

GALİP AKTAŞ BPN CASE ÇALIŞMASI

Problemlı Kod - 1 :

Aşağıdaki calculatePrice metodu, onlarca satırda yayılmış karmaşık if-else yapıları, çeşitli indirim ve vergi hesaplamaları içermektedir. Metodu okumak, anlamak ve bakımını yapmak zor. Nasıl bir çözüm üretirisiniz ?

```
Calculate Price Method ▾

1  public double calculatePrice(Order order) {
2      double basePrice = 0.0;
3
4      // Baz fiyat hesaplama
5      for (String product : order.getProducts()) {
6          if (product.equals("Laptop")) {
7              basePrice += 7000;
8          } else if (product.equals("Phone")) {
9              basePrice += 3000;
10         } else if (product.equals("Monitor")) {
11             basePrice += 1500;
12         } else {
13             basePrice += 500;
14         }
15     }
16
17     // İndirim hesaplama
18     if (order.getCustomerEmail().endsWith("@corporate.com")) {
19         basePrice = basePrice * 0.90;
20     } else {
21         if (basePrice > 10000) {
22             basePrice = basePrice - 500;
23         } else if (basePrice > 5000) {
24             basePrice = basePrice - 200;
25         } else {
26             basePrice = basePrice * 0.95;
27         }
28     }
29
30     // Vergi hesaplama
31     if (order.getShippingAddress().contains("Istanbul")) {
32         basePrice += basePrice * 0.18;
33     } else {
34         basePrice += basePrice * 0.08;
35     }
36
37     // Daha pek çok if-else...
38     return basePrice;
39 }
40
```

Çözüm - 1 :

Sorunlar:

1 - Tek Sorumluluk İlkesi (SRP) İhlali:

calculatePrice() metodu birden fazla işlem yapıyordu:

Ürün fiyatlarını hesaplama,

İndirimleri belirleme,

Vergi oranlarını ekleme,

Tek bir metot, çok fazla sorumluluk taşıdığı için okunabilirliği ve test edilebilirliği düşük.

2 - Magic Numbers Kullanımı:

Kod içerisinde sabit değerler (örneğin: 200, 0.95, 0.18) doğrudan yazılmıştı.

Değerlerin değiştirilmesi gerektiğinde kodun içine müdahale etmek zorunlu hale geliyor.

3 - Hardcoded Değerlerin Kullanılması:

Vergi oranları, indirim yüzdeleri ve ürün fiyatları gibi değerler kod içine gömülü.

Konfigüre edilebilir olmaması nedeniyle değiştirilmesi kodda değişiklik gerektiriyor.

4 - If-Else Bloklarının Aşırı Kullanımı:

Cok fazla if-else bloğu var, bu da kodun okunabilirliğini düşürüyor.

Örneğin, product.equals("Laptop") gibi ifadeler yerine daha sürdürülebilir bir yapı kurulmalı.

5 - Finansal Hassasiyet Eksikliği:

double veri tipi kullanılmış, ancak finansal işlemlerde BigDecimal kullanımı daha doğrudur.

BigDecimal, hassas hesaplamalar için daha güvenilir ve hatasız sonuçlar üretir.

6 - Null Güvenliği Eksikliği:

order.getProducts(), order.getShippingAddress() gibi çağrılar null olabilir ve hata fırlatabilir.

NullPointerException (NPE) hatalarını önlemek için null check uygulanmalıdır.

Yapılan İyileştirmeler:

Kodun veritabanı destekli, tamamen dinamik ve sürdürülebilir hale gelmesi için aşağıdaki adımları uyguladım:

1 - Kod Modüller Hale Getirildi:

calculatePrice() metodundaki işlemler farklı metotlara ayrıldı:

calculateBasePrice() → Ürün fiyatlarını hesaplar.

applyDiscount() → İndirimleri belirler.

applyTax() → Vergileri ekler.

Böylece kod okunaklı hale geldi ve her metot tek bir sorumluluk üstlendi.

2 - Sabit Değerler Veritabanına Taşındı:

ProductConfig Tablosu(Ürünlerin Fiyatları), DiscountConfig Tablosu (İndirim Koşulları) ve TaxConfig Tablosu (Vergi Oranları) tabloları oluşturuldu.

Ürün fiyatları, indirim oranları ve vergi yüzdeleri gibi değerler Hibernate ile yönetilebilir hale getirildi.

Kod içinde sabit değer kalmadı, her şey veritabanından okunuyor.

3 - Spring Data JPA Kullanıldı:

Hibernate kullanılarak ProductConfigEntity, DiscountConfigEntity ve TaxConfigEntity sınıfları oluşturuldu.

Bu entity'ler sabit değerleri yönetmek için repository ve service katmanlarıyla entegre edildi.

4 - Stream API ile Kod Temizlendi:

for döngüsü yerine Stream API kullanıldı.

reduce() metodu ile daha okunaklı ve fonksiyonel kod yazıldı.

5 - Null Kontrolleri Eklendi:

Optional.ofNullable() ile null güvenliği sağlandı.

Böylece NullPointerException hatalarının önüne geçildi.

6- BigDecimal Kullanımı ile Finansal Hassasiyet Artırıldı:

double yerine BigDecimal kullanıldı, böylece hassas hesaplamalarda hata payı sıfıra indirildi.

Vergi ve indirim oranları BigDecimal ile hassas işlemler yapacak şekilde uyarlandı.

Tablolar:

ProductConfig Tablosu (Ürünlerin Fiyatları):

id	product	price
1	Laptop	7000
2	Phone	3000
3	Monitor	1500

DiscountConfig Tablosu (İndirim Koşulları):

id	discount_type	key_value	discount_value
1	eMail	@corporate.com	0.90
2	bulk	10000	500
3	bulk	5000	200

TaxConfig Tablosu (Vergi Oranları):

id	city	tax_rate
1	istanbul	0.18
2	default	0.08

sırasıyla entity, repository ve servis sınıfları:

```
1 import jakarta.persistence.*;
2 import java.math.BigDecimal;
3
4 @Entity
5 @Table(name = "product_config")
6 public class ProductConfigEntity {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
11
12    @Column(name = "product_name", unique = true, nullable = false)
13    private String productName;
14
15    @Column(name = "price", nullable = false)
16    private BigDecimal price;
17
18    public String getProductName() { return productName; }
19    public BigDecimal getPrice() { return price; }
20}
21
```

```
1 import jakarta.persistence.*;
2 import java.math.BigDecimal;
3
4 @Entity
5 @Table(name = "discount_config")
6 public class DiscountConfigEntity {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
11
12    @Column(name = "discount_type", nullable = false)
13    private String discountType;
14
15    @Column(name = "key_value", nullable = false)
16    private String keyValue;
17
18    @Column(name = "discount_value", nullable = false)
19    private BigDecimal discountValue;
20
21    public String getDiscountType() { return discountType; }
22    public String getKeyValue() { return keyValue; }
23    public BigDecimal getDiscountValue() { return discountValue; }
24
25 import jakarta.persistence.*;
26 import java.math.BigDecimal;
27
28 @Entity
29 @Table(name = "tax_config")
30 public class TaxConfigEntity {
31
32     @Id
33     @GeneratedValue(strategy = GenerationType.IDENTITY)
34     private Long id;
35
36     @Column(name = "city", unique = true, nullable = false)
37     private String city;
38
39     @Column(name = "tax_rate", nullable = false)
40     private BigDecimal taxRate;
41
42     public String getCity() { return city; }
43     public BigDecimal getTaxRate() { return taxRate; }
44}
45
```

```

1 // repository class kodları tek bir ekran görüntüsünde:
2
3 //product config repo
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import java.util.Optional;
6
7 public interface ProductConfigRepository extends JpaRepository<ProductConfigEntity, Long> {
8     Optional<ProductConfigEntity> findByProductName(String productName);
9 }
10
11 //discount config repo
12 import org.springframework.data.jpa.repository.JpaRepository;
13 import java.util.Optional;
14
15 public interface DiscountConfigRepository extends JpaRepository<DiscountConfigEntity, Long> {
16     Optional<DiscountConfigEntity> findByKeyValue(String keyValue);
17 }
18
19
20 // tax config repo
21 import org.springframework.data.jpa.repository.JpaRepository;
22 import java.util.Optional;
23
24 public interface TaxConfigRepository extends JpaRepository<TaxConfigEntity, Long> {
25     Optional<TaxConfigEntity> findByCity(String city);
26 }
27
28

