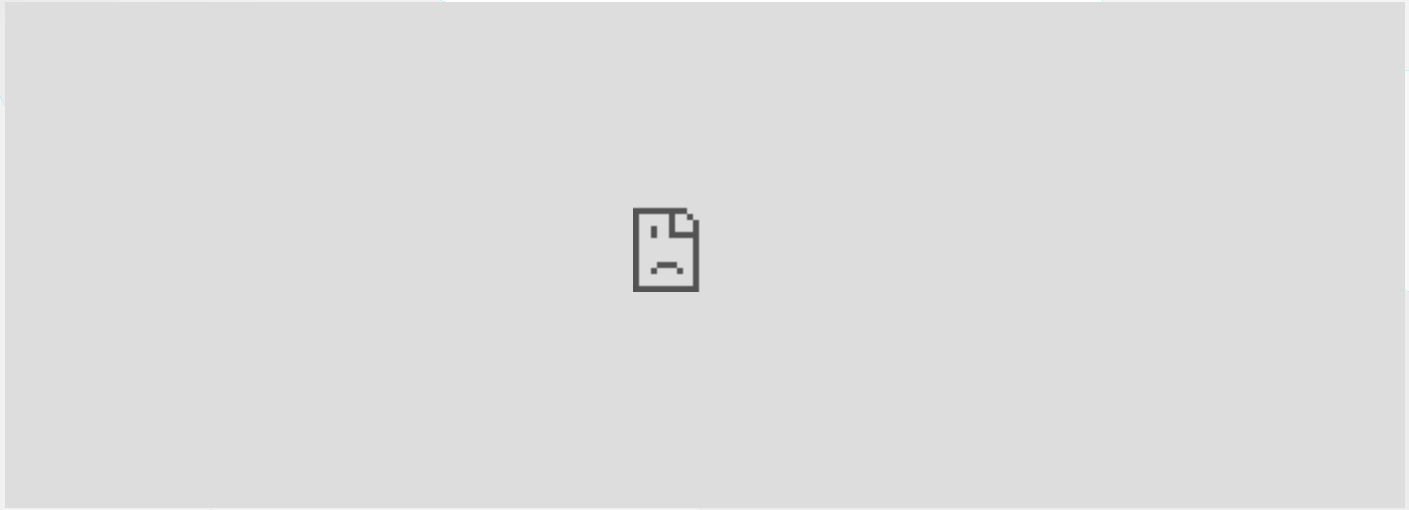
CEN206 Nesne Yönelimli Programlama

Hafta-13 (Yeniden Yapılandırma Teknikleri)

Bahar Dönemi, 2025-2026

İndir DOC-PDF, DOC-DOCX, SLAYT



Hafta-13 Genel Bakış

Yeniden Yapılandırma Teknikleri (Tüm 66 Teknik)

Modül	Konu	Teknikler
A	Metot Oluşturma	9 teknik
B	Nesneler Arası Özellikleri Taşıma	8 teknik
C	Verileri Düzenleme	15 teknik
D	Koşulları Sadeleştirme	8 teknik
E	Metot Çağrılarını Sadeleştirme	14 teknik

Kaynak: https://refactoring.guru/refactoring/techniques

Refactoring (Yeniden Yapılandırma) Nedir?

- **Refactoring** (Yeniden Yapılandırma), mevcut bir kod tabanının iç yapısını, dış davranışını değiştirmeden yeniden düzenlemeye yönelik disiplinli bir tekniktir.
- Her dönüşüm ("refactoring" olarak adlandırılır) küçüktür, ancak bir dizi dönüşüm önemli bir yeniden yapılandırma üretebilir.
- Amaç, kodu **anlaşılması daha kolay**, **değiştirilmesi daha ucuz** ve **genişletilmesi daha güvenli** hale getirmektir.

Neden Refactoring Yapılır?

- **Yazılım tasarımını iyileştirir** -- Refactoring yapılmadığında, bir programın tasarımı zamanla bozulur.
- **Yazılımın anlaşılmasını kolaylaştırır** -- Amacını açıkça ileten kod, bakımı daha kolaydır.
- **Hataları bulmaya yardımcı olur** -- Kod yapısını netleştirmek, hataları daha görünür kılar.
- **Daha hızlı programlamaya yardımcı olur** -- İyi tasarım, yeni özelliklerin hızla geliştirilmesini sağlar.

Ne Zaman Refactoring Yapılır?

- **Üçler Kuralı**: Bir şeyi ilk kez yaparsanız, sadece yaparsınız. İkinci kez, tekrardan rahatsız olursunuz ama yine de yaparsınız. Üçüncü kez, refactoring yaparsınız.
- **Özellik eklerken**: Yeni özelliğin eklenmesini kolaylaştırmak için refactoring yapın.
- **Hata düzeltirken**: Hatayı belirgin hale getirmek için refactoring yapın.
- **Kod inceleme sırasında**: Okunabilirliği ve bakım kolaylığını artırmak için refactoring yapın.


Refactoring Techniques
A: Composing Methods (9)
Extract Method
Inline Method
Extract Variable
Replace Temp with Query
Replace Method with Method Object
Substitute Algorithm
B: Moving Features Between Objects (8)
Move Method
Move Field
Extract Class
Inline Class
Hide Delegate
Introduce Local Extension
C: Organizing Data (15)
Encapsulate Field
Encapsulate Collection
Replace Type Code with Subclasses
Replace Type Code with State/Strategy
Replace Magic Number with Constant
D: Simplifying Conditionals (8)
Decompose Conditional
Consolidate Conditional
Replace Conditional with Polymorphism
Introduce Null Object
Guard Clauses
Introduce Assertion
E: Simplifying Method Calls (14)
Rename Method
Introduce Parameter Object
Preserve Whole Object
Separate Query from Modifier
Replace Constructor with Factory Method
Replace Error Code with Exception
F: Dealing with Generalization (12)
Pull Up Method
Pull Up Field
Push Down Method
Extract Superclass
Replace Inheritance with Delegation

Modül A: Composing Methods (Metot Oluşturma)

Daha İyi Metot Yapısı için 9 Teknik

Composing Methods (Metot Oluşturma)

Bu yeniden yapılandırmalar, metotların nasıl oluşturulduğuyla ilgilenir. Yeniden yapılandırmanın büyük bölümü, kodu düzgün paketlemek için metotları oluşturmakla ilgilidir. Neredeyse her zaman temel sorun **çok uzun metotlardır**.

1. Extract Method (Metot Çıkarma)
2. Inline Method (Satır İçi Metot)
3. Extract Variable (Değişken Çıkarma)
4. Inline Temp (Geçici Değişkeni Satır İçine Alma)
5. Replace Temp with Query (Geçici Değişkeni Sorgu ile Değiştirme)
6. Split Temporary Variable (Geçici Değişkeni Bölme)
7. Remove Assignments to Parameters (Parametrelere Atamayı Kaldırma)
8. Replace Method with Method Object (Metodu Metot Nesnesi ile Değiştirme)
9. Substitute Algorithm (Algoritmayı Değiştirme)

A1. Extract Method (Metot Çıkarma)

Nedir: Bir kod parçasını açıklayıcı bir isimle ayrı bir metoda taşıyın.

Problem: Birlikte gruplanabilecek bir kod parçanız var. Uzun metotları anlamak ve bakımını yapmak zordur.

