GEDOPLAN

Implementierung von Domain-Driven Design (DDD) mit Jakarta EE

Jens Seekamp, GEDOPLAN GmbH

Kernaussagen von DDD

Domain-Driven Design (DDD) definiert durch

E. Evans

Domain-Driven Design: Tackling Complexity in the Heart of Software

Addison-Wesley, 2004

DDD-Ansatz

- objektorientiertes Entwurfskonzept für komplexe, unternehmensweite Anwendungs-Systeme
- Fokus der Software-Entwicklung liegt auf Geschäftslogik
- Software-Design und -Implementierung basieren auf Domänen-Modell
- Zusammenarbeit zwischen technischen und fachlichen Experten

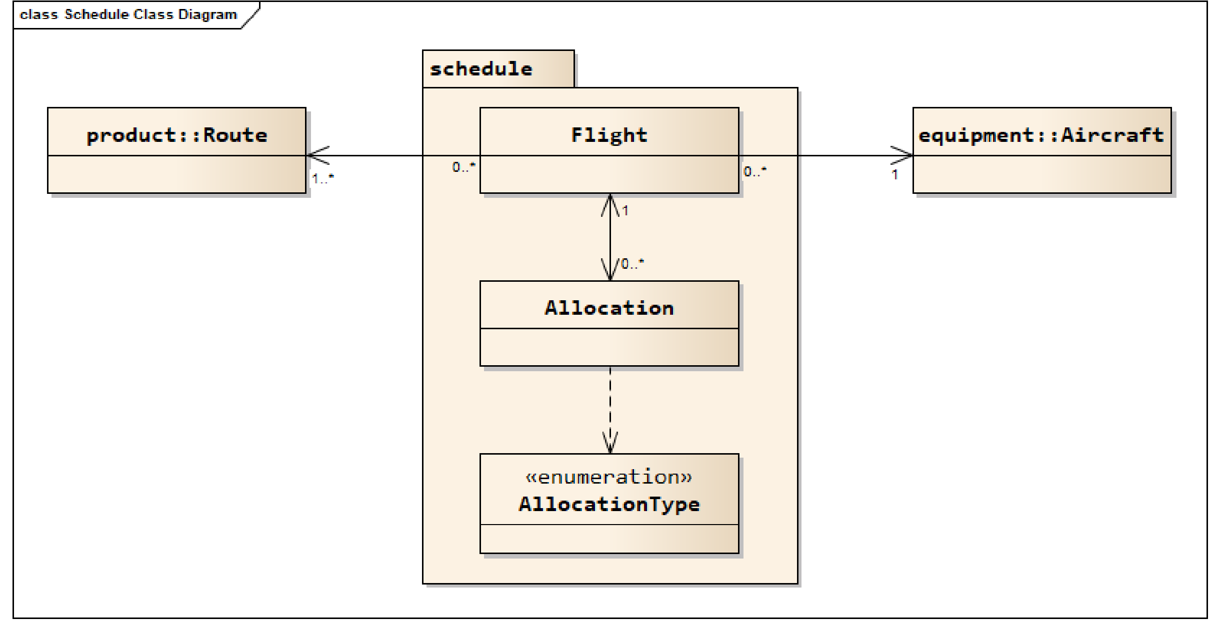
Konzepte und Begriffe des DDD

- *Domain* = unser Geschäftsbereich
- *Domain Model* = abstrakte Beschreibung unseres Geschäftsbereiches
- *Ubiquitous Language* = wie wir miteinander sprechen
- *Bounded Context* = Räume unseres "Software-Gebäudes"
- *Building Block* = Bausteine für unser "Software-Gebäude"

DDD-Fallbeispiel ("Referenz-Implementierung")

- Domain: Flight Information System (FIS)
- Domain Model: UML, Software-Requirement-Specifications, ...
- Ubiquitous Language:

- Bounded Context:



Building Blocks des DDD

- Domänen-Objekte für Daten
  - Domain Aggregate, Domain Entity, Domain Entity Identifier, Domain Value, Domain Attribute
- Domänen-Funktionalität
  - Domain Factory, Domain Repository, Domain Service
- Datenaustausch
  - Composite Domain Object, Domain Event
- Geschäftsvorgänge
  - Application Service

Domain Entity (DDD: *entity*)