```

```

1 //product config service
2 import org.springframework.stereotype.Service;
3 import java.math.BigDecimal;
4
5 @Service
6 public class ProductConfigService {
7
8     private final ProductConfigRepository productConfigRepository;
9
10    public ProductConfigService(ProductConfigRepository productConfigRepository) {
11        this.productConfigRepository = productConfigRepository;
12    }
13
14    public BigDecimal getProductPrice(String productName) {
15        return productConfigRepository.findByProductName(productName)
16            .map(ProductConfigEntity::getPrice)
17            .orElseThrow(() -> new RuntimeException("Product not found: " + productName));
18    }
19

```

```

59 //tax config service
60 import org.springframework.stereotype.Service;
61 import java.math.BigDecimal;
62
63 @Service
64 public class TaxConfigService {
65
66     private final TaxConfigRepository taxConfigRepository;
67
68    public TaxConfigService(TaxConfigRepository taxConfigRepository) {
69        this.taxConfigRepository = taxConfigRepository;
70    }
71
72    public BigDecimal getTaxRate(String city) {
73        return taxConfigRepository.findByCity(city)
74            .map(TaxConfigEntity::getTaxRate)
75            .orElse(new BigDecimal("0.08")); // Varsayılan vergi oranı
76    }
77
78

```

```

21 //discount config service
22 import org.springframework.stereotype.Service;
23 import java.math.BigDecimal;
24
25 import org.springframework.stereotype.Service;
26 import java.math.BigDecimal;
27 import java.util.Optional;
28
29 @Service
30 public class DiscountConfigService {
31
32     private final DiscountConfigRepository discountConfigRepository;
33
34     public DiscountConfigService(DiscountConfigRepository discountConfigRepository) {
35         this.discountConfigRepository = discountConfigRepository;
36     }
37
38     public BigDecimal getDiscount(String customerEmail, BigDecimal basePrice) {
39         // Eğer müşteri kurumsal e-posta kullanıyorsa, yüzdelen indirim uygula
40         Optional<DiscountConfigEntity> corporateDiscount = discountConfigRepository.findByKeyValue("@corporate.com");
41         if (customerEmail.endsWith("@corporate.com")) {
42             return corporateDiscount.map(DiscountConfigEntity::getDiscountValue).orElse(BigDecimal.ONE);
43         }
44
45         // Eğer sipariş tutarı belirli eşikleri geçiyorsa, sabit indirim uygula
46         Optional<DiscountConfigEntity> bulkDiscount1000 = discountConfigRepository.findByKeyValue("10000");
47         Optional<DiscountConfigEntity> bulkDiscount5000 = discountConfigRepository.findByKeyValue("5000");
48
49         if (basePrice.compareTo(new BigDecimal("10000")) > 0) {
50             return basePrice.subtract(bulkDiscount1000.map(DiscountConfigEntity::getDiscountValue).orElse(BigDecimal.ZERO));
51         } else if (basePrice.compareTo(new BigDecimal("5000")) > 0) {
52             return basePrice.subtract(bulkDiscount5000.map(DiscountConfigEntity::getDiscountValue).orElse(BigDecimal.ZERO));
53         }
54
55         return basePrice; // Eğer hiçbir indirim kriterine uymuyorsa fiyatı değiştirme
56     }
57 }

```

kodun son hali aşağıdaki şekilde oldu:

```

1 import java.math.BigDecimal;
2 import java.util.List;
3 import java.util.Optional;
4
5 public class PriceCalculator {
6
7     private final ProductConfigService productConfigService;
8     private final DiscountConfigService discountConfigService;
9     private final TaxConfigService taxConfigService;
10
11    public PriceCalculator(ProductConfigService productConfigService, DiscountConfigService discountConfigService, TaxConfigService taxConfigService) {
12        this.productConfigService = productConfigService;
13        this.discountConfigService = discountConfigService;
14        this.taxConfigService = taxConfigService;
15    } //servisler ile veri tabanından sabit değerler çekilerek alınıyor
16
17    public BigDecimal calculatePrice(Order order) {
18        BigDecimal basePrice = calculateBasePrice(order);
19        basePrice = applyDiscount(order, basePrice);
20        basePrice = applyTax(order, basePrice);
21        return basePrice;
22    }
23
24    //private metodlar ile her bir işlem ayrı ayrı metodlarda yapılıyor
25    //İç içe if ve for yapıları yerine stream api kullandım
26    private BigDecimal calculateBasePrice(Order order) {
27        return Optional.ofNullable(order.getProducts())//null check ekledim
28            .orElse(List.of())
29            .stream()
30            .map(productConfigService::getProductPrice)
31            .reduce(BigDecimal.ZERO, BigDecimal::add);
32    }
33
34    private BigDecimal applyDiscount(Order order, BigDecimal basePrice) {
35        return discountConfigService.getDiscount(order.getCustomerEmail(), basePrice);
36    }
37
38    private BigDecimal applyTax(Order order, BigDecimal basePrice) {
39        return basePrice.add(basePrice.multiply(taxConfigService.getTaxRate(order.getShippingAddress())));
40    }
41 }

```

Problemlı Kod - 2 :

Aşağıdaki kod, çok katmanlı iç içe if-else yapılarıyla karmaşık ve anlaşılması güç bir yapı oluşturmuş. Nasıl bir çözüm üretirsiniz ?

Shipping Cost Calculator ▾

```
1 ~ public String getShippingCost(int weight, String shippingRegion, boolean isFragile) {  
2 ~     if (shippingRegion.equals("EU")) {  
3 ~         if (weight < 5) {  
4 ~             if (isFragile) {  
5 ~                 return "10 USD";  
6 ~             } else {  
7 ~                 return "5 USD";  
8 ~             }  
9 ~         } else {  
10 ~             if (isFragile) {  
11 ~                 return "20 USD";  
12 ~             } else {  
13 ~                 return "15 USD";  
14 ~             }  
15 ~         }  
16 ~     } else {  
17 ~         if (weight < 5) {  
18 ~             if (isFragile) {  
19 ~                 return "15 USD";  
20 ~             } else {  
21 ~                 return "10 USD";  
22 ~             }  
23 ~         } else {  
24 ~             if (isFragile) {  
25 ~                 return "25 USD";  
26 ~             } else {  
27 ~                 return "20 USD";  
28 ~             }  
29 ~         }  
30 ~     }  
31 ~ }  
32 }
```

Çözüm – 2:

Sorunlar:

Kod derin bir iç içe if-else yapısına sahip, bu da okunabilirliği zorlaştırmıyor.

