2.3. Enterprise Java Beans

Enterprise JavaBeans (EJBs) stellen seit 1998 eine Komponententechnologie innerhalb eines Java-EE-Servers dar und sind dementsprechend Teil der Java Enterprise Edition (Java EE). Innerhalb des Servers laufen sie in einem sogenannten EJB-Container, der mehrere Dienste zur Verfügung stellt und einen Großteil der technischen Arbeit übernimmt. Die EJBs dienen der einfachen und schnellen Implementierung von mehrschichtigen, verteilten Systemen, wie in diesem Projekt. So ist es durch sie möglich sowohl die Anwendungslogik als auch die Persistierung vorzunehmen. In EJB 3.0, das 2006 erschienen ist, werden drei Hauptkomponenten unterschieden: Session Beans, Message Driven Beans und Persistent Entities. Den Begriff der Persistent Entity wird in der Spezifikation von EJB 3.0 nicht durchgehend verwendet, da aber Ihns u. a. [15] diesen Begriff durchgängig benutzen, wird er auch in dieser Arbeit verwendet.

2.3.1. Session Beans

Session Beans werden genutzt, um die Interaktion des Nutzers mit dem System abzubilden. Sie werden in zustandslose (stateless) und zustandsbasierte (stateful) Session Beans unterteilt. Beide Session-Bean-Typen bestehen aus einer Bean-Klasse, die die Implementierung der Methoden beinhaltet, und einem Remote Business Interface oder einem Local Business Interface. Die Bean-Klasse wird entweder mit `@Stateless` oder mit `@Stateful` annotiert, je nachdem, ob es sich um eine stateless oder stateful Session Bean handelt. Das Remote Business Interface ist die Schnittstelle für Zugriffe über Prozess- und Rechnergrenzen hinweg, wohingegen es sich bei dem Local Business Interface um die Schnittstelle für die lokalen Zugriffe aus demselben Java-Prozess handelt. Stateless Session Beans haben keinen Zustand in Bezug auf den Client, da die Verbindung zwischen Client und Bean nur während des Methodenaufrufs besteht. Bei stateful Session Beans ist das anders, da diese für die Dauer des Geschäftsprozesses exklusiv einem Client zugeordnet sind. Bei den Bean-Klassen handelt es ab Version 3.0 um POJOs (Plain Old Java Objects). Dieser Begriff wurde von Fowler [10] geprägt und bezeichnet grundsätzlich ein ganz normales Java-Objekt, dass weder von einer vorspezifizierten Klasse erbt, noch ein vorspezifiziertes Interface implementieren muss. Dies ist in EJB 3.0 nicht mehr der Fall, da die Interfaces frei definiert werden können und lediglich eins der beiden implementiert werden muss und dieses dann lediglich annotiert wird.

2.3.2. Message-Driven Beans

Bei Message-Driven Beans handelt es sich wie bei den Session Beans um POJOs. Sie realisieren eine asynchrone Nachrichten-Kommunikation mit EJBs im Gegensatz zu der synchronen Kommunikation der Session Beans. Diese blockieren die

Objekte, die sie aufrufen so lange sie die aufgerufene Methode ausführen. Erst nach Beenden dieser kann das Objekt weiterarbeiten. Ist dies nicht gewünscht, ist mit Message-Driven Beans eine asynchrone Kommunikation möglich. Dann muss das Objekt nicht auf die Bean warten und kann weiterarbeiten.

Das Versenden von Nachrichten wird über sogenannte „Message-oriented Middleware“ (MOM) vorgenommen. Teil der MOM ist der „Message-Broker“, der die Nachrichten empfängt, zwischenspeichert und weiterleitet. Durch ihn sind Sender und Empfänger entkoppelt. Durch die lose Koppelung der Komponenten kann eine Performanzsteigerung erzielt werden, da der Sender nun keine Zeit mehr mit Warten verbringen muss. Außerdem ist das Messaging zuverlässiger als die synchrone Kommunikation, da hier der Empfänger nicht unbedingt verfügbar sein muss. Ist der Empfänger beispielsweise ein Server, der gerade abgestürzt ist, so wiederholt der Message-Broker die Nachricht so lange, bis der Server wieder empfangsbereit ist. Bei einer synchronen Kommunikation geht die Nachricht allerdings verloren. Außerdem kann mit Messaging-Systemen ein Broadcast realisiert werden, d.h. ein Nachrichtenversand an mehrere Empfänger gleichzeitig.

