MTAT.03.229
Enterprise System Integration
Lecture 3: Services and Presentation layer

Luciano García-Bañuelos
University of Tartu
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 of work
- Services as a means to coordinate the integration of aggregates
- Adapters as a way to integrate with external actors
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
Factory method on classes

```java
@Entity @Getter
public class PurchaseOrder {

    @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;
    }
}
```
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
Validation via “self-encapsulation”

- Every access, even from within the same class, goes through accessor methods

```java
public class EmailAddress {
    private String address;

    public static EmailAddress of(String anAddress) {
        EmailAddress newAddress = new EmailAddress();
        newAddress.setAddress(anAddress);
        return newAddress;
    }

    private void setAddress(String anAddress) {
        if (anAddress == null)
            throw new IllegalArgumentException("The email address is required.");

        if (Pattern.matches("\w+([-]+)\w+@\w+.\w+", anAddress))
            throw new IllegalArgumentException("Email address format is invalid.");

        this.address = anAddress;
    }
}
```
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.status.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();
    }
}
Validation in different contexts

// When creating a Purchase Order
PurchaseOrder po = PurchaseOrder.of(...);

DataBinder binder = new DataBinder(po);
binder.addValidators(new PurchaseOrderValidator());
binder.validate();

if (!binder.getBindingResult().hasErrors())
    orderRepo.save(po);
DDD is also a matter of coding style
Module pattern

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

```java
@Entity
@Getter
public class PurchaseOrder {

@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)
Presentation layer
The picture

- How to organize the code that implements the interactions with end-users?

- MVC Pattern
  - Model: consists of application data and business logic
  - View: any output representation of data (e.g., Chart diagrams, GUIs Java swing, HTML pages)
  - Controller: Mediates user interaction, converting it to commands for the model or view

- Worth asking:
  - Layered architecture?
  - Logical boundaries?
A closer look to MVC

• The MVC pattern has its roots on the early attempts to developing Graphical User Interfaces
  ◦ Introduced in Smalltalk-76, in Xerox Park, in the 70’s
  ◦ Re-implemented later in Smalltalk-80’s class library, in the 80’s

**Model**
- Encapsulates application state
- Responds to state queries
- Exposes application functionality
- Notifies views changes

**View**
- Renders the models
- Requests updates from models
- Sends user gestures to controller
- Allows controller to select view

**Controller**
- Defines application behavior
- Maps user actions to model updates
- Selects view for response
- One for each functionality

Source: Oracle
Java Web Applications

Client
(e.g., Web browser)

Web Server
Servlet container
Servlet

URL
http://example.com/app?name=Adam

HTTP request
Get
Post
Update
Delete

HTTP response
200, 201, 404, 500, etc.

+ content (e.g., HTML, XML, JSON)
Hello world, Servlets!

```java
public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        String name = request.getParameter("name");

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        out.println("<HTML>");
        out.println("<HEAD><TITLE>Hello world</TITLE></HEAD>" );
        out.println("<BODY><H1>Hello " + name + "</H1></BODY>" );
        out.println("</<HTML>" );
    }
}
```

http://example.com/app?name=Adam
MVC revisited: The Struts 2 architecture

Client

HTTP request

Web Server

JSP/Servlet container

Dispatcher filter

Interceptor

Servlet

Servlet

Servlet

Action

View

compiled into

Model

Model

Model

View

View

View

Servlet

Servlet

Servlet

Servlet

Servlet

Servlet

Servlet

Servlet
Spring MVC architecture

Client

HTTP request

Dispatcher

Model

View

View logical name

Model

View logical name

URL

Handler mapping

Controller

View resolver

Web Server

Spring MVC container

HTTP request

HTTP response
Hexagonal architecture
The service layer (module)

```java
@Service
public class SalesService {
    @Autowired
    PurchaseOrderRepository orderRepo;
    @Autowired
    PlantInventoryEntryRepository plantRepo;
    @Autowired
    SalesIdentifierFactory idFactory;

    public void createPO(PlantInventoryEntryDTO plantDTO, BusinessPeriodDTO periodDTO) {
        PurchaseOrder po = PurchaseOrder.of(
            idFactory.nextPurchaseOrderID(),
            PlantInventoryEntryID.of(plantDTO.getId()),
            periodDTO.asBusinessPeriod());

        DataBinder binder = new DataBinder(po);
        binder.addValidators(new PurchaseOrderValidator());
        binder.validate();

        if (!binder.getBindingResult().hasErrors())
            orderRepo.save(po);
    }
}
```

Validation code
@Controller
@RequestMapping("/dashboard")
public class DashboardController {
    @Autowired
    PlantCatalogService plantCatalog;
    @Autowired
    SalesService salesService;

    @RequestMapping("/catalog/form")
    String getQueryForm(Model model) {
        model.addAttribute("catalogQuery", new CatalogQueryDTO());
        return "dashboard/catalog/query-form";
    }

    ...
}

@Controller
@RequestMapping("/dashboard")
public class DashboardController {

...
@RequestMapping("/catalog/query")
String executeQuery(CatalogQueryDTO query, Model model) {
    ...
    return "dashboard/catalog/query-result";
}

@RequestMapping("/orders")
String createPO(PurchaseOrderDTO poDTO, Model model) {
    ...
    return "redirect:/dashboard";
}
}
Domain object & Data Transfer Object

```java
@Entity
@Getter
public class PlantInventoryEntry {
    @Id
    private String id;
    private String name;
    private String description;
    @Column(precision=8, scale=2)
    private BigDecimal price;
}

@Data
public class PlantInventoryEntryDTO {
    private String _id;
    private String name;
    private String description;
    private BigDecimal price;
}
```

```java
@Service
public class PlantInventoryEntryAssembler {

    public PlantInventoryEntryDTO toResource(PlantInventoryEntry plant) {
        PlantInventoryEntryDTO dto = new PlantInventoryEntryDTO();
        dto.set_id(plant.getId());
        ...
        return dto;
    }

    public List<PlantInventoryEntryDTO> toResources(List<PlantInventoryEntry> plants) {
        return plants.stream().map(p -> toResource(p)).collect(Collectors.toList());
    }
}
```