Modul 06 - Tactical DDD Building Blocks

Die Bausteine des Domain-Driven Design

Lernziele

- Entity, Value Object und Aggregate unterscheiden können
- Aggregate-Regeln verstehen und korrekt anwenden
- Domain Services, Domain Events, Factories und Repositories einordnen
- Die "Primitive Obsession" als Anti-Pattern erkennen
- Alle Building Blocks in Java 17+ idiomatisch umsetzen

Verbindung zum Event Storming

Die gelben Sticky Notes werden jetzt zu Code

Event-Storming-Element	→	Building Block
Aggregate (gelb)	→	Aggregate Root (Klasse mit Invarianten)
Domain Event (orange)	→	Domain Event (Java Record)
Command (blau)	→	Command-Objekt (Java Record)
Policy (lila)	→	Event Handler / Domain Service
Akteur	→	Auslöser eines Use Case

> In der Realität weichen wir davon aber durchaus ab.

Überblick - Tactical DDD Building Blocks

Baustein	Zweck	Java-Umsetzung
Entity	Objekt mit Identität und Lebenszyklus	Klasse mit ID, equals/hashCode by ID
Value Object	Unveränderlicher Wert ohne Identität	Java `record` mit Validierung
Aggregate	Konsistenzgrenze mit Root-Entity	Klasse mit Invarianten + Event-Liste
Domain Service	Zustandslose domänenübergreifende Logik	Klasse ohne State
Domain Event	Fachliches Ereignis (Vergangenheitsform)	Java `record` (immutable)
Factory	Komplexe Erzeugungslogik	Statische Methode oder eigene Klasse
Repository	Collection-ähnlicher Zugriff auf Aggregates	Java Interface (Port)

Was macht eine Entity aus?

- Hat eine eindeutige Identität (ID), die über den Lebenszyklus bestehen bleibt
- Gleichheit wird über die ID bestimmt, nicht über Attribute
- Hat einen Lebenszyklus: Erstellung → Änderung → ggf. Archivierung
- Enthält Geschäftslogik als Methoden

Im Immobilien-CRM sind das z.B. `BrokerageProcess`, `Viewing` oder `Contact` - jeweils identifiziert durch eine `UUID`.

Entity in Java

```
public class Contact {

    private final UUID id;
    private String firstName;
    private String lastName;
    private Emailadresse email;
    private ContactType type; // OWNER, PROSPECT

    public Contact(UUID id, String firstName,
                   String lastName, ContactType type) {
        this.id = Objects.requireNonNull(id);
        this.firstName = Objects.requireNonNull(firstName);
        this.lastName = Objects.requireNonNull(lastName);
        this.type = Objects.requireNonNull(type);
    }
}
```

- Konstruktor validiert alle Pflichtfelder
- Objekt ist nie in einem ungültigen Zustand

Gleichheit über ID

```
public class Contact {
    // ... fields and constructor

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Contact other)) return false;
        return id.equals(other.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }
}
```

- Zwei Kontakte mit gleicher ID sind dasselbe Objekt
- Auch wenn Vorname oder E-Mail sich geändert haben
- `instanceof` Pattern Matching (Java 16+)

Value Objects

Nicht jedes Objekt braucht eine ID. Value Objects beschreiben Eigenschaften, Maße oder Konzepte - ohne eigene Identität.

- Gleichheit durch Wertevergleich aller Attribute
- Unveränderlich (immutable) - Änderung erzeugt neues Objekt
- In Java 17+: ideal als Record umsetzbar

Value Object	Beschreibt
`Address`	Straße, PLZ, Ort
`AskingPrice`	Betrag + Währung
`Commission`	Prozentsatz

Value Object als Java Record

```
public record Address(String street, String postalCode, String city) {

    // Compact constructor: validation
    public Address {
        Objects.requireNonNull(street, "Street must not be null");
        Objects.requireNonNull(postalCode, "Postal code must not be null");
        Objects.requireNonNull(city, "City must not be null");
        if (!postalCode.matches("\\d{5}")) {
            throw new IllegalArgumentException("Invalid postal code: " + postalCode);
        }
    }
}
```