Çözüm: Parçayı, metodun amacını açıklayan bir isimle bir metoda dönüştürün.

Ne zaman kullanılır: Metotlar 10-15 satırı aştığında; kod bölümleri belirgin bir görev gerçekleştirdiğinde; mantık tekrarlandığında.

Ne zaman KULLANILMAZ: Çıkarılan kod gerçekten tek seferlik bir işlem olduğunda; yeni metot aşırı parametre gerektireceğinde.

A1. Extract Method (Metot Çıkarma) -- Önce

```
public class Order {
    private String name;
    private List<LineItem> items;

    public void printOwing() {
        double outstanding = 0.0;

        // print banner
        System.out.println("**************************");
        System.out.println("***** Customer Owes ******");
        System.out.println("**************************");

        // calculate outstanding
        for (LineItem item : items) {
            outstanding += item.getAmount();
        }

        // print details
        System.out.println("name: " + name);
        System.out.println("amount: " + outstanding);
    }
}
```

A1. Extract Method (Metot Çıkarma) -- Sonra

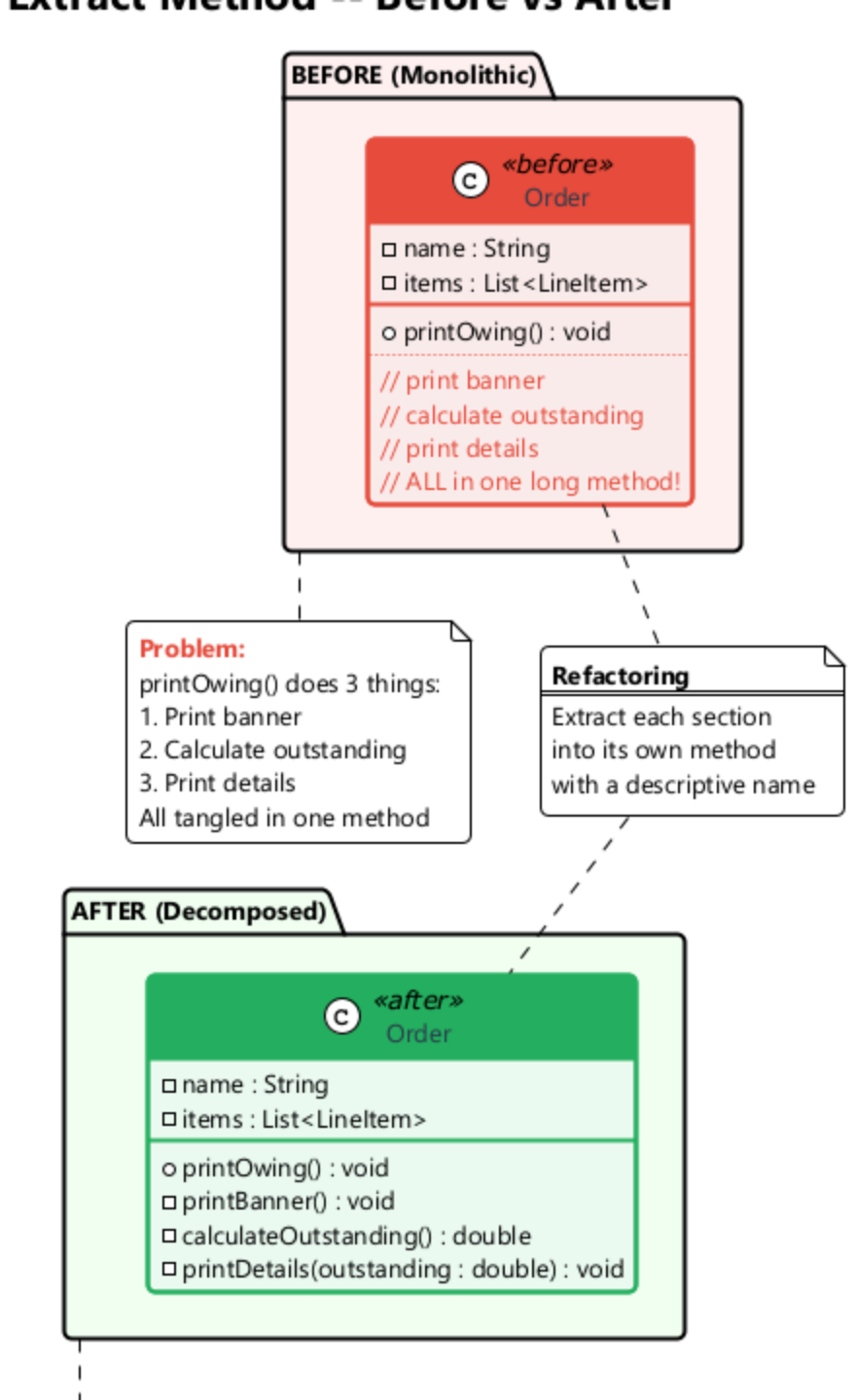
```
public class Order {
    private String name;
    private List<LineItem> items;

    public void printOwing() {
        printBanner();
        double outstanding = calculateOutstanding();
        printDetails(outstanding);
    }

    private void printBanner() {
        System.out.println("**************************");
        System.out.println("***** Customer Owes ******");
        System.out.println("**************************");
    }

    private double calculateOutstanding() {
        double outstanding = 0.0;
        for (LineItem item : items) {
            outstanding += item.getAmount();
        }
        return outstanding;
    }

    private void printDetails(double outstanding) {
        System.out.println("name: " + name);
        System.out.println("amount: " + outstanding);
    }
}
```


BEFORE (Monolithic)
«before»
Order
name : String
items : List<LineItem>
printOwing() : void
// print banner
// calculate outstanding
// print details
// ALL in one long method!
Problem:
printOwing() does 3 things:
1. Print banner
2. Calculate outstanding
3. Print details
All tangled in one method
Refactoring
Extract each section
into its own method
with a descriptive name
AFTER (Decomposed)
«after»
Order
name : String
items : List<LineItem>
printOwing() : void
printBanner() : void
calculateOutstanding() : double
printDetails(outstanding : double) : void

A2. Inline Method (Satır İçi Metot)

Nedir: Bir metot çağrısını metodun gövdesiyle değiştirin ve metodu silin.

Problem: Bir metodun gövdesi, metot adı kadar açık olduğunda veya metot gövdesi önemsiz derecede basit olduğunda, dolaylılık herhangi bir değer katmaz.

Çözüm: Metodun gövdesini çağıranların gövdesine yerleştirin ve metodu kaldırın.

Ne zaman kullanılır: Metot gövdeleri metot adlarından daha net olduğunda; basit delege metotları amacına hizmet etmediğinde.

Ne zaman KULLANILMAZ: Metot alt sınıflarda geçersiz kılındığında (satır içine almak polimorfizmi bozar); metodun basitliğin önemli olduğu birçok yerden çağrıldığında.

A2. Inline Method (Satır İçi Metot) -- Önce / Sonra

```
// Before
class PizzaDelivery {
    int getRating() {
        return moreThanFiveLateDeliveries() ? 2 : 1;
    }
    boolean moreThanFiveLateDeliveries() {
        return numberOfLateDeliveries > 5;
    }
}
```

```
// After
class PizzaDelivery {
    int getRating() {
        return numberOfLateDeliveries > 5 ? 2 : 1;
    }
}
```

Gereksiz metot sayısını en aza indirerek kodu daha anlaşılır hale getirirsiniz.

