4 - Architecture Modulaire et Découplage

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
- Utiliser 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 d'**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
class OrderService(
    private val orderRepository: OrderRepository,
    private val productRepository: ProductRepository,
    private val userRepository: UserRepository,
    private val mailSender: JavaMailSender // □ Dépendance directe
) {
    private val logger = LoggerFactory.getLogger(javaClass)
    @Transactional
    fun createOrder(dto: CreateOrderDto): Order {
        // 1. Validation et création
        val user = userRepository.findById(dto.userId)
```

```
            .orElseThrow { UserNotFoundException(dto.userId) }
        val order = Order(user = user)
        dto.items.forEach { itemDto ->
            val product = productRepository.findById(itemDto.productId)
                .orElseThrow { ProductNotFoundException(itemDto.productId) }
            if (product.stock < itemDto.quantity) {
                throw InsufficientStockException(product.id!!)
            }
            product.decreaseStock(itemDto.quantity)
            order.addItem(OrderItem(
                product = product,
                quantity = itemDto.quantity,
                unitPrice = product.price
            ))
        }
        val 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 {
            sendOrderConfirmationEmail(savedOrder)
        } catch (e: Exception) {
            logger.error("Failed to send email for order ${savedOrder.id}", e)
            // Que faire ? Rollback ? Continuer ?
        }
        return savedOrder
    }
    private fun sendOrderConfirmationEmail(order: Order) {
        val message = mailSender.createMimeMessage()
        val helper = MimeMessageHelper(message, true, "UTF-8")
        helper.setTo(order.user.email)
        helper.setSubject("Order Confirmation #${order.id}")
        helper.setText(buildEmailContent(order), true)
        mailSender.send(message)
        logger.info("Email sent for order ${order.id}")
    }
    private fun buildEmailContent(order: Order): String {
        return """
            <html>
                <body>
                    <h1>Order Confirmation</h1>
                    <p>Order ID: ${order.id}</p>
                    <p>Total: €${order.totalAmount}</p>
                </body>
            </html>
        """.trimIndent()
    }
}
```

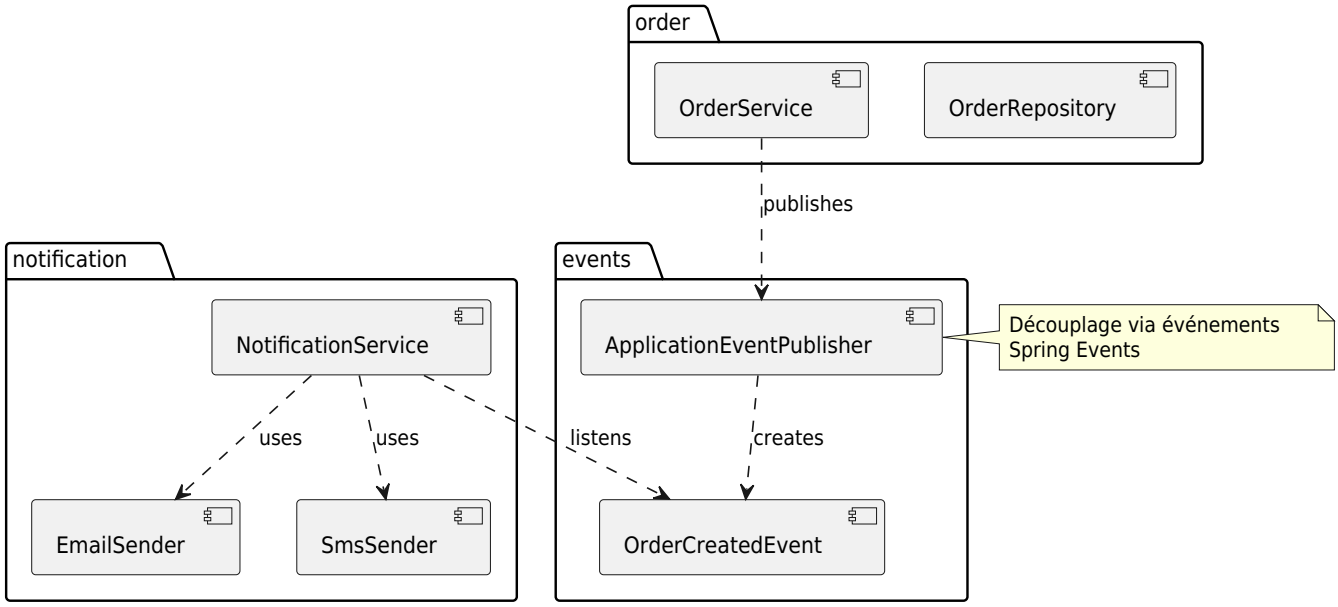
0.2 Problèmes identifiés

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

