Operationen im Allgemeinen unter transaktionalem Schutz stehen. Zur Realisierung dieses Schutzes wird auf die Transaktionsmechanismen des EJB-Containers zurückgegriffen.

[2]Man beachte die Beschreibung der Paketstruktur in Abschnitt 2.1.

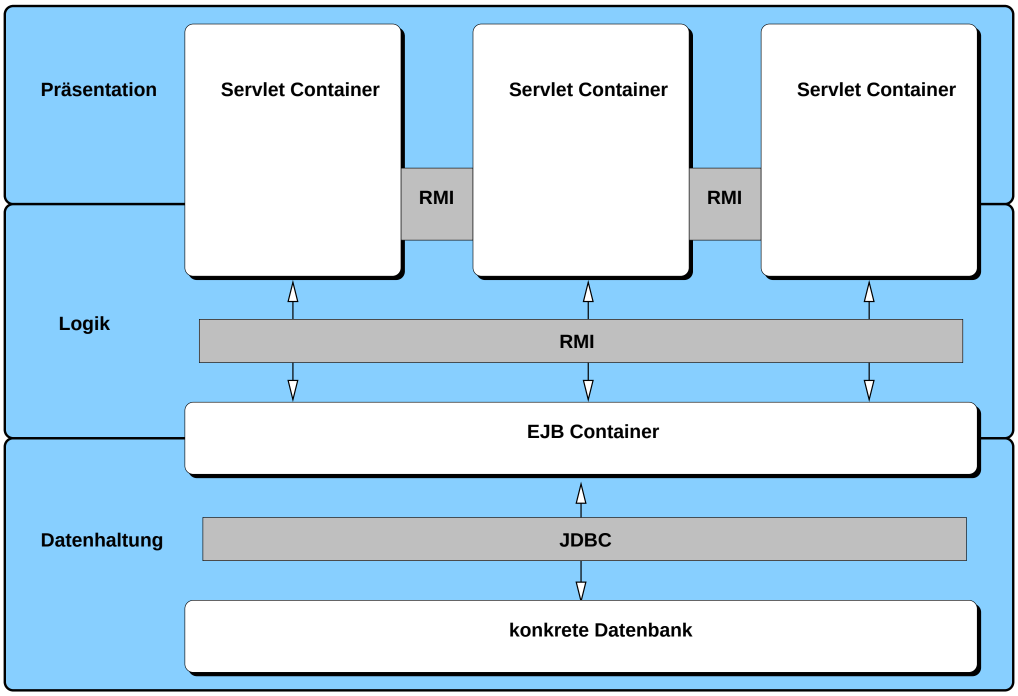

Abbildung 3: Architektur des Systems

Der EJB Container wird aus der Logikschicht aus über RMI angesprochen.

In der Präsentationsschicht wird die Servlettechnologie eingesetzt, da diese eine komfortable Programmierumgebung für Webschnittstellen darstellt. Aus Gründen der Skalierbarkeit können mehrere Servlet Container parallel auf unterschiedlichen Rechnern eingesetzt werden. Falls Kommunikation zwischen Servlets in unterschiedlichen Containern nötig werden sollte, so muß diese über RMI oder CORBA realisiert werden.

1.4 Einschränkungen gegenüber Systemspezifikation

Es wurde aus Zeitgründen entschieden, Bundlekontingente nicht in die Entwurfsspezifikation zu integrieren.