- Record = automatisch immutable + `equals()` / `hashCode()` by value
- Compact Constructor für Validierung - kein new-Keyword im Body
- Keine Getter-Boilerplate: `address.postalCode()` statt `address.getPostalCode()`

Primitive Obsession - ein Anti-Pattern

Exzessiver Einsatz von Primitives statt Value Objects gilt als Anti Pattern im Domain Driven Design.

```
public class BrokerageProcess {
    private String ownerId;                 // Welches Format?
    private double purchasePrice;           // Welche Währung?
    private double commission;              // Prozent oder absolut?
    private String street, postalCode, city; // Braucht man immer zusammen.
}
```

Value Objects statt Primitives

```
public class BrokerageProcess {
    private UUID ownerId;
    private AskingPrice askingPrice;
    private Commission commission;
    private Address address;
}
```

Value Objects sind typsicher und drücken ihre Rolle durch ihren Typ aus.

Entity vs. Value Object - Entscheidungshilfe

Kriterium	Entity	Value Object
Identität	Ja (ID)	Nein
Gleichheit	Per ID	Per Wert
Veränderlich	Ja	Nein (immutable)
Lebenszyklus	Ja	Nein - wird ersetzt
Java-Umsetzung	Klasse	Record

> Faustregel: Im Zweifel Value Object bevorzugen!
>
> Nur wenn ein Objekt über die Zeit getrackt werden muss, dann Entity.

Aggregates und ihre Grenzen

Ein Aggregate ist ein **Cluster von Entities und Value Objects mit einer Root-Entity**. Es bildet eine **Konsistenzgrenze**: Invarianten innerhalb eines Aggregats werden sofort garantiert, zwischen Aggregates gilt Eventual Consistency.

Die **Aggregate Root** ist der einzige Einstiegspunkt von außen.
Sie kontrolliert alle Änderungen, hat eine global eindeutige ID und stellt sicher, dass das Aggregat immer in einem gültigen Zustand ist.

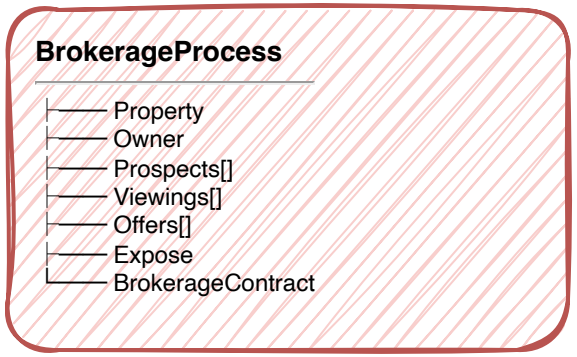
Aggregate-Regeln - Die 7 Gebote

1. Nur die Root ist von außen referenzierbar
2. Innere Objekte dürfen nicht direkt weitergegeben werden (→ Kopie oder Unmodifiable)
3. Cross-Aggregate-Referenz nur per ID, nie per Objektreferenz
4. Innerhalb eines Aggregats: Immediate Consistency
5. Zwischen Aggregates: Eventual Consistency (über Domain Events)
6. Aggregates sollten klein gehalten werden
7. Ein Repository pro Aggregate - nie für innere Entities

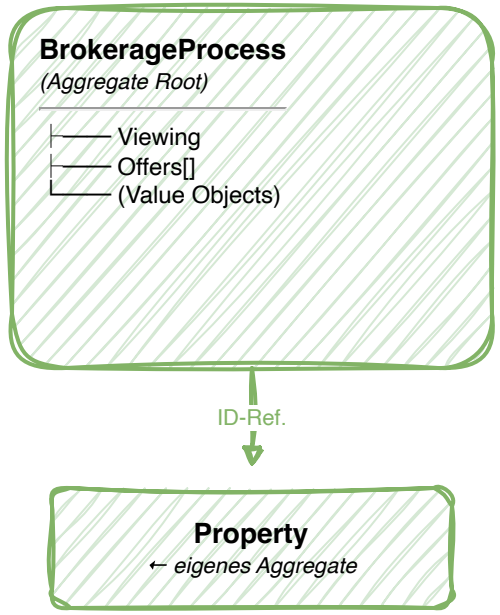
Aggregate Sizing - Wie groß ist richtig?