```
│       └── OrderCreatedEvent.kt
│
├── notification/                    # Domaine Notification
│   ├── domain/
│   │   ├── NotificationChannel.kt
│   │   └── NotificationLog.kt
│   ├── service/
│   │   ├── NotificationService.kt
│   │   └── sender/
│   │       ├── NotificationSender.kt (interface)
│   │       ├── EmailNotificationSender.kt
│   │       └── ConsoleNotificationSender.kt
│   ├── listener/
│   │   └── OrderNotificationListener.kt
│   ├── repository/
│   │   └── NotificationLogRepository.kt
│   └── config/
│       └── NotificationConfig.kt
│
├── product/                         # Domaine Product
│   ├── domain/
│   ├── service/
│   └── ...
│
└── user/                            # Domaine User
    ├── 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 org.springframework.context.ApplicationEvent
import java.math.BigDecimal
import java.time.Instant
import java.util.*

/**
 * Événement publié lors de la création d'une commande
 */
class OrderCreatedEvent(
    source: Any,
    val orderId: UUID,
    val userId: UUID,
    val userEmail: String,
    val totalAmount: BigDecimal,
    val items: List<OrderItemInfo>,
    val createdAt: Instant = Instant.now()
) : ApplicationEvent(source) {
    data class OrderItemInfo(
        val productName: String,
        val quantity: Int,
        val unitPrice: BigDecimal
    )
}
```

2.2 Publication de l'événement dans OrderService

```
package com.ecommerce.order.service

import com.ecommerce.order.domain.Order
import com.ecommerce.order.domain.OrderItem
import com.ecommerce.order.dto.CreateOrderDto
import com.ecommerce.order.event.OrderCreatedEvent
import com.ecommerce.order.repository.OrderRepository
import com.ecommerce.product.repository.ProductRepository
import com.ecommerce.user.repository.UserRepository
import com.ecommerce.exception.*
import org.slf4j.LoggerFactory
import org.springframework.context.ApplicationEventPublisher
import org.springframework.stereotype.Service
```

```
import org.springframework.transaction.annotation.Transactional
import java.util.*

@Service
class OrderService(
    private val orderRepository: OrderRepository,
    private val productRepository: ProductRepository,
    private val userRepository: UserRepository,
    private val eventPublisher: ApplicationEventPublisher // Injection de l'event
publisher
) {
    private val logger = LoggerFactory.getLogger(javaClass)
    @Transactional
    fun createOrder(dto: CreateOrderDto): Order {
        logger.info("Creating order for user ${dto.userId}")
        // 1. Validation
        val user = userRepository.findById(dto.userId)
            .orElseThrow { UserNotFoundException(dto.userId) }
        require(dto.items.isNotEmpty()) {
            "Order must contain at least one item"
        }
        // 2. Création de la commande
        val order = Order(user = user)
        dto.items.forEach { itemDto ->
            val product = productRepository.findById(itemDto.productId)
                .orElseThrow { ProductNotFoundException(itemDto.productId) }
            if (product.stock < itemDto.quantity) {
                throw InsufficientStockException(product.id!!)
            }
            product.decreaseStock(itemDto.quantity)
            order.addItem(OrderItem(
                product = product,
                quantity = itemDto.quantity,
                unitPrice = product.price
            ))
        }
        // 3. Sauvegarde
        val savedOrder = orderRepository.save(order)
        logger.info("Order ${savedOrder.id} created successfully")
        // 4. Publication de l'événement
        // APRÈS le commit de la transaction (voir @TransactionalEventListener)
        val event = OrderCreatedEvent(
            source = this,
            orderId = savedOrder.id!!,
            userId = user.id!!,
            userEmail = user.email,
            totalAmount = savedOrder.totalAmount,
            items = savedOrder.items.map { item ->
                OrderCreatedEvent.OrderItemInfo(
                    productName = item.product.name,
                    quantity = item.quantity,
                    unitPrice = item.unitPrice
                )
            }
        )
        eventPublisher.publishEvent(event)
```

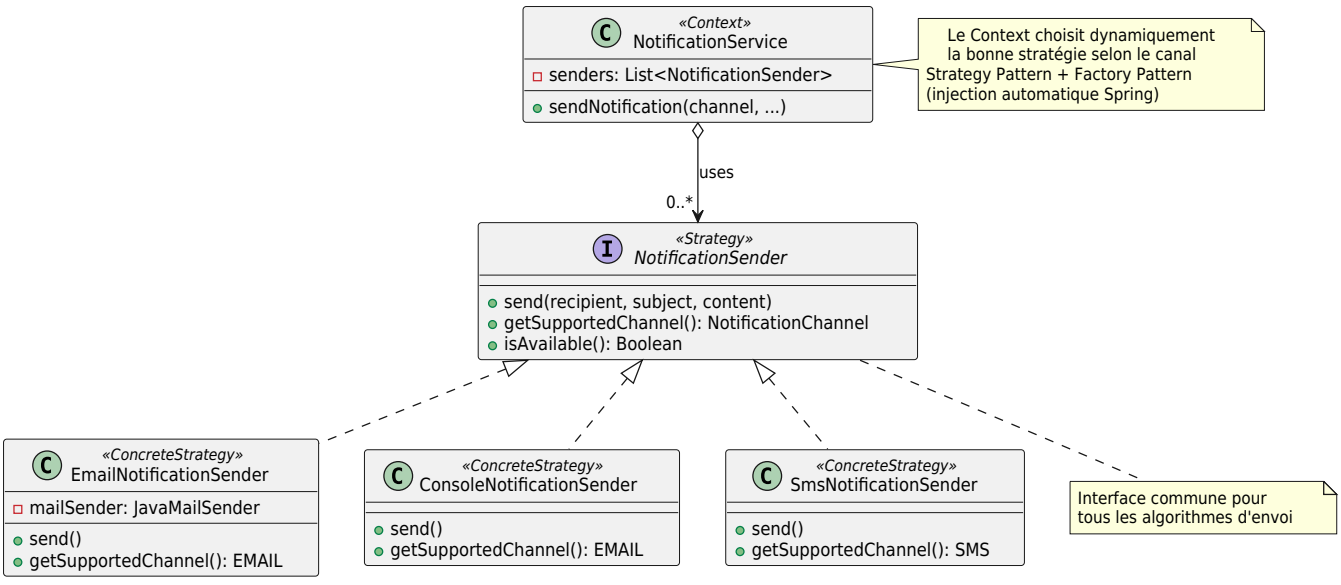
```
        logger.info("OrderCreatedEvent published for order ${savedOrder.id}")
        return savedOrder
    }
    fun getOrder(orderId: UUID): Order {
        return orderRepository.findById(orderId)
            .orElseThrow { OrderNotFoundException(orderId) }
    }
    fun getUserOrders(userId: UUID): List<Order> {
        return orderRepository.findByUserId(userId)
    }
}
```

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
 */
interface NotificationSender {
    /**
     * Envoie une notification
     * @param recipient Destinataire (email, numéro de téléphone...)
     * @param subject Sujet de la notification
     * @param content Contenu de la notification
     */
    fun send(recipient: String, subject: String, content: String)
    /**
     * Canal supporté par cette implémentation
     */
    fun getSupportedChannel(): NotificationChannel
    /**
     * Vérifie si l'envoi est disponible
     */
    fun isAvailable(): Boolean
}
```