2 Entwurfsspezifikation

2.1 Paketbildung

Das Gesamtsystem wird in mehrere Java Packages aufgeteilt. In Abbildung 4 ist diese Struktur dargestellt. Alle Pakete besitzen das Präfix *vis2000*. Die horizontale Aufteilung der Pakete spiegelt die in Abschnitt 1.3 beschriebene Architektur wider.

database Datenhaltung

logic Geschäftslogik

presentation Präsentation

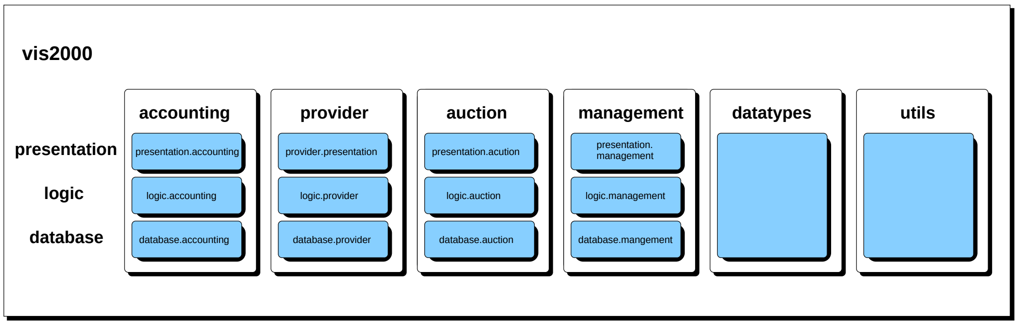

Abbildung 4: Die Paketstruktur des Systems

Diese Pakete enthalten Unterpakete, die sich aus einer vertikalen Aufteilung, erfolgt nach logischen Gruppen aus dem Systemmodell, ergeben. Die Unterpakete sind:

accounting Dieses Unterpaket enthält kunden und kaufbezogene Klassen,

auction Dieses Unterpaket enthält auktionsbezogene Klassen ,

provider Dieses Unterpaket enthält auf den Anbieter bezogene Klassen,

management Dieses Unterpaket enthält auf den Systemverwalter bezogene Klassen.

Klassen die keinem Unterpaket zugeordnet werden können, bzw. von mehreren Paketen aus benutzt werden, können jeweils dem übergeordneten Paket zugeordnet werden. Die Klasse *logic.LSystemuser* wird beispielsweise von *logic.accounting.LCustomer* und *logic.provider.LProvider* benutzt.

Es ist zu anzumerken, daß die horizontalen Schnitte in der Paketstruktur nicht exakt mit den Schnitten in der Architektur übereinstimmen, die weiter oben beschrieben sind. Operationen, die unter transaktionalem Schutz stehen, werden in den Entity- und Session Beans realisiert, um die Transaktionsunterstützung des EJB Containers auszunutzen. Daher befindet sich die Implementierung dieser Operationen teilweise auch in den Paketen *database.** und nicht nur, wie anzunehmen, in den Paketen *logic.**. Die Paketgrenzen sind also eher technisch als logisch motiviert.

Außerdem existieren folgende Pakete:

utils Dieses Paket enthält projektweit wiederverwendbare Klassen enhalten,[3]

datatypes Dieses Paket enthält projektspezifische Datentypen.

2.2 Interaktion der Schichten

Im Abbildung 5 ist ein einfaches Kollaborationsdiagramm dargestellt und beschrieben, um einen ersten Überblick über die Arbeitsweise des Systems zu erhalten.

Dargestellt ist ein Kunde, der sich in Schritt 1 die Beschreibung einer Veranstaltung anschaut. In der Darstellung befindet sich ein Verweis auf ein im Rahmen dieser Veranstaltung angebotenes Ticketkontingent. Der Kunde wählt dieses Kontingent in Schritt 2 zur Betrachtung aus. In Schritt 3 wählt er das Kontingent schließlich zum Kauf aus.

Anhand dieses Beispiels läßt sich erkennen, wie der Informationsfluß im System stattfindet.

[3] Dieses Paket ist noch leer.

1 Browser des Kunden sendet eine Anfrage an das System, um eine Veranstaltung, eindeutig identifiziert über eine ID, darzustellen. Die Darstellung dieser Veranstaltung wird von einem *JSPEvent* Objekt durchgeführt.