A3. Extract Variable (Değişken Çıkarma)

Nedir: Bir ifadenin sonucunu (veya parçalarını) kendi kendini açıklayan ayrı değişkenlere yerleştirin.

Problem: Anlaşılması zor bir ifadeniz var, özellikle koşullu ifadelerde veya çok parçalı aritmetik işlemlerde.

Çözüm: İfade sonuçlarını kendi kendini açıklayan değişkenlere atayın.

Ne zaman kullanılır: Karmaşık koşullar netleştirme gerektirdiğinde; uzun aritmetik ifadeler ara sonuçlardan yoksun olduğunda.

Ne zaman KULLANILMAZ: Basit, apaçık ifadeler; çıkarılan değişken yalnızca bir kez kullanıldığında.

A3. Extract Variable (Değişken Çıkarma) -- Önce / Sonra

```java
// Before
public class Product {
    private int quantity;
    private double itemPrice;

    public double getPrice() {
        return quantity * itemPrice
            - Math.max(0, quantity - 500) * itemPrice * 0.05
            + Math.min(quantity * itemPrice * 0.1, 100.0);
    }
}
```

```java
// After
public class Product {
    private int quantity;
    private double itemPrice;

    public double getPrice() {
        double basePrice = quantity * itemPrice;
        double quantityDiscount = Math.max(0, quantity - 500) * itemPrice * 0.05;
        double shipping = Math.min(basePrice * 0.1, 100.0);
        return basePrice - quantityDiscount + shipping;
    }
}
```

A4. Inline Temp (Geçici Değişkeni Satır İçine Alma)

Nedir: Geçici bir değişkene yapılan tüm referansları ifadenin kendisiyle değiştirin.

Problem: Basit bir ifadeyle bir kez atanan bir geçici değişkeniniz var ve bu geçici değişken diğer yeniden yapılandırmaları engelliyor.

Çözüm: Bu geçici değişkene yapılan tüm referansları ifadeyle değiştirin.

Ne zaman kullanılır: Geçici değişken başka bir yeniden yapılandırmayı engellediğinde (örn. Replace Temp with Query); değişken herhangi bir netlik katmadığında.

Ne zaman KULLANILMAZ: İfade yan etkilerle birden fazla kez değerlendirildiğinde; değişkeni kaldırmak okunabilirliği azalttığında.

```
// Before
double basePrice = order.basePrice();
return (basePrice > 1000);

// After
return (order.basePrice() > 1000);
```

A5. Replace Temp with Query (Geçici Değişkeni Sorgu ile Değiştirme)

Nedir: İfadeyi bir metoda çıkarın ve geçici değişkene yapılan tüm referansları metot çağrısıyla değiştirin.

Problem: Bir ifadenin sonucunu tutmak için geçici bir değişken kullanıyorsunuz.

Çözüm: İfadeyi ayrı bir metoda taşıyın. Geçici değişkene yapılan tüm referansları metot çağrısıyla değiştirin. Yeni metot daha sonra diğer metotlarda da kullanılabilir.

Ne zaman kullanılır: Bir geçici değişken tek bir metotta kullanılıyorsa ancak ifade diğer metotlarda da yararlıysa.

Ne zaman KULLANILMAZ: İfadenin yan etkileri olduğunda veya tekrarlanmaması gereken performans açısından kritik hesaplamalar olduğunda.

A5: Replace Temp with Query – Önce / Sonra

```java
// Before
public class Product {
    private int quantity;
    private double itemPrice;

    double getPrice() {
        double basePrice = quantity * itemPrice;
        if (basePrice > 1000) {
            return basePrice * 0.95;
        }
        return basePrice * 0.98;
    }
}
```

```java
// After
public class Product {
    private int quantity;
    private double itemPrice;

    double getPrice() {
        if (basePrice() > 1000) {
            return basePrice() * 0.95;
        }
        return basePrice() * 0.98;
    }

    private double basePrice() {
        return quantity * itemPrice;
    }
}
```

A6. Split Temporary Variable (Geçici Değişkeni Bölme)

Nedir: Her atama için ayrı bir geçici değişken kullanın.

Problem: Döngü değişkeni veya toplama değişkeni olmayan bir yerel değişkeniniz var ve birden fazla kez atanıyor.

Çözüm: Her atama için ayrı bir geçici değişken oluşturun. Her geçici değişken yalnızca bir kez atanmalıdır.

Ne zaman kullanılır: Bir geçici değişken iki farklı şey için kullanıldığında, okuyucuyu kafa karıştırır. Her amaç için ayrı bir değişken kullanın.

Ne zaman KULLANILMAZ: Doğal olarak yeniden atanan döngü sayaçları veya toplayıcı değişkenler.

A6. Split Temporary Variable -- Önce / Sonra

```
// Before
public class Rectangle {
    double height;
    double width;

    void calculate() {
        double temp = 2 * (height + width);
        System.out.println("Perimeter: " + temp);
        temp = height * width;
        System.out.println("Area: " + temp);
    }
}
```

```
// After
public class Rectangle {
    double height;
    double width;

    void calculate() {
        double perimeter = 2 * (height + width);
        System.out.println("Perimeter: " + perimeter);
        double area = height * width;
        System.out.println("Area: " + area);
    }
}
```

A7. Remove Assignments to Parameters (Parametrelere Atamayı Kaldırma)

Nedir: Bir parametreye atama yapmak yerine yerel bir değişken kullanın.

Problem: Metodun gövdesinde bir parametreye değer atanıyor.

Çözüm: Parametre yerine yerel bir değişken kullanın.

Ne zaman kullanılır: Bir metot parametre değerini değiştirdiğinde, giriş değerleri ile yerel çalışma değişkenleri arasındaki sınırı bulanıklaştırır.

Ne zaman KULLANILMAZ: Parametreyi değiştirmenin amaçlanan davranış olduğu referansa göre geçiş yapan dillerde.

```
// Before
int discount(int inputVal, int quantity) {
    if (quantity > 50) inputVal -= 2;
    return inputVal;
}

// After
int discount(int inputVal, int quantity) {
    int result = inputVal;
    if (quantity > 50) result -= 2;
    return result;
}
```

A8. Replace Method with Method Object (Metodu Metot Nesnesi ile Değiştirme)

Nedir: Uzun bir metodu ayrı bir sınıfa dönüştürün, böylece yerel değişkenler alan haline gelir.

Problem: Yerel değişkenlerin birbirine öylesine bağlı olduğu uzun bir metodunuz var ki Extract Method uygulayamıyorsunuz.

Çözüm: Metodu ayrı bir sınıfa dönüştürün, böylece yerel değişkenler sınıfın alanları olur. Ardından metodu aynı sınıf içinde birkaç metoda bölebilirsiniz.

Ne zaman kullanılır: Extract Method karmaşık yerel değişken bağımlılıkları nedeniyle engellendiğinde.

Ne zaman KULLANILMAZ: Metot önce diğer, daha basit yeniden yapılandırmalarla sadeleştirilebildiğinde.