3.2 Implémentation Console (pour dev/test)

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

import com.ecommerce.notification.domain.NotificationChannel
import org.slf4j.LoggerFactory
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty
import org.springframework.stereotype.Component

/**
 * Implémentation de test qui affiche les notifications dans la console
 * Activée quand notification.email.enabled=false
 */
@Component
@ConditionalOnProperty(
    name = ["notification.email.enabled"],
    havingValue = "false",
    matchIfMissing = true
)
class ConsoleNotificationSender : NotificationSender {
    private val logger = LoggerFactory.getLogger(javaClass)
    override fun send(recipient: String, subject: String, content: String) {
        logger.info("""
            ╔═══════════════════════════════════════════════════════════════
            ║ 📧 CONSOLE EMAIL NOTIFICATION
            ╠═══════════════════════════════════════════════════════════════
            ║ To: $recipient
            ║ Subject: $subject
            ╠═══════════════════════════════════════════════════════════════
            ║ $content
            ╚═══════════════════════════════════════════════════════════════
```

```
        """.trimIndent())
    }
    override fun getSupportedChannel() = NotificationChannel.EMAIL
    override fun isAvailable() = true
}
```

3.3 Implémentation Email (pour prod)

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

import com.ecommerce.notification.domain.NotificationChannel
import jakarta.mail.internet.MimeMessage
import org.slf4j.LoggerFactory
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty
import org.springframework.mail.javamail.JavaMailSender
import org.springframework.mail.javamail.MimeMessageHelper
import org.springframework.stereotype.Component

/**
 * Implémentation réelle avec JavaMailSender
 * Activée quand notification.email.enabled=true
 */
@Component
@ConditionalOnProperty(
    name = ["notification.email.enabled"],
    havingValue = "true"
)
class EmailNotificationSender(
    private val mailSender: JavaMailSender
) : NotificationSender {
    private val logger = LoggerFactory.getLogger(javaClass)
    override fun send(recipient: String, subject: String, content: String) {
        try {
            val message: MimeMessage = mailSender.createMimeMessage()
            val helper = MimeMessageHelper(message, true, "UTF-8")
            helper.setTo(recipient)
            helper.setSubject(subject)
            helper.setText(content, true) // true = HTML
            mailSender.send(message)
            logger.info("Email sent successfully to $recipient")
        } catch (e: Exception) {
            logger.error("Failed to send email to $recipient", e)
            throw RuntimeException("Email sending failed", e)
        }
    }
    override fun getSupportedChannel() = NotificationChannel.EMAIL
    override fun isAvailable(): Boolean {
        return try {
            // Vérifier si le serveur SMTP est configuré
            mailSender.createMimeMessage()
            true
        } catch (e: Exception) {
            logger.warn("Email sender is not available", e)
```

```
            false
        }
    }
}
```