1.1 Die entsprechenden Daten holt das *JSPEvent* über *aEventProxy*.

1.1.1 *aEventProxy* greift dazu auf die EntityBean *aEventBean* zu
dort ist vermerkt, welche Ticketkontingente der Veranstaltung zugeordnet sind.

2 Der Kunde schaut sich die Darstellung der Veranstaltung an. Das dafür verfügbare Ticketkontingent wird als Hyperlink dargestellt. Sobald der Kunde diesen Hyperlink anwählt, wird die, dieses Ticketkontingent identifizierende ID, zur Darstellung an das System gesendet.

2.1 Verantwortlich für die Darstellung ist jetzt ein Objekt der Klasse *JSPTicketContingent*. Dieses Objekt greift auf die Daten des Ticketkontingentes über *TicketContingentProxy* zu.

2.1.1 *TicketContingentProxy* holt die Daten aus *TicketContingentBean*.

3 Schließlich entscheidet sich der Kunde, ein Ticket aus diesem Kontingent zu erwerben. Die ID des Ticketkontingentes wird an das System zum Kauf übermittelt.

3.1 Die Informationen über den Kaufwunsch gehen an *JSPTicketContingentProxy*, welches die Informationen auswertet.

3.1.1 Die Ergebnisse dieser Auswertungen (z.B. Verringerung der Ticketanzahl) werden durch Zugriff auf *TicketContingentBean* persistent gemacht.

2.3 Datenbankschicht

Die Datenhaltung wird durch Enterprise Java Beans realisiert. Der Zugriff auf die Daten erfolgt über eine definierte Schnittstelle durch die EntityBeans, welche ein Objekt vom Typ *PrimaryKey* als Primärschlüssel haben. *PrimaryKey* ist ein Interface, welches von als Primärschlüssel dienenden Klassen implementiert wird. Die im System vorkommenden Primärschlüssel sind in einer Vererbungshierarchie organisiert, um zusammengehörende Primärschlüssel zu gruppieren. Damit sind z.B. *ContingentPK* und *EventPK* vom Typ *ItemPK*.

2.3.1 Datenbankschema

Da die Persistenz der Beans von dem EJB-Container verwaltet wird, wird das Datenbankschema automatisch erstellt und ergibt sich aus der Spezifikation der Entity Beans. Im Gegensatz zur Systemspezifikation werden Anbieter und Kunde in der Klasse *SystemUser* zusammengefaßt und anhand einer internen Markierung unterschieden.

2.4 Logikschicht

2.4.1 das ID Konzept

Die ID Klasse wurde eingeführt, um eine Abstraktion von den Primärschlüsseln der Datenbank zu bieten. Die Java String Repräsentation kann von der Präsentationsschicht

