Cargo-Tracker: DDD auf Basis von Java EE

Dirk Ehms
GameDuell GmbH
Berlin

Schlüsselworte
Domain-Driven Design, DDD, Java EE, Cargo Tracker

Einleitung

Die Herangehensweise des Domain-Driven Designs (DDD) wurde erstmalig im gleichnamigen Buch von Eric Evans vor mehr als 10 Jahren (2003) beschrieben. DDD stellt eine Menge von Prinzipien und Patterns zur Verfügung. Diese unterstützen insbesondere den Modellierungsprozess während der Softwareentwicklung. Dabei lässt sich DDD sehr gut mit Ansätzen wie Test-Driven Development (TDD), Behavior-Driven Development (BDD) und agilen Prozessen kombinieren.

Die aktuelle Version der Java Enterprise Edition (JEE) bietet durch ihr leichtgewichtiges Programmiermodell optimale Voraussetzungen, um unter Anwendung der DDD Philosophie ein Softwaresystem zu entwickeln. Das zeigt auch Oracles aktuelle JEE-Blueprint-Applikation „Cargo Tracker“, welche vollständig ein komplexes Anwendungsbeispiel aus Evans Buch umsetzt.

„Cargo Tracker“ implementiert die Sendungsverfolgung von Frachtgut rund um den Globus. Das dabei verwendete Domänenmodell zeigt Abb. 1. Die Applikation veranschaulicht auf praktische Weise das Mapping der DDD-Bausteine auf die verschiedenen Programmierschnittstellen (API) von JEE. Allerdings erschließt sich dieses Mapping nicht immer auf dem ersten Blick und muss teilweise per Reverse Engineering extrahiert werden.