3.4 NotificationService avec Injection des senders

```
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 org.slf4j.LoggerFactory
import org.springframework.stereotype.Service
import java.time.Instant

/**
 * Service de notification avec Pattern Factory
 * Sélectionne automatiquement le bon sender selon le canal
 */
@Service
class NotificationService(
    private val notificationSenders: List<NotificationSender>, // Spring injecte
TOUS les senders
    private val logRepository: NotificationLogRepository
) {
    private val logger = LoggerFactory.getLogger(javaClass)
    /**
     * Envoie une notification via le canal spécifié
     */
    fun sendNotification(
        channel: NotificationChannel,
        recipient: String,
        subject: String,
        content: String
    ) {
        logger.info("Sending $channel notification to $recipient")
        // Pattern Factory : sélectionner le bon sender
        val sender = notificationSenders.firstOrNull {
            it.getSupportedChannel() == channel
        } ?: run {
            logger.error("No sender found for channel $channel")
            logFailure(channel, recipient, subject, "No sender available for this
channel")
            return
        }
        // Vérifier la disponibilité
        if (!sender.isAvailable()) {
            logger.warn("Sender for $channel is not available")
            logFailure(channel, recipient, subject, "Sender unavailable")
            return
        }
```

```
        // Envoi
        try {
            sender.send(recipient, subject, content)
            logSuccess(channel, recipient, subject)
        } catch (e: Exception) {
            logger.error("Failed to send $channel notification to $recipient", e)
            logFailure(channel, recipient, subject, e.message ?: "Unknown error")
        }
    }
    private fun logSuccess(
        channel: NotificationChannel,
        recipient: String,
        subject: String
    ) {
        val log = NotificationLog(
            channel = channel,
            recipient = recipient,
            subject = subject,
            status = "SUCCESS",
            errorMessage = null,
            sentAt = Instant.now()
        )
        logRepository.save(log)
    }
    private fun logFailure(
        channel: NotificationChannel,
        recipient: String,
        subject: String,
        errorMessage: String
    ) {
        val log = NotificationLog(
            channel = channel,
            recipient = recipient,
            subject = subject,
            status = "FAILED",
            errorMessage = errorMessage,
            sentAt = Instant.now()
        )
        logRepository.save(log)
    }
    /**
     * Récupère l'historique des notifications pour un destinataire
     */
    fun getNotificationHistory(recipient: String): List<NotificationLog> {
        return logRepository.findByRecipientOrderBySentAtDesc(recipient)
    }
}
```

3.5 Entité NotificationLog

```
package com.ecommerce.notification.domain

import jakarta.persistence.*
```

```
import java.time.Instant
import java.util.*

@Entity
@Table(name = "notification_logs")
class NotificationLog(
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    var id: UUID? = null,
    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    val channel: NotificationChannel,
    @Column(nullable = false)
    val recipient: String,
    @Column(nullable = false)
    val subject: String,
    @Column(nullable = false)
    val status: String, // SUCCESS, FAILED
    @Column(length = 1000)
    val errorMessage: String?,
    @Column(nullable = false)
    val sentAt: Instant
)

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

```
package com.ecommerce.notification.repository

import com.ecommerce.notification.domain.NotificationLog
import org.springframework.data.jpa.repository.JpaRepository
import org.springframework.stereotype.Repository
import java.util.*

