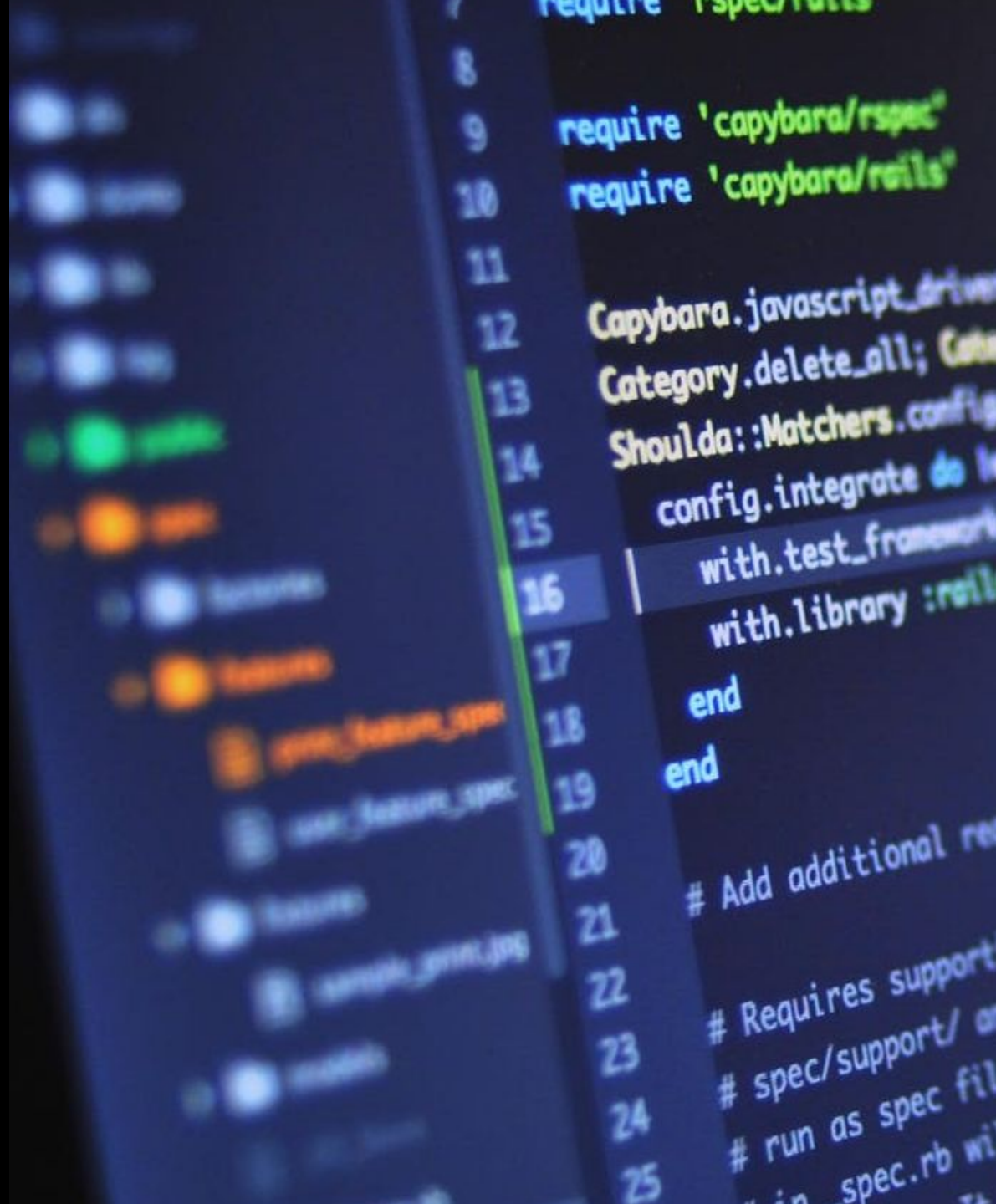
Arquitectura Hexagonal con Java/Quarkus

01-02-03 Y 04 DE JULIO
ESPAÑA - 2024



Diseño del Núcleo (Dominio)

Introduction to Domain-Driven Design

DOMAIN MODEL

The domain model is the core of Domain-Driven Design, representing the problem domain and its complex business rules.