Klein anfangen!

Zu groß:



Gut geschnitten:



Sizing: Faustregel

- Entities, die nur zusammen mit der Root geändert werden → ins Aggregate
- Entities, die eigenständig geladen werden müssen → eigenes Aggregate
- Wenn in Zweifel → kleineres Aggregate, verbunden per ID

Codebeispiel

```
public class BrokerageProcess {

    private final UUID id;
    private final UUID propertyId; // Reference by ID!
    private AskingPrice askingPrice;
    private ProcessStatus status;
    private final List<Viewing> viewings = new ArrayList<>();
    private final List<Offer> offers = new ArrayList<>();
    private final List<BrokerageEvent> domainEvents = new ArrayList<>();

    public UUID addViewing(
            String prospect, LocalDateTime timestamp) {
        var viewing = new Viewing(
            UUID.randomUUID(), prospect, timestamp);
        this.viewings.add(viewing);
        this.status = ProcessStatus.VIEWING;
        return viewing.getId();
    }
}
```

Invarianten schützen

```
public class BrokerageProcess {

    public void setStatusToNotaryAppointment() {
        boolean hasAcceptedOffer = offers.stream().anyMatch(Offer::isAccepted);

        if (!hasAcceptedOffer) {
            // Invariante: Kein Notartermin ohne angenommenes Angebot
            throw new IllegalStateException("Notary appointment requires an accepted offer");

        }
        this.status = ProcessStatus.NOTARY_APPOINTMENT;
        domainEvents.add(new NotaryAppointmentScheduled(this.id));
    }

    // Außenstehende können Liste nicht modifizieren, weil unmodifiable
    public List<Viewing> getViewings() {
        return Collections.unmodifiableList(viewings);
    }
}
```

Domain Event Collection Pattern: Events sammeln

```java
public class BrokerageProcess {

    private final List<BrokerageEvent> domainEvents = new ArrayList<>();

    // Wird von Business-Methoden aufgerufen
    protected void registerEvent(BrokerageEvent event) {
        this.domainEvents.add(event);
    }

    public List<BrokerageEvent> getDomainEvents() {
        return Collections.unmodifiableList(domainEvents);
    }

    public void clearDomainEvents() {
        this.domainEvents.clear();
    }
}
```

Domain Services

Manche Geschäftslogik passt in keine Entity und kein Value Object.
Für diese Fälle gibt es **Domain Services**: zustandslos, in der Domain-Schicht angesiedelt, oft über Aggregate-Grenzen hinweg operierend.

Abgrenzung von Domain zum Application Service

	Domain Service	Application Service
Schicht	Domain	Application
Enthält	Geschäftslogik	Orchestrierung
Zustand	Stateless	Stateless
Spring	Kein Spring nötig	`@Service`, `@Transactional`
Beispiel	Provisionsberechnung	ScheduleViewingUseCase

Beispiel: CommissionCalculator

```
// Diese Klasse kommt ohne Abhängigkeiten zu Spring aus
public class CommissionCalculator {

    public Commission calculate(AskingPrice price,
                                BigDecimal commissionRate,
                                SplitModel model) {
        var amount = price.amount()
            .multiply(commissionRate)
            .divide(BigDecimal.valueOf(100), 2, RoundingMode.HALF_UP);

        return switch (model) {
            case BUYER_PAYS_ALL ->
                new Commission(amount, BigDecimal.ZERO);
            case FIFTY_FIFTY ->
                new Commission(amount.divide(BigDecimal.TWO), amount.divide(BigDecimal.TWO));
            case SELLER_PAYS_ALL ->
                new Commission(BigDecimal.ZERO, amount);
        };
    }
}
```

Domain Events - Fachliche Ereignisse

Domain Events beschreiben Dinge, die in der Domäne passiert sind.
Immer in der **Vergangenheitsform, immer unveränderlich**.

Sie enthalten alle relevanten Daten des Ereignisses und ermöglichen
lose Kopplung - sowohl intern als auch als Basis für spätere Integrationsereignisse.

