Java Persistence API

JPA ist eine Sammlung von Schnittstellen, welche es uns erlaubt, Java-**Objekte** samt ihren Beziehungen in einem Datastore abzuspeichern (persistent zu machen).

Genau gesagt besteht JPA aus diesen drei Teilen:

1. The Java Persistence API
2. The query language
3. Object/relational mapping metadata

Punkt 1

enthält wie gesagt Schnittstellendefinitionen und Annotations.

Punkt 2

definiert eine eigene Abfragesprache.

Sie ist leicht zu lernen, da sie sehr ähnlich aufgebaut ist wie SQL. Im Unterschied zu SQL werden aber keine Tabellenabfragen, sondern Objektabfragen gemacht. Das heisst, wir brauchen keine Joins sondern können direkt Objektreferenzen benutzen.

Punkt 3

hilft beim Abspeichern von Objekten in einer relationalen Datenbank.

Relationale Datenbanken funktionieren mittels Tabellen und Fremdschlüsselbeziehungen. Abfrageergebnisse sind Mengen die sich aus einer Abfrage (über eine oder mehrere Tabellen hinweg) ergeben. Beziehungen werden typisch durch weitere Fremdschlüsselabfragen aufgelöst.

Objekte sind Dinge die miteinander in Beziehung stehen. Wenn Kunden eine Adresse haben, dann soll mit dem laden des Kunden automatisch die zugehörige Adresse vorhanden, und verbunden sein. Kurz, Beziehungen müssen einfach existieren, ohne weitere Abfragen.

Diese Anforderung führt zusammen mit relationalen DBs zum sogenannten Object-relational impedance mismatch. JPA hilft, diesen Mismatch transparent zu machen.

Die Idee ist, dass wir unser Objekte mit JPA-Annotations versehen, so dass es klar ist, was abgespeichert werden muss, welche DB-Tabelle für ein Objekt benutzt wird und wie die einzelnen Fields mit Tabellenspalten verknüpft sind. Zudem müssen wir für all Verknüpfungen die Art der Beziehungen angeben.

Beachten Sie bitte, dass JPA nicht nur mir relationalen DBs funktioniert. JPA besteht nur aus Schnittstellendefinitionen, JPA-**Implementierungen** können für beliebige Datastores (OO-DB, Filesystems, XML-Files etc.) geschrieben werden.

Repository-Layer in Serveranwendungen

Sie haben schon gehört, dass das Repository Layer die Aufgabe hat, Daten aus einem Datastore zu besorgen und allfällige Zustandsänderungen der Business Objects im Datastore persistent zu machen. Es dient als Abstraktionsschicht für den Datenzugriff.

OK, was heisst das und warum brauchen wir das?

Wenn wir den Zustand eines Business-Objects ändern, ändert sich seine Darstellung in Speicher (RAM). Es muss aber sichergestellt werden, dass diese Änderung auch auf dem Datastore ankommt, also persistent wird. Wir möchten nicht bei jeder Mutation daran denken müssen, dass wir den Datastore über die Änderung benachrichtigen müssen. Das wäre zu viel, und vor allen sehr fehleranfälliger Code, der zudem eigentlich gar nichts mit der Business-Funktion des Objektes zu tun hat.

Wir lösen das Problem in zwei Stufen. Erstens definieren wir das Business-Object als Interface welches nur Lesefunktionen hat. In unserer Applikation (dem Service-Layer) können wir die Objekte folgerichtig gar nicht ändern. Dann brauchen wir uns auch keine Gedanken zu machen, dass der Datastore über Änderungen benachrichtigt werden muss.

Zweitens definieren wir im Repository-Layer Funktionalität, welche Business-Objects und weitere Parameter erhält, die Modifikationen vornimmt und den Datastore über die Änderungen informiert. Damit haben wir das Problem der Datenkonsistenz an einem zentralen Ort gekapselt und erledigt.

Und wie bitteschön soll das funktionieren, wenn unsere Business-Objects nur Interface sind?

Sie haben Recht. Natürlich müssen wir eine Implementation für die Business-Objects erstellen. Diese Enthalten dann alle benötigten Fields, Konstruktoren, Lesefunktionen aus dem Interface und die nötigen Setter-Funktionen für allfällige Zustandsänderungen. Beachten Sie, dass wir uns hier konkret festlegen müssen, welche Art von Datastore wir benutzen wollen. Die JPA-Annotations können für verschiedene BDs unterschiedlich sein.

Eine Adresse als Beispiel

Stellen Sie sich vor, wir haben eine Adresse. Wir müssen Adressen erstellen, lesen, ändern und löschen können. Genau das ist die Aufgabe des «AddressRepository». Es stellt die CRUD Funktionalität zur Verfügung. (CRUD: create, read, update, delete).

Da die Adressimplementierung Datastorespezifisch ist, muss auch die AddressRepository Implementierung Datastorespezifisch sein. Selbstverständlich können Sie auch mehrere Repositoryimplementierungen haben. Sie müssen sich dann allerdings entscheiden, welche Implementierung Sie benutzen wollen.