A8. Replace Method with Method Object -- Önce

```
public class Order {
    public double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation using all three variables...
        primaryBasePrice = quantity * itemPrice;
        secondaryBasePrice = primaryBasePrice * 0.9;
        tertiaryBasePrice = primaryBasePrice * 0.8;
        return primaryBasePrice + secondaryBasePrice + tertiaryBasePrice;
    }
}
```

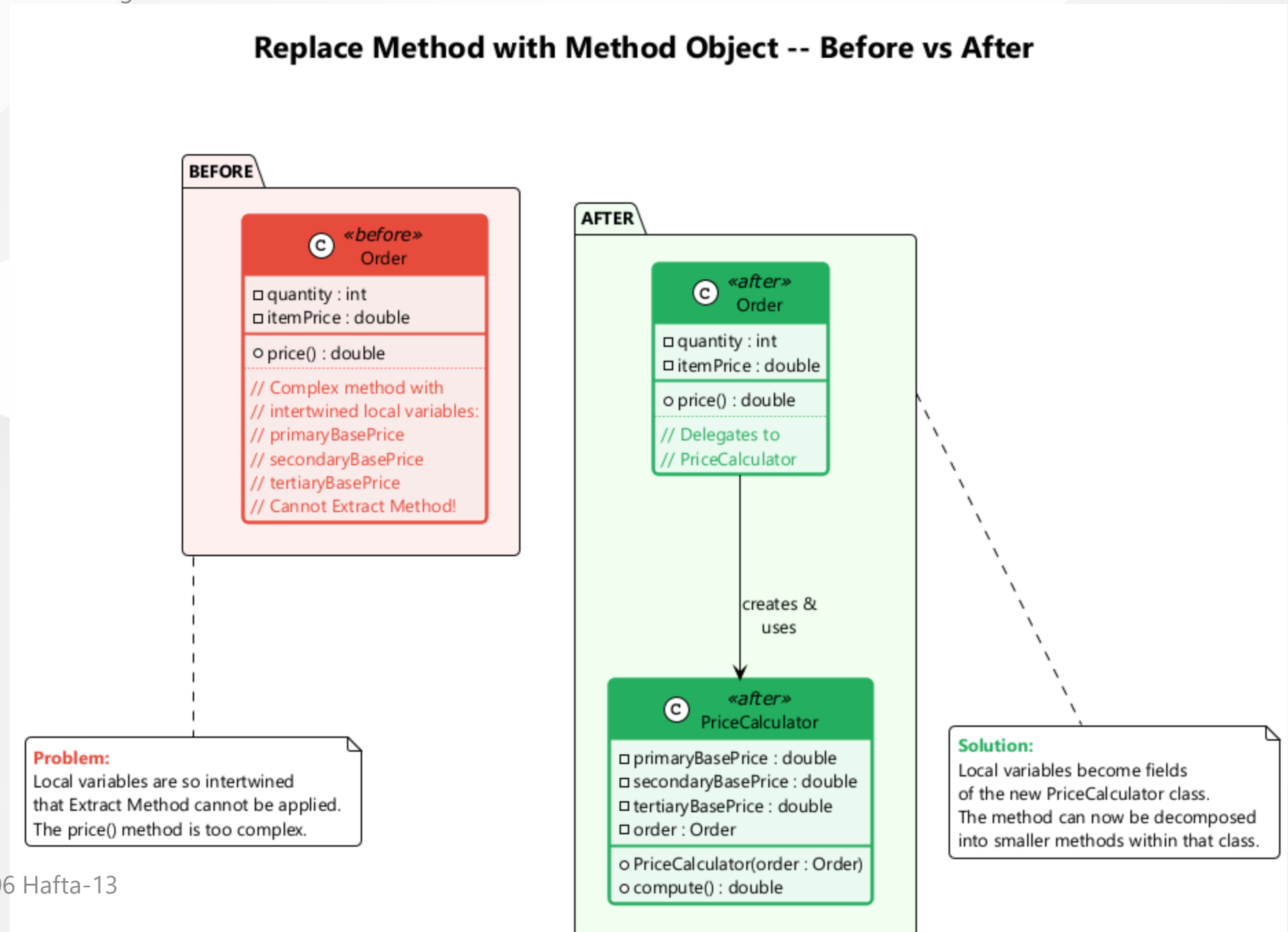
A8. Replace Method with Method Object -- Sonra

```
public class PriceCalculator {
    private double primaryBasePrice;
    private double secondaryBasePrice;
    private double tertiaryBasePrice;
    private Order order;

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

    public double compute() {
        primaryBasePrice = order.getQuantity() * order.getItemPrice();
        secondaryBasePrice = primaryBasePrice * 0.9;
        tertiaryBasePrice = primaryBasePrice * 0.8;
        return primaryBasePrice + secondaryBasePrice + tertiaryBasePrice;
    }
}
```

Metodu Metot Nesnesi ile Değiştirme - Diyagram



A9. Substitute Algorithm (Algoritmayı Değiştirme)

Nedir: Bir metodun gövdesini yeni, daha açık bir algoritma ile değiştirin.

Problem: Mevcut bir algoritmayı daha açık veya daha verimli bir algoritmayla değiştirmek istiyorsunuz.

Çözüm: Algoritmayı uygulayan metodun gövdesini yeni bir algoritma ile değiştirin.

Ne zaman kullanılır: Bir metodun yaptığı şeyi başarmak için daha basit veya daha verimli bir yol bulduğunuzda; mevcut algoritma aşırı karmaşık olduğunda.

Ne zaman KULLANILMAZ: Mevcut algoritma zaten optimal ve iyi anlaşılır olduğunda.

A9. Substitute Algorithm -- Önce / Sonra

```
// Before
String foundPerson(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) { return "Don"; }
        if (people[i].equals("John")) { return "John"; }
        if (people[i].equals("Kent")) { return "Kent"; }
    }
    return "";
}
```

```
// After
String foundPerson(String[] people) {
    List<String> candidates = List.of("Don", "John", "Kent");
    for (String person : people) {
        if (candidates.contains(person)) {
            return person;
        }
    }
    return "";
}
```

Modül A -- Özet

Composing Methods (Metot Oluşturma): Temel Noktalar

#	Teknik	Amaç
1	Extract Method (Metot Çıkarma)	Uzun metotları daha küçük, isimlendirilmiş parçalara bölme
2	Inline Method (Satır İçi Metot)	Gereksiz dolaylılığı kaldırma
3	Extract Variable (Değişken Çıkarma)	Karmaşık ifadeleri okunabilir hale getirme
4	Inline Temp	Engelleyen geçici değişkenleri kaldırma

Modül B: Moving Features Between Objects (Özellikleri Nesneler Arasında Taşıma)

Doğru Sorumluluk Dağılımı için 8 Teknik

Modül B Ana Hatları

Moving Features Between Objects (Özellikleri Nesneler Arasında Taşıma)

Bu yeniden yapılandırmalar, işlevlerin sınıflar arasında nasıl dağıtılacağını ele alır. Temel kararlar **sorumlulukların nereye konulacağıdır**. Bir sınıfın çok fazla sorumluluğu varsa Extract Class kullanın. Bir sınıf yetersiz kalıyorsa Inline Class kullanın.

1. Move Method (Metodu Taşıma)
2. Move Field (Alanı Taşıma)
3. Extract Class (Sınıf Çıkarma)
4. Inline Class (Satır İçi Sınıf)
5. Hide Delegate (Temsilciyi Gizleme)
6. Remove Middle Man (Aracıyı Kaldırma)
7. Introduce Foreign Method (Yabancı Metot Ekleme)
8. Introduce Local Extension (Yerel Uzantı Ekleme)

B1. Move Method (Metodu Taşıma)

Nedir: Metodu en çok kullanan sınıfta oluşturun ve orijinali yönlendirin veya kaldırın.