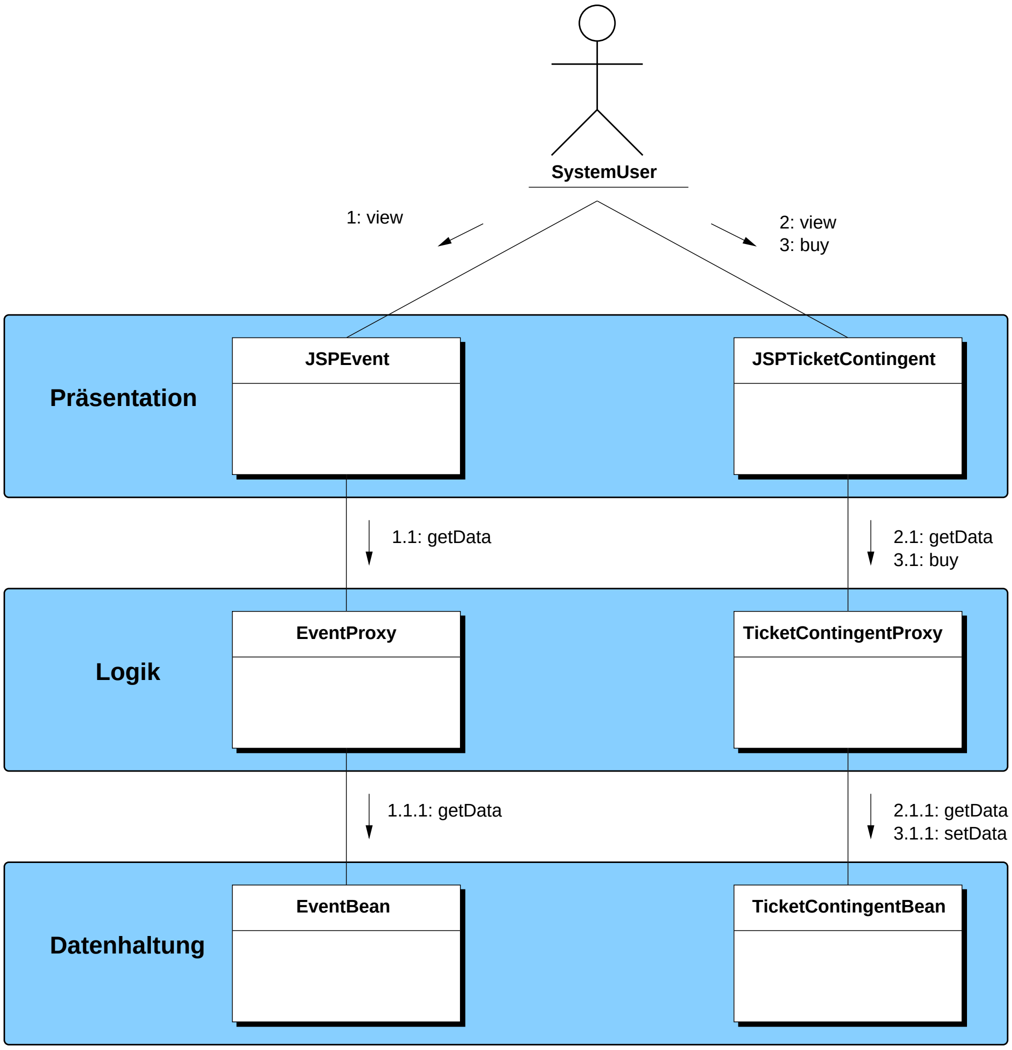

Abbildung 5: Interaktion der Schichten am Beispiel eines Kaufs

genutzt werden, um in eine HTML Seite als Parameter für folgende Http-Anfragen eingebunden zu werden. Anhand dieses Strings kann die ID und damit der Primärschlüssel wiederhergestellt werden.

Eine ID Instanz bildet einen Wrapper um einen Primärschlüssel und bietet folgende Operationen.

$encode :: ID \rightarrow String$ Die ID läßt sich als String darstellen und mit $decode :: String \rightarrow ID$ aus diesem String wiederherstellen. Eine ID Instanz wird mittels $initialize :: PrimaryKey \rightarrow ID$ mit einem PrimaryKey Objekt initialisiert, der sich mit $restore :: ID \rightarrow PrimaryKey$ wiederherstellen läßt.

$\forall id \in ID : decode(encode(id)) = id$

Die Stringrepräsentation ist eindeutig umkehrbar.

$\forall key \in PrimaryKey : restore(initialize(key)) = key$

Die Umwandlung in eine ID ist für einen PrimaryKey ebenfalls identitätserhaltend. Die Operationen *decode* und *initialize* werden durch geeignete Konstruktoren der Klasse ID realisiert, *encode* entspricht der Methode *toString()* und *restore* der Methode *getPrimaryKey()*.

2.4.2 Konzept der L-Objekte und der Operationsklassen

Um Zugriff auf ein EntityBean zu erhalten, muß man eine sogenannte Finder-Methode des EJB-Containers aufrufen. Diese Methode akzeptiert unterschiedliche Argumente, die sich zur Identifizierung eines oder mehrerer Beans eignen. Die Methode liefert dann ein oder mehrere Referenzen auf das BeanRemote Interface implementierende Objekte zurück, mit denen dann Zugriff auf die Beans möglich ist.

