Parfait ! Maintenant j'ai une vision complète. Voici ma proposition pour la **Séance 4 : Architecture modulaire et découplage avec la notification par email**.

—

#

4 - Architecture Modulaire et Découplage

Séance 4 (4h)

##

Objectifs pédagogiques

- Comprendre les limites d'une architecture monolithique
- Appliquer les principes SOLID (notamment DIP et OCP)
- Mettre en place un système d'événements avec Spring Events
- Découvrir les design patterns : Strategy, Observer, Factory
- Introduire la notion de modules/packages par domaine métier
- Tester les composants découplés

##

Contexte : User Story

US-042 : Notification email lors de la création d'une commande

En tant que client **Je veux** recevoir un email récapitulatif immédiatement après avoir passé une commande **Afin de** avoir une confirmation et les détails de ma commande

Critères d'acceptation :

- □ Email envoyé automatiquement lors de la création d'une commande
- □ Email contient : numéro de commande, liste des produits, montant total
- □ La création de commande ne doit pas échouer si l'envoi d'email échoue
- □ Le système doit être extensible pour ajouter d'autres notifications (SMS, push...)
- □ Logs de toutes les notifications envoyées

##

Partie 0 : État des lieux - Le piège du couplage fort (20min)

0.1 Implémentation naïve (anti-pattern)

⚠ **NE PAS FAIRE - Exemple de mauvaise pratique**

```
@Service
@RequiredArgsConstructor
public class OrderService {
    private final OrderRepository orderRepository;
    private final ProductRepository productRepository;
    private final JavaMailSender mailSender; // □ Dépendance directe
    @Transactional
    public Order createOrder(CreateOrderDto dto) {
        // 1. Validation et création
        User user = userRepository.findById(dto.getUserId())
            .orElseThrow(() -> new UserNotFoundException(dto.getUserId()));
        Order order = new Order(user);
        for (OrderItemDto itemDto : dto.getItems()) {
            Product product = productRepository.findById(itemDto.getProductId())
                .orElseThrow(() -> new
ProductNotFoundException(itemDto.getProductId()));
            if (product.getStock() < itemDto.getQuantity()) {
                throw new InsufficientStockException(product.getId());
            }
            product.decreaseStock(itemDto.getQuantity());
            order.addItem(new OrderItem(product, itemDto.getQuantity(),
product.getPrice()));
        }
        Order savedOrder = orderRepository.save(order);
        // □ PROBLÈME 1 : Logique métier mélangée avec l'envoi d'email
        // □ PROBLÈME 2 : Si l'email échoue, la transaction est rollback
        // □ PROBLÈME 3 : Impossible de tester la création sans email
        // □ PROBLÈME 4 : Pour ajouter SMS, il faut modifier cette classe
        try {
            MimeMessage message = mailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(message, true);
            helper.setTo(user.getEmail());
            helper.setSubject("Order Confirmation #" + savedOrder.getId());
            helper.setText(buildOrderEmailContent(savedOrder), true);
            mailSender.send(message);
        } catch (Exception e) {
            log.error("Failed to send order confirmation email", e);
            // ⚠□ On ignore l'erreur mais la commande est déjà sauvegardée
        }
        return savedOrder;
    }
    private String buildOrderEmailContent(Order order) {
        // Construction du HTML de l'email...
        return "...";
    }
}
```