- veränderbares Geschäftsobjekt
  - hat unveränderbare Identität
  - Zustand als Attribute
  - Beziehungen zu anderen Geschäftsobjekten
  - Verhalten in Form von Methoden
    - also nicht anämisch = "blutleer"
  - oftmals (aber nicht notwendig) persistent

Domain Entity

class Domain Entity

«interface»
Comparable<T>

«interface»
DomainEntity<I>

+ getId() :I
+ getVersion() :Long

«abstract»
DomainEntityImpl<I>

- id :I
- version :Long

+ equals(Object) :boolean
+ hashCode() :int
+ toString() :String
+ compareTo(DomainEntity<I>) :int

«Domain Entity»
Customer

- id
- number
- turnover

+ processInvoicePayment(CustomerTurnover) :void

1 0..*

«Domain Entity»
Contact

Company

PrivateClient

Domain Entity

- JPA-Entity

```
@Entity
@Table(name = "SC_FLIGHT")
@Access(AccessType.FIELD)
public class Flight extends DomainEntityImpl<FlightId> {

  @EmbeddedId
  private FlightId id;

  @NotNull @Valid
  @Embedded
  private FlightPrice price;

  @OneToMany(...)
  private Set<Allocation> offeredWithAllocations;

  public AllocationNumberOfAvailableSeats
    determineAvailableSeats(AllocationType allocationType)
  { ...
```

Object Relational Mapping (ORM)

Attribut-Typen sind eingebettete Klassen

Bean Validation inkl. Propagation

Beziehungen mit Rollennamen

Funktionalität auf Attributen

Domain Entity Identifier

- Identifikation für ein Geschäftsobjekt
  - Schlüssel-Attribut(e)
  - initialisierender Konstruktor
  - nur Getter (keine Setter)
  - serialisierbar

- unveränderbar (immutable)

Domain Entity Identifier

class fis.sales.domain.model.rootcustomer.customer.attribute-id

«interface»
Serializable

«interface»
DomainEntityId

«utility»
UniqueOidGenerator

+ generate() :Long

«abstract»
DomainAttributeImpl<V>

+ equals(Object) :boolean
+ hashCode() :int
+ toString() :String
+ compareTo(DomainAttributeImpl<V>) :int

«Domain Entity Id»
CustomerId

\- value :Long

+ CustomerId(Long) :CustomerId
+ getValue() :Long

Domain Entity Identifier

- JPA-Embeddable

```
@AllArgsConstructor
@Embeddable
@Access(AccessType.FIELD)
public class FlightId
  extends DomainAttributeImpl<Long>
  implements DomainEntityId {

  private static final long serialVersionUID = 1L;

  @GeneratedValue(...)
  private Long id;

  @Override
  public Long getValue()
  {
    return this.id;
  }
}
```

Code-Generierung mit Lombok

Serializable

Best Practice:
generierte, technische, einteilige ID

Domain Value (DDD: *value object*)

- unveränderbares Geschäftsobjekt
  - hat keine Identität
  - Zustand als Attribute
  - Beziehungen zu anderen Geschäftsobjekten
  - Verhalten in Form von Methoden
  - oftmals (aber nicht notwendig) Bestandteil von persistenten Geschäftsobjekten

- quasi unveränderbar (immutable)

Domain Value

class fis.fundamentals.domain.model.address-value

«interface»
DomainValue

«abstract»
DomainValueImpl

«Domain Value»
Address

- street :AddressStreet
- houseNumber :AddressHouseNumber
- postOfficeBox :AddressPostOfficeBox
- postCode :AddressPostCode
- city :AddressCity

+ create(AddressComposite) :Address
+ getAttributes() :AddressComposite
- setAttributes(AddressComposite) :void
+ selectAssociatedCountry() :Country
+ print() :String

«Domain Entity»
Customer

- homeAddress :Address
- invoiceAddress :Address

+locatedInCountry
0..*
1

«Domain Entity»
Country

Domain Value

- JPA-Embeddable

```
@AllArgsConstructor
@Getter
@EqualsAndHashCode
@ToString
@Embeddable
@Access(AccessType.FIELD)
public class Address extends DomainValueImpl {

  @Embedded
  private AddressCity city;

  @ManyToOne
  private Country locatedInCountry;
```

