MTAT.03.229
Enterprise System Integration

Lecture 3

**DDD: Aggregates, Repositories, Identifiers, and Factories**

Orlenys López-Pintado

University of Tartu

Slides by Marlon Dumas and Luciano García-Bañuelos
Hexagonal architecture

Last week: Domain model
- Structural patterns:
  - Entities, Value objects and Aggregates
- Data access abstraction
  - Repositories and Specifications

This week:
- Aggregate as the unit for structuring applications
- Explicit Identifiers
- Factories
- Validation
Rentit’s domain model
More on Aggregates

• As the domain model grows, the underlying interdependencies become complex
  ◦ How to cope with transactions? What about concurrency and distribution?

• Aggregates represent a logical partitioning of the domain model and exhibit our understanding of the system interdependencies
  ◦ Boundaries for consistency
  ◦ As so, they provide natural boundaries for transaction management, distribution, etc.

• Rules of thumb
  ◦ Keep all interaction with the aggregate happen only via the aggregate root
  ◦ Reference other aggregates by identity
  ◦ Keep control on object’s life-cycle
  ◦ Model invariants inside consistency boundaries
    ◦ We may check invariants after every change
Let no association go straight to the aggregate’s inner classes. Every interaction must pass through the aggregate root class!
Use identifiers to external entities
Entity identifiers

Having an ad-hoc, reified form of identifier may become handy when working with distributed applications

• Common identity creation strategies:
  ◦ User provides unique values as input to the application (e.g. username)
  ◦ Application internally generates identifiers using algorithms that ensure uniqueness
  ◦ Application relies on persistence infrastructure to generate identifiers
  ◦ Another system, application or bounded context generate identifiers
Examples of identifier generation

Random string

```java
public PlantInventoryEntryID nextIdentifier() {
    String newID = java.util.UUID.randomUUID().toString();
    return new PlantInventoryEntryID(newID);
}
```

Oracle + Hibernate

```java
public PlantInventoryEntryID nextIdentifier() {
    Long newID = (Long) this.session().
        createSQLQuery("select plant_inventory_entry_seq.nextval " +
                        "as plant_id from dual")
        .addScalar("plant_id", Hibernate.LONG)
        .uniqueResult();
    return new PlantInventoryEntryID(newID);
}
```
Aggregate’s lifecycle: Object creation

• Several well-known creational patterns: Factory method, Abstract factory and Build patterns described by GoF

• Where should we place factories:
  ◦ Factory method on Aggregate root?
  ◦ Abstract factory as a domain service?
  ◦ Internal objects are created as a side effect of domain model methods?

• What is the minimal set of data required at construction time?
Identify the minimal required data

```java
<<interface>>
SalesAggregateFactory

createPurchaseOrder(CustomerID, PlantInventoryEntryID, ContactPerson, Address, BusinessPeriod)
```
@Entity @Getter

public class PurchaseOrder {
    @Enumerated(EnumType.STRING)
    POSTatus status;
    .... //attributes defined here

protected PurchaseOrder() {}}

public static PurchaseOrder of(PurchaseOrderID id, PlantInventoryEntryID plantId, ...) {
    PurchaseOrder po = new PurchaseOrder();
    po.id = id;
    po.plantId = plantId;
    ...
    po.status = POSTatus.PENDING;
    ... // call the PO validator
    return po;
}

}
Validation

• We should be just-enough paranoid
  ◦ Protecting the domain model against invalid data coming from outside
  ◦ Report any inconsistency found in input data or any state reached within the aggregate
  ◦ The use of assertion-like validation rules document assumptions (current understanding of the domain)

• Again, one important question to ask is where to put validation

• Let us also ask when we are supposed to validate the aggregates
public class PurchaseOrderValidator implements Validator {

    public void validate(Object object, Errors errors) {
        PurchaseOrder po = (PurchaseOrder) object;

        if (po.getId() == null)
            errors.rejectValue("id", "Purchase Order id cannot be null");
        if (po.getPlantId() == null)
            errors.rejectValue("plantId", "Plant id cannot be null");

        if (po.getStatus().equals(POStatus.PENDING_CONFIRMATION)) {
            if (po.getReservationId() == null)
                errors.rejectValue("reservationId", "Purchase order's ...");
        }

        errors.pushNestedPath("rentalPeriod");
        ValidationUtils.invokeValidator(periodValidator, po.getRentalPeriod(), errors);
        errors.popNestedPath();
    }
}
DDD is also a matter of coding style
Module pattern

Let the code organization speak about the domain!
Intention-revealing interfaces Pattern

```java
@Entity
public class PurchaseOrder {

    @Enumerated(EnumType.STRING)
    @Enumerated(EnumType.STRING)
    POStatus status;

    protected PurchaseOrder() {
    }

    public static PurchaseOrder of(PurchaseOrderID id, PlantInventoryEntryID plantId, ...) {
        PurchaseOrder po = new PurchaseOrder();
        po.id = id;
        po.plantId = plantId;
        ...
        po.status = POStatus.PENDING;
        return po;
    }

    public static PurchaseOrder of(Long a, Long b, LocalDate c, LocalDate d) {
        PurchaseOrder po = new PurchaseOrder();
        po.id = a;
        po.plantId = b;
        ...
        po.status = POStatus.PENDING;
        return po;
    }
}
```
Side-effect-free functions pattern

Place as much of the logic of program as possible in operations that return results with no observable side effects

```java
public void updateTotalCost(BigDecimal price) {
    int numberOfWorkingDays = 0;
    for (LocalDate date = rentalPeriod.getStartDate(); date.isBefore(rentalPeriod.getEndDate()); date = date.plusDays(1)) {
        // All the logic to compute the number of working days...
    }
    total = price.multiply(BigDecimal.valueOf(numberOfWorkingDays));
}
```

Refactor this code (to BusinessPeriod)