Beispiele: `ViewingCompleted`, `OfferAccepted`,
`BrokerageCompleted`

Domain Event als Record

```
public record ViewingCompleted(
    UUID brokerageProcessId,
    UUID viewingId,
    LocalDateTime timestamp
) {
    public ViewingCompleted {
        Objects.requireNonNull(brokerageProcessId);
        Objects.requireNonNull(viewingId);
        Objects.requireNonNull(timestamp);
    }
}
```

Records bieten sich an, weil sie unverändlich und kompakt zu definieren sind.

Interne Domain Events vs. öffentliche Integrations-Events

> Nicht jedes Domain Event ist ein Integrations-Event!

Aspekt	Domain Event (intern)	Modul-/Integrations-Event (öffentlich)
Zweck	Fachliche Änderung im Modell signalisieren	Andere BCs/Module informieren
Gültigkeit	Innerhalb des BC	Über die BC-/Modulgrenze
Typen	Domain-nahe Typen möglich	Primitive/stabile Vertragstypen bevorzugt
API-Status	Internes Modellartefakt	Öffentlicher Vertrag
Einführung im Kurs	Dieses Modul	Modul 12/13

Factory - Erzeugung komplexer Objekte

Als statische Factory-Methode auf dem Aggregate Root

```
public class BrokerageProcess {

    // Private constructor
    private BrokerageProcess(UUID id, UUID propertyId,
            Address address, AskingPrice price, Commission commission) {
        this.id = id;
        this.propertyId = propertyId;
        this.address = address;
        this.askingPrice = price;
        this.commission = commission;
        this.status = ProcessStatus.NEW;
    }

    public static BrokerageProcess create(
            UUID propertyId, Address address,
            AskingPrice price, Commission commission) {
        var process = new BrokerageProcess(
            UUID.randomUUID(), propertyId, address, price, commission);
        process.registerEvent(new BrokerageStarted(process.id));
        return process;
    }
}
```

- Privater Konstruktor → Erstellung nur über Factory-Methode
- Event wird direkt bei Erstellung registriert
- Objekt ist sofort in einem gültigen Zustand

Das Repository als Port

Repositories abstrahieren die **Persistenz**:

Die Domäne arbeitet mit einer Schnittstelle, **ohne die Datenbank zu kennen**.

Definiert wird das Interface in der Domain-Schicht (Port), implementiert in der Infrastructure-Schicht (Adapter).

Repository-Interface

```
// Domain layer: pure Java, no Spring!
public interface BrokerageProcessRepository {

    Optional<BrokerageProcess> findById(UUID id);

    void save(BrokerageProcess process);

    void deleteById(UUID id);
}
```

- Kein `JpaRepository`, keine Spring-Abhängigkeit!
- Spricht die Ubiquitous Language: `save`, `findById`
- Rückgabetyp: Domain-Objekt, nicht JPA-Entity

ID-Strategien

Wie generiert man Aggregate-IDs?