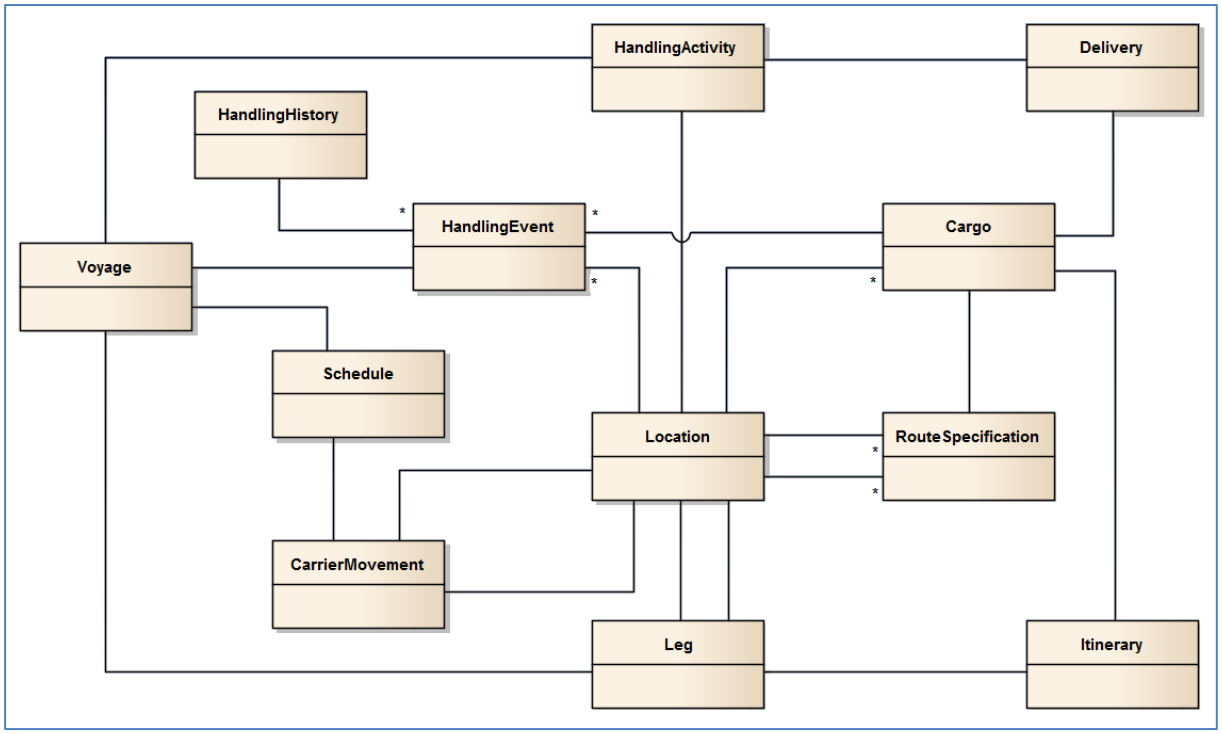

Abb. 1: Domänenmodell

Konzepte des Domain Driven Designs

Die Vorzüge beim Einsatz von DDD sind vor allem komplexen Geschäftsfeldern vorbehalten. Dabei erfolgt eine Rückbesinnung auf die Konzepte der Objektorientierte Analyse und Design (OOAD). Das modellgetriebene Vorgehen von DDD rückt das Domänenmodell ins Zentrum der Betrachtung. Dabei handelt es sich um ein erweitertes Modell (Rich Domain Model), bei dem die Domänenobjekte nicht nur Zustand sondern auch Verhalten besitzen.

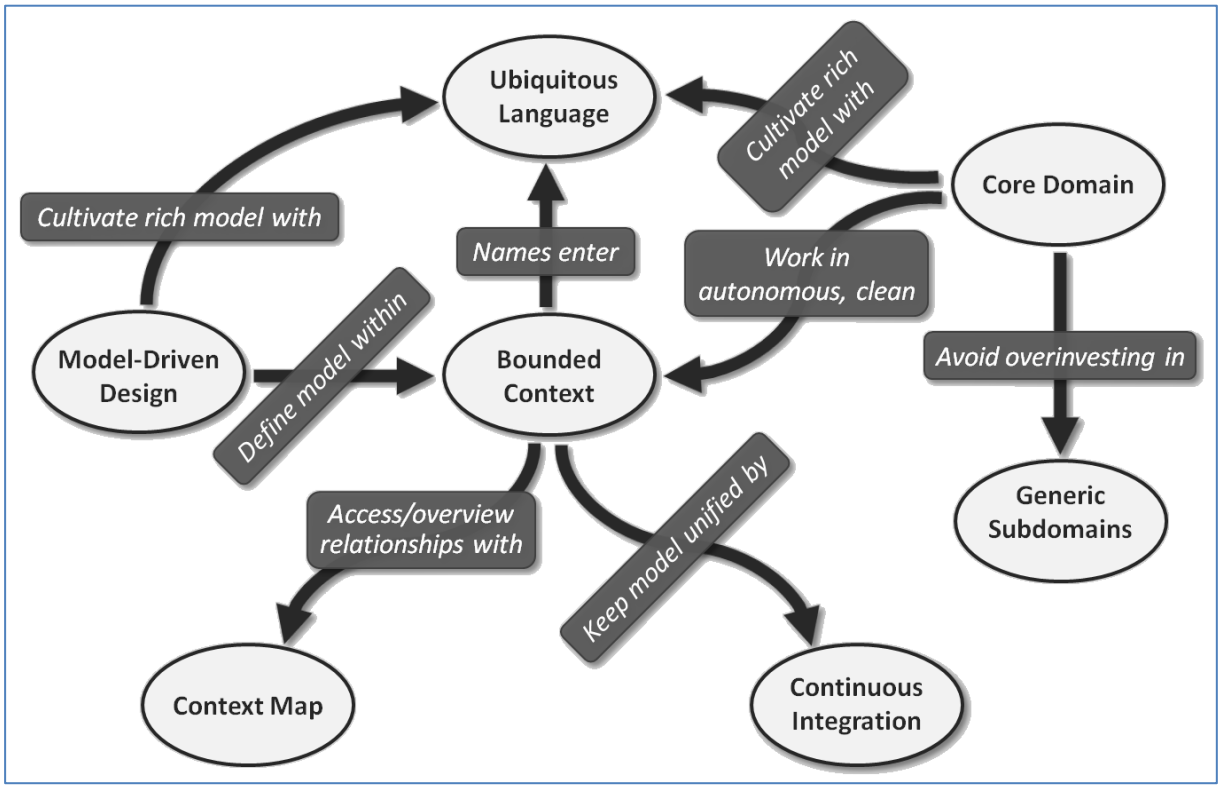

Abb. 2: DDD-Konzepte

Es erfolgt eine direkte Abbildung des Modells in den Quellcode der Anwendung. Das heißt Änderungen des Domänenmodells führen auch zu Änderungen im Code und umgekehrt. Einen Überblick der DDD-Konzepte und ihre Beziehungen zeigt Abb. 2. Zur Vereinfachung wurde auf die Darstellung der unterschiedlichen Patterns zur Umsetzung des Context Mappings in der Abbildung verzichtet. Die einzelnen Konzepte werden im Folgenden kurz erklärt.