Die Kommunikation von Message-Driven Beans wird im Allgemeinen per „Java Message Service“ (JMS) vorgenommen. Die JMS-Spezifikation besteht aus „Service Provider Interface“ und API, wobei nur diese für den Anwendungsentwickler interessant ist. Die Empfänger müssen sich beim Message-Broker über „Destinations“ registrieren, die Nachrichtenwarteschlangen darstellen. Es können sich mehrere Empfänger bei einer Destination registrieren.

Im Gegensatz zu Session Beans bestehen Message-Driven Beans ausschließlich aus der Bean-Klasse, d.h. es werden keine weiteren Interfaces benötigt. Sie sind wie stateless Session Beans zustandslos in Bezug auf den Client. Die Kommunikation zwischen Clients und Message-Driven Beans erfolgt über den EJB-Container, der zugleich JMS-Provider ist. Er ruft eine bestimmte Methode der Message-Driven Bean auf, der er die Nachricht als Parameter übergibt. Diese Methode kann aufgrund der asynchronen Kommunikation weder Ergebnisse noch Exceptions an den Sender schicken. Stattdessen muss wieder eine neue Nachricht an den Sender verschickt werden.

2.3.3. Persistent Entities

Unternehmenssoftware benötigt häufige eine dauerhafte Speicherung der anfallenden Daten in einer Datenbank. Diese Speicherung wird als Persistierung bezeichnet. Da ein EJB-System objekt-orientiert ist, sind die Daten auch in Objekten gespeichert. Diese Objekte werden als *Persistent Entities* bezeichnet und im Allgemeinen in einer relationalen Datenbank gespeichert. In der Datenbank werden die Daten in Tabellen und Spalten gespeichert. Das Übertragen der Objektdaten auf Datenbankspalten wird als *Object Relational Mapping* bezeichnet.

Wichtig ist, dass die Attribute eines Objektes nicht immer einfach in die Spalten geschrieben werden können. Insbesondere Beziehungen zwischen Objekten sind komplexer zu übertragen. Die Abbildung kann in der orm.xml oder per Annotation in den sogenannten Persistent Entities vorgenommen werden.

Bei den Persistent Entities handelt es sich wie bei den anderen Komponenten um POJOs. Sie werden ähnlich wie die Session und Message-Driven Beans per Annotation als Entities definiert. Alternativ können sie in der persistence.xml definiert werden. Mit `@Entity` wird aus einem POJO eine Persistent Entity. Um die Objekte in einer relationalen Datenbank zu speichern, muss eine ID angegeben werden. Dies kann ebenfalls per Annotation vorgenommen werden. So wird das Attribut, das als ID genutzt werden soll, mit `@Id` annotiert. Außerdem kann man die Generierung dieser ID per `@GeneratedValue` an die Datenbank delegieren. Um Beziehungen darzustellen, können ebenfalls Annotationen benutzt werden.

Für unidirektionale Eins-zu-Eins-Beziehungen annotiert man das Attribut, das einem Objekt einer anderen Persistent Entity entspricht, mit der man in Beziehung steht, mit `@OneToOne`. Außerdem wird der Name der Fremdschlüsselspalte angegeben. Sofern das Datenbankschema automatisch vom Persistence Provider vorgenommen wurde, so wird das Namensschema <Referenzierte Tabelle>_<Primärschlüsselspalte> verwendet. Beim Persistence Provider handelt es sich um eine Implementierung der Java Persistence API (JPA). Der Persistence Provider wird vom Anwendungsserver gestellt und ist für die Persistierung der Persistent Entities zuständig. Für eine unidirektionale Ein-zu-Viele-Beziehung nutzt man dementsprechend die `@OneToMany`-Annotation. Diese wird an das Attribut geschrieben, dass die Viele-Seite realisiert. Dieses muss für eine JPA-Unterstützung einer Java-Collection-Klasse angehören, wie z.B. java.util.List. Außerdem sind noch die Annotationen `@ManyToOne` und `@ManyToMany` möglich.