UBIQUITOUS LANGUAGE

A shared language between the business and development team, ensuring consistent terminology and understanding of the domain.

BOUNDED CONTEXTS

Defining clear boundaries and responsibilities for different parts of the domain, to manage complexity and enable parallel development.

AGGREGATES

Grouping related entities and value objects into cohesive units, ensuring data consistency and transactional integrity.

BY UNDERSTANDING THESE CORE CONCEPTS OF DOMAIN-DRIVEN DESIGN, YOU CAN EFFECTIVELY DESIGN AND IMPLEMENT COMPLEX BUSINESS SYSTEMS THAT ALIGN WITH THE DOMAIN EXPERTS' UNDERSTANDING.

Designing the Core (Domain)

- **IDENTIFY ENTITIES AND AGGREGATES**

  Identify the key business objects or entities that represent the core domain concepts. Group related entities into logical units called aggregates, which define the boundaries of transactional consistency.

- **DEFINE VALUE OBJECTS**

  Identify value objects, which are immutable objects that represent a conceptual value. Value objects have no identity and are defined solely by their attributes.

- **CREATE DOMAIN SERVICES**

  Identify domain services, which are stateless operations that encapsulate core domain logic that does not naturally fit within an entity or value object.

Entities and Aggregates

UNIQUE IDENTITY

Entities are objects with a distinct and persistent identity, allowing them to be uniquely identified and distinguished from other objects.

COLLECTIONS OF ENTITIES

Aggregates are groups of related entities and value objects that are treated as a single unit for the purposes of data consistency and transaction management.

