MTAT.03.229
Enterprise System Integration
Lecture 2: Domain model and data access layer

Luciano García-Bañuelos
University of Tartu
The Anatomy of an Enterprise System

Application layers

• Users interact with the system through a **presentation layer** (aka **user interface** or UI)

• The **application logic** (aka business logic) determines what the system actually does:
  ◦ Enforces business rules
  ◦ Coordinates business processes

• The **data access layer** facilitates the access to persistent data manipulated by the application logic.
  ◦ Includes access to databases, search engines, document managers and/or a file system.
Hexagonal architecture

- Allows many disparate clients to interact with the system:
  - The **Domain model** is only accessible via an internal Application API (inner hexagon)
  - Need a new client (with different integration requirements)? Just add an **Adapter**
  - Use also adapters to integrate with external applications and infrastructure services
Domain-driven design (DDD)

• Domain-driven design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts.

• Its premises are:
  ◦ A development project must primarily focus in the core domain and domain logic
  ◦ The model is central to any complex design
  ◦ The development relies on creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem
The Framework of DDD

• Ubiquitous language

• Strategic design
  ◦ Bounded Context
  ◦ Context Maps
  ◦ Patterns: Shared kernel, Anti-corruption layer, etc.

• Tactic design
  ◦ Structural patterns: Entities, Value Objects and Aggregates
  ◦ Life cycle patterns: Repositories and Factories
  ◦ Behavior patterns: Services, Domain events, Event sourcing (CQRS)
  ◦ Style patterns: Modules, Intention Revealing Interfaces, Specifications
So when we change the customs clearance point, we need to redo the whole routing plan.

Right. We’ll delete all the rows in the shipment table with the cargo id. Then, we’ll pass the origin, destination and new clearance point into the Routing service, and it will repopulate the table.

Delete the rows? OK, whatever. Anyway, if we didn’t have a customs clearance point at all before, we’ll have to do the same thing.

A language structured around the application domain
- Understood by domain experts and software development team
- Essential in communicating insights from the domain that must be transferred into the model.
Strategic design

• In order to tackle complexity, the domain is decomposed in subdomains, usually based on Business capabilities
  ◦ Identify the core subdomain and other supporting subdomains
  ◦ Characterize the of relationships between subdomains

• The usage of the ubiquitous language used in the core domain delimits a bounded context
  ◦ Use the ubiquitous language in the process of modeling the domain
  ◦ Strive to keep the consistency of the domain model with the code
  ◦ In fact, the code itself is another representation of the model

• We need to identify and describe the nature of the relationships between bounded contexts by means of Context Maps
  ◦ Teams working in different context bounds should take care (or at least aware of) potential interdependencies
Rentit’s domain

Sub-domain

Domain

- Reservations
  - Plant reservation
- Transportation
  - Plant transportation
- Invoicing
- Maintenance
  - Maintenance plan
- Sales (core)
  - Plant catalog and purchase orders

Bounded Context

DDD & DATA ACCESS   LUCIANO GARCÍA-BAÑUELOS 7
Exercise: Rentit

- **Sales**
  - Querying the plant catalog
  - Creating a purchase order
  - Processing of a purchase order
  - Plant transportation
  - Invoicing

- **Maintenance**
  - Creation of yearly maintenance plan
  - Scheduling of maintenance tasks

- **Reservations**
  - How to coordinate rental and maintenance (implicit)
Customers query Rentit’s plant catalog. For each plant, the plant catalog provides a name, a description, and a daily rental price. Each entry in the plant catalog is referred to as plant inventory entry.

There might be multiple plants for the same entry, each one having a different serial number. An actual plant is referred to as a plant inventory item.
Creating a Purchase Order

A **customer** can **check a plant's availability** for a period of time. If the plant is available, he/she can **create a Purchase Order** with the following information: customer’s company, contact person (e.g. name, email), construction site’s address, **plant inventory entry**’s identifier, the start and end dates of the rental period.
A PO is initially considered as created. A clerk checks the customer’s credit status with the department of finances, and the plant's availability. If everything is OK, the clerk notifies the customer that the PO has been accepted and updates the cost of the PO. Otherwise, the clerk notifies the rejection of the PO with an explanatory note.
The overall picture
Tactic Design

Repositories

Entities

Value objects

Factories

Aggregates

Modules

Model driven design

Services

Value objects

Entities

Repositories

Aggregates

Factories

access with

maintain integrity with

act as root of

encapsulate with

encapsulate with

encapsulate with

access with

encapsulate with
**Building blocks in the Model**

<table>
<thead>
<tr>
<th>Entity</th>
<th>Value Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Have identity that must be preserved all along their life-cycle</td>
<td>Things within the model that are not uniquely identified</td>
</tr>
<tr>
<td>Distinguished from other similar objects by their identity and not</td>
<td>A pair of value objects are equal if the value of their properties match</td>
</tr>
<tr>
<td>based in their properties</td>
<td></td>
</tr>
<tr>
<td>Their properties are mutable</td>
<td>Are immutable, i.e. replace rather than update</td>
</tr>
<tr>
<td>Involve structure and behavior (encapsulate domain logic)</td>
<td>Involve also structure and some additional validation logic</td>
</tr>
</tbody>
</table>
Back to our running example

Classify the classes in your domain model as Entities or Value objects

- In the figure, entities are shown in grey and value objects in blue

About associations:
- Mind traversal directions
- Eliminate nonessential associations
- Uncover composition relations (strong life cycle dependencies)
Aggregates