Um Persistent Entities zu persistieren, benötigt man einen Entity-Manager. Dieser wird durch den Persistence Provider gestellt. Durch die Nutzung des Entity-Managers entfällt der technische Persistierungscode, denn die Persistierung wird vom Persistence Provider übernommen. Der Entity-Manager stellt als zentraler Teil von JPA alle Dienste für die Persistierung bereit.

Um Abfragen zu realisieren benutzt man mit JPQL eine an SQL angelehnte Abfragesprache. Für eine Query nutzt man den Entity-Manager und z.B. dessen Methode `createQuery(String qlString)`.

2.3.4. EJB-Komponentenarchitektur

Enterprise Java Beans existieren, wie bereits beschreiben, im sogenannten EJB-Container eines Java-EE-Anwendungsservers. Der EJB-Container übernimmt die Erzeugung, Verwaltung und Zerstörung der EJBs automatisch, so dass der

Entwickler sich um solche technischen Details nicht kümmern muss. Der Zugriff auf EJBs kann von außerhalb des Containers und auch von außerhalb des Anwendungsservers vonstatten gehen, in dem passende Technologien, wie z.B. RMI, eingesetzt werden. Der Anwendungsserver beinhaltet nicht nur einen EJB-Container, sondern enthält weitere Container (z.B. Servlet-Container und Webserver-Container) und bietet verschiedene weitere Dienste an. Der Anwendungsserver stellt die Laufzeitumgebung für eine große Vielfalt von Softwareartefakten, die verwaltet werden müssen. Dabei fasst Java EE eine große Menge von APIs und Spezifikationen zusammen, die aber bezüglich der Implementierung beliebig kombiniert werden könne. So nutzt WebSphere 7.0 als JPA-Persistence Provider beispielsweise standardmäßig Apache OpenJPA. Die Implementierung kann aber auch ausgetauscht werden, so wären auch Hibernate oder TopLink denkbar. Diesbezüglich gibt es von Seiten des Java-EE-Anwendungsservers keine Vorgaben. Die Anwendungsserver bieten außerdem eine große Vielfalt an unterstützten Kommunikationsprotokollen, wie HTTP(S), RMI, SOAP, etc. Der hier implementierte Webservice nutzt beispielsweise SOAP via HTTP.

Dementsprechend bietet die EJB-Technologie eine einfache Art und Weise für die Implementierung von service-orientierten, verteilten Business-Systemen, wie es auch bei dem hier entwickelten System der Fall ist.

2.4. Weiterführendes

Diese Arbeit gliedert sich in die momentane Forschung im Bereich der Metriken im *Lehr- und Forschungsgebiet Informatik 3 Softwarekonstruktion* ein. Deshalb folgt eine Beschreibung des sich in der Entwicklung befindenden Metrikwerkzeugs MeDIC [29]. Dies ist insbesondere deshalb wichtig, da in Zukunft eine Integration des in dieser Arbeit vorgestellten Systems in MeDIC vorgesehen ist. Es folgt ein kurzer Abschnitt zu *SOA (Service Oriented Architecture)*, da das System auch service-orientiert ist.

2.4.1. MeDIC

Um eine große Menge von Metriken zu definieren, zu pflegen, zu dokumentieren und weiterzuentwickeln, reicht es nicht aus, Metriken rein textuell zu beschreiben, da dies einige Nachteile mit sich bringt.

Erstens sind rein textuelle Beschreibungen der Messvorschriften selten präzise genug, um Fehler bei der Verwendung der Metriken zu vermeiden. Falsch durchgeführte Messungen aufgrund falsch verstandener Messvorschriften führen zu unbrauchbaren Ergebnissen. Des Weiteren können rein textuelle Beschreibungen leicht dazu führen, dass unterschiedliche Personen die Beschreibungen unterschiedlich interpretieren, so dass auch unterschiedliche Ergebnisse erzielt