Problem: Bir metot, kendi sınıfından çok başka bir sınıfta kullanılıyor.

Çözüm: Metodu en çok kullanan sınıfta yeni bir metot oluşturun, ardından eski metodun gövdesini oraya taşıyın. Eski metodu basit bir delege haline getirin veya tamamen kaldırın.

Ne zaman kullanılır: Bir metot, kendi nesnesinin verilerinden çok başka bir nesnenin verilerine eriştiğinde.

Ne zaman KULLANILMAZ: Metot, mevcut sınıfının birkaç özelliğini eşit olarak kullandığında.

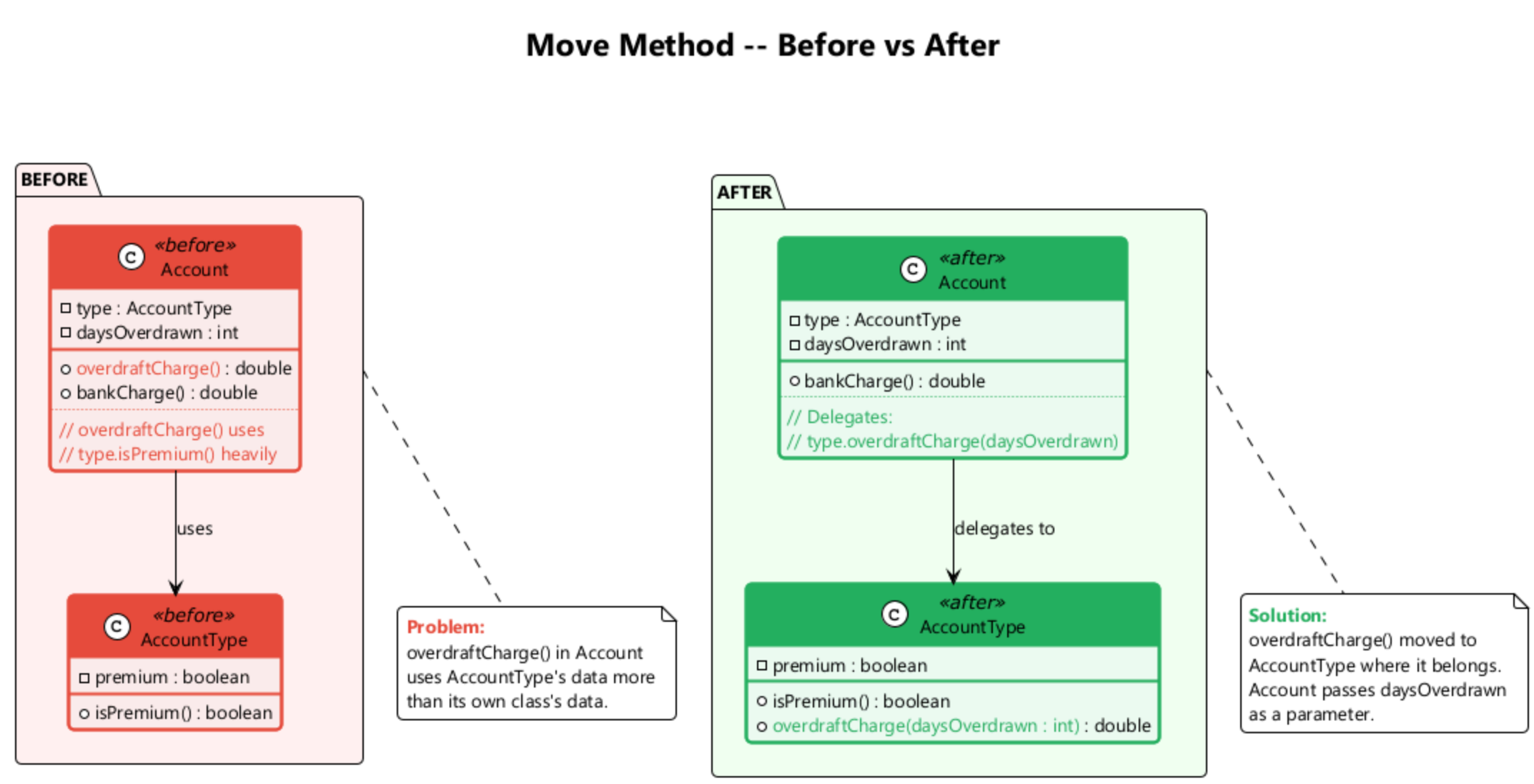
```
// Before
class Account {
    private AccountType type;
    private int daysOverdrawn;

    double overdraftCharge() {
        if (type.isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7) {
                result += (daysOverdrawn - 7) * 0.85;
            }
            return result;
        }
        return daysOverdrawn * 1.75;
    }

    double bankCharge() {
        double result = 4.5;
        if (daysOverdrawn > 0) {
            result += overdraftCharge();
        }
        return result;
    }
}
```

```
// After -- overdraftCharge moved to AccountType
class AccountType {
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7) {
                result += (daysOverdrawn - 7) * 0.85;
            }
            return result;
        }
        return daysOverdrawn * 1.75;
    }
```

Metodu Taşıma - Önce/Sonra Diyagramı



B2: Move Field (Alanı Taşıma)

Nedir: Bir alanı, onu daha sık kullanan sınıfa taşıyın.

Problem: Bir alan, kendi sınıfından çok başka bir sınıfta kullanılıyor.

Çözüm: Yeni sınıfta bir alan oluşturun ve eski alanın tüm kullanıcılarını yeni alana yönlendirin.

Ne zaman kullanılır: Bir alan, başka bir sınıfın metotları tarafından daha fazla kullanıldığında; Extract Class işleminin parçası olarak.

Ne zaman KULLANILMAZ: Alan, mevcut sınıfın kimliği için merkeziyken.

```
// Before
class Account {
    private double interestRate;
    private AccountType type;
}

// After -- interestRate moved to AccountType
class AccountType {
    private double interestRate;
    double getInterestRate() { return interestRate; }
}
class Account {
    private AccountType type;
    double getInterestRate() { return type.getInterestRate(); }
}
```

B3. Extract Class (Sınıf Çıkarma)

Nedir: Belirli bir sorumluluk için alanlar ve metotlar içeren yeni bir sınıf oluşturun.

Problem: Bir sınıf iki sınıfın işini yaptığında, hantallık ortaya çıkar. Sınıfın çok fazla sorumluluğu vardır.

Çözüm: Yeni bir sınıf oluşturun ve ilgili alanları ve metotları eski sınıftan yeni sınıfa taşıyın.

Ne zaman kullanılır: Bir sınıfın doğal olarak gruplanan ve kendi başına durabilecek özellikleri olduğunda.

Ne zaman KULLANILMAZ: Sınıf hala uyumlu olduğunda ve bölmek gereksiz bağımlılık yaratacağında.