Ein naiver Ansatz wäre in jedem Schritt die beteiligten Beans per Finder-Methode zu suchen und dann zu benutzen. Die Erfahrungen aus der Entwicklung des Prototyps[4] zeigten daß dieser Ansatz zu zeitintensiv ist und eine unakzeptables Anwortverhalten zur Folge hat. Daher wurde entschieden ein Caching zu realisieren.

Ein L-Objekt dient als Stellvertreter (Proxy) für eine EntityBean der Datenbankschicht. Dieses L-Objekt bietet Zugriff auf die Daten des EntityBeans und bietet Operationen an, um dieses EntityBean zu verändern.

Zugriff auf ein solches L-Objekt erhält man durch Benutzung der abstrakten Fabrik *ItemFactory*. Diese Fabrik nutzt beim ersten Aufruf der Methode *LObject getItem(ID id)* die Finder-Methoden des EJB Container, um eine Referenz auf eine Entity Bean zu erhalten. Mit dieser Referenz wird ein L-Objekt initialisiert, das dann an den Aufrufer zurückgegeben wird. Weitere Aufrufe mit der gleichen ID kann die Fabrik direkt mit dem zwischengespeicherten L-Objekt bedienen.

L-Objekte können Daten aus den Entity Beans zur Verbesserung des Antwortverhaltens zwischenspeichern, wenn keine Konsistenzprobleme auftreten. Schreibzugriffe werden auf jeden Fall unmittelbar auf dem entsprechenden Entity Beans ausgeführt.

Falls das von einem L-Objekt referenzierte Entity Bean nebenläufig aus dem System entfernt wird, kann es vorkommen, daß das referenzierende L-Objekt davon keine Kenntnis hat. Der nächste Zugriff auf das L-Objekt und damit auf das Bean wird scheitern. Ein L-Objekt signalisiert dem aufrufenden Objekt diesen Fehlerfall mit einer Ausnahme vom Typ *StaleObjectException* mit.

[4] die Dokumentation zum Prototyp findet sich unter [7]

Falls nur bestimmte Systemnutzer eine Methode aufrufen dürfen sollen, enthält die Methodensignatur der L-Objekte ein weiteres Argument in dem der aufrufende Systemnutzer zu übergeben ist, so daß eine Zugriffskontrolle realisiert werden kann. Zugriffsschutzverletzungen werden mit Ausnahmen vom Typ *AccessViolationException* signalisiert.

Alle Operationen, die sich nicht einzelnen Klassen zuordnen lassen, finden sich in der Operationsklasse wieder. Diese Klasse bietet statische Methoden an, der dann die ID der beteiligten L-Objekte übergeben werden.

2.5 Präsentationsschicht

Für jedes L-Objekt, das darstellbar sein soll, existiert in der Präsentationsschicht eine Java Server Page (JSP). JSP erlauben es, statischen HTML Code mit speziellen Tags anzureichern, um dynamische Inhalte aus Java Objekten darzustellen.

Falls das Geschäftsobjekt auch veränderbar sein soll, existiert zusätzlich ein Objekt der Klasse *EditableClassname,* das die Eingaben des Benutzer aus der Http Anfrage extrahiert, auf Konsistenz prüft und an die zuständigen L-Objekte der Logikschicht weiterleitet.

2.6 Ablauf einer Auktion

In Abbildung 6 ist der beispielhafte Ablauf einer Auktion abgebildet. Im folgenden eine Erklärung der Abläufe.