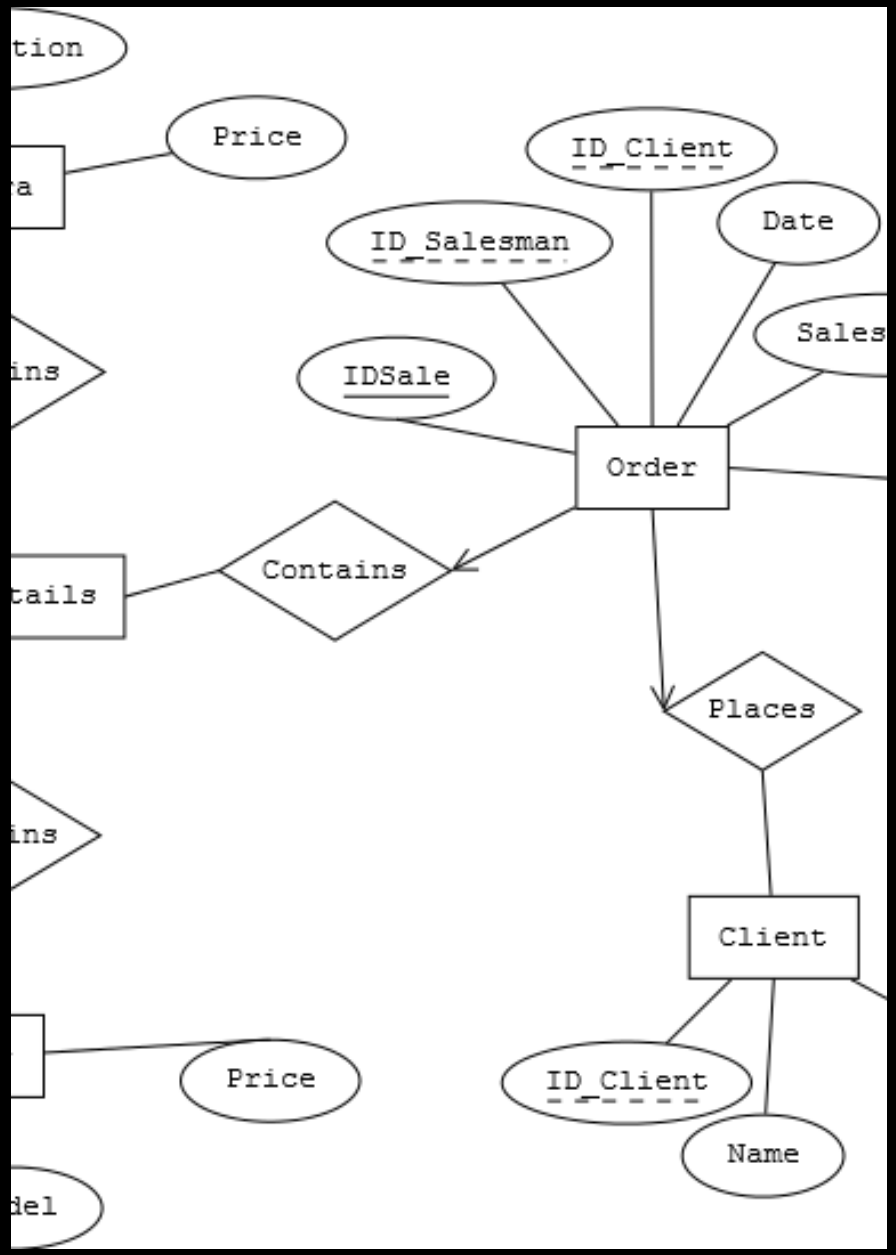
CONSISTENCY BOUNDARY

Aggregates establish a consistency boundary, ensuring that all the entities and value objects within the aggregate maintain a valid and coherent state as a whole.

TRANSACTION MANAGEMENT

Aggregates are the basic unit of transaction in a domain-driven design, allowing for atomic updates and ensuring data integrity.

ENTITIES AND AGGREGATES ARE FUNDAMENTAL CONCEPTS IN DOMAIN-DRIVEN DESIGN, WHERE ENTITIES REPRESENT DISTINCT OBJECTS WITH UNIQUE IDENTITIES, AND AGGREGATES ARE COLLECTIONS OF RELATED ENTITIES AND VALUE OBJECTS THAT ARE TREATED AS A SINGLE UNIT FOR CONSISTENCY AND TRANSACTION MANAGEMENT.



Order and Customer Entities

In an e-commerce system, the Order and Customer entities are crucial for managing the entire sales process. The Order entity represents a customer's purchase, including details such as the items ordered, quantity, total cost, and order status. The Customer entity, on the other hand, stores information about the individuals or organizations making purchases, including their personal details, contact information, and purchase history.

Ejemplo de Entidad Order

```java
package com.example.hexagonal.domain;

import java.util.ArrayList;
import java.util.List;

public class Order {
    private String id;
    private String customerId;
    private OrderStatus status;
    private List<OrderItem> items;

    public Order(String id, String customerId) {
        this.id = id;
        this.customerId = customerId;
        this.status = OrderStatus.PENDING;
        this.items = new ArrayList<>();
    }

    public void addItem(OrderItem item) {
        items.add(item);
    }

    public void removeItem(OrderItem item) {
        items.remove(item);
    }

    public void updateStatus(OrderStatus status) {
        this.status = status;
    }

    // Getters and setters
}

public class OrderItem {
    private String productId;
    private int quantity;
    private double price;

    public OrderItem(String productId, int quantity, double price) {
        this.productId = productId;
        this.quantity = quantity;
        this.price = price;
    }

    // Getters and setters
}

public enum OrderStatus {
    PENDING,
    CONFIRMED,
    SHIPPED,
    DELIVERED,
    CANCELLED
}
```

Ejemplo de Entidad Customer

```java
package com.example.hexagonal.domain;

public class Customer {
    private String id;
    private String name;
    private String email;
    private Address address;

    public Customer(String id, String name, String email, Address address) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.address = address;
    }

    // Getters and setters
}
```

Aggregates and Business Rules

AGGREGATE CONCEPT

Aggregates are the fundamental building blocks in Domain-Driven Design (DDD). They encapsulate a cluster of related entities that should be treated as a single unit.

