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
Aggregate as unit of consistency

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

Diagram:

- Customer
  - CustomerID
    - customer
  - ContactPerson
    - contact
- PlantReservation
  - PlantReservationID
    - reservation
  - PlantInventoryEntry
    - PlantInventoryEntryID
    - plant
- PurchaseOrder
  - rentalPeriod
- Address
  - siteAddr
  - notes
- Comment
- BusinessPeriod
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

<<interface>>
SalesAggregateFactory

createPurchaseOrder(CustomerID, PlantInventoryEntryID, ContactPerson, Address, BusinessPeriod)
Factory method on classes

```java
@Entity @Getter
public class PurchaseOrder {
    @Enumerated(EnumType.STRING)
    POStatus status;

    protected PurchaseOrder() {} // Protected constructor

    public static PurchaseOrder of(PurchaseOrderId id, PlantInventoryEntryId plantId, ...) {
        PurchaseOrder po = new PurchaseOrder();
        po.id = id;
        po.plantId = plantId;
        ... // Other fields
        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+\..*", anAddress))
            throw new IllegalArgumentException("Email address format is invalid.");

        this.address = anAddress;
    }
}
```
Explicit validator: Spring validation

```java
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 ..."blurp");
            }
        }
        errors.pushNestedPath("rentalPeriod");
        ValidationUtils.invokeValidator(periodValidator, po.getRentalPeriod(), errors);
        errors.popNestedPath();
    }
}
```

No exception is thrown!
Can be contextualized!
Can navigate the composition hierarchy
// 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 the 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));
}
```
Presentation layer
• 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

Source: Oracle
Java Web Applications

Client
(e.g., Web browser)

HTTP verbs
Get
Post
Update
Delete

HTTP request

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

Servlet container

Servlet

Http response

Status code
200, 201, 404, 500, etc.

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

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

```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>");
    }
}
```
MVC revisited: The Struts 2 architecture
Spring MVC architecture

Client → HTTP request → Dispatcher → Model → View

HTTP response

Dispatcher → URL → Controller → View resolver → View

Model → View logical name

Handler mapping

Web Server

Spring MVC container

Interc. → Interc. → Interc. → URL
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
The controller

```java
@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";
    }

    ...
}
```
The controller

```java
@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

@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;
}

@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());
    }
}