- *Model-Driven Design* überführt ein Domänenmodel in die Strukturen und das Design eines Softwaresystems.
- *Bounded Context* definiert den Kontext in dem ein Domänenmodel seine Anwendung findet.
- *Ubiquitous Language* ist eine einheitliche Fachsprache, welche innerhalb eines *Bounded Context* definiert wird. Sie dient als Grundlage der Kommunikation zwischen Fachexperten und Entwicklung. Das Domänenmodel wird als Kern der *Ubiquitous Language* verwendet. Durch ihre konsequente Verwendung wird die Übersetzung der fachlichen Terminologie in eine technische Entsprechung vermieden und so das Risiko von Missverständnissen verringert.
- *Context Map* definiert Schnittstellen zwischen verschiedenen *Bounded Contexts* und ermöglicht so die Kommunikation über Kontextgrenzen hinweg.

- *Core Domain* beschreibt einen Teil des Geschäftsfeldes eines Unternehmens mit starken Einfluss auf den Unternehmenserfolg. Aus diesem Grund sollte hier auch der Schwerpunkt bei der Softwareentwicklung liegen.
- *Generic Subdomains* sind untergeordnete Geschäftsfelder, die bei der Entwicklung nur nachrangig berücksichtigt werden sollten.
- *Continuous Integration* unterstützt die Konsistenz des Domänenmodells mit dem Quellcode aufrecht zu erhalten.