0.2 Problèmes identifiés

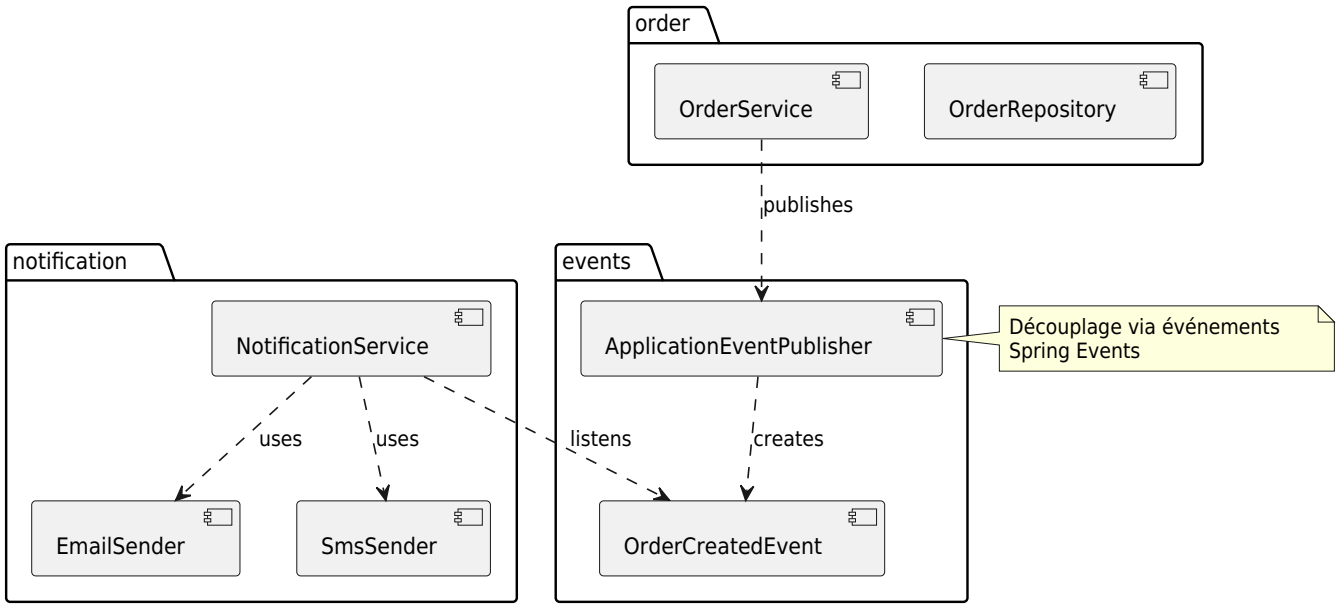
Discussion collective (10min) :

Identifier les problèmes de cette implémentation :

- Violation du Single Responsibility Principle
- Couplage fort entre domaines métier (Order ↔ Email)
- Testabilité compromise
- Gestion d'erreur problématique (transaction vs notification)
- Extensibilité limitée (ajout SMS, push...)
- Performance (envoi synchrone bloquant)

0.3 Objectif de la séance

Transformer cette architecture monolithique en une architecture modulaire et découplée



##

Partie 1 : Réorganisation en packages par domaine (30min)

1.1 Structure modulaire proposée

```
src/main/java/com/ecommerce/
├── order/                          # Domaine Order
│   ├── domain/
│   │   ├── Order.java
│   │   ├── OrderItem.java
│   │   └── OrderStatus.java
│   ├── dto/
│   │   ├── CreateOrderDto.java
│   │   └── OrderResponseDto.java
│   ├── repository/
│   │   └── OrderRepository.java
│   ├── service/
│   │   └── OrderService.java
│   ├── controller/
│   │   └── OrderController.java
```

```
│   └── event/
│       └── OrderCreatedEvent.java
│
├── notification/                    # Domaine Notification
│   ├── domain/
│   │   ├── NotificationChannel.java
│   │   └── NotificationLog.java
│   ├── service/
│   │   ├── NotificationService.java
│   │   └── sender/
│   │       ├── NotificationSender.java (interface)
│   │       ├── EmailNotificationSender.java
│   │       └── ConsoleNotificationSender.java
│   ├── listener/
│   │   └── OrderNotificationListener.java
│   ├── repository/
│   │   └── NotificationLogRepository.java
│   └── config/
│       └── NotificationConfig.java
│
├── product/                         # Domaine Product (existant)
│   ├── domain/
│   ├── service/
│   └── ...
│
└── user/                            # Domaine User (existant)
    ├── domain/
    ├── service/
    └── ...
```

Principe de packaging par domaine (DDD-lite) :

- Chaque package = un domaine métier cohérent
- Limite les dépendances croisées
- Facilite l'extraction future en microservices
- Améliore la lisibilité et la maintenabilité

1.2 Exercice de refactoring

Exercice 1 (20min) :

Réorganiser votre code existant selon cette structure :

- Déplacer les classes `Order*` vers `com.ecommerce.order.*`
- Déplacer les classes `Product*` vers `com.ecommerce.product.*`
- Déplacer les classes `User*` vers `com.ecommerce.user.*`
- Corriger les imports
- Vérifier que tous les tests passent après refactoring

Validation :