@Repository
interface NotificationLogRepository : JpaRepository<NotificationLog, UUID> {
    fun findByRecipientOrderBySentAtDesc(recipient: String): List<NotificationLog>
}
```

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 org.slf4j.LoggerFactory
import org.springframework.scheduling.annotation.Async
import org.springframework.stereotype.Component
import org.springframework.transaction.event.TransactionalEventListener
import org.springframework.transaction.event.TransactionPhase
import java.math.BigDecimal

/**
 * Écoute les événements OrderCreatedEvent et envoie des notifications
 *
 * @TransactionalEventListener : attend le COMMIT de la transaction
 * @Async : traitement asynchrone (ne bloque pas la réponse HTTP)
 */
@Component
class OrderNotificationListener(
    private val notificationService: NotificationService
) {
    private val logger = LoggerFactory.getLogger(javaClass)
    /**
     * Gère l'événement de création de commande
     * Phase AFTER_COMMIT : exécuté APRÈS le commit de la transaction
     */
    @Async
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    fun handleOrderCreated(event: OrderCreatedEvent) {
        logger.info("Received OrderCreatedEvent for order ${event.orderId}")
        try {
            // Envoi de la notification email
            notificationService.sendNotification(
                channel = NotificationChannel.EMAIL,
                recipient = event.userEmail,
                subject = "Order Confirmation #${event.orderId}",
                content = buildEmailContent(event)
            )
        } catch (e: Exception) {
            // Si l'email échoue, la commande reste créée
            logger.error("Failed to send notification for order ${event.orderId}",
e)
        }
    }
    private fun buildEmailContent(event: OrderCreatedEvent): String {
        val html = StringBuilder()
        html.append("<!DOCTYPE html>")
        html.append("<html><head><meta charset='UTF-8'></head><body>")
        html.append("<h1>Order Confirmation</h1>")
        html.append("<p>Thank you for your order!</p>")
        html.append("<p><strong>Order ID:</strong> ${event.orderId}</p>")
        html.append("<p><strong>Order Date:</strong> ${event.createdAt}</p>")
        html.append("<h2>Order Details</h2>")
        html.append("<table border='1' cellpadding='10' cellspacing='0'>")
        html.append("<tr><th>Product</th><th>Quantity</th><th>Unit
Price</th><th>Total</th></tr>")
        event.items.forEach { item ->
            val itemTotal = item.unitPrice.multiply(BigDecimal(item.quantity))
```

```
            html.append("<tr>")
            html.append("<td>${item.productName}</td>")
            html.append("<td>${item.quantity}</td>")
            html.append("<td>€${item.unitPrice}</td>")
            html.append("<td>€$itemTotal</td>")
            html.append("</tr>")
        }
        html.append("</table>")
        html.append("<p><strong>Total Amount: €${event.totalAmount}</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
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 io.mockk.*
import org.assertj.core.api.Assertions.*
import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.DisplayName
import org.junit.jupiter.api.Test

@DisplayName("NotificationService - Unit Tests")
class NotificationServiceTest {
    private lateinit var logRepository: NotificationLogRepository
    private lateinit var emailSender: NotificationSender
    private lateinit var smsSender: NotificationSender
    private lateinit var notificationService: NotificationService
    @BeforeEach
    fun setUp() {
        logRepository = mockk(relaxed = true)
        emailSender = mockk()
        smsSender = mockk()
        every { emailSender.getSupportedChannel() } returns
NotificationChannel.EMAIL
        every { smsSender.getSupportedChannel() } returns NotificationChannel.SMS
        notificationService = NotificationService(
            notificationSenders = listOf(emailSender, smsSender),
            logRepository = logRepository
        )
    }
    @Test
    @DisplayName("Should send notification when sender is available")
    fun `sendNotification with available sender should send successfully`() {
        // Given
        every { emailSender.isAvailable() } returns true
        every { emailSender.send(any(), any(), any()) } just Runs
        // When
        notificationService.sendNotification(
            NotificationChannel.EMAIL,
            "test@example.com",
            "Test Subject",
            "Test Content"
        )
        // Then
        verify(exactly = 1) {
            emailSender.send("test@example.com", "Test Subject", "Test Content")
        }
        verify(exactly = 1) {
```

```
            logRepository.save(match { it.status == "SUCCESS" })
        }
    }
    @Test
    @DisplayName("Should log failure when sender throws exception")
    fun `sendNotification when sender fails should log error`() {
        // Given
        every { emailSender.isAvailable() } returns true
        every { emailSender.send(any(), any(), any()) } throws
RuntimeException("SMTP error")
        // When
        notificationService.sendNotification(
            NotificationChannel.EMAIL,
            "test@example.com",
            "Test",
            "Content"
        )
        // Then
        verify(exactly = 1) {
            logRepository.save(match {
                it.status == "FAILED" && it.errorMessage?.contains("SMTP error") ==
true
            })
        }
    }
    @Test
    @DisplayName("Should not send when no sender for channel")
    fun `sendNotification with unsupported channel should log failure`() {
        // Given
        // Pas de sender pour PUSH
        // When
        notificationService.sendNotification(
            NotificationChannel.PUSH,
            "test@example.com",
            "Test",
            "Content"
        )
        // Then
        verify(exactly = 0) {
            emailSender.send(any(), any(), any())
            smsSender.send(any(), any(), any())
        }
        verify(exactly = 1) {
            logRepository.save(match {
                it.status == "FAILED" &&
                it.errorMessage?.contains("No sender available") == true
            })
        }
    }
    @Test
    @DisplayName("Should not send when sender is unavailable")
    fun `sendNotification when sender unavailable should log failure`() {
        // Given
        every { emailSender.isAvailable() } returns false
        // When
        notificationService.sendNotification(
```

```
            NotificationChannel.EMAIL,
            "test@example.com",
            "Test",
            "Content"
        )
        // Then
        verify(exactly = 0) { emailSender.send(anyString(), anyString(),
anyString()) }
        verify(exactly = 1) {
            logRepository.save(match {
                it.status == "FAILED" &&
                it.errorMessage?.contains("unavailable") == true
            })
        }
    }
}
```