1. Ein Anbieter möchte einem Ticketkontingent eine Auktion zuordnen.
2. Eine Auktionsschnittstelle wird instantiiert und dem Ticketkontingent zugeordnet.
3. Die Auktion wird mit den Auktionsparametern initialisiert.
4. Während der Initialisierung registriert sich die Auktion mit Eintrittszeit und Auktionsende bei einem Timer.
5. Der Timer benachrichtigt die Auktion über den Startzeitpunkt.
6. Die Auktion startet sich.
7. Der Timer benachrichtigt die Auktion über den Zeitverlauf, indem er in regelmäßigen Zeitintervallen die Methode *actionPerformed()* aufruft.
8. Ein Kunde gibt ein Gebot ab.
9. Der Timer benachrichtigt die Auktion über das Auktionsende.
10. Die Auktion beendet sich und speichert die Daten der erfolgten Auktionsabschlüsse intern zur weiteren Bearbeitung.
11. Die Auktion meldet sich beim Timer ab und kann aus dem System entfernt werden.

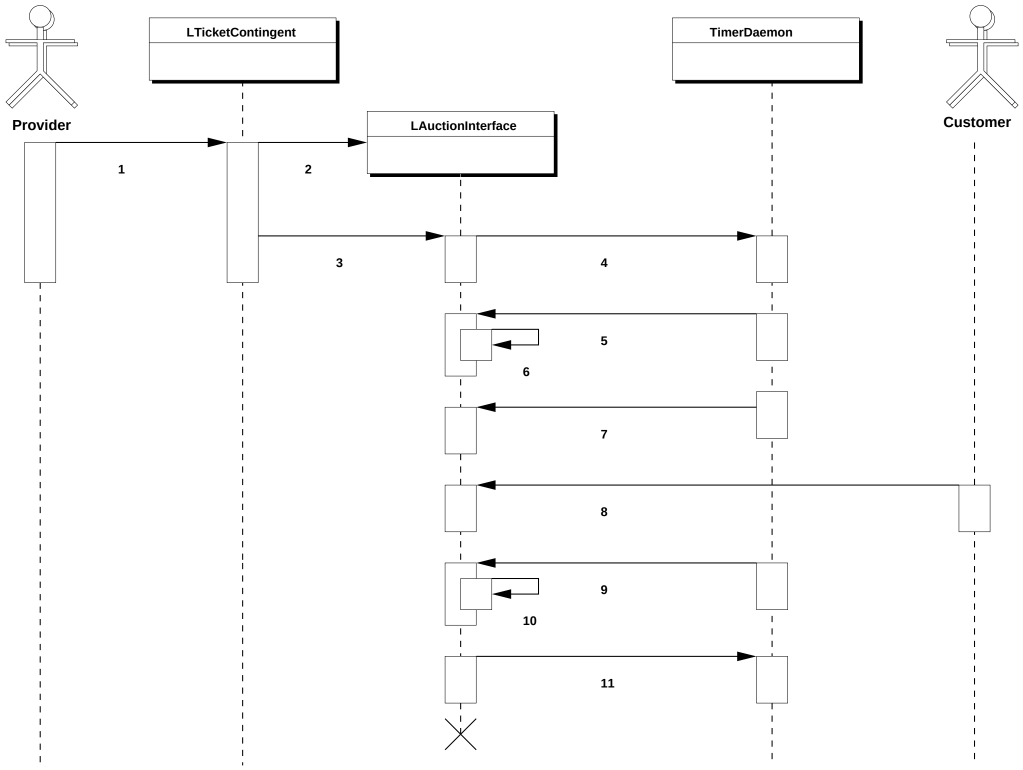

Abbildung 6: Ablauf einer Auktion

Literatur

[1] Projektdokumentation `http://www.inf.fu-berlin.de/~vis2000/doc`

[2] Java Servlets `http://java.sun.com/products/servlet/index.html`

[3] Enterprise Java Beans `http://java.sun.com/products/ejb`

[4] Jakarta Projekt `http://java.apache.org`

[5] Jonas EJB Container `http://www.evidian.com/jonas`

[6] Together Homepage `http://www.togethersoft.com`

[7] Prototyp `http://www.inf.fu-berlin.de/~vis2000/prototype/`