BUSINESS RULE ENCAPSULATION

Aggregates are responsible for enforcing complex business rules within the domain model. They ensure consistency and maintain the integrity of the data.

ORDERAGGREGATE EXAMPLE

The OrderAggregate is an example of an aggregate that encapsulates the business rules related to managing orders, including order creation, modification, and processing.

AGGREGATE BOUNDARIES

Defining the boundaries of an aggregate is crucial. The aggregate boundary determines the scope of the business rules and the level of consistency that must be maintained.

AGGREGATE CONSISTENCY

Aggregates ensure consistency within the domain model by providing a single entry point for operations and guaranteeing that all business rules are applied correctly.

Ejemplo de Agregado OrderAggregate

```java
package com.example.hexagonal.domain;

public class OrderAggregate {
    private Order order;

    public OrderAggregate(Order order) {
        this.order = order;
    }

    public void confirmOrder() {
        if (order.getStatus() == OrderStatus.PENDING) {
            order.updateStatus(OrderStatus.CONFIRMED);
        } else {
            throw new IllegalStateException("Order cannot be confirmed.");
        }
    }

    public void shipOrder() {
        if (order.getStatus() == OrderStatus.CONFIRMED) {
            order.updateStatus(OrderStatus.SHIPPED);
        } else {
            throw new IllegalStateException("Order cannot be shipped.");
        }
    }

    public void deliverOrder() {
        if (order.getStatus() == OrderStatus.SHIPPED) {
            order.updateStatus(OrderStatus.DELIVERED);
        } else {
            throw new IllegalStateException("Order cannot be delivered.");
        }
    }

    // Other business logic
}
```

Value Objects

IMMUTABLE OBJECTS

Value objects are immutable, meaning their state cannot be modified after creation, ensuring consistency and thread-safety.

DESCRIBE DOMAIN ASPECTS

Value objects represent aspects of the domain without their own identity, such as Money, Address, or Date.

NO IDENTITY

Value objects are not identified by a unique identifier, but rather by the combination of their properties, allowing for value-based equality comparison.

SIMPLE AND FOCUSED

Value objects are simple, single-purpose objects that encapsulate a specific concept or value, making them easy to understand and use.

VALUE OBJECTS ARE AN ESSENTIAL PART OF DOMAIN-DRIVEN DESIGN, PROVIDING A CLEAR AND CONSISTENT WAY TO REPRESENT KEY DOMAIN CONCEPTS AND VALUES.

Ejemplo de Value Object Money

```java
package com.example.hexagonal.domain;

import java.math.BigDecimal;
import java.util.Currency;

public class Money {
    private BigDecimal amount;
    private Currency currency;

    public Money(BigDecimal amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public BigDecimal getAmount() {
        return amount;
    }

    public Currency getCurrency() {
        return currency;
    }

    // Getters and other methods
}
```

Ejemplo de Value Object Addres

```java
package com.example.hexagonal.domain;

public class Address {
    private String street;
    private String city;
    private String state;
    private String zipCode;

    public Address(String street, String city, String state, String zipCode) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
    }

    // Getters and other methods
}
```

Domain Services

WHAT ARE DOMAIN SERVICES?

Domain services encapsulate business logic that cannot be attributed to a single entity or value object, such as the OrderService.

ROLE OF DOMAIN SERVICES

Domain services handle complex business rules, workflows, and calculations that span multiple domain objects.

SEPARATING CONCERNS

Domain services help separate concerns by keeping the business logic out of entities and value objects, making the domain model more focused and maintainable.

REUSABILITY

Domain services can be reused across different parts of the application, promoting code reuse and consistency.

TESTABILITY

Domain services, being isolated from the UI and infrastructure, are easier to test independently, improving overall application testability.