5.2 Test d'intégration avec capture d'événements

```
package com.ecommerce.order.controller

import com.ecommerce.order.event.OrderCreatedEvent
import org.assertj.core.api.Assertions.*
import org.junit.jupiter.api.BeforeEach
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.test.web.servlet.post
import org.springframework.transaction.annotation.Transactional

@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("test")
@Transactional
@DisplayName("OrderController - Integration Tests with Events")
class OrderControllerEventIntegrationTest {
    @Autowired
    private lateinit var mockMvc: MockMvc
    @Autowired
    private lateinit var testEventListener: TestEventListener
    @BeforeEach
    fun setUp() {
        testEventListener.reset()
    }
```

```
    @Test
    @DisplayName("POST /orders should publish OrderCreatedEvent")
    fun `createOrder should publish event after successful creation`() {
        // Given
        val orderRequest = """
            {
                "userId": "${setupUserId()}",
                "items": [
                    {
                        "productId": "${setupProductId()}",
                        "quantity": 2
                    }
                ]
            }
        """.trimIndent()
        // When
        mockMvc.post("/orders") {
            contentType = MediaType.APPLICATION_JSON
            content = orderRequest
        }.andExpect {
            status { isCreated() }
        }
        // Then - Vérifier que l'événement a été publié
        Thread.sleep(500) // Attendre le traitement asynchrone
        val events = testEventListener.getReceivedEvents()
        assertThat(events).hasSize(1)
        val event = events[0]
        assertThat(event.userEmail).isNotEmpty()
        assertThat(event.totalAmount).isGreaterThan(java.math.BigDecimal.ZERO)
        assertThat(event.items).isNotEmpty()
    }
    private fun setupUserId(): String {
        // Créer un utilisateur de test
        // TODO: implémenter
        return java.util.UUID.randomUUID().toString()
    }
    private fun setupProductId(): String {
        // Créer un produit de test
        // TODO: implémenter
        return java.util.UUID.randomUUID().toString()
    }
    /**
     * Configuration de test pour capturer les événements
     */
    @TestConfiguration
    class TestConfig {
        @Bean
        @Primary
        fun testEventListener(): TestEventListener {
            return TestEventListener()
        }
    }
    /**
     * Listener de test pour vérifier la publication d'événements
     */
    class TestEventListener {
```

```
        private val receivedEvents = mutableListOf<OrderCreatedEvent>()
        @EventListener
        fun handleEvent(event: OrderCreatedEvent) {
            receivedEvents.add(event)
        }
        fun getReceivedEvents(): List<OrderCreatedEvent> = receivedEvents.toList()
        fun reset() {
            receivedEvents.clear()
        }
    }
}
```

5.3 Test unitaire du Listener

```
package com.ecommerce.notification.listener

import com.ecommerce.notification.domain.NotificationChannel
import com.ecommerce.notification.service.NotificationService
import com.ecommerce.order.event.OrderCreatedEvent
import io.mockk.*
import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.DisplayName
import org.junit.jupiter.api.Test
import java.math.BigDecimal
import java.util.*

@DisplayName("OrderNotificationListener - Unit Tests")
class OrderNotificationListenerTest {
    private lateinit var notificationService: NotificationService
    private lateinit var listener: OrderNotificationListener
    @BeforeEach
    fun setUp() {
        notificationService = mockk(relaxed = true)
        listener = OrderNotificationListener(notificationService)
    }
    @Test
    @DisplayName("Should send email notification when order is created")
    fun `handleOrderCreated should send email notification`() {
        // Given
        val event = OrderCreatedEvent(
            source = this,
            orderId = UUID.randomUUID(),
            userId = UUID.randomUUID(),
            userEmail = "customer@example.com",
            totalAmount = BigDecimal.valueOf(100.00),
            items = emptyList()
        )
        // When
        listener.handleOrderCreated(event)
        // Then
        verify(exactly = 1) {
            notificationService.sendNotification(
                NotificationChannel.EMAIL,
```

```
                "customer@example.com",
                match { it.contains("Order Confirmation") },
                any()
            )
        }
    }
    @Test
    @DisplayName("Should include order details in email content")
    fun `handleOrderCreated should include all order information`() {
        // Given
        val orderId = UUID.randomUUID()
        val event = OrderCreatedEvent(
            source = this,
            orderId = orderId,
            userId = UUID.randomUUID(),
            userEmail = "customer@example.com",
            totalAmount = BigDecimal("250.00"),
            items = listOf(
                OrderCreatedEvent.OrderItemInfo("Product A", 2,
BigDecimal("100.00")),
                OrderCreatedEvent.OrderItemInfo("Product B", 1,
BigDecimal("50.00"))
            )
        )
        // When
        listener.handleOrderCreated(event)
        // Then
        verify {
            notificationService.sendNotification(
                NotificationChannel.EMAIL,
                "customer@example.com",
                any(),
                match { content ->
                    content.contains(orderId.toString()) &&
                    content.contains("Product A") &&
                    content.contains("Product B") &&
                    content.contains("250.00")
                }
            )
        }
    }
}
```

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)