Bausteine des Domain-Driven Designs

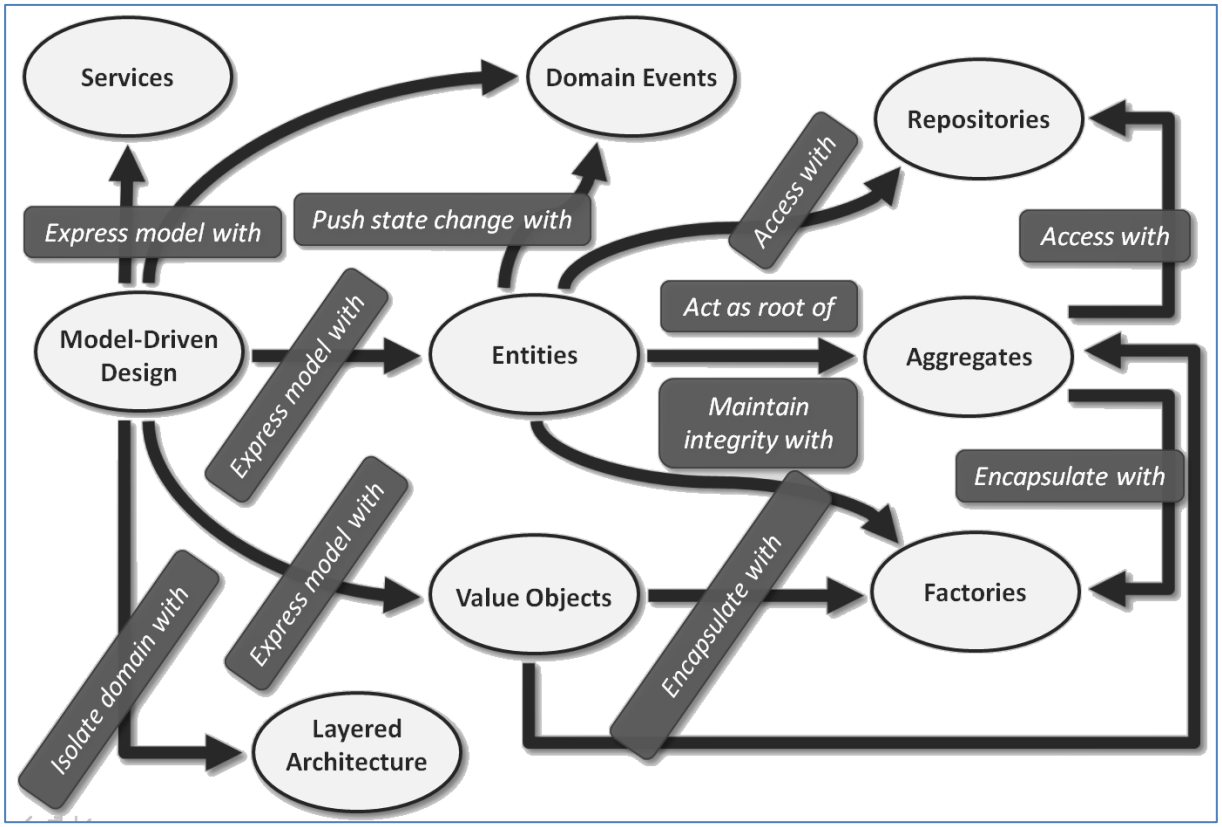

Abb. 3: DDD-Bausteine

Ausgehend von einem modellgetriebenen Design definiert DDD eine Menge von Bausteinen (Abb. 3). Diese Bausteine werden verwendet, um das Domänenmodell der Anwendung auszudrücken und das dazugehörige Verhalten zu modellieren.

- *Entities* (Entitäten) sind Domänenobjekte, die nicht durch ihre Eigenschaften sondern durch ihre Identität definiert werden. Diese Identität bleibt während des gesamten Lebensdauer unverändert.
- *Value Objects* (Werteobjekte) sind unveränderliche Objekte ohne Identität und werden dazu verwendet um Attribute von *Entities* zu beschreiben.
- *Aggregates* sind Gruppen von individuellen aber in enger Beziehung stehender Domänenobjekte (*Entities, Value Objects*), welche als Einheit betrachtet werden. Genau ein *Entity* aus dem Objektverbund wird als *Aggregate Root* ausgezeichnet und definiert den Zugangspunkt auf das *Aggregate*.

- *Domain Events* unterstützen die Entkopplung von Teilen der Anwendung durch das Senden einer Nachricht bei Zustandsänderungen von Domänenobjekten.
- *Repositories* werden für den Zugriff auf Instanzen von *Aggregates* verwendet.
- *Factories* kommen bei der Erzeugung komplexer Domänenobjekte zum Einsatz.
- (Domain) *Services* sind zustandslos und besitzen ein definiertes Verhalten.
- *Layered Architecture* (Schichtenarchitektur) dient als Mittel der Strukturierung von DDD-Applikationen.

Technologie Mapping

Die verfügbaren Java EE Technologien bieten unterschiedliche Möglichkeiten, um die Bausteine von DDD (Abb. 3) zu implementieren. Dabei kommen in den einzelnen Schichten der Schichtenarchitektur unterschiedliche Technologien zum Einsatz. Außerdem finden die mit Java 5 eingeführten Annotationen häufig Verwendung. Die im folgenden dargestellten Codefragmente sind der Beispielanwendung „Cargo Tracker“ entnommen und können dort vollständig nachgeschlagen werden.

Das Herz der Anwendung schlägt im *Domain Layer*. Hier erfolgt die Implementierung der Geschäftslogik mit den dazugehörigen Domänenobjekten. Die Zuweisung von Objektinstanzen zwischen *Services*, *Factories* und *Repositories* erfolgt durch Contexts and Dependency Injection (CDI). Die Verwendung der Annotation `@Inject` ermöglicht den Zugriff auf benötigte Instanzen (Listing 1), welche durch den Applikationsserver gemanagt werden.

```
@ApplicationScoped
public class HandlingEventFactory implements Serializable {

    @Inject
    private CargoRepository cargoRepository;
    @Inject
    private VoyageRepository voyageRepository;
    @Inject
    private LocationRepository locationRepository;

    public HandlingEvent createHandlingEvent(...) {...}
    ...
}
```

Listing 1: Contexts and Dependency Injection

Weiterhin macht „Cargo Tracker“ im *Domain Layer* intensiven Gebrauch vom Java Persistence API (JPA). Dabei werden *Entities* mit der gleichnamigen Annotation `@Entity` and *Value Objects* mit `@Embeddable` gekennzeichnet (Listing 2 und 3).

```
@Entity
public class Cargo implements Serializable {...}
```

Listing 2: Entities annotieren

```
@Embeddable
public class Itinerary implements Serializable {...}
```

Listing 3: Value Objects annotieren

Durch die Verwendung von JPA Annotationen wird allerdings eine Abhängigkeit des *Domain Layers* zu einer spezifischen Technology (Relationale Datenbanken) erzeugt. Diese Abhängigkeit kann durch die alternative Definition des JPA Mappings in einer externen XML Datei vermieden werden (Listing 4).