B3. Extract Class (Sınıf Çıkarma) -- Önce / Sonra

```
// Before
class Person {
    private String name;
    private String officeAreaCode;
    private String officeNumber;

    String getTelephoneNumber() {
        return "(" + officeAreaCode + ") " + officeNumber;
    }
}
```

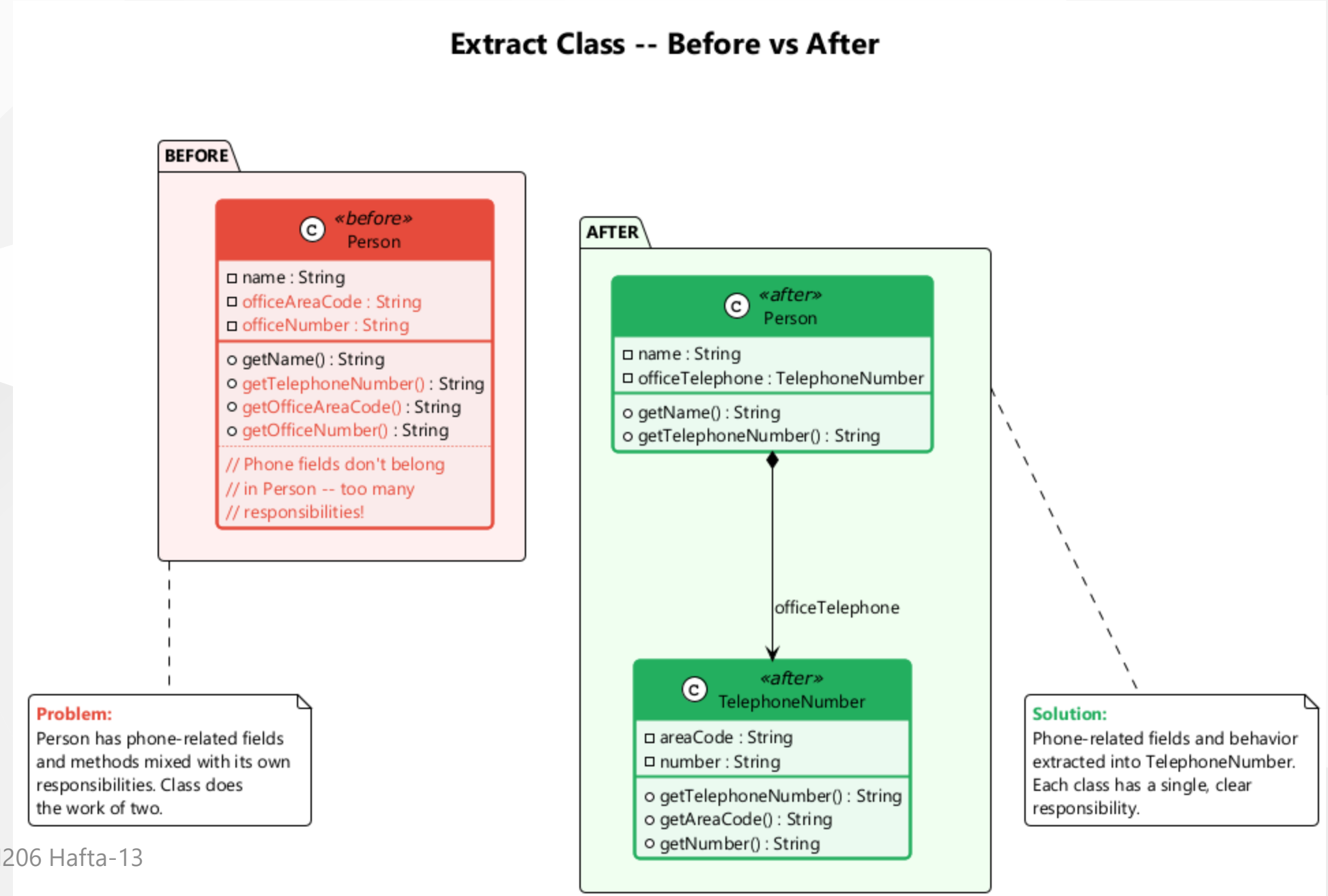
```
// After
class Person {
    private String name;
    private TelephoneNumber officeTelephone = new TelephoneNumber();

    String getTelephoneNumber() {
        return officeTelephone.getTelephoneNumber();
    }
}

class TelephoneNumber {
    private String areaCode;
    private String number;

    String getTelephoneNumber() {
        return "(" + areaCode + ") " + number;
    }
}
```

Sınıf Çıkarma - Önce/Sonra Diyagramı



B4. Inline Class (Satır İçi Sınıf)

Nedir: Yetersiz bir sınıfın tüm özelliklerini başka bir sınıfa taşıyın ve silin.

Problem: Bir sınıf neredeyse hiçbir şey yapmıyor, hiçbir şeyden sorumlu değil ve ek sorumluluk planlanmıyor.

Çözüm: Sınıfın tüm özelliklerini başka birine taşıyın ve boş sınıfı silin.

Ne zaman kullanılır: Yeniden yapılandırmalar bir sınıfı ince bir sarıcı dışında her şeyden arındırdığında.

Ne zaman KULLANILMAZ: Sınıfın hala açık bir sorumluluğu olduğunda veya büyümesi beklendiğinde.

```
// Before
class Person {
    private TelephoneNumber officeTelephone;
    String getAreaCode() { return officeTelephone.getAreaCode(); }
}
class TelephoneNumber { /* only areaCode and number */ }

// After -- TelephoneNumber merged back into Person
class Person {
    private String officeAreaCode;
    private String officeNumber;
    String getAreaCode() { return officeAreaCode; }
}
```

B5. Hide Delegate (Temsilciyi Gizleme)

Nedir: Temsilciyi istemciden gizlemek için sunucu üzerinde bir delege metodu oluşturun.

Problem: İstemci, A nesnesinden B nesnesini alır, ardından B nesnesinin bir metodunu çağırır (Demeter yasası ihlali: `a.getB().doSomething()`).

Çözüm: A sınıfında, çağrıyı B'ye devreden bir metot oluşturun ve istemcinin B hakkında bilgi sahibi olmasını engelleyin.

Ne zaman kullanılır: İstemciler nesne zincirleri arasında gezindiğinde (mesaj zincirleri).

Ne zaman KULLANILMAZ: Temsilci yaygın olarak bilinen, kararlı bir API olduğunda.

```
// Before
class Person {
    Department department;
    Department getDepartment() { return department; }
}
// Client code: person.getDepartment().getManager();

// After
class Person {
    Department department;
    Person getManager() { return department.getManager(); }
}
// Client code: person.getManager();
```

Nedir: İstemcinin temsilciyi doğrudan çağırmasını sağlayın, yönlendirme metotlarını kaldırın.

Problem: Bir sınıf, basitçe başka bir nesneye devreden çok fazla metoda sahiptir. Sunucu sınıf bir "aracı" haline gelmiştir.

Çözüm: Aracı metotları silin ve istemcinin son nesneyi doğrudan çağırmasına izin verin.

Ne zaman kullanılır: Bir sınıf basit delege metotlarla dolu olduğunda; Hide Delegate aşırıya kaçtığında.

Ne zaman KULLANILMAZ: Temsilciyi gizlemek kapsülleme için önemli olduğunda.

```
// Before
class Person {
    Department department;
    Person getManager() { return department.getManager(); }
    // many more delegations...
}
// Client: person.getManager();

// After
class Person {
    Department department;
    Department getDepartment() { return department; }
}
```

B7. Introduce Foreign Method (Yabancı Metot Ekleme)

Nedir: Sunucu nesnesini argüman olarak geçirerek istemci sınıfa bir yardımcı metot ekleyin.