• An aggregate is a cluster of entities and value objects, treated as one single unit
  ◦ It has one root entity that is known as the **Aggregate Root**
  ◦ External objects interact with the aggregate only via the aggregate root
  ◦ As a result of the above
    ◦ Any required change on the internal objects is always mediated by the aggregate root!
    ◦ The aggregate root is responsible for ensuring aggregate invariants

• Note that any entities/value object within an aggregate can refer to other external entities
Aggregates in our running example
More on aggregates

- <<Aggregate Root>>
  - Engine
  - Car
    - Wheel
    - Tire
    - Position

- Customer

- PlantInventoryItem
  - condition: EquipmentCondition

- PlantReservation
  - bookPlantForRental(PlantInventoryItem BusinessPeriod)

- Ensures a plant is not reserved for two or more overlapping periods

- Ensures a plant is not reserved for rental unless the plant is serviceable
Repositories

• A repository is an abstraction over the persistence support
  ◦ Provides basic operations for storing/retrieving objects of an aggregate
  ◦ Promotes a clean separation between domain and the technical issues related to persistence management
  ◦ Defines a unit of work
    ◦ A repository is usually associated with one single data source
    ◦ In case of multiple data sources, the transaction boundary should be the aggregate itself

• Although not a rule, we will prefer to use a single repository per aggregate
  ◦ Not always appropriate: How can we check the availability of plants?
Example repository

```
<inteface>
PlantCatalog

fetchAllPlants()
findPlantsByName(string)
findAvailablePlants(PlantInventoryEntry, BusinessPeriod)
```

PlantInventoryEntry
- name: string
- description: string
- dailyPrice: Money

PlantInventoryItem
- condition: EquipmentCondition

PlantReservation
- bookPlantForRental(PlantInventoryItem, BusinessPeriod)

BusinessPeriod
- startDate: Date
- endDate: Date
Specification pattern

```
components * Specification

expr 1

CompositeSpecification LeafSpecification NegationSpecification

DisjunctionSpecification ConjunctionSpecification
```
public List<Invoice> selectSatisfying(InvoiceSpecification spec) {
    List<Invoice> result = new ArrayList();
    for (Invoice invoice: invoiceRepo.findAll())
        if (spec.isSatisfiedBy(invoice)) result.add(invoice);
    return result;
}

invoices = invoiceRepository.selectSatisfying(new DelinquentInvoice());
Domain model and Data access
Simple Mapping

```java
@Entity
public class Plant {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    @Column(name="DESC")
    private String description;
    private Float price;
    @Lob
    private Image photo;
}
```
Mapping of associations

@Entity
public class PurchaseOrderLine {
    @ManyToOne
    private Plant plant;
}

<table>
<thead>
<tr>
<th>PurchaseOrder</th>
<th>PurchaseOrderLine</th>
<th>Plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>contact: String</td>
<td>start: Date</td>
<td>name: String</td>
</tr>
<tr>
<td>total: Float</td>
<td>end: Date</td>
<td>description: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PURCHASEORDERLINE</th>
<th>PLANT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>NAME</td>
</tr>
<tr>
<td>STARTD</td>
<td>ENDD</td>
</tr>
<tr>
<td>ID</td>
<td></td>
</tr>
</tbody>
</table>

Plant name: String
String description: String
Float price: Float
@Entity
public class PurchaseOrder {

    @OneToMany(cascade = CascadeType.ALL)
    private Set<PurchaseOrderLine> poLines = new HashSet<PurchaseOrderLine>();
}

eenum CascadeType {
    ALL, MERGE, PERSIST, REFRESH, REMOVE
}
Querying domain objects
Basic querying with Spring Data repositories

@Repository
public interface PlantRepository extends JpaRepository<br />List<Plant> findByNameLikeAndPriceBetween(String name, Float low, Float hi);

<table>
<thead>
<tr>
<th>Textual operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
</tr>
<tr>
<td>Or</td>
</tr>
<tr>
<td>Between</td>
</tr>
<tr>
<td>LessThan</td>
</tr>
<tr>
<td>GreaterThan</td>
</tr>
<tr>
<td>After</td>
</tr>
<tr>
<td>Before</td>
</tr>
<tr>
<td>IsNull</td>
</tr>
<tr>
<td>IsNotNull/NotNull</td>
</tr>
<tr>
<td>Like</td>
</tr>
<tr>
<td>NotLike</td>
</tr>
<tr>
<td>StartingWith</td>
</tr>
<tr>
<td>EndingWith</td>
</tr>
<tr>
<td>Containing</td>
</tr>
<tr>
<td>OrderBy</td>
</tr>
<tr>
<td>Not</td>
</tr>
<tr>
<td>In</td>
</tr>
<tr>
<td>NotIn</td>
</tr>
<tr>
<td>True</td>
</tr>
<tr>
<td>False</td>
</tr>
</tbody>
</table>

@Repository
public interface PlantRepository extends JpaRepository<br />List<Plant> finderMethod(String name, Float low, Float hi);

DDD & DATA ACCESS LUCIANO GARCÍA-BAÑUELOS
QueryDSL

public class InventorySpecifications {
    static QPlantInventoryEntry plant = QPlantInventoryEntry.plantInventoryEntry;
    static QPlantReservation reservation = QPlantReservation.plantReservation;

    public static BooleanExpression isAvailableFor(BusinessPeriod period) {
        return plant.notIn(
            new JPASubQuery().from(reservation)
                .where(...
                )
                .list(reservation.plant));
    }

    public static BooleanExpression nameContains(String keyword) {
        return plant.name.lower().contains(keyword.toLowerCase());
    }
}

plantRepo.findAll(nameContains("Mini").and(isAvailableFor(period)))