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
    http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
    version="2.0">

    <entity class="...domain.model.cargo.Cargo" access="FIELD">
        ...
    </entity>

    <embeddable class="...domain.model.cargo.Itinerary" access="FIELD">
        ...
    </embeddable>
</entity-mappings>
```

Listing 4: JPA-XML-Mapping

Weitaus schlanker als der *Domain Layer* ist der *Application Layer*. Aus DDD Perspektive werden dem *Application Layer* die verschiedenen *Application Services* zugeordnet. Diese enthalten keine Geschäftslogik, welche den *Domain Services* im *Domain Layer* vorbehalten ist. Stattdessen sind sie für die Umsetzung nichtfunktionaler Anforderungen wie Transaktionsmanagement oder die Prüfung von Zugriffsrechten verantwortlich. Darüber hinaus können sie auch Aufrufe zum *Domain Layer* bündeln und auf die Weise die Granularität von Schnittstellen anpassen. Entsprechend ihren Aufgaben lassen sich *Application Services* gut mit Enterprise Java Beans (EJB) umsetzen. Zu diesem Zweck werden die Services mit der Annotation `@Stateless` versehen (Listing 5).

```
@Stateless
public class DefaultBookingService implements BookingService {

    @Inject
    private CargoRepository cargoRepository;
    @Inject
    private LocationRepository locationRepository;
    @Inject
    private RoutingService routingService;
    ...
}
```

Listing 5: Application Service

Die Anbindung an externe Systeme erfolgt im *Infrastructure Layer*. Für die meisten Applikationen steht hier der Datenzugriff auf Datenbanksysteme im Vordergrund. Dazu werden die im *Domain Layer* definierten *Repository* Interfaces unter Verwendung von JPA implementiert. Die Annotation `@PersistenceContext` injiziert die benötigte EntityManager-Instanz und ermöglicht den Zugriff auf persistente Daten (Listing 6). Neben *Repositories* befinden sich im *Infrastructure Layer* noch *Infrastructure Services*. Diese verwenden verschiedene Client-Implementierungen (JAX-RS, JAX-WS, JMS, JavaMail), um den Zugriff auf externe Systeme zu erlangen.

```
@ApplicationScoped
public class JpaCargoRepository implements CargoRepository, Serializable {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public Cargo find(TrackingId trackingId) {...}
    ...
}
```

Listing 6: JPA Repository

Die *User Interface* Schicht implementiert die Benutzschnittstelle, wobei „Cargo Tracker“ Java Server Faces (JSF) verwendet werden. Außerdem wird in dieser Schicht für die Anbindung aufrufender Systeme Sorge getragen. Für die Unterstützung der unterschiedlichen Kommunikationsprotokolle werden JAX-RS (Listing 7) und Websockets (Listing 8) verwendet.

```
@Path("/cargo")
public class CargoMonitoringService {

    @Inject
    private CargoRepository cargoRepository;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public JsonArray getAllCargo() {...}
}
```

Listing 7: JAX-RS Service

```
@ServerEndpoint("/tracking")
public class RealtimeCargoTrackingService {...}
```

Listing 8: Websocket Service

Die dargestellte Anwendung der unterschiedlichen Technologien zeigt nur einen Ausschnitt der Möglichkeiten beim Einsatz von JEE zur Umsetzung von DDD. Sie dient hoffentlich als Anreiz um das Thema Domain-Driven Design weiter zu vertiefen. In jedem Fall lohnt es sich im Quellcode der Applikation „Cargo Tracker“ auf Entdeckungsreise zu gehen und DDD hautnah zu erleben.

Kontaktadresse:
Dirk Ehms
GameDuell GmbH
Taubenstrasse 24-25
D-10117 Berlin

Telefon:	+49 (0) 30-288 768 210
Fax:	+49 (0) 30-288 768 299
E-Mail	dirk.ehms@gameduell.de
Internet:	inside.gameduell.de