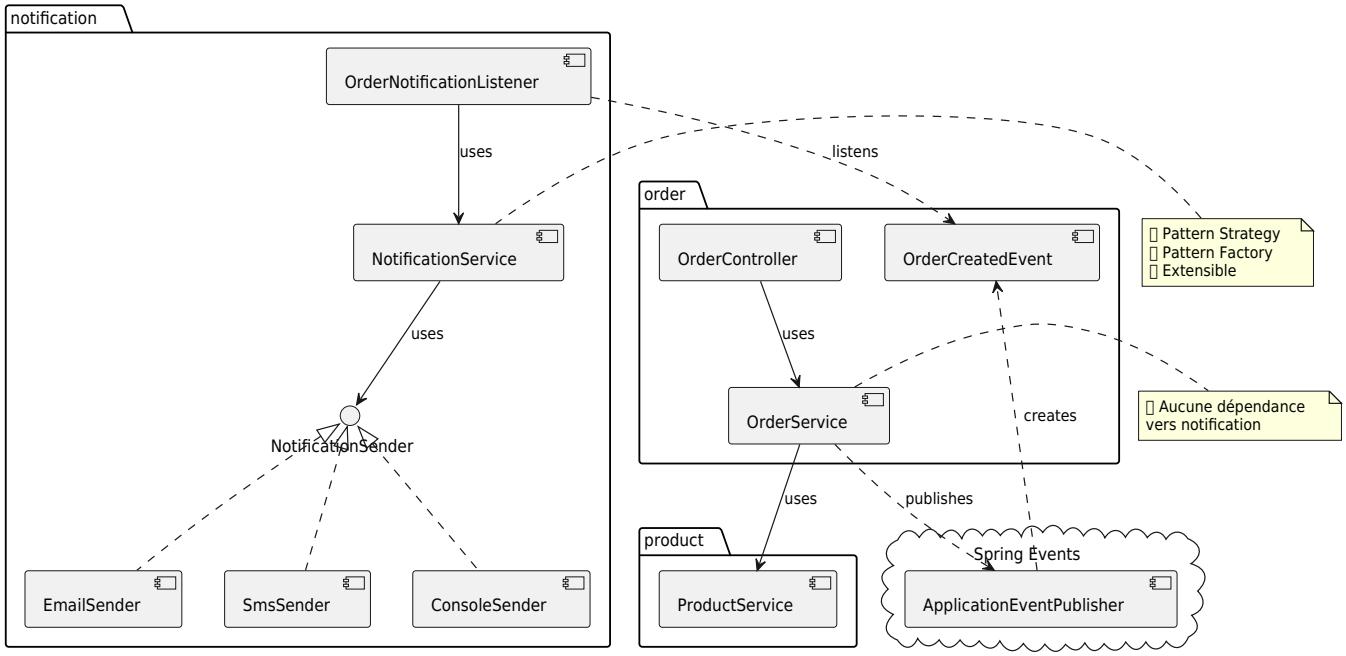
Challenge : Ajouter un canal SMS sans modifier le code existant (principe Open/Closed)

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

import com.ecommerce.notification.domain.NotificationChannel
import org.slf4j.LoggerFactory
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty
import org.springframework.stereotype.Component

@Component
@ConditionalOnProperty(
    name = ["notification.sms.enabled"],
    havingValue = "true"
)
class SmsNotificationSender : NotificationSender {
    private val logger = LoggerFactory.getLogger(javaClass)
    override fun send(recipient: String, subject: String, content: String) {
        // Intégration avec Twilio, AWS SNS, etc.
        logger.info("Sending SMS to $recipient: $content")
        // Implémentation simplifiée pour la demo
    }
    override fun getSupportedChannel() = NotificationChannel.SMS
    override fun isAvailable() = true
}
```

Points à noter :

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

Récapitulatif : Architecture finale



Partie 7 : Visualiser les emails avec MailPit (Bonus : 10min)

Mailpit = Serveur SMTP de test avec interface web moderne

- Capture tous les emails envoyés par l'application
- Interface web pour visualiser les emails
- Aucune configuration SMTP complexe
- Parfait pour le développement

7.1 Ajouter Mailpit au docker-compose.yml

Ajouter ce service dans votre fichier **`docker-compose.yml`** :

```
  mailpit:
    image: axllent/mailpit:latest
    container_name: ecommerce-mailpit
    ports:
      - "1025:1025"  # SMTP
      - "8025:8025"  # Web UI
    networks:
      - ecommerce-network
```

```
# Démarrer Mailpit
docker-compose up -d mailpit

# Vérifier que Mailpit est démarré
docker ps | grep mailpit
```

7.2 Configuration Spring

Modifier le fichier **src/main/resources/application-dev.properties** :

```
# Mailpit configuration
spring.mail.host=localhost
spring.mail.port=1025
spring.mail.username=
spring.mail.password=
spring.mail.properties.mail.smtp.auth=false
spring.mail.properties.mail.smtp.starttls.enable=false

# Notification settings
notification.email.enabled=true
notification.email.from=noreply@ecommerce-demo.com

# Logs pour voir les envois
logging.level.org.springframework.mail=DEBUG
```

Modifier également **src/test/resources/application-test.properties** :