Standard-Code (Lombok)

Attribut-Typen sind eingebettete Klassen

Domain Aggregate (DDD: *aggregate*)

- Sammlung von Geschäftsobjekten in Form eines Objekt-Graphen
  - mit einem definierten Geschäftsobjekt als Wurzel-Knoten
  - mit Beziehungen zwischen den enthaltenen Geschäftsobjekten

- Aggregat stellt Konsistenz bei Änderungen an enthaltenen Geschäftsobjekten sicher
  - externer Zugriff nur in definierter Weise über den Wurzel-Knoten
  - oftmals mit kaskadierenden Persistenz-Operationen erreichbar

Domain Aggregate

class Domain Aggregate

Comparable<T>
«interface»
DomainEntity<I>

«interface»
DomainAggregateRoot<I>

«abstract»
DomainEntityImpl<I>

«abstract»
DomainAggregateRootImpl<I>

«Domain Aggregate Root»
Customer

1

*

«Domain Entity»
Contact

«Domain Value»
Address

Domain Aggregate

- typisches (einfaches) Beispiel: Kompositions-Beziehung

```
@Entity
public class Booking extends DomainAggregateRootImpl<BookingId> {

  @OneToMany(mappedBy = "subordinatedToBooking",
    fetch = FetchType.EAGER,
    cascade = {CascadeType.PERSIST, CascadeType.REMOVE},
    orphanRemoval = true)
  private Set<BookingPosition> composedByBookingPositions;
```

Kaskadierung

```
@Entity
public class BookingPosition extends DomainEntityImpl<BookingPositionId>
{

  @ManyToOne
  @JoinColumn(...)
  private Booking subordinatedToBooking;
```

Domain Attribute (DDD: *value object*)

- dedizierter, spezifischer Datentyp für ein fachliches Attribut
  - eine Instanzvariable `value`
  - initialisierender Konstruktor
  - nur Getter (kein Setter)
  - serialisierbar

- unveränderbar (immutable)

- keine Verwendung vordefinierter Java-Typen für fachliche Attribute
  - `Country.isoCode` ≠ `Airport.iataCode` (obwohl beide `String`)
- Aufwandschätzung für Change Requests wird vereinfacht

Domain Attribute

class fis.sales.domain.model.rootcustomer.customer.attribute-attribute

«interface»
Serializable

«interface»
DomainAttribute<V>
+ getValue() :V

«interface»
Comparable<T>
+ compareTo(T) :int

«abstract»
DomainAttributeImpl<V>
+ equals(Object) :boolean
+ hashCode() :int
+ toString() :String
+ compareTo(DomainAttributeImpl<V>) :int

«Domain Attribute»
CustomerNumber
- value :Integer
+ CustomerNumber(Integer) :CustomerNumber
+ getValue() :Integer

«abstract»
DomainAttributeMoneyAmountImpl
+ getMoneyAmount() :MoneyAmount
+ *getCurrencyUnit() :CurrencyUnit*

«Domain Attribute»
CustomerTurnover
- value :BigDecimal
+ CustomerTurnover(BigDecimal) :CustomerTurnover
+ getValue() :BigDecimal
+ getCurrencyUnit() :CurrencyUnit

EUR

Domain Attribute

- JPA-Embeddable als Wrapper um einen einfachen Java-Datentyp

```
/** IATA-Code für einen {@link Airport} */
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@AllArgsConstructor
@Embeddable
@Access(AccessType.FIELD)
public class AirportIataCode extends DomainAttributeImpl<String> {

  private static final long serialVersionUID = 1L;

  @Getter
  @NonNull
  @NotNull
  @Pattern(regexp = "[A-Z]{3}")
  @Column(name = "IATA_CODE")
  private String value;
}
```

Typ-Beschreibung

rein deklarative Programmierung

Serializable

top-down propagierte Bean Validation

Composite Domain Object

- passive Datenstruktur zur Kapselung beliebiger Geschäftsdaten
  - gemäß JavaBeans-Spezifikation
  - beinhaltet unveränderbare Domänen-Objekte
    - Domain Entity Identifier, Domain Attribute
  - Schachtelung und Java-Collections möglich
  - transient, serialisierbar

- veränderbar nur bei Bedarf, sonst unveränderbar (immutable)