Ejemplo de Servicio de Dominio para Gestionar Pedidos

```java
package com.example.hexagonal.domain;

import javax.enterprise.context.ApplicationScoped;
import java.util.ArrayList;
import java.util.List;

@ApplicationScoped
public class OrderService {

    private List<Order> orders = new ArrayList<>();

    public void createOrder(Order order) {
        orders.add(order);
    }

    public void addItemToOrder(String orderId, OrderItem item) {
        Order order = findOrderById(orderId);
        if (order != null) {
            order.addItem(item);
        }
    }

    public void updateOrderStatus(String orderId, OrderStatus status) {
        Order order = findOrderById(orderId);
        if (order != null) {
            order.updateStatus(status);
        }
    }

    public Order findOrderById(String orderId) {
        return orders.stream().filter(order -> order.getId().equals(orderId)).findFirst().orElse(null);
    }

    public List<Order> getAllOrders() {
        return orders;
    }
}
```

Implementing Business Rules

ENCAPSULATE BUSINESS LOGIC

Business rules are defined within entities and domain services to centralize the system's core logic and ensure consistent behavior.

VALIDATE INPUTS AND OUTPUTS

Domain services validate incoming data and outgoing results to uphold the integrity of the system and prevent invalid states.

ENFORCE INVARIANTS

Entities maintain invariants, which are rules that must always hold true, to ensure the data is in a valid and consistent state.

SEPARATE CONCERNS

Business rules are defined in entities and domain services, separating them from other concerns like user interface, data persistence, or communication.

BY IMPLEMENTING BUSINESS RULES WITHIN THE CORE OF THE SYSTEM, THE DOMAIN-DRIVEN DESIGN APPROACH ENSURES THE SOFTWARE OPERATES CORRECTLY AND CONSISTENTLY, ALIGNING WITH THE BUSINESS REQUIREMENTS.

Implementación de reglas de negocio: Definición y validación de reglas dentro de las entidades y servicios

```java
package com.example.hexagonal.domain;

public class Order {
    // Existing code...

    public void validateItemQuantity(OrderItem item) {
        if (item.getQuantity() <= 0) {
            throw new IllegalArgumentException("Item quantity must be greater than zero.");
        }
    }
}
```

Design Patterns: Factory and Repository

THE FACTORY PATTERN

The Factory pattern is used to create complex objects without exposing the creation logic to the client. It provides a way to create objects of a common interface or superclass, without knowing the exact class of the object being created.

THE REPOSITORY PATTERN

The Repository pattern provides an abstraction for data persistence operations, such as creating, reading, updating, and deleting entities. It acts as a mediator between the domain model and the data mapping layer, making the application's code more maintainable and testable.

Ejemplo patron Factory

```java
package com.example.hexagonal.domain;

import java.util.UUID;

public class OrderFactory {

    public Order createOrder(String customerId) {
        return new Order(UUID.randomUUID().toString(), customerId);
    }
}
```

Repository: Para la persistencia de entidades

```java
package com.example.hexagonal.domain;

import java.util.List;

public interface OrderRepository {
    void save(Order order);
    Order findById(String id);
    List<Order> findAll();
}
```

Practical Exercise with Java and Quarkus

- **INTRODUCING THE PROBLEM DOMAIN**

  Explore a real-world problem domain and identify the core entities, their relationships, and the business rules that govern them.

- **MODELING THE DOMAIN WITH JAVA AND QUARKUS**

  Implement the domain model using Java classes and leverage the Quarkus framework for its features, such as dependency injection, configuration management, and rapid development.

- **APPLYING DOMAIN-DRIVEN DESIGN PATTERNS**

  Incorporate patterns like Aggregate, Repository, and Service to ensure the application's architecture aligns with the domain model and follows best practices.

- **IMPLEMENTING THE APPLICATION LOGIC**

  Develop the application logic that interacts with the domain model, handling user requests, data persistence, and any other necessary functionality.

- **TESTING THE DOMAIN MODEL AND APPLICATION**

  Write comprehensive unit and integration tests to ensure the domain model and application logic are working as expected, and refine the implementation as needed.

- **DEPLOYING THE QUARKUS APPLICATION**

  Package the Quarkus application and deploy it to a target environment, leveraging Quarkus's capabilities for easy containerization and cloud-native deployment.