Sabit değerler doğrudan kodun içine gömülümiş.

Yeni bir bölge eklemek gerektiğinde kodun içine müdahale etmek gerekiyor.

getShippingCost() metodu hem iş kurallarını hem de fiyatları yönetiyor.

Yeni bir bölge veya yeni fiyatlandırma eklemek gerektiğinde kod karmaşık hale gelecek.

Gelecekte fiyat değişiklikleri yapılrsa kodun içini değiştirmek gerekecek.

Çözüm yöntemim:

Tüm sabit değerleri veritabanına taşı: Değişiklikler için kodu güncellemek gerekmeyecek.

Shipping fiyatlarını yönetmek için veritabanında yeni tablo oluşturur.

If-Else bloklarını temizleyip daha temiz bir yapı kuracağım.

Tüm hesaplamaları bağımsız servis katmanlarında yöneteceğim.

1 adet tablo ile sorun çözülebilir.

ShippingCostConfig tablosu:

id	region	weight_threshold	is_fragile	cost
1	EU	5	true	10
2	EU	5	false	5
3	EU	10000	true	20
4	EU	10000	false	15
5	OTHER	5	true	15
6	OTHER	5	false	10
7	OTHER	10000	true	25
8	OTHER	10000	false	20

***Not: bu tabloya ayrıca currency kolonu da ekleyebilirim. Farklı ülkelerdeki farklı para birimleri için ancak kısa tutmak için tüm para brimlerini orijinal koddaki gibi USD kabul ederek ilerliyorum.

***Not2: orijinal kodda ağırlık 5'ten büyük mü ya da küçük mü diye bir kontrol yapılıyor. 5'ten büyük bir ağırlığı temsilen tabloda "10000" değerini kullandım.

Entity:

```
1 import jakarta.persistence.*;
2 import java.math.BigDecimal;
3
4 @Entity
5 @Table(name = "shipping_cost_config")
6 public class ShippingCostConfigEntity {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
11
12    @Column(name = "region", nullable = false)
13    private String region;
14
15    @Column(name = "weight_threshold", nullable = false)
16    private int weightThreshold;
17
18    @Column(name = "is_fragile", nullable = false)
19    private boolean isFragile;
20
21    @Column(name = "cost", nullable = false)
22    private BigDecimal cost;
23
24    public String getRegion() { return region; }
25    public int getWeightThreshold() { return weightThreshold; }
26    public boolean isFragile() { return isFragile; }
27    public BigDecimal getCost() { return cost; }
28
29 }
```

Repository:

```
1 import org.springframework.data.jpa.repository.JpaRepository;
2 import java.util.Optional;
3
4 public interface ShippingCostConfigRepository extends JpaRepository<ShippingCostConfigEntity, Long> {
5     Optional<ShippingCostConfigEntity> findByRegionAndWeightThresholdAndIsFragile(String region, int weightThreshold, boolean isFragile);
6 }
7
```

Service:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class ShippingCostConfigService {
6
7     private final ShippingCostConfigRepository shippingCostConfigRepository;
8
9     public ShippingCostConfigService(ShippingCostConfigRepository shippingCostConfigRepository) {
10         this.shippingCostConfigRepository = shippingCostConfigRepository;
11     }
12
13     public BigDecimal getShippingCost(String region, int weight, boolean isFragile) {
14         int weightThreshold = (weight < 5) ? 5 : 10000;
15         return shippingCostConfigRepository.findByRegionAndWeightThresholdAndIsFragile(region, weightThreshold, isFragile)
16             .map(ShippingCostConfigEntity::getCost)
17             .orElseThrow(() -> new RuntimeException("Shipping cost not found for given parameters"));
18     }
19 }
20
```

price calculator son hali:

```
1 import java.math.BigDecimal;
2
3 public class ShippingCostCalculator {
4
5     private final ShippingCostConfigService shippingCostConfigService;
6
7     public ShippingCostCalculator(ShippingCostConfigService shippingCostConfigService) {
8         this.shippingCostConfigService = shippingCostConfigService;
9     }
10
11     public BigDecimal calculateShippingCost(int weight, String shippingRegion, boolean isFragile) {
12         return shippingCostConfigService.getShippingCost(shippingRegion, weight, isFragile);
13     }
14 }
15
```

Problemlı Kod - 3 :

Aşağıda InvoiceService, doğrudan SqlDatabase sınıfını new yaparak kullanıyor. Başka bir veri kaynağına (ör. NoSQL veya Mock DB) geçmek istersek kodu değiştirmek gerekiyor. Ayrıca test yazmak da zorlaşıyor. Nasıl bir çözüm üretirsiniz ?

```
Invoice Service ▾

1 ~ public class InvoiceService {
2     private SqlDatabase sqlDatabase = new SqlDatabase();
3
4 ~     public void createInvoice(Invoice invoice) {
5         sqlDatabase.connect();
6         sqlDatabase.insertInvoice(invoice);
7         sqlDatabase.disconnect();
8     }
9 }
10
11 ~ class SqlDatabase {
12     public void connect() { /* ... */ }
13     public void insertInvoice(Invoice inv) { /* ... */ }
14     public void disconnect() { /* ... */ }
15 }
16
```

Çözüm – 3:

Sorun:

InvoiceService Sınıfı SqlDatabase'e Sıkı Bağlı (Tightly Coupled)

Eğer NoSQL, Mock DB veya başka bir veri kaynağına geçmek istersek, kodun içinde değişiklik yapmamız gereklidir.

Kodun bağımsız ve genişletilebilir olması gereklidir, şu an sadece SqlDatabase ile çalışabilen durumda.

Test yazmak zorlaşıyor, çünkü SqlDatabase'i test etmek için gerçek bir veritabanına bağlanmak gereklidir.

SOLID Prensiplerine Aykırı: Açık Kapalı Prensibi (Open/Closed Principle - OCP) İhlali
Gelecekte başka bir veri kaynağı eklemek istediğimizde mevcut kodu değiştirmek yerine, genişletebilmeliyiz.

Çözüm:

Bir Interface tanımlayarak soyutlama yapacağım.

SQL ve NoSQL İçin ayrı Implementations yazacağım.

DatabaseService interface'sı:

```
1  public interface DatabaseService {
2      void connect();
3      void insertInvoice(Invoice invoice);
4      void disconnect();
5  }
```

Sql DB Service(DatabaseService interface'sinin bir implemantasyonu):

```
1 import org.springframework.stereotype.Service;
2
3 @Service
4 public class SqlDatabaseService implements DatabaseService {
5
6     @Override
7     public void connect() {
8         System.out.println("SQL Database connected.");
9     }
10
11    @Override
12    public void insertInvoice(Invoice invoice) {
13        System.out.println("Invoice inserted into SQL Database: " + invoice);
14    }
15
16    @Override
17    public void disconnect() {
18        System.out.println("SQL Database disconnected.");
19    }
20 }
```

No SQL DB Service(DatabaseService interface'sinin bir diğer implemantasyonu):