- ✅ `mvn clean verify` passe au vert
- ✅ Aucune dépendance cyclique entre packages

- Les controllers importent uniquement les services de leur domaine

##

Partie 2 : Spring Events pour le découplage (1h)

2.1 Création de l'événement métier

```
package com.ecommerce.order.event;

import lombok.Getter;
import org.springframework.context.ApplicationEvent;

import java.math.BigDecimal;
import java.time.Instant;
import java.util.List;
import java.util.UUID;

/**
 * Événement publié lors de la création d'une commande
 */
@Getter
public class OrderCreatedEvent extends ApplicationEvent {
    private final UUID orderId;
    private final UUID userId;
    private final String userEmail;
    private final BigDecimal totalAmount;
    private final List<OrderItemInfo> items;
    private final Instant createdAt;
    public OrderCreatedEvent(Object source, UUID orderId, UUID userId, String
userEmail,
                             BigDecimal totalAmount, List<OrderItemInfo> items) {
        super(source);
        this.orderId = orderId;
        this.userId = userId;
        this.userEmail = userEmail;
        this.totalAmount = totalAmount;
        this.items = items;
        this.createdAt = Instant.now();
    }
    @Getter
    public static class OrderItemInfo {
        private final String productName;
        private final int quantity;
        private final BigDecimal unitPrice;
        public OrderItemInfo(String productName, int quantity, BigDecimal
unitPrice) {
            this.productName = productName;
            this.quantity = quantity;
            this.unitPrice = unitPrice;
        }
```

```
    }
}
```

2.2 Publication de l'événement dans OrderService

```
package com.ecommerce.order.service;

import com.ecommerce.order.event.OrderCreatedEvent;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@RequiredArgsConstructor
@Slf4j
public class OrderService {
    private final OrderRepository orderRepository;
    private final ProductRepository productRepository;
    private final UserRepository userRepository;
    private final ApplicationEventPublisher eventPublisher; // □ Injection Spring
    @Transactional
    public Order createOrder(CreateOrderDto dto) {
        // 1. Validation
        User user = userRepository.findById(dto.getUserId())
            .orElseThrow(() -> new UserNotFoundException(dto.getUserId()));
        // 2. Création de la commande
        Order order = new Order(user);
        for (OrderItemDto itemDto : dto.getItems()) {
            Product product = productRepository.findById(itemDto.getProductId())
                .orElseThrow(() -> new
ProductNotFoundException(itemDto.getProductId()));
            if (product.getStock() < itemDto.getQuantity()) {
                throw new InsufficientStockException(product.getId());
            }
            product.decreaseStock(itemDto.getQuantity());
            order.addItem(new OrderItem(product, itemDto.getQuantity(),
product.getPrice()));
        }
        Order savedOrder = orderRepository.save(order);
        // 3. □ Publication de l'événement (découplé)
        publishOrderCreatedEvent(savedOrder, user);
        log.info("Order created successfully: {}", savedOrder.getId());
        return savedOrder;
    }
    private void publishOrderCreatedEvent(Order order, User user) {
        List<OrderCreatedEvent.OrderItemInfo> itemInfos = order.getItems().stream()
            .map(item -> new OrderCreatedEvent.OrderItemInfo(
                item.getProduct().getName(),
                item.getQuantity(),
                item.getUnitPrice()
            ))
            .toList();
```

```
        OrderCreatedEvent event = new OrderCreatedEvent(
            this,
            order.getId(),
            user.getId(),
            user.getEmail(),
            order.getTotalAmount(),
            itemInfos
        );
        eventPublisher.publishEvent(event);
        log.debug("OrderCreatedEvent published for order: {}", order.getId());
    }
}
```

2.3 Avantages de cette approche

Bénéfices du découplage par événements :

- ✅ `OrderService` n'a aucune dépendance vers `notification`
- ✅ Transaction commit AVANT le traitement de l'événement
- ✅ Si l'email échoue, la commande reste créée
- ✅ Testable indépendamment
- ✅ Extensible : ajout de listeners sans modifier `OrderService`
- ✅ Respect du principe Open/Closed (SOLID)

##

Partie 3 : Pattern Strategy pour les canaux de notification (1h)