Problem: Kullandığınız bir yardımcı sınıf, ihtiyacınız olan metodu içermiyor ve sınıfa metot ekleyemiyorsunuz.

Çözüm: Metodu istemci sınıfta oluşturun ve yardımcı sınıfın bir örneğini ilk argüman olarak geçin.

Ne zaman kullanılır: Değiştiremediğiniz bir sınıfta yalnızca bir veya iki ekstra metoda ihtiyacınız olduğunda.

Ne zaman KULLANILMAZ: Birden fazla metoda ihtiyacınız olduğunda -- bunun yerine Introduce Local Extension kullanın.

```
// Before
Date newStart = new Date(previousEnd.getYear(),
    previousEnd.getMonth(), previousEnd.getDate() + 1);

// After
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date date) {
    return new Date(date.getYear(), date.getMonth(), date.getDate() + 1);
}
```

B8. Introduce Local Extension (Yerel Uzantı Ekleme)

Nedir: İhtiyacınız olan ekstra metotları içeren yeni bir sınıf (alt sınıf veya sarıcı) oluşturun.

Problem: Bir yardımcı sınıf, ihtiyacınız olan birkaç metottan yoksun ve sınıfa metot ekleyemiyorsunuz.

Çözüm: Ekstra metotları içeren yeni bir sınıf oluşturun. Bu uzantı sınıfını orijinalin bir alt sınıfı veya sarıcısı yapın.

Ne zaman kullanılır: Değiştiremediğiniz bir sınıfta birden fazla ekstra metoda ihtiyacınız olduğunda.

Ne zaman KULLANILMAZ: Yalnızca bir veya iki metot gerektiğinde (bunun yerine Introduce Foreign Method kullanın).

B8. Introduce Local Extension (Yerel Uzantı Ekleme) -- Örnek

```
// Wrapper approach
class MfDateWrapper {
    private Date original;

    public MfDateWrapper(Date date) {
        this.original = date;
    }

    public MfDateWrapper(String dateString) {
        this.original = new Date(dateString);
    }

    public Date nextDay() {
        return new Date(original.getYear(), original.getMonth(), original.getDate() + 1);
    }

    public int dayOfYear() {
        // custom implementation
        Calendar cal = Calendar.getInstance();
        cal.setTime(original);
        return cal.get(Calendar.DAY_OF_YEAR);
    }

    // delegate all original methods...
    public int getYear() { return original.getYear(); }
    public int getMonth() { return original.getMonth(); }
    public int getDate() { return original.getDate(); }
}
```

Modül B -- Özet

Moving Features Between Objects (Özellikleri Nesneler Arasında Taşıma): Temel Noktalar

#	Teknik	Amaç
1	Move Method (Metodu Taşıma)	Metodu kullandığı verilerin yanına yerleştirme
2	Move Field (Alanı Taşıma)	Alanı, onu kullanan metotların yanına yerleştirme
3	Extract Class (Sınıf Çıkarma)	Aşırı yüklenmiş sınıfları bölme
4	Inline Class (Satır İçi Sınıf)	Yetersiz sınıfları birleştirme

Modül C: Organizing Data (Verileri Düzenleme)

Daha İyi Veri Yönetimi için 15 Teknik

Organizing Data (Verileri Düzenleme)

Bu yeniden yapılandırmalar, ilkel türleri zengin sınıf temsilleriyle değiştirerek ve sınıf ilişkilerini sadeleştirerek veri işlemeyi kolaylaştırır.

1. Self Encapsulate Field (Alanı Kendi İçinde Kapsülleme)
2. Replace Data Value with Object (Veri Değerini Nesne ile Değiştirme)
3. Change Value to Reference (Değeri Referansa Çevirme)
4. Change Reference to Value (Referansı Değere Çevirme)
5. Replace Array with Object (Diziyi Nesne ile Değiştirme)
6. Duplicate Observed Data (Gözlenen Veriyi Çoğaltma)
7. Change Unidirectional Association to Bidirectional (Tek Yönlü İlişkiyi Çift Yönlü Yapma)
8. Change Bidirectional Association to Unidirectional (Çift Yönlü İlişkiyi Tek Yönlü Yapma)
9. Replace Magic Number with Symbolic Constant (Sihirli Sayıyı Sembolik Sabit ile Değiştirme)
10. Encapsulate Field (Alanı Kapsülleme)
11. Encapsulate Collection (Koleksiyonu Kapsülleme)
12. Replace Type Code with Class (Tür Kodunu Sınıf ile Değiştirme)
13. Replace Type Code with Subclasses (Tür Kodunu Alt Sınıflarla Değiştirme)
14. Replace Type Code with State/Strategy (Tür Kodunu State/Strategy ile Değiştirme)

C1. Self Encapsulate Field (Alanı Kendi İçinde Kapsülleme)

Nedir: Sınıf içinde bile bir alana yalnızca getter ve setter metotları aracılığıyla erişin.

Problem: Sınıf içinde özel bir alana doğrudan erişiyorsunuz, bu da alanın davranışını daha sonra değiştirmeyi zorlaştırıyor.

Çözüm: Alan için bir getter ve setter oluşturun ve alana erişmek için yalnızca bunları kullanın.

Ne zaman kullanılır: Alt sınıflarda alan erişimini geçersiz kılmak istediğinizde; doğrulama veya tembel başlatma eklemek istediğinizde.

Ne zaman KULLANILMAZ: Doğrudan erişim daha basit olduğunda ve dolaylı erişim için öngörülebilir bir ihtiyaç olmadığında.

C1. Self Encapsulate Field -- Önce / Sonra

```
// Before
class IntRange {
    private int low, high;

    boolean includes(int arg) {
        return arg >= low && arg <= high;
    }
}
```

```
// After
class IntRange {
    private int low, high;

    int getLow() { return low; }
    int getHigh() { return high; }

    boolean includes(int arg) {
        return arg >= getLow() && arg <= getHigh();
    }
}

class CappedRange extends IntRange {
    private int cap;

    @Override
    int getHigh() {
        return Math.min(super.getHigh(), cap);
    }
}
```

C2. Replace Data Value with Object (Veri Değerini Nesne ile Değiştirme)

Nedir: Bir veri alanını, kendi alanları ve davranışı olan bir nesneye dönüştürün.

Problem: Ek veri veya davranışa ihtiyaç duyan bir veri alanınız var (örn. telefon numarası için biçimlendirme, doğrulama).

Çözüm: Yeni bir sınıf oluşturun, eski alanı ve davranışını sınıfa yerleştirin ve orijinal sınıfta bu sınıfın bir nesnesini saklayın.

Ne zaman kullanılır: Basit bir alan kendi davranışına ihtiyaç duyacak şekilde geliştiğinde.

Ne zaman KULLANILMAZ: Veri gerçekten ek davranışa ihtiyacı olmayan basit bir değer olduğunda.

C2. Replace Data Value with Object -- Önce / Sonra

```
// Before
class Order {
    private String customer; // just a name string

    String getCustomer() { return customer; }
    void setCustomer(String customer) { this.customer = customer; }
}
```

```
// After
class Order {
    private Customer customer;

    String getCustomerName() { return customer.getName(); }
    void setCustomer(String customerName) {
        this.customer = new Customer(customerName);
    }
}

class Customer {
    private String name;

    Customer(String name) { this.name = name; }
    String getName() { return name; }
}
```

C3. Change Value to Reference (Değeri Referansa Çevirme)

Nedir: Aynı değer nesnelerini tek bir referans nesnesine dönüştürün.