```
1 import org.springframework.stereotype.Service;
2
3 @Service
4 public class NoSqlDatabaseService implements DatabaseService {
5
6     @Override
7     public void connect() {
8         System.out.println("NoSQL Database connected.");
9     }
10
11    @Override
12    public void insertInvoice(Invoice invoice) {
13        System.out.println("Invoice inserted into NoSQL Database: " + invoice);
14    }
15
16    @Override
17    public void disconnect() {
18        System.out.println("NoSQL Database disconnected.");
19    }
20 }
21 |
```

Güncellenmiş InvoiceService sınıfı(artık DatabaseService adındaki interface'e bağlı ve sadece sql database'e bağımlı değil):

```
1 import org.springframework.beans.factory.annotation.Autowired;
2 import org.springframework.stereotype.Service;
3
4 @Service
5 public class InvoiceService {
6
7     private final DatabaseService databaseService;
8
9     @Autowired
10    public InvoiceService(DatabaseService databaseService) {
11        this.databaseService = databaseService;
12    }
13
14    public void createInvoice(Invoice invoice) {
15        databaseService.connect();
16        databaseService.insertInvoice(invoice);
17        databaseService.disconnect();
18    }
19 }
```

Hangi Database Kullanılacağını Belirlemek için de application.properties veya application.yml dosyasında hangi veri tabanı kullanacağımızı belirleyebiliriz ardından da interface'in implemantasyonlarından hangisini DB olarak kullanmak istiyorsak o sınıfa @Primary annotasyonunu ekleyeceğiz.

Yine benzer şekilde test yazabilmek için de yine DatabaseService interface'inden bir Mock DB servis implemantasyonu yazarak testlerde kullanabiliriz.

Sonuç olarak veritabanı bağımlılığı kaldırılarak farklı veri tabanlarını kolayca ekleyebileceğimiz esnek ve kolay test edilebilir bir hale getirdim.

Problemli Kod - 4 :

Finans uygulamalarında, para hesaplamalarında kesinlik (precision) kritik öneme sahiptir. Aşağıdaki kodda double veya float kullanarak yapılan işlemler, hatalı sonuçlara neden olabilir (ör. "0.9999999998" gibi). Nasıl bir çözüm üretirsiniz ?

Account Balance Calculation ▾

```
1 public double calculateAccountBalance(double initialBalance, double deposit, double feePercentage) {
2     double newBalance = initialBalance + deposit;
3     double fee = newBalance * feePercentage / 100;
4     return newBalance - fee;
5 }
6
```

Çözüm – 4:

double, kayan noktalı (floating-point) bir sayı formatıdır ve hassas hesaplamlarda hata üretebilir. Bankacılık ve finans sistemlerinde en küçük hata bile çok yüksek zarara neden olabilir. Finansal işlemler için her zaman BigDecimal kullanılması gereklidir.

```
1 import java.math.BigDecimal;
2 import java.math.RoundingMode;
3
4 public class AccountBalanceCalculator {
5
6     public BigDecimal calculateAccountBalance(BigDecimal initialBalance, BigDecimal deposit, BigDecimal feePercentage) {
7
8         BigDecimal newBalance = initialBalance.add(deposit);
9
10        //burada divide metodunun 2. parametresi olan 2 değeri istenirse 3,4,5 gibi değerler de girilerek daha hassas hesaplama da yapılabilir
11        //bu değer virgülden sonraki basamak sayısını gösterir.
12        BigDecimal fee = newBalance.multiply(feePercentage).divide(BigDecimal.valueOf(100), 2, RoundingMode.HALF_UP);
13
14     return newBalance.subtract(fee);
15 }
16
17 }
```

Problemli Kod – 5:

Birden fazla finansal ürün veya ülke pazarında faaliyet gösteren bir uygulamada, vergi (KDV, stopaj) ya da komisyon oranlarının doğrudan koda gömülmesi bakım ve güncelleme zorlukları doğurur. Nasıl bir çözüm üretirsiniz ?

Tax And Commission Calculation ▾

```
1  public BigDecimal calculateTax(BigDecimal amount) {
2      // Farklı ülkeler için sabit vergi oranları koda gömülü:
3  ✓      if (amount.compareTo(BigDecimal.valueOf(10000)) > 0) {
4          return amount.multiply(BigDecimal.valueOf(0.15)); // %15 vergi
5  ✓      } else {
6          return amount.multiply(BigDecimal.valueOf(0.10)); // %10 vergi
7      }
8  }
9
10 ✓ public BigDecimal calculateCommission(BigDecimal transactionAmount) {
11     // Sabit komisyon tutarı
12     return transactionAmount.multiply(BigDecimal.valueOf(0.02)); // %2 komisyon
13 }
14
```

Çözüm - 5 :

Bu oranları veritabanından veya konfigürasyon dosyasından çekerek yönetilebilir hale getirebiliriz. Ben şahsen bu oranları veri tabanında tutmak yerine konfigürasyon dosyalarında tutmayı tercih ederim. Konfigürasyon dosyalarında tutulan bu gibi sabit değerlerin değiştirilmesi durumda uygulama yeniden başlatılması gereklidir(veri tabanında tutarsak bu gerekmez) ancak bu gibi sabit değerler de sık sık değişen değerler olmadığı için, uygulamanın performanslı ve hızlı çalışması açısından gereksiz DB sorgu maliyeti eklemeyi doğru bulmuyorum. Yine de verileri DB'de tutup, gereksiz veri tabanı sorgu maliyetini düşürmek için Redis'te cache'lemek de bir seçenek olabilir. Ben konfigürasyon dosyasında tutma mantığı ile ilerledim bu soruda.

Örnek application.yml :

```
1  tax:
2    rates:
3      TR: 0.10
4      USA: 0.12
5      GERMANY: 0.15
6      HIGH_AMOUNT_THRESHOLD: 10000
7    commission:
8      rates:
9        CREDIT_CARD: 0.02
10       LOAN: 0.03
11       INVESTMENT: 0.015
```

TaxAndComissionConfigService:

```
1  import org.springframework.beans.factory.annotation.Value;
2  import org.springframework.stereotype.Service;
3  import java.math.BigDecimal;
4  import java.util.Map;
5
6  @Service
7  public class TaxAndCommissionConfigService {
8
9      @Value("#{${tax.rates}}")
10     private Map<String, BigDecimal> taxRates;
11
12     @Value("#{${commission.rates}}")
13     private Map<String, BigDecimal> commissionRates;
14
15     @Value("${tax.rates.HIGH_AMOUNT_THRESHOLD}")
16     private BigDecimal highAmountThreshold;
17
18     public BigDecimal getTaxRate(String country, BigDecimal amount) {
19         BigDecimal taxRate = taxRates.get(country);
20         if (taxRate == null) {
21             throw new RuntimeException("Tax rate not found for country: " + country);
22         }
23         return (amount.compareTo(highAmountThreshold) > 0) ? taxRate.add(BigDecimal.valueOf(0.05)) : taxRate;
24     }
25
26     public BigDecimal getCommissionRate(String productType) {
27         BigDecimal commissionRate = commissionRates.get(productType);
28         if (commissionRate == null) {
29             throw new RuntimeException("Commission rate not found for product type: " + productType);
30         }
31         return commissionRate;
32     }
33 }
```

TaxAndComissionCalculator:

```
1 import java.math.BigDecimal;
2 import org.springframework.stereotype.Service;
3
4 @Service
5 public class TaxAndCommissionCalculator {
6
7     private final TaxAndCommissionConfigService configService;
8
9     public TaxAndCommissionCalculator(TaxAndCommissionConfigService configService) {
10         this.configService = configService;
11     }
12
13     public BigDecimal calculateTax(BigDecimal amount, String country) {
14         BigDecimal taxRate = configService.getTaxRate(country, amount);
15         return amount.multiply(taxRate);
16     }
17
18     public BigDecimal calculateCommission(BigDecimal transactionAmount, String productType) {
19         BigDecimal commissionRate = configService.getCommissionRate(productType);
20         return transactionAmount.multiply(commissionRate);
21     }
22 }
```

Problemli Kod – 6:

Aşağıdaki kod parçasında, ödeme veya fatura sisteminde döviz çevirimi yapan bir sınıf doğrudan bir “ExchangeRateService”e sıkı sıkıya bağlı. Ayrıca desteklenen tüm döviz çiftleri de koda gömülü (if-else veya switch yapısı). Nasıl bir çözüm üretirsiniz ?

Payment Service ▾

```
1 ~ public class PaymentService {
2     private ExchangeRateService exchangeRateService = new ExchangeRateService();
3
4     public BigDecimal convertCurrency(BigDecimal amount, String from, String to) {
5         if (from.equals("USD") && to.equals("EUR")) {
6             BigDecimal rate = exchangeRateService.getRate("USD", "EUR");
7             return amount.multiply(rate);
8         } else if (from.equals("EUR") && to.equals("USD")) {
9             BigDecimal rate = exchangeRateService.getRate("EUR", "USD");
10            return amount.multiply(rate);
11        }
12        // ... daha fazla if-else döviz çifti
13        throw new UnsupportedOperationException("Conversion not supported");
14    }
15 }
16
17 ~ class ExchangeRateService {
18     public BigDecimal getRate(String from, String to) {
19         // API veya sabit tablo
20         return BigDecimal.valueOf(0.90);
21     }
22 }
```

Çözüm - 6 :

Sorun:

Dependency Injection (Bağımlılık Enjeksiyonu) yerine doğrudan nesne oluşturulmuş. Farklı bir döviz kuru sağlayıcısına geçmek istersek (örn: başka bir API, Redis, veritabanı) kodu değiştirmek gereklidir.

Yeni bir para birimi eklemek istediğimizde kodu değiştirmemiz gerekiyor.

Mock test yazmak zorlaşır.

Çözüm:

ExchangeRateService bir interface olarak tanımlanmalıdır.

Farklı kur sağlayıcıları (API, veritabanı, Redis) için ayrı implementasyonlar olmalıdır.

Spring'in @Autowired annotationu ile dependency injection kullanılmalıdır.

Döviz kurları(yeni para birimleri), veritabanında veya bir API servisinden dinamik olarak çekilmeli.

Öncelikle farklı kaynaklardan veri alabilmek için bir interface oluşturduğum:

```
1 import java.math.BigDecimal;
2
3 public interface ExchangeRateService {
4     BigDecimal getRate(String from, String to);
5 }
```

yukarıdaki interface'in bir imlemantasyonu olarak bir api'den döviz kuru alan servis örneği:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class ApiExchangeRateService implements ExchangeRateService {
6
7     @Override
8     public BigDecimal getRate(String from, String to) {
9         // Gerçek bir API çağrısı yapılabilir (ör. Open Exchange Rates API)
10        return BigDecimal.valueOf(1.10); // Simüle edilen API cevabı
11    }
12 }
```

Benzer şekilde farklı imlemantasyonlar da olabilir DB'den alan vs, ya da test yazmak için Mock imlemantasyonlar da yazılabılır. Ben örnek olarak yukarıda api'den verileri çeken servisi gösterdim sadece.

PaymentService son hali aşağıdaki gibi olur ve sıkı sıkıya bağımlı hali ve desteklenen döviz tiplerinin kodda yazılı olması sorunu giderildi:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class PaymentService {
6
7     private final ExchangeRateService exchangeRateService;
8
9     public PaymentService(ExchangeRateService exchangeRateService) {
10         this.exchangeRateService = exchangeRateService;
11     }
12
13     public BigDecimal convertCurrency(BigDecimal amount, String from, String to) {
14         if (from.equals(to)) {
15             return amount;
16         }
17
18         BigDecimal rate = exchangeRateService.getRate(from, to);
19         return amount.multiply(rate);
20     }
21 }
```

Problemlı Kod - 7 :

Aşağıdaki PaymentProcessing sınıfı, bir finans uygulamasında ödeme onayı, fraud kontrolü, API entegrasyonu ve muhasebe kaydına kadar tüm işleri üstleniyor. Kod büyük ölçüde sınıf içi metod sayısı ve sorumluluklar birbirine girecek. Nasıl bir çözüm üretirisiniz ?

Payment Processing

```
1 ~ public class PaymentProcessing {
2 ~     public void processCreditCardPayment(String cardNumber, String expiration, BigDecimal amount) {
3 ~         // Fraud check
4 ~         if (!fraudCheck(cardNumber, amount)) {
5 ~             throw new RuntimeException("Fraud suspected!");
6 ~         }
7 ~         // External payment gateway çağrıları
8 ~         boolean success = callPaymentGateway(cardNumber, expiration, amount);
9 ~         // Muhasebe kaydı
10 ~        if (success) {
11 ~            recordTransaction(cardNumber, amount);
12 ~        }
13 ~    }
14
15 ~    private boolean fraudCheck(String cardNumber, BigDecimal amount) {
16 ~        // Basit bir kontrol
17 ~        return amount.compareTo(BigDecimal.valueOf(5000)) < 0;
18 ~    }
19
20 ~    private boolean callPaymentGateway(String cardNumber, String expiration, BigDecimal amount) {
21 ~        // 3rd party API entegrasyonu
22 ~        return true;
23 ~    }
24
25 ~    private void recordTransaction(String cardNumber, BigDecimal amount) {
26 ~        // Muhasebe sistemine kayıt
27 ~        System.out.println("Transaction recorded for " + cardNumber + ":" + amount);
28 ~    }
29 ~}
```

Çözüm – 7:

Bu kodun düzenlenmesinde birden fazla yöntem olabilir, mesela PaymentProcessing sınıfı bir “orchestrator” gibi çalışıp diğer fraud, muhasebe, api entegrasyonu gibi işlemler ayrı birer servis olarak ele alınabilir. Ya da Chain of Responsibility Pattern ya da Decorator Pattern gibi desenler de çözüm olarak kullanılabilir. Tüm bu yöntemlerde temel amaç kodun karmaşıklığını azaltmak ve kolay yönetilebilir olmasını sağlamak.

Ben şahsen Chain of Responsibility Pattern kullanmayı daha mantıklı buluyorum bu senaryoda. Bu pattern’ı kullanarak, ödeme sürecini adım adım işleyen bir zincir gibi oluşturabiliriz ve bu da ödeme gibi kritik bir işlem için en mantıklı çözüm gibi duruyor bana göre.

Bir handler interface’ı oluşturdum:

```
1 import java.math.BigDecimal;
2
3 public interface PaymentHandler {
4     void handle(PaymentContext context);
5 }
```

Bu interface sayesinde her bir aşama için ayrı handler sınıfları oluşturacağım.

Fraud kontrolü handler’ı:

```
1 import org.springframework.stereotype.Component;
2 import java.math.BigDecimal;
3
4 @Component
5 public class FraudCheckHandler implements PaymentHandler {
6
7     @Override
8     public void handle(PaymentContext context) {
9         if (context.getAmount().compareTo(BigDecimal.valueOf(5000)) > 0) {
10             throw new RuntimeException("Fraud suspected!");
11         }
12         context.next();
13     }
14 }
```

Ödeme gateway handler'i:

```
1 import org.springframework.stereotype.Component;
2 import java.math.BigDecimal;
3
4 @Component
5 public class PaymentGatewayHandler implements PaymentHandler {
6
7     @Override
8     public void handle(PaymentContext context) {
9         boolean success = callPaymentGateway(context.getCardNumber(), context.getExpiration(), context.getAmount());
10        if (!success) {
11            throw new RuntimeException("Payment failed!");
12        }
13        context.next();
14    }
15
16    private boolean callPaymentGateway(String cardNumber, String expiration, BigDecimal amount) {
17        // Burada gerçek bir ödeme API'si çağrılabilir
18        return true; // Simüle edilen başarı durumu
19    }
20
21 }
```

Ödeme işlemi başarılı olursa işlem zincirin bir sonraki aşamasına devrediliyor.

Muhasebe handler'i:

```
1 import org.springframework.stereotype.Component;
2 import java.math.BigDecimal;
3
4 @Component
5 public class AccountingHandler implements PaymentHandler {
6
7     @Override
8     public void handle(PaymentContext context) {
9         System.out.println("Transaction recorded for " + context.getCardNumber() + ": " + context.getAmount());
10        context.next();
11    }
12 }
13
```

PaymentContext (işlemi hangi aşamada olduğunu takip eder ve sırayla handler'ları çalıştırır):

```
1 import java.math.BigDecimal;
2 import java.util.List;
3
4 public class PaymentContext {
5     private final String cardNumber;
6     private final String expiration;
7     private final BigDecimal amount;
8     private final List<PaymentHandler> handlers;
9     private int currentHandlerIndex = 0;
10
11     public PaymentContext(String cardNumber, String expiration, BigDecimal amount, List<PaymentHandler> handlers) {
12         this.cardNumber = cardNumber;
13         this.expiration = expiration;
14         this.amount = amount;
15         this.handlers = handlers;
16     }
17
18     public void next() {
19         if (currentHandlerIndex < handlers.size()) {
20             handlers.get(currentHandlerIndex++).handle(this);
21         }
22     }
23
24     public String getCardNumber() { return cardNumber; }
25     public String getExpiration() { return expiration; }
26     public BigDecimal getAmount() { return amount; }
27 }
```

PaymentProcessing sınıfı:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3 import java.util.List;
4
5 @Service
6 public class PaymentProcessing {
7
8     private final List<PaymentHandler> handlers;
9
10    public PaymentProcessing(List<PaymentHandler> handlers) {
11        this.handlers = handlers;
12    }
13
14    public void processCreditCardPayment(String cardNumber, String expiration, BigDecimal amount) {
15        PaymentContext context = new PaymentContext(cardNumber, expiration, amount, handlers);
16        context.next();
17    }
18 }
```

Bu yapı ile esnek modüler bir kod yapısı oluşturulur ve ihtiyaca göre koda yeni handler sınıfları kolaylıkla eklenebilir. Mesela ödeme yapılip muhasebesel işlemleri de bittikten sonra bir bildirim gönderilecek diyelim, bu bildirim SMS ya da mail şeklinde bu zincirin en son halkasına bir notification handler olarak kolaylıkla eklenip kullanılabilir.

***Not: Yukarıdaki handler'ların hangi sırayla çalıştırılacağına kontrolü için @Order annotasyonu kullanılmalı. Örneğin fraud kontrolü birinci sırada çalışacak, sonra payment ve en son da muhasebe süreçleri sırayla çalışacaksa FraudCheckHandler sınıfının başına @Order(1) annotasyonu koymalı, aynı şekilde sırayla diğerlerine de @Order(2) ve @Order(3) koymalı. Yukarıdaki ekran görüntülerini aldığım kodlarda koymayı unuttuğum için burada ayrıca açıklamak istedim.

Problemlı Kod – 8:

Karmaşık ve Yanlış Yönetilen Dağıtık İşlem (Distributed Transaction)

Bir ödeme işleminde aynı anda birkaç farklı finansal kaynağa (ör. birden fazla veritabanı, mikroservis, harici ödeme ağ geçidi) yazma işlemi yapılıyor. Kod aşağıdaki gibi “yarı dağıtık” bir mantık içeriyor. Aşağıdaki ipucunu değerlendirerek nasıl bir çözüm üretirsiniz ?

Payment Facade ▾

```
1 public class PaymentFacade {
2     private AccountRepository accountRepository = new AccountRepository(); // DB1
3     private TreasuryService treasuryService = new TreasuryService(); // Harici mikros.
4     private LoanRepository loanRepository = new LoanRepository(); // DB2
5
6     public void processPayment(String accountId, BigDecimal amount) {
7         // 1. Müşteri hesabından para düş
8         accountRepository.debitAccount(accountId, amount);
9
10        // 2. Ödeme hazinede takip edilsin
11        treasuryService.registerIncoming(amount);
12
13        // 3. Müşterinin ilgili kredisine (varsıa) aktarılsın
14        loanRepository.creditLoan(accountId, amount);
15
16        // Hata yönetimi yok, transaction boundaries net değil!
17    }
18 }
```

İpucu:

- Transaction Boundary Sorunu: Müşteri hesabından para düşütken sonra treasury veya loan servisinde hata oluşursa, sistemi geri almak için “rollback” mekanizması bulunmuyor.
- Çoklu Kaynak: İki ayrı veritabanı ve bir harici servis, tek bir “acid transaction” ile yönetilmiyor.
- Code Smell: Tek bir metot içinde, farklı veri kaynaklarına dokunan “spaghetti” çağrılar ve net olmayan hata/rollback yönetimi.

Çözüm – 8:

Böyle bir durumda SAGA Pattern en doğru çözüm olur. Roll back mekanizması bu sayede sağlanmış olur. Kodu SAGA Pattern'e uygun bir hale dönüştüreceğim.