```
# Mailpit pour les tests
spring.mail.host=localhost
spring.mail.port=1025

notification.email.enabled=true
notification.email.from=test@ecommerce-demo.com
```

7.3 Améliorer les logs dans EmailNotificationSender

Modifier la méthode **send()** dans **EmailNotificationSender.kt** :

```
override fun send(recipient: String, subject: String, content: String) {
    try {
        val message = mailSender.createMimeMessage()
        val helper = MimeMessageHelper(message, true, "UTF-8")
        helper.setFrom(fromEmail)
        helper.setTo(recipient)
        helper.setSubject(subject)
        helper.setText(content, true)
        mailSender.send(message)
        logger.info("□ Email sent to: $recipient - Subject: $subject")
        logger.info("□ View in Mailpit: http://localhost:8025")
    } catch (e: Exception) {
        logger.error("□ Failed to send email to $recipient", e)
        throw RuntimeException("Email sending failed", e)
    }
}
```

7.4 Test manuel

```
# 1. Démarrer Mailpit (si pas déjà fait)
docker-compose up -d mailpit

# 2. Lancer l'application avec le profil dev
mvn spring-boot:run -P dev

# 3. Créer une commande pour déclencher l'envoi d'email
curl -X POST http://localhost:8080/api/orders \
  -H "Content-Type: application/json" \
  -d '{
    "customerId": 1,
    "customerEmail": "test@example.com",
    "items": [
      {
        "productId": 1,
        "quantity": 2,
        "price": 29.99
      }
    ]
  }'

# 7.4 Ouvrir l'interface Mailpit
open http://localhost:8025
# Ou dans votre navigateur : http://localhost:8025
```

Vérifications à effectuer :

```
1. Ouvrir http://localhost:8025 dans votre navigateur
2. Vérifier qu'un email apparaît dans la liste
3. Cliquer sur l'email pour voir son contenu
4. Vérifier que le contenu HTML est correct
5. Vérifier le sujet : "Order Confirmation #XXX"
6. Vérifier les détails de la commande dans l'email
```

7.5 Interface Mailpit

L'interface web de Mailpit (http://localhost:8025) permet de :

- Voir tous les emails envoyés
- Rechercher dans les emails
- Prévisualiser le HTML et le texte brut
- Voir les pièces jointes
- Supprimer les emails
- Tester le responsive design des emails

7.6 Configuration pour la production

⚠ **Mailpit est uniquement pour le développement !**

En **développement** (Mailpit) :

```
spring.mail.host=localhost
spring.mail.port=1025
notification.email.enabled=true
```

En **production** (SMTP réel - exemple avec Gmail) :

```
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

notification.email.enabled=true
notification.email.from=${EMAIL_FROM}
```

Bonnes pratiques :

- Ne **jamais** commiter les credentials SMTP dans le code
- Utiliser des **variables d'environnement**
- Activer **TLS/SSL** en production
- Utiliser des **app passwords** (Gmail, Outlook, etc.)

7.7 Commandes utiles

```
# Démarrer uniquement Mailpit
docker-compose up -d mailpit

# Voir les logs de Mailpit
docker logs -f ecommerce-mailpit

# Redémarrer Mailpit
docker-compose restart mailpit

# Arrêter Mailpit
docker-compose stop mailpit

# Supprimer le conteneur Mailpit
docker-compose down mailpit
```

7.8 Dépannage

Problème : Les emails n'apparaissent pas dans Mailpit

```
# 1. Vérifier que Mailpit est démarré
docker ps | grep mailpit

# 2. Vérifier les logs de l'application
# Rechercher : "Email sent to:" ou "Failed to send email"

# 3. Vérifier que le profil dev est actif
# Dans les logs au démarrage : "The following profiles are active: dev"

# 4. Tester la connexion SMTP
telnet localhost 1025
```

Problème : “Connection refused” sur le port 1025

```
# Vérifier que le port 1025 n'est pas déjà utilisé
lsof -i :1025

# Si occupé, changer le port dans docker-compose.yml et application-dev.properties
```

7.3 Améliorer les logs dans EmailNotificationSender

```
override fun send(recipient: String, subject: String, content: String) {
    try {
        val message = mailSender.createMimeMessage()
        val helper = MimeMessageHelper(message, true, "UTF-8")
        helper.setFrom(fromEmail)
        helper.setTo(recipient)
        helper.setSubject(subject)
        helper.setText(content, true)
        mailSender.send(message)
        logger.info("□ Email sent to: $recipient")
        logger.debug("□ Check MailHog UI: http://localhost:8025")
    } catch (e: Exception) {
        logger.error("□ Failed to send email", e)
        throw RuntimeException("Email sending failed", e)
    }
}
```

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
- MockK Documentation
- Kotlin Data Classes

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
http://slamwiki2.kobject.net/eadl/bloc3/dev_av/td4?rev=1762755255

Last update: **2025/11/10 07:14**