Das führt zu folgendem Klassendiagramm:

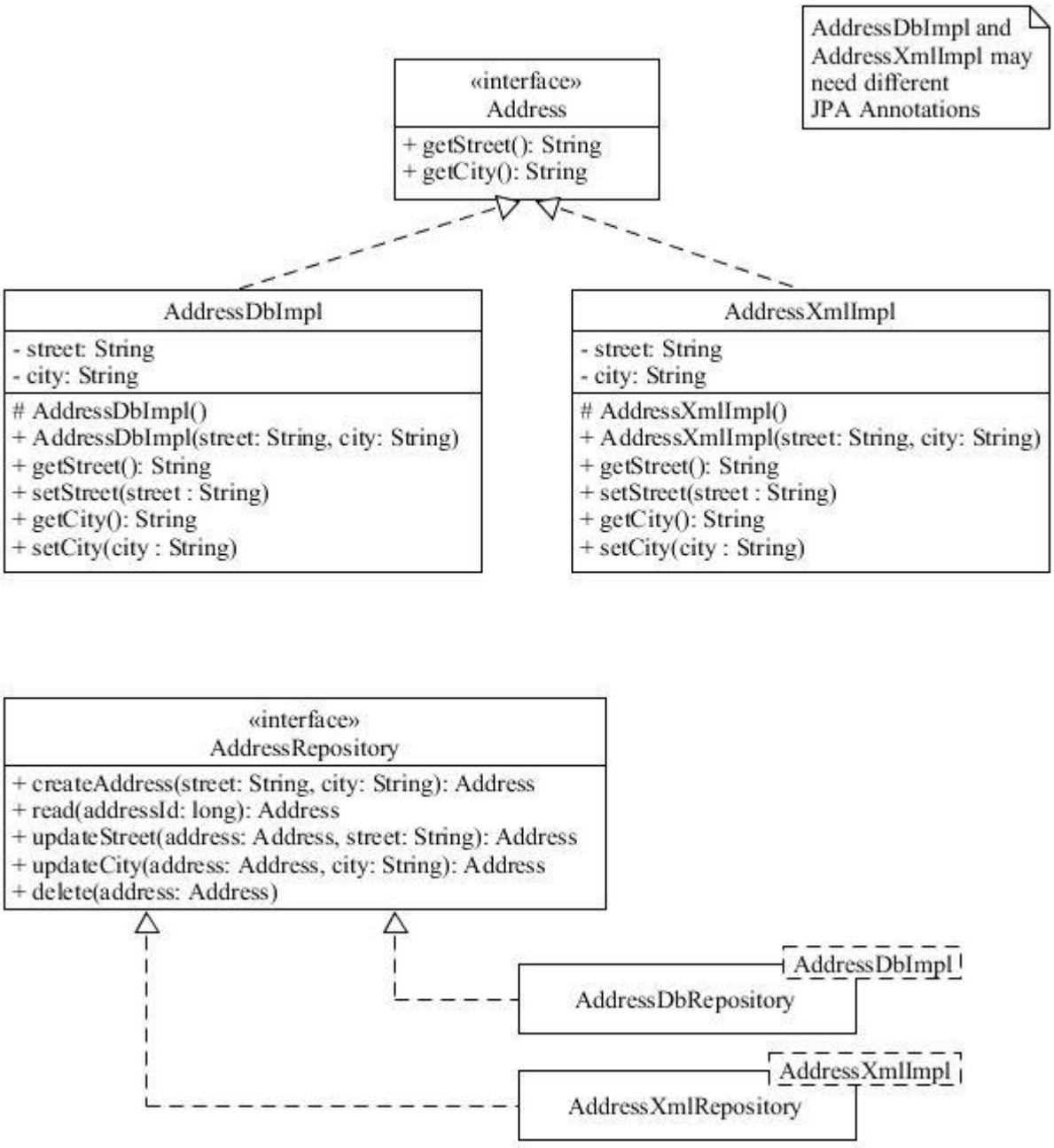

Abbildung 1 Klassendiagramm Adresse

JPA und Repositories in Spring

Das obenstehende Klassendiagramm wirkt sehr aufwendig. Da wir aber nur eine Art von Datastore (eine relationale DB) ansteuern wollen, fallen schon mal zwei Klassen weg. Spring befreit uns auch davon, eine konkrete Repository-Implementierung zu schreiben. Es reicht, wenn wir unser Repositoryinterface von JpaRepository<AddressDbImpl, Long> ableiten. Spring generiert uns dann im Hintergrung „automagicaly" eine Implementierung. Um das Wunder noch zu vergrössern, reicht es oft, unser Interface mit den gewünschten Abfragefunktionen zu versehen, Spring generiert dann aus den Funktionsnamen automatisch die nötige JPA-Abfrage.

«Customer»

```
/** Business object abstraction for a customer */
public interface Customer {

      /** @return the customer id or null if not yet persisted on the data store */
      Long getId();

      /** @return the customers name */
      String getName();

      /** @return the customers street */
      String getStreet();

      /** @return the customers location */
      String getCity();

      /** @return a list of memos sorted by time, descending */
      List<Memo> getMemos();
}
```

«CustomerRepository»

```
/** The Customer repository */
public interface CustomerRepository extends JpaRepository<CustomerImpl, Long> {

       /**
        * Create a new Customer object
        * @return a new Customer object
        */
       default Customer create(String name, String street, String city) {
              CustomerImpl customer = new CustomerImpl(name, street, city);
              return save(customer);       // Persist changes to data store
                                           // and return the new customer
       }

       /**
        * Update a given Customer object
        * @return the updated customer
        */
       default Customer update(Customer customer, String name, String street, String city) {
              CustomerImpl customerImpl = (CustomerImpl) customer;
              customerImpl.setName(name);
              customerImpl.setStreet(street);
              customerImpl.setCity(city);
              return save(customerImpl);   // Persist changes to data store
                                           // return the updated customer
       }
      }
```

«CustomerImpl»

Was jetzt noch fehlt, ist die Klasse «CustomerImpl». Beachten Sie, dass «CustomerImpl» momentan keine Funktionalität bezüglich der referenzierten Memos beinhaltet. Wir werden das später erledigen.