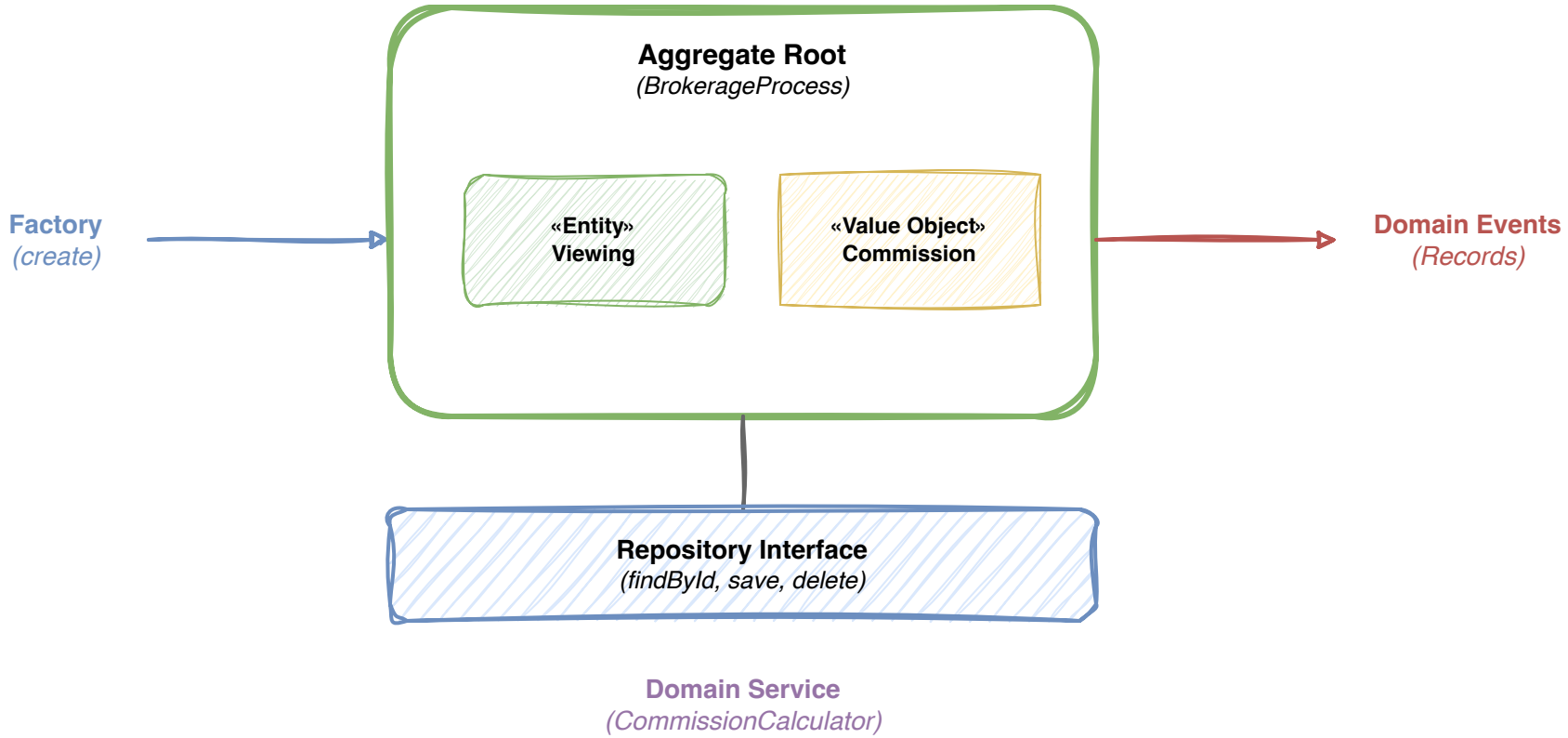
Strategie	Vorteile	Nachteile
UUID.randomUUID()	Einfach, keine DB nötig, verteilt	Nicht sortierbar, 36 Chars
UUIDv7 (zeitbasiert)	Sortierbar + einzigartig	Java-Library nötig
DB-Sequence	Kompakt, sortierbar	Kopplung an DB
Fachliche ID	Lesbar (`IMM-2024-0042`)	Eindeutigkeit schwerer sicherbar

Empfehlung für diesen Workshop

```
UUID id = UUID.randomUUID();
```

> ID wird im Domain Layer erzeugt (Factory-Methode), nicht von der Datenbank vergeben.

Zusammenspiel der Building Blocks



Hands-on: Lab 04

Building Blocks im Immobilien-CRM implementieren

Zusammenfassung

- Entity = Identität + Lebenszyklus, Gleichheit per ID
- Value Object = immutabel, Gleichheit per Wert → Java Record
- Aggregate = Konsistenzgrenze, Zugriff nur über Root, klein halten!
- Domain Service = zustandslose Geschäftslogik (kein Spring nötig)
- Domain Event = was passiert ist, immutabel, Record
- Factory = garantiert gültigen Initialzustand
- Repository = Java Interface in der Domain-Schicht (Port)
- Primitive Obsession vermeiden → Value Objects nutzen!

> Im nächsten Modul überführen wir diese Building Blocks
> in eine Clean Architecture (Modul 07).