Problem: Tek bir sınıfın birden fazla aynı örneği var ve bunları tek bir nesneyle değiştirmeniz gerekiyor (örn. aynı müşteri için birden fazla Customer nesnesi).

Çözüm: Değer nesnelerini referans nesnelerine dönüştürün. Her mantıksal varlık için yalnızca bir örneğin var olmasını sağlamak için bir fabrika veya kayıt defteri kullanın.

Ne zaman kullanılır: Birden fazla eşit nesne tutarlılık için aynı fiziksel nesne olması gerektiğinde.

Ne zaman KULLANILMAZ: Nesnelerin bağımsız olarak değiştirilebilir olması gerektiğinde.

C3. Change Value to Reference -- Önce / Sonra


```
// Before -- each Order creates its own Customer
class Order {
    private Customer customer;

    Order(String customerName) {
        customer = new Customer(customerName);
    }
}
```

```
// After -- Orders share a single Customer reference
class Customer {
    private static Map<String, Customer> instances = new HashMap<>();

    static Customer getNamed(String name) {
        if (!instances.containsKey(name)) {
            instances.put(name, new Customer(name));
        }
        return instances.get(name);
    }

    private String name;
    private Customer(String name) { this.name = name; }
}

class Order {
    private Customer customer;

    Order(String customerName) {
        customer = Customer.getNamed(customerName);
    }
}
```

C4: Change Reference to Value (Referansı Değere Çevirme)

Nedir: Bir referans nesnesini değer nesnesine dönüştürün.

Problem: Küçük, değişmez ve yönetimi zahmetli bir referans nesneniz var (yaşam döngüsü yönetimi, kayıt defteri vb. gerektirir).

Çözüm: Onu bir değer nesnesine dönüştürün. Değişmez yapın ve equals/hashCode metodlarını geçersiz kılın.

Ne zaman kullanılır: Referans yönetimi maliyeti haklı görülmediğinde; nesne küçük ve nadiren değiştiğinde.

Ne zaman KULLANILMAZ: Nesnenin sistem genelinde paylaşılması ve değiştirilmesi gerektiğinde.

```
// Before -- Currency as reference object with registry
Currency usd = Currency.get("USD");

// After -- Currency as value object
class Currency {
    private String code;
    Currency(String code) { this.code = code; }
    public boolean equals(Object o) {
        return o instanceof Currency && ((Currency)o).code.equals(code);
    }
    public int hashCode() { return code.hashCode(); }
}
```

C5. Replace Array with Object (Diziyi Nesne ile Değiştirme)

Nedir: Farklı veri türleri içeren bir diziyi, isimlendirilmiş alanlara sahip bir nesneyle değiştirin.

Problem: Elemanları farklı anlamlar taşıyan bir diziniz var (örn. `row[0]` isim, `row[1]` yaş).

Çözüm: Diziyi, her eleman için bir alana sahip bir nesneyle değiştirin.

Ne zaman kullanılır: Dizi elemanları farklı kavramları temsil ettiğinde.

Ne zaman KULLANILMAZ: Dizi gerçekten aynı türdeki elemanların bir listesini temsil ettiğinde.

```
// Before
String[] row = new String[2];
row[0] = "Liverpool";
row[1] = "15";

// After
class Performance {
    private String name;
    private int wins;
    String getName() { return name; }
    int getWins() { return wins; }
}
Performance row = new Performance();
row.setName("Liverpool");
row.setWins(15);
```

C6. Duplicate Observed Data (Gözlenen Veriyi Çoğaltma)

Nedir: Alan verisini alan nesnelerine kopyalayın ve iki kopya arasında senkronizasyon sağlamak için bir gözlemci kurun.

Problem: Alan verisi yalnızca GUI sınıflarında saklanıyor ve alan metotlarının buna erişmesi gerekiyor.

Çözüm: Veriyi bir alan nesnesine kopyalayın ve iki kopya arasında senkronizasyon sağlamak için bir gözlemci kurun.

Ne zaman kullanılır: GUI denetimleri, iş mantığına erişilebilir olması gereken alan verisi tuttuğunda.

Ne zaman KULLANILMAZ: Veri yalnızca görüntüleme amaçlı olup alan mantığı olmadığında.

C6. Duplicate Observed Data -- Önce / Sonra

```
// Before -- domain data trapped in GUI
class IntervalWindow extends Frame {
    TextField startField;
    TextField endField;
    TextField lengthField;

    void calculateLength() {
        int start = Integer.parseInt(startField.getText());
        int end = Integer.parseInt(endField.getText());
        lengthField.setText(String.valueOf(end - start));
    }
}
```

```
// After -- domain object with observer
class Interval implements Observable {
    private int start, end;

    int getLength() { return end - start; }

    void setStart(int start) {
        this.start = start;
        notifyObservers();
    }
    void setEnd(int end) {
        this.end = end;
        notifyObservers();
    }
}

class IntervalWindow extends Frame implements Observer {
    Interval interval = new Interval();
    // GUI syncs with domain model via Observer pattern
}
```

C7. Change Unidirectional Association to Bidirectional (Tek Yönlü İlişkiyi Çift Yönlü Yapma)

Nedir: Her iki sınıfın da birbirine referans verebilmesi için geri işaretçi ekleyin.

Problem: Her birinin diğerinin özelliklerini kullanması gereken iki sınıfınız var, ancak yalnızca tek yönlü bir ilişki mevcut.

Çözüm: Geri referans ekleyin ve bağlantı değiştiğinde her iki yönü de güncelleyin.

Ne zaman kullanılır: Her iki nesnenin de birbirine gezinmesi gerektiğinde.

Ne zaman KULLANILMAZ: Çift yönlü bağlantı gereksiz karmaşıklık eklediğinde; mümkünse tek yönlü bağlantıları tercih edin.

C7. Change Unidirectional to Bidirectional -- Örnek

```
// Before -- Order knows Customer, but Customer doesn't know Orders
class Order {
    private Customer customer;
    Customer getCustomer() { return customer; }
    void setCustomer(Customer customer) { this.customer = customer; }
}

// After -- bidirectional association
class Order {
    private Customer customer;
    Customer getCustomer() { return customer; }
    void setCustomer(Customer customer) {
        if (this.customer != null) this.customer.removeOrder(this);
        this.customer = customer;
        if (customer != null) customer.addOrder(this);
    }
}

class Customer {
    private Set<Order> orders = new HashSet<>();
    Set<Order> getOrders() { return Collections.unmodifiableSet(orders); }
    void addOrder(Order order) { orders.add(order); }
    void removeOrder(Order order) { orders.remove(order); }
}
```

Yönlü Yapma)

Nedir: Çift yönlü ilişkinin gereksiz yönünü kaldırın.

Problem: Çift yönlü bir ilişkiniz var ama bir sınıf artık diğerinin özelliklerine ihtiyaç duymuyor.

Çözüm: İhtiyaç duyulmayan ilişki ucunu kaldırın.

Ne zaman kullanılır: Bir sınıf artık diğerine referansı kullanmadığında; bağımlılık yapısını sadeleştirmek için.

Ne zaman KULLANILMAZ: Her iki yön aktif olarak kullanıldığında.

```
// Before -- bidirectional
class Order {
    Customer customer;
}
class Customer {
    Set<Order> orders;
}

// After