MTAT.03.229
Enterprise System Integration

Lecture 2: Domain model and data access layer

Orlenys López Pintado
University of Tartu

Slides by LUCIANO GARCÍA-BAÑUELOS
Classical Three-Tier Architecture

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.
Issues with 3-tier architecture

• What if I need to access the application via mobile devices?
• What if I need to access the application via a chat or voice-to-text interface?
• What if I want to access the application logic from another system?
• What if I change the data layer? For example:
  ◦ Add or modify logging
  ◦ Add or modify caching
  ◦ Get user data from LDAP
  ◦ Change data storage system, e.g. move some data to NoSQL
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
Principles of Hexagonal Architecture

• Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.
  ◦ Alistair Cockburn

• Simple Example:
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
  ◦ We define a common language / vocabulary
  ◦ We may capture it as a domain model

• 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.
Running Example

• Throughout the course, we will use a scenario of a construction company (called BuildIT) which hires construction equipment (so-called “plants”) from another company (called RentIT).


• We’ll start by building a system that supports the order-to-cash process of RentIT
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-domains

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

Sub-domain

Bounded Context
Example: Let’s build Rentit’s model

• 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.

Note: In Domain-Driven Design, we will allow ourselves to specify operations in addition to attributes, e.g. fetchPlants().
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.
Processing a Purchase Order

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
## 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 based in their properties</td>
<td>A pair of value objects are equal if the value of their properties match</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
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

```java
<interface>
    PlantCatalog
</interface>

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
```
Homework 1

• Refactor an initial piece of RentIT’s domain model to introduce value objects
• Map the domain model to JPA classes
• Identify and specify the repositories
• Write tests
Initial domain model