- Data Transfer Object (DTO) zum kompakten Datenaustausch
  - Parameter- und Rückgabetypen von Services
  - use-case-spezifische Komposition von fachlichen Daten

Composite Domain Object

class Composite Domain Object

«interface»
Serializable

«interface»
CompositeDomainObject

«abstract»
CompositeDomainObjectImpl

«Composite Domain Object»
CustomerComposite

- number :CustomerNumber
- turnover :CustomerTurnover
- rebate :RebateComposite
- invoiceAddress :Address

«Composite Domain Object»
RebateComposite

- flightClass :FlightClass
- rate :RebateRate

DomainValueImpl
<<Composite Domain Object>>
Address

- street :AddressStreet
- houseNumber :AddressHouseNumber
- postCode :AddressPostCode
- city :AddressCity

Composite Domain Object

- JavaBeans-Klasse

```
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class CustomerComposite extends CompositeDomainObjectImpl {

  private static final long serialVersionUID = 1L;

  @Valid
  private CustomerNumber number;

  @Valid
  private RebateComposite rebate;

  @Valid
  private AddressComposite invoiceAddress;
}
```

rein deklarative Programmierung

Serializable

Propagation der Bean Validation

Domain Factory (DDD: *factory*)

- Kapselung der Funktionalität zur Erzeugung von Domain Entities und Domain Values
  - separate Factory-Klasse
  - statische Factory-Methode in einer Geschäftsobjekt-Klasse

- Delegation der Objekt-Erzeugung an spezialisierte Factory-Klasse oder -Methode
  - Erzeugung von kompletten Aggregaten oder komplexen Objekten
  - Lokalisierung der Geschäftsobjekt-Erzeugung
  - Austauschbarkeit der Implementierung

Domain Factory

class Domain Factory

«interface»
DomainFactory<E,I,C>

«abstract»
DomainFactoryImpl<E,I,C>
\+ *create(C) :E*

«Domain Factory»
CustomerFactory
\+ create(CustomerComposite) :Customer

DomainEntityImpl<I>
«Domain Entity»
Customer

DomainAttributeImpl<V>
«Domain Entity Id»
CustomerId

CompositeDomainObjectImpl
«Composite Domain Object»
CustomerComposite

Domain Factory

- CDI-Bean

```
@ApplicationScoped
public class BookingFactory
  extends DomainFactoryImpl<Booking, BookingId, BookingComposite> {

  @Inject
  BookingPositionFactory factoryBookingPosition;

  public Booking createConfirmedBooking(
    @NotNull @Valid CustomerId idCustomer,
    @NotNull @Valid FlightId idFlight,
    @NotNull @Valid Set<BookingPositionComposite> positions,
    @NotNull @Valid List<PassengerCreationComposite> passengers)
  {
    ...
```

Delegation an injizierte "Dienste"

Parameter-Validierung

Domain Repository (DDD: *repository*)

- Kapselung der Funktionalität zum Persistieren und Wiederauffinden von Domain Entities
  - separate Repository-Klasse
  - CRUD-Operationen mittels injiziertem JPA-Entity-Manager
  - spezielle Daten-Abfragen mit JPA Criteria API (oder z. B. Querydsl)

- Delegation der Objekt-Persistenz an spezialisierte Repository-Klasse
  - Abstraktion von Persistenz-Framework
  - generische Implementierung der Standard-CRUD-Funktionalität
  - Lokalisierung der Geschäftsobjekt-Persistenz
  - Austauschbarkeit der Implementierung

Domain Repository

class Domain Repository

«interface»
DomainRepository<E,I>

+ insert(E) :I
+ update(E) :void
+ delete(E) :boolean
+ findById(I) :E

«abstract»
DomainRepositoryImpl<E,I>

- entityManager :EntityManager

+ insert(E) :I
+ update(E) :void
+ delete(E) :boolean
+ findById(I) :E

«Domain Repository»
CustomerRepository

+ selectRebateForCustomer(FlightClass, CustomerNumber) :CustomerRebateComposite

DomainEntityImpl<I>

«Domain Entity»
Customer

DomainAttributeImpl<V>

«Domain Entity Id»
CustomerId

Domain Repository

- CDI-Bean

```
public abstract class DomainRepositoryImpl<E,I> {

  @Inject
  EntityManager entityManager;

  public I insert(E entity)
  { ... this.entityManager.persist(entity) ...
```