Saga Orchestrator:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class PaymentSagaManager {
6
7     private final AccountService accountService;
8     private final TreasuryService treasuryService;
9     private final LoanService loanService;
10
11    public PaymentSagaManager(AccountService accountService, TreasuryService treasuryService, LoanService loanService) {
12        this.accountService = accountService;
13        this.treasuryService = treasuryService;
14        this.loanService = loanService;
15    }
16
17    public void processPayment(String accountId, BigDecimal amount) {
18        try {
19            accountService.debitAccount(accountId, amount);
20            treasuryService.registerIncoming(amount);
21            loanService.creditLoan(accountId, amount);
22        } catch (Exception e) {
23            rollback(accountId, amount);
24            throw new RuntimeException("Payment failed, rolling back transaction: " + e.getMessage());
25        }
26    }
27
28    private void rollback(String accountId, BigDecimal amount) {
29        // birbirinden ayrı DB yada microservis ler için roll back, en sondan en başa doğru
30        loanService.compensateCreditLoan(accountId, amount);
31        treasuryService.compensateRegisterIncoming(amount);
32        accountService.compensateDebitAccount(accountId, amount);
33    }
34}
35
```

Account Service:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class AccountService {
6
7     public void debitAccount(String accountId, BigDecimal amount) {
8         System.out.println("Debiting account: " + accountId + " by " + amount);
9         // DB1'e yazma işlemi burada yapılır
10    }
11
12    public void compensateDebitAccount(String accountId, BigDecimal amount) {
13        System.out.println("Rolling back debit for account: " + accountId);
14        // DB1'de yapılan işlem geri alınır
15    }
16 }
17
```

Treasury Service:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class TreasuryService {
6
7     public void registerIncoming(BigDecimal amount) {
8         System.out.println("Registering incoming payment: " + amount);
9         // Harici mikroservise çağrı yapar
10    }
11
12    public void compensateRegisterIncoming(BigDecimal amount) {
13        System.out.println("Rolling back treasury record for amount: " + amount);
14        // Hazine kaydını geri al
15    }
16 }
17
```

Loan Service:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class LoanService {
6
7     public void creditLoan(String accountId, BigDecimal amount) {
8         System.out.println("Crediting loan for account: " + accountId + " by " + amount);
9         // DB2'ye yazma işlemi
10    }
11
12    public void compensateCreditLoan(String accountId, BigDecimal amount) {
13        System.out.println("Rolling back credit loan for account: " + accountId);
14        // DB2'deki kredi işlemini geri al
15    }
16 }
17
```

Kodu bu şekilde düzenleyerek sorunu giderdim.

Problemlı Kod - 9 :

Yüksek Hacimli Gerçek Zamanlı İşlemlerde Yanlış Concurrency Yönetimi

Bir borsada, yüksek hacimli emir (order) eşleştirme / trade işlemleri gerçekleştiren kod, aynı entity üzerinde eş zamanlı güncellemeleri (concurrent updates) doğru yönetemiyor. Örneğin, TradeEngine sınıfı şu şekilde:

Aşağıdaki ipucunu değerlendirerek nasıl bir çözüm üretirsiniz ?

Trade Engine ▾

```
1 ~ public class TradeEngine {  
2     private OrderBook orderBook; // In-memory data structure  
3  
4 ~     public void matchOrders(Order incomingOrder) {  
5 ~         for (Order existing : orderBook.getOrders()) {  
6 ~             if (existing.isMatch(incomingOrder)) {  
7 ~                 executeTrade(existing, incomingOrder);  
8 ~                 existing.setStatus(OrderStatus.FILLED);  
9 ~                 incomingOrder.setStatus(OrderStatus.FILLED);  
10 ~                break;  
11 ~            }  
12 ~        }  
13 ~    }  
14 ~  
15 ~    private void executeTrade(Order existing, Order incoming) {  
16 ~        // Trade logic  
17 ~        BigDecimal price = existing.getPrice();  
18 ~        BigDecimal total = price.multiply(BigDecimal.valueOf(existing.getQuantity()));  
19 ~        // Müşteri hesap güncellemeleri, komisyon vb.  
20 ~    }  
21 ~}  
22 ~
```

İpucu:

- Concurrency Sorunu: orderBook in-memory ve eş zamanlı güncellemeleri korumak için yeterli kilit (lock) veya senkronizasyon stratejisi yok.
- Yüksek hacimli işlemlerde, aynı anda gelen çoklu emirler orderBook'u güncelliyor olabilir. Sonuç: Duplicated trade, missed match, hatalı hesap güncellemeleri vs.
- Code Smell: Tek bir sınıfta hem emir işleme (match), hem hesap güncelleme ve trade execution mantığı var. Ayrıca senkronizasyon kuralları belirsiz.

Çözüm – 9:

Bu sorunun çözümünde asenkron veri işleme ve queue mantığı kullanmak gereklidir çünkü büyük ölçekli ve yüksek trafik alan bir borsa kodundan bahsediyoruz bunu yönetmek için de genellikle best practice olarak kafka ve pessimistic locking kullanılıyor. Buna göre kodu yeniden şu şekilde düzenleyebiliriz:

Kafka Producer (Emirleri Kuyruğa Alma):

```
1 import org.springframework.kafka.core.KafkaTemplate;
2 import org.springframework.stereotype.Service;
3
4 @Service
5 public class TradeEventProducer {
6
7     private final KafkaTemplate<String, Order> kafkaTemplate;
8
9     public TradeEventProducer(KafkaTemplate<String, Order> kafkaTemplate) {
10         this.kafkaTemplate = kafkaTemplate;
11     }
12
13     public void publishOrder(Order order) {
14         // Aynı orderId'ye sahip mesajlar aynı partition'a giderek sıralı işlenecek
15         kafkaTemplate.send("trade-orders", order.getOrderId(), order);
16     }
17 }
18 }
```

Kafka Consumer:

```
1 import org.springframework.kafka.annotation.KafkaListener;
2 import org.springframework.stereotype.Service;
3 import org.springframework.transaction.annotation.Transactional;
4
5 @Service
6 public class TradeEventConsumer {
7
8     private final TradeEngine tradeEngine;
9     private final OrderRepository orderRepository;
10
11     public TradeEventConsumer(TradeEngine tradeEngine, OrderRepository orderRepository) {
12         this.tradeEngine = tradeEngine;
13         this.orderRepository = orderRepository;
14     }
15
16     @KafkaListener(topics = "trade-orders", groupId = "trade-group")
17     @Transactional
18     public void consumeTrade(Order order) {
19         try {
20             // Pessimistic Locking ile order kilitlenerek işleniyor
21             Order lockedOrder = orderRepository.findByIdWithLock(order.getId())
22                 .orElseThrow(() -> new RuntimeException("Order not found!"));
23
24             tradeEngine.matchOrders(lockedOrder);
25         } catch (Exception e) {
26             System.err.println("Trade processing failed: " + e.getMessage());
27         }
28     }
29 }
30 }
```

TradeEngine: order eşleşmelerini ve trade işlemlerini yönetiyor:

```
import org.springframework.stereotype.Service;
import java.math.BigDecimal;

@Service
public class TradeEngine {

    private final OrderRepository orderRepository;

    public TradeEngine(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }

    public void matchOrders(Order incomingOrder) {
        for (Order existing : orderRepository.findOpenOrders()) {
            if (existing.isMatch(incomingOrder)) {
                executeTrade(existing, incomingOrder);
                existing.setStatus(OrderStatus.FILLED);
                incomingOrder.setStatus(OrderStatus.FILLED);
                orderRepository.save(existing);
                orderRepository.save(incomingOrder);
                break;
            }
        }
    }

    private void executeTrade(Order existing, Order incoming) {
        BigDecimal price = existing.getPrice();
        BigDecimal total = price.multiply(BigDecimal.valueOf(existing.getQuantity()));

        // Trade işlemleri burada kaydedilebilir
        System.out.println("Trade executed: " + existing.getId() + " with " + incoming.getId());
    }
}
```

Order Repository:

```
1 import org.springframework.data.jpa.repository.JpaRepository;
2 import org.springframework.data.jpa.repository.Lock;
3 import jakarta.persistence.LockModeType;
4 import java.util.List;
5 import java.util.Optional;
6
7 public interface OrderRepository extends JpaRepository<Order, Long> {
8
9     @Lock(LockModeType.PESSIMISTIC_WRITE)
10    Optional<Order> findByIdWithLock(Long id);
11
12    List<Order> findOpenOrders();
13 }
14
```

order entity:

```
1 import jakarta.persistence.*;
2 import java.math.BigDecimal;
3
4 @Entity
5 @Table(name = "orders")
6 public class Order {
7
8     @Id
9     @GeneratedValue(strategy = GenerationType.IDENTITY)
10    private Long id;
11
12    private String orderId;
13    private BigDecimal price;
14    private int quantity;
15
16    @Enumerated(EnumType.STRING)
17    private OrderStatus status;
18
19    public boolean isMatch(Order incomingOrder) {
20        return this.price.equals(incomingOrder.getPrice()) && this.status == OrderStatus.OPEN;
21    }
22
23    // Getter ve Setter metotları...
24 }
```

Kafka Kullanımı: Order işlemlerinin asenkron ve sıralı çalışmasını sağlıyor.