3.1 Interface NotificationSender

```
package com.ecommerce.notification.service.sender;

import com.ecommerce.notification.domain.NotificationChannel;

/**
 * Contrat pour l'envoi de notifications
 * Pattern Strategy
 */
public interface NotificationSender {
    /**
     * Envoie une notification
     * @param recipient Destinataire (email, numéro de téléphone...)
     * @param subject Sujet de la notification
     * @param content Contenu de la notification
     */
    void send(String recipient, String subject, String content);
    /**
     * Canal supporté par cette implémentation
     */
```

```
    NotificationChannel getSupportedChannel();
    /**
     * Vérifie si l'envoi est disponible
     */
    boolean isAvailable();
}
```

3.2 Implémentation Console (pour dev/test)

```
package com.ecommerce.notification.service.sender;

import com.ecommerce.notification.domain.NotificationChannel;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.stereotype.Component;

@Component
@Slf4j
@ConditionalOnProperty(name = "notification.email.enabled", havingValue = "false",
matchIfMissing = true)
public class ConsoleNotificationSender implements NotificationSender {
    @Override
    public void send(String recipient, String subject, String content) {
        log.info("""
            ========== EMAIL NOTIFICATION ==========
            To: {}
            Subject: {}
            ---
            {}
            ========================================
            """, recipient, subject, content);
    }
    @Override
    public NotificationChannel getSupportedChannel() {
        return NotificationChannel.EMAIL;
    }
    @Override
    public boolean isAvailable() {
        return true;
    }
}
```

3.3 Implémentation Email (pour production)

```
package com.ecommerce.notification.service.sender;

import com.ecommerce.notification.domain.NotificationChannel;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.mail.MailException;
```

```
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.stereotype.Component;

import jakarta.mail.MessagingException;
import jakarta.mail.internet.MimeMessage;

@Component
@RequiredArgsConstructor
@Slf4j
@ConditionalOnProperty(name = "notification.email.enabled", havingValue = "true")
public class EmailNotificationSender implements NotificationSender {
    private final JavaMailSender mailSender;
    @Override
    public void send(String recipient, String subject, String content) {
        try {
            MimeMessage message = mailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(message, true,
"UTF-8");
            helper.setTo(recipient);
            helper.setSubject(subject);
            helper.setText(content, true); // true = HTML
            helper.setFrom("noreply@ecommerce.com");
            mailSender.send(message);
            log.info("Email sent successfully to {}", recipient);
        } catch (MessagingException | MailException e) {
            log.error("Failed to send email to {}: {}", recipient, e.getMessage());
            throw new NotificationException("Email sending failed", e);
        }
    }
    @Override
    public NotificationChannel getSupportedChannel() {
        return NotificationChannel.EMAIL;
    }
    @Override
    public boolean isAvailable() {
        try {
            mailSender.createMimeMessage();
            return true;
        } catch (Exception e) {
            log.warn("Email sender not available: {}", e.getMessage());
            return false;
        }
    }
}
```

3.4 Service de notification avec Pattern Factory

```
package com.ecommerce.notification.service;

import com.ecommerce.notification.domain.NotificationChannel;
import com.ecommerce.notification.domain.NotificationLog;
import com.ecommerce.notification.repository.NotificationLogRepository;
import com.ecommerce.notification.service.sender.NotificationSender;
```

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;

import java.time.Instant;
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;

@Service
@Slf4j
public class NotificationService {
    private final Map<NotificationChannel, NotificationSender> senders;
    private final NotificationLogRepository logRepository;
    //  Pattern Factory : injection de toutes les implémentations
    public NotificationService(List<NotificationSender> senderList,
                               NotificationLogRepository logRepository) {
        this.senders = senderList.stream()
            .collect(Collectors.toMap(
                NotificationSender::getSupportedChannel,
                Function.identity()
            ));
        this.logRepository = logRepository;
        log.info("NotificationService initialized with {} senders: {}",
            senders.size(), senders.keySet());
    }
    public void sendNotification(NotificationChannel channel, String recipient,
                                 String subject, String content) {
        NotificationSender sender = senders.get(channel);
        if (sender == null) {
            log.error("No sender found for channel: {}", channel);
            logFailedNotification(channel, recipient, subject, "No sender
configured");
            return;
        }
        if (!sender.isAvailable()) {
            log.warn("Sender for channel {} is not available", channel);
            logFailedNotification(channel, recipient, subject, "Sender
unavailable");
            return;
        }
        try {
            sender.send(recipient, subject, content);
            logSuccessfulNotification(channel, recipient, subject);
        } catch (Exception e) {
            log.error("Failed to send notification via {}: {}", channel,
e.getMessage());
            logFailedNotification(channel, recipient, subject, e.getMessage());
        }
    }
    private void logSuccessfulNotification(NotificationChannel channel, String
recipient, String subject) {
        NotificationLog log = NotificationLog.builder()
            .channel(channel)
            .recipient(recipient)
            .subject(subject)
```

```
            .status("SUCCESS")
            .sentAt(Instant.now())
            .build();
        logRepository.save(log);
    }
    private void logFailedNotification(NotificationChannel channel, String
recipient,
                                       String subject, String errorMessage) {
        NotificationLog log = NotificationLog.builder()
            .channel(channel)
            .recipient(recipient)
            .subject(subject)
            .status("FAILED")
            .errorMessage(errorMessage)
            .sentAt(Instant.now())
            .build();
        logRepository.save(log);
    }
}
```

3.5 Entité NotificationLog

```
package com.ecommerce.notification.domain;

import jakarta.persistence.*;
import lombok.*;

import java.time.Instant;
import java.util.UUID;

@Entity
@Table(name = "notification_logs")
@Getter
@Setter
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class NotificationLog {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;
    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private NotificationChannel channel;
    @Column(nullable = false)
    private String recipient;
    @Column(nullable = false)
    private String subject;
    @Column(nullable = false)
    private String status; // SUCCESS, FAILED
    @Column(length = 1000)
    private String errorMessage;
    @Column(nullable = false)
    private Instant sentAt;
```

```
}
```

```
package com.ecommerce.notification.domain;

public enum NotificationChannel {
    EMAIL,
    SMS,
    PUSH
}
```

##

Partie 4 : Listener d'événements (30min)

4.1 OrderNotificationListener

```
package com.ecommerce.notification.listener;

import com.ecommerce.notification.domain.NotificationChannel;
import com.ecommerce.notification.service.NotificationService;
import com.ecommerce.order.event.OrderCreatedEvent;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.context.event.EventListener;
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Component;
import org.springframework.transaction.event.TransactionPhase;
import org.springframework.transaction.event.TransactionalEventListener;

@Component
@RequiredArgsConstructor
@Slf4j
public class OrderNotificationListener {
    private final NotificationService notificationService;
    /**
     *  TransactionalEventListener : exécuté APRÈS le commit de la transaction
     *  @Async : traitement asynchrone (ne bloque pas la création de commande)
     */
    @Async
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    public void handleOrderCreated(OrderCreatedEvent event) {
        log.info("Handling OrderCreatedEvent for order: {}", event.getOrderId());
        String subject = "Order Confirmation #" + event.getOrderId();
        String content = buildOrderEmailContent(event);
        notificationService.sendNotification(
            NotificationChannel.EMAIL,
            event.getUserEmail(),
            subject,
            content
```

```
        );
    }
    private String buildOrderEmailContent(OrderCreatedEvent event) {
        StringBuilder html = new StringBuilder();
        html.append("<html><body>");
        html.append("<h1>Order Confirmation</h1>");
        html.append("<p>Dear Customer,</p>");
        html.append("<p>Your order
<strong>#").append(event.getOrderId()).append("</strong> has been confirmed.</p>");
        html.append("<h2>Order Details</h2>");
        html.append("<table border='1' cellpadding='10'>");
        html.append("<tr><th>Product</th><th>Quantity</th><th>Unit
Price</th><th>Total</th></tr>");
        event.getItems().forEach(item -> {
            html.append("<tr>");
            html.append("<td>").append(item.getProductName()).append("</td>");
            html.append("<td>").append(item.getQuantity()).append("</td>");
            html.append("<td>€").append(item.getUnitPrice()).append("</td>");
            html.append("<td>€").append(item.getUnitPrice().multiply(new
java.math.BigDecimal(item.getQuantity()))).append("</td>");
            html.append("</tr>");
        });
        html.append("</table>");
        html.append("<p><strong>Total Amount:
€").append(event.getTotalAmount()).append("</strong></p>");
        html.append("<p>Thank you for your order!</p>");
        html.append("</body></html>");
        return html.toString();
    }
}
```

4.2 Configuration pour @Async

```
package com.ecommerce.notification.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableAsync;

@Configuration
@EnableAsync
public class NotificationConfig {
    // Configuration par défaut de Spring pour @Async
    // Un ThreadPoolTaskExecutor sera créé automatiquement
}
```

4.3 Configuration des propriétés

```
# application-dev.properties
notification.email.enabled=false  # Console en dev

# application-test.properties
```

```
notification.email.enabled=false  # Console en test

# application-prod.properties
notification.email.enabled=true   # Vrai email en prod

# Configuration Spring Mail (seulement si enabled=true)
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=${SMTP_USERNAME}
spring.mail.password=${SMTP_PASSWORD}
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

##

Partie 5 : Tests du système découplé (1h)

5.1 Test unitaire du NotificationService

```
package com.ecommerce.notification.service;

import com.ecommerce.notification.domain.NotificationChannel;
import com.ecommerce.notification.repository.NotificationLogRepository;
import com.ecommerce.notification.service.sender.NotificationSender;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.List;

import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
@DisplayName("NotificationService - Unit Tests")
class NotificationServiceTest {
    @Mock
    private NotificationSender emailSender;
    @Mock
    private NotificationLogRepository logRepository;
    private NotificationService notificationService;
    @BeforeEach
    void setUp() {
when(emailSender.getSupportedChannel()).thenReturn(NotificationChannel.EMAIL);
        when(emailSender.isAvailable()).thenReturn(true);
        notificationService = new NotificationService(List.of(emailSender),
logRepository);
    }
    @Test
```

```
    @DisplayName("Should send notification via correct sender")
    void sendNotification_WithValidChannel_ShouldUseSender() {
        // When
        notificationService.sendNotification(
            NotificationChannel.EMAIL,
            "test@example.com",
            "Test Subject",
            "Test Content"
        );
        // Then
        verify(emailSender).send("test@example.com", "Test Subject", "Test
Content");
        verify(logRepository).save(any());
    }
    @Test
    @DisplayName("Should log when sender is unavailable")
    void sendNotification_WhenSenderUnavailable_ShouldLogFailure() {
        // Given
        when(emailSender.isAvailable()).thenReturn(false);
        // When
        notificationService.sendNotification(
            NotificationChannel.EMAIL,
            "test@example.com",
            "Test",
            "Content"
        );
        // Then
        verify(emailSender, never()).send(anyString(), anyString(), anyString());
        verify(logRepository).save(argThat(log ->
            log.getStatus().equals("FAILED") &&
            log.getErrorMessage().contains("unavailable")
        ));
    }
}
```

5.2 Test d'intégration avec capture d'événements

```
package com.ecommerce.order.controller;

import com.ecommerce.order.event.OrderCreatedEvent;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Primary;
import org.springframework.context.event.EventListener;
import org.springframework.http.MediaType;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.transaction.annotation.Transactional;
```

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

import static org.assertj.core.api.Assertions.assertThat;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("test")
@Transactional
@DisplayName("Order Creation - Integration Tests")
class OrderCreationIntegrationTest {
    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private TestEventListener eventListener;
    @Test
    @DisplayName("Should publish OrderCreatedEvent after order creation")
    void createOrder_ShouldPublishEvent() throws Exception {
        // Given
        String requestBody = """
            {
                "userId": "...",
                "items": [
                    {
                        "productId": "...",
                        "quantity": 2
                    }
                ]
            }
            """;
        // When
        mockMvc.perform(post("/orders")
                .contentType(MediaType.APPLICATION_JSON)
                .content(requestBody))
            .andExpect(status().isCreated());
        // Then
        assertThat(eventListener.getReceivedEvents()).hasSize(1);
        OrderCreatedEvent event = eventListener.getReceivedEvents().get(0);
        assertThat(event.getUserId()).isNotNull();
        assertThat(event.getItems()).hasSize(1);
    }
    /**
     * Configuration de test pour capturer les événements
     */
    @TestConfiguration
    static class TestConfig {
        @Bean
        @Primary
        public TestEventListener testEventListener() {
            return new TestEventListener();
        }
    }
```

```
    /**
     * Listener de test pour vérifier la publication d'événements
     */
    static class TestEventListener {
        private final List<OrderCreatedEvent> receivedEvents = new ArrayList<>();
        @EventListener
        public void handleEvent(OrderCreatedEvent event) {
            receivedEvents.add(event);
        }
        public List<OrderCreatedEvent> getReceivedEvents() {
            return receivedEvents;
        }
        public void reset() {
            receivedEvents.clear();
        }
    }
}
```

5.3 Test du Listener avec Mockito

```
package com.ecommerce.notification.listener;

import com.ecommerce.notification.domain.NotificationChannel;
import com.ecommerce.notification.service.NotificationService;
import com.ecommerce.order.event.OrderCreatedEvent;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import java.math.BigDecimal;
import java.util.List;
import java.util.UUID;

import static org.mockito.ArgumentMatchers.*;
import static org.mockito.Mockito.verify;

@ExtendWith(MockitoExtension.class)
@DisplayName("OrderNotificationListener - Unit Tests")
class OrderNotificationListenerTest {
    @Mock
    private NotificationService notificationService;
    @InjectMocks
    private OrderNotificationListener listener;
    @Test
    @DisplayName("Should send email notification when order is created")
    void handleOrderCreated_ShouldSendEmailNotification() {
        // Given
        OrderCreatedEvent event = new OrderCreatedEvent(
            this,
            UUID.randomUUID(),
            UUID.randomUUID(),
```

```
            "customer@example.com",
            BigDecimal.valueOf(100.00),
            List.of()
        );
        // When
        listener.handleOrderCreated(event);
        // Then
        verify(notificationService).sendNotification(
            eq(NotificationChannel.EMAIL),
            eq("customer@example.com"),
            contains("Order Confirmation"),
            anyString()
        );
    }
}
```

Exercice 2 (45min) :

Compléter la suite de tests :

- Test du `ConsoleNotificationSender`
- Test d'intégration complet : création commande → vérification log notification
- Test de gestion d'erreur : email invalide
- Test de performance : vérifier que l'envoi est bien asynchrone (temps de réponse < 500ms)

Validation :

- Tous les tests passent
- Couverture > 80% sur le package `notification`
- Tests asynchrones correctement gérés

##

Partie 6 : Extension - Ajout d'un nouveau canal (SMS) (20min - Bonus)

Challenge : Ajouter un canal SMS sans modifier le code existant (principe Open/Closed)

```
package com.ecommerce.notification.service.sender;

import com.ecommerce.notification.domain.NotificationChannel;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.stereotype.Component;

@Component
@Slf4j
@ConditionalOnProperty(name = "notification.sms.enabled", havingValue = "true")
public class SmsNotificationSender implements NotificationSender {
```

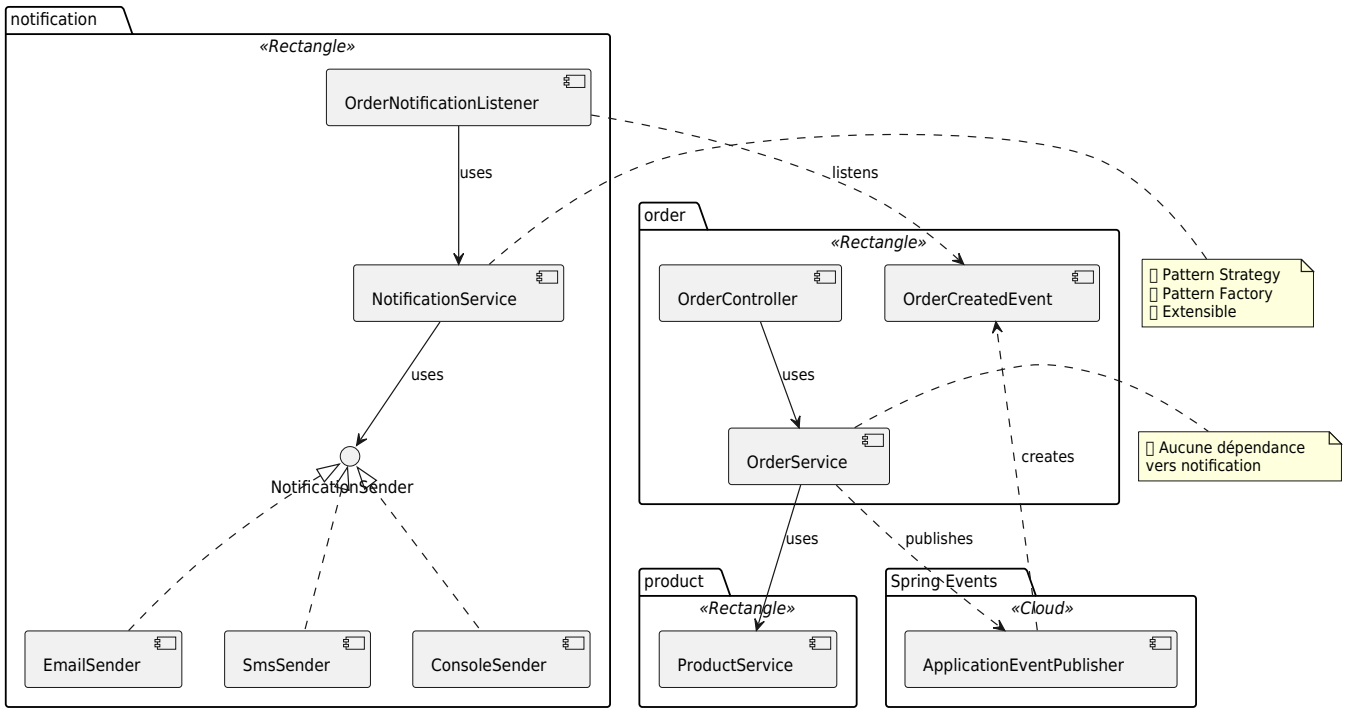
```
    @Override
    public void send(String recipient, String subject, String content) {
        // Intégration avec Twilio, AWS SNS, etc.
        log.info("Sending SMS to {}: {}", recipient, content);
        // Implémentation simplifiée pour la demo
    }
    @Override
    public NotificationChannel getSupportedChannel() {
        return NotificationChannel.SMS;
    }
    @Override
    public boolean isAvailable() {
        return true;
    }
}
```

Points à noter :

- Aucune modification dans `NotificationService`
- Spring injecte automatiquement le nouveau sender
- Activation via configuration (`notification.sms.enabled`)

##

Récapitulatif : Architecture finale



##

Livrables attendus

Priorités (4h)

Architecture modulaire (30min) :

- Packages réorganisés par domaine (order, notification, product, user)
- Pas de dépendances cycliques
- Tests passent après refactoring

Système de notification (2h) :

- `OrderCreatedEvent` implémenté
- Publication d'événement dans `OrderService`
- Interface `NotificationSender` + 2 implémentations (Console + Email)
- `NotificationService` avec Pattern Factory
- `OrderNotificationListener` avec `@TransactionalEventListener`
- Entité `NotificationLog` pour audit
- Configuration multi-environnements

Tests (1h) :

- Tests unitaires de `NotificationService`
- Tests unitaires de `OrderNotificationListener`
- Test d'intégration avec capture d'événements
- Vérification que l'envoi est asynchrone
- Couverture > 70% sur le package notification

Documentation (30min) :

- Diagramme d'architecture dans le README
- Documentation des patterns utilisés
- Guide de configuration des notifications

Bonus (si temps)

- Ajout du canal SMS
- Template d'email avec Thymeleaf
- Retry automatique en cas d'échec
- Dashboard des notifications dans H2 console
- Métriques Prometheus (nombre d'emails envoyés)

##

Concepts clés à retenir

Design Patterns appliqués

- **Observer** : Spring Events pour la communication inter-domaines
- **Strategy** : `NotificationSender` avec différentes implémentations
- **Factory** : Injection automatique de tous les senders
- **Dependency Inversion** : `OrderService` ne dépend que d'abstractions

Principes SOLID

- **S**ingle Responsibility : chaque service a une responsabilité unique
- **O**pen/Closed : ajout de canaux sans modifier le code existant
- **L**iskov Substitution : toutes les implémentations respectent le contrat
- **I**nterface Segregation : interface minimale `NotificationSender`
- **D**ependency Inversion : dépendances vers abstractions, pas implémentations

Architecture

- **Packaging par domaine** : prépare la transition vers les microservices
- **Event-driven** : découplage temporel et organisationnel
- **Async processing** : performances et résilience
- **Configuration externalisée** : flexibilité environnements

##

Ressources

- Spring Events Documentation
- Refactoring Guru - Design Patterns
- Martin Fowler - Event-Driven Architecture
- Baeldung - Spring Events
- Spring @Async Documentation