Delegation an JPA-Entity-Manager

```
@ApplicationScoped
public class CustomerRepository
  extends DomainRepositoryImpl<Customer, CustomerId> {

  public CustomerRebateComposite selectRebateForCustomer
    (@NotNull CustomerNumber filterCustomerNumber,
     @NotNull FlightClass filterFlightClass)
  { ... new JPAQueryFactory(this.entityManager).select(customer) ...
```

Daten-Abfrage => Selektion mit Projektion

Framework Querydsl

Domain Service (DDD: *service*)

- Kapselung objekt-übergreifender Funktionalität auf Domain Entities und Domain Values
  - Operationen die nicht eindeutig zu einem bestimmten Geschäftsobjekt gehören
    - insbesondere weil sie mehrere Entities umspannen
  - Nutzung von Factories und Repositories per Injektion
  - Nutzung der Methoden von Entities

- Delegation der Geschäftslogik an spezialisierte Service-Klasse
  - Lokalisierung der Geschäftslogik
  - Austauschbarkeit der Implementierung

Domain Service

class Domain Service

«interface»
DomainService

«abstract»
DomainServiceImpl

«Domain Service»
CustomerReportService

+ generateReport(CustomerReportData, CustomerReportTemplate) :void

CustomerReportData

CustomerReportTemplate

Domain Service

- CDI-Bean (statuslos)

```
@ApplicationScoped
public class CustomerReportService extends DomainServiceImpl {

  @Inject
  private CountryRepository repositoryCountry;

  @Inject
  private FtpService ftpService;

  public void generateReport(
    CustomerReportData data, CustomerReportTemplate template)
  {
    ...
```

Nutzung injizierter "Dienste"

Orchestrierung der Geschäftslogik der Domäne

Rich Domain Entity – Injektions-Fähigkeit

- JPA-Entities haben keine Injektions-Fähigkeit
- CDI-Bean-Manager unterstützt programmatische Injektion

- Utility-Klasse (Standard CDI 1.1, vereinfachte Darstellung):

```
public abstract class Injector {

  public static void injectFields(Object entity) {

    BeanManager beanManager = CDI.current().getBeanManager();
    AnnotatedType annotatedType =
      beanManager.createAnnotatedType(entity.getClass());
    InjectionTarget injectionTarget =
      beanManager.createInjectionTarget(annotatedType);
    CreationalContext creationalContext =
      beanManager.createCreationalContext(null);
    injectionTarget.inject(entity, creationalContext);
```

Rich Domain Entity – Anreicherung

- Rich Domain Entities realisieren vielfältige fachliche Funktionalität

```
@Entity
public class Booking extends DomainAggregateRootImpl<BookingId> {

  @Inject
  @Transient
  private BookingPositionRepository repositoryBookingPosition;

  protected Booking() {
    Injector.injectFields(this);
  }

  public void addPassenger(...) {
    ...
    BookingPosition bookingPosition =
      this.factoryBookingPosition.create(...);
    ...
    this.repositoryBookingPosition.insert(bookingPosition);
    ...
```

Injektion benötigter "Domänen-Dienste"

JPA + CDI

Domain Event (DDD: *event*)

- passive Datenstruktur zur Kapselung eines Geschäftsereignisses
  - "etwas" dass in der Domäne passiert
  - Fachexperten und Benutzer interessieren sich für dieses Ereignis

- unveränderbar (immutable)

- starke Entkopplung
  - asynchroner Nachrichtenaustausch möglich
  - anstelle von (stärker koppelnden) Methodenaufrufen

Domain Event

class Domain Event

«interface»
Serializable

«interface»
DomainEvent

«abstract»
DomainEventImpl

«Event Sender»
FlightIsCanceledEventSender

+ send(FlightIsCanceledEvent) :void

«Event Receiver»
FlightIsCanceledEventReceiver

+ receive(FlightIsCanceledEvent) :void

«Domain Event»
FlightWasCanceledEvent

- idFlight :NumberId
- reason :CancellationReason

+ FlightIsCanceledEvent(NumberId, CancellationReason) :FlightIsCanceledEvent
+ getIdFlight() :NumberId
+ getReason() :CancellationReason