PESSIMISTIC_WRITE Locking: Aynı order'ın eşzamanlı olarak birden fazla işlemde güncellenmesini engelliyor.

Transactional Consumer: İşlemler başarısız olursa Kafka mesajı tekrar kuyruğa koyuyor.

Idempotency & Ordering: Duplicate işlem riskini ve sırasız işlem hatalarını ortadan kaldırıyor.

Trade sistemleri gibi yüksek hacimli ve eşzamanlı işlem yapılan sistemlerde, bir order üzerinde aynı anda birçok işlem gerçekleştirilebilir.

Optimistic Locking (@Version), çakışmayı işlem tamamlandıktan sonra fark eder ve çakışma olduğunda tekrar denemek gerekir (retry).

Ancak borsa gibi sistemlerde, her işlem milisaniyeler içinde gerçekleştiği için, çakışmalar çok sık olur ve sürekli retry yapmak büyük bir performans kaybına neden olur.

Pessimistic Locking (PESSIMISTIC_WRITE) kullanarak, aynı order üzerinde çalışan işlemlerin sırasını garanti altına alıyoruz ve trade işlemlerinin çakışmadan, güvenli bir şekilde yapılmasını sağlıyoruz.

Problemlı Kod - 10 :

Muhasebe Kurallarıyla Karışık, Çok Katmanlı Validation ve Denetim (Audit)

Aşağıdaki “AccountingService” sınıfı, bir finans kurumunda farklı raporlama ve muhasebe standartları (IFRS, US GAAP vb.) gereği birçok validation, business rule ve audit kaydını iç içe yapıyor:

Aşağıdaki ipucunu değerlendirerek nasıl bir çözüm üretirsiniz ?

```
Accounting Service ▾

1 v  public class AccountingService {
2 v      public void postGeneralLedgerEntry(GeneralLedgerEntry entry) {
3         // 1. Validation
4         if (entry.getDebit().compareTo(BigDecimal.ZERO) < 0
5             || entry.getCredit().compareTo(BigDecimal.ZERO) < 0) {
6             throw new IllegalArgumentException("Amounts cannot be negative");
7         }
8 v         if (!entry.getAccountType().matches("ASSET|LIABILITY|EQUITY")) {
9             throw new IllegalArgumentException("Invalid account type");
10        }
11
12        // 2. IFRS kuralı (basit örnek)
13        if (entry.isUnderIFRS()) {
14            if (entry.getAccountType().equals("EQUITY") && entry.getDebit().compareTo(BigDecimal.valueOf(100000)) > 0) {
15                // Örnek bir kural
16                throw new IllegalStateException("Equity hesapları IFRS limit aşımı!");
17            }
18        }
19
20        // 3. Logging & Audit
21        System.out.println("Posting to ledger: " + entry);
22
23        // 4. Kalemleri DB'ye kaydet
24        // ...
25    }
26
27 }
```

İpucu:

- Validation + İş Mantığı + Denetim tek bir metotta toplanmış.
- Farklı regülasyonlar, raporlama standartları (IFRS, US GAAP, local GAAP vb.) ile eklendikçe if-else yiğini artacak.
- Muhasebe veya raporlama yazılımında denetim (audit) kayıtları çok önemlidir, ancak burada gelişigüzel “println” ile yapılmış gibi duruyor.

Çözüm – 10:

Validation, Business Rules ve Audit işlemlerini farklı servisler haline getireceğim. Yeni finansal regülasyonlar geldiğinde, modüler olarak eklenebilecek bir yapı olacak. Audit loglarını bir veritabanına kaydedeceğim.

AccountingService:

```
1 import org.springframework.stereotype.Service;
2 import org.springframework.transaction.annotation.Transactional;
3
4 @Service
5 public class AccountingService {
6
7     private final AccountingValidationService validationService;
8     private final AccountingRuleService ruleService;
9     private final AuditService auditService;
10
11     public AccountingService(AccountingValidationService validationService,
12                             AccountingRuleService ruleService,
13                             AuditService auditService) {
14         this.validationService = validationService;
15         this.ruleService = ruleService;
16         this.auditService = auditService;
17     }
18
19     @Transactional
20     public void postGeneralLedgerEntry(GeneralLedgerEntry entry) {
21         validationService.validateEntry(entry); // Validation işlemi
22         ruleService.applyIFRSRules(entry); // İş mantığı kontrolü
23         auditService.logTransaction(entry); // Audit kaydı
24     }
25 }
26 }
```

Validation Servisi:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class AccountingValidationService {
6
7     public void validateEntry(GeneralLedgerEntry entry) {
8         if (entry.getDebit().compareTo(BigDecimal.ZERO) < 0 || entry.getCredit().compareTo(BigDecimal.ZERO) < 0) {
9             throw new IllegalArgumentException("Amounts cannot be negative");
10        }
11
12        if (!entry.getAccountType().matches("ASSET|LIABILITY|EQUITY")) {
13            throw new IllegalArgumentException("Invalid account type");
14        }
15    }
16 }
17 }
```

Business Rules Servisi:

```
1 import org.springframework.stereotype.Service;
2 import java.math.BigDecimal;
3
4 @Service
5 public class AccountingRuleService {
6
7     public void applyIFRSRules(GeneralLedgerEntry entry) {
8         if (entry.isUnderIFRS()) {
9             if (entry.getAccountType().equals("EQUITY") && entry.getDebit().compareTo(BigDecimal.valueOf(10000)) > 0) {
10                 throw new IllegalStateException("Equity hesapları IFRS limit aşımı!");
11             }
12         }
13     }
14 }
15
```

Audit Servisi:

```
1 import org.springframework.stereotype.Service;
2
3 @Service
4 public class AuditService {
5
6     public void logTransaction(GeneralLedgerEntry entry) {
7         // örnek bir metod, normalde bu kayıtlar bir DB'ye kaydedilir.
8         System.out.println("Audit Log: Posting to ledger: " + entry);
9     }
10 }
11
```

Validation, Business Rules ve Audit işlemlerini ayrı katmanlara ayırarak modüler hale getirildi.

Yeni bir finansal regülasyon geldiğinde sadece ilgili servise ekleme yaparak kodu değiştirebiliriz.

Ayrıca Audit kayıtlarını **Kafka gibi bir mesaj kuyruğuna** gönderebiliriz.

Böylece finansal işlemlerden ayrı olarak bir 'audit pipeline' çalıştırarak sistem yükünü azaltabiliriz.