«enumeration»
CancellationReason

STORM
PILOTE_STRIKE
OTHER

Domain Event

- Mischform von Domain Value und Composite Domain Object
- Übertragung als CDI-Event oder JMS-Message

```
@AllArgsConstructor
@Getter
@EqualsAndHashCode
@ToString
public final class FlightWasCanceledEvent extends DomainEventImpl {

  @NotNull
  @Valid
  private final FlightId idFlight;

  @NotNull
  private final CancellationReason reason;
}
```

rein deklarative Programmierung

Application Service (DDD: *service*)

- Kapselung der Anwendungsfälle (use case)
  - enthält keine Geschäftslogik, sondern orchestriert diese
  - Nutzung von Services, Factories und Repositories per Injektion
  - Nutzung der Methoden von Entities

- Delegation der Anwendungslogik an spezialisierte Service-Klasse
  - dedizierte (logische) Schnittstelle zwischen Frontend und Backend
  - zentrale Realisierung technischer Querschnittsaufgaben
    - Transaktionen, Exception-Handling, Logging, ...

Application Service

class Application Service

«interface»
ApplicationService

«interface»
BookFlightAppSvc

+ bookFlight(FlightBookingRequest) :FlightBookingResponse

«abstract»
ApplicationServiceImpl

«Application Service»
BookFlightAppSvcImpl

- factoryBooking :BookingFactory
- repositoryBooking :BookingRepository

+ bookFlight(FlightBookingRequest) :FlightBookingResponse

«Composite Domain Object»
FlightBookingRequest

- idFlight :NumberId
- offeredBooking :OfferedBookingComposite
- customerNumber :CustomerNumber
- passengers :List<PassengerComposite>

«Composite Domain Object»
FlightBookingResponse

- idBooking :NumberId
- bookingStateTraceLink :java.net.URI

Application Service

- CDI-Bean (`@ApplicationScoped`, statuslos, `@Transactional`)
- Stateless-EJB

```
@Stateless
@LogException
public class BookFlightAppSvcImpl
  extends ApplicationServiceImpl
  implements BookFlightAppSvc {

  @Inject
  private BookingFactory factoryBooking;

  @Inject
  private BookingRepository repositoryBooking;

  public FlightBookingResponse bookFlight
    (FlightBookingRequest request)
  {
    ...
```

Interceptors für "technischen Rahmen"

Injektion von "Domänen-Diensten"

Orchestrierung der Geschäftslogik der Domäne

Building Block Hierarchy

class Building Block Hierarchy

«Event Receiver»
FlightIsCanceledEventReceiver

trigger

receive

«Application Service»
CreateCustomerReportAppSvc

request

response

«Composite Domain Object»
CustomerComposite

«Domain Event»
FlightWasCanceledEvent

send

«Event Sender»
FlightIsCanceledEventSender

orchestrate

orchestrate

orchestrate

orchestrate

business
data
projection

«Domain Service»
CustomerReportService

«Domain Factory»
CustomerFactory

call

call

«Domain Repository»
CustomerRepository

create / call

call

CRUD

call

«Domain Aggregate Root»
Customer

compose

compose / call

compose / call

compose

«Domain Entity Id»
CustomerId

«Domain Entity»
Contact

relate

«Domain Value»
Address

«Domain Attribute»
CustomerTurnover

Clean Architecture

R. C. Martin: Clean Architecture (Prentice Hall, 2018)

- domain
- application
- adapter

Business Components

Component Based Architecture

gerichteter, azyklischer Graph (DAG)

cmp FIS Component Diagram

«Application System»
Flight Information System (FIS)

«Business Component»
Order

«Business Component»
Sales

«Business Component»
Schedule

«Business Component»
Product

«Business Component»
Equipment

«library»
Fundamentals

+API
+API
+API
+API

Software-Architektur (im Kunden-Projekt)

cmp Software Architecture

Application System

Inbound

Deployment Unit (war)

DDD

Deployment Unit (war)

DDD

Deployment Unit (war)

DDD

Outbound

DB

ESB

ERP

Vielen Dank!

Schulungen zum Thema:

www.gedoplan-it-training.de

Projekt-Beratung und -Unterstützung:

www.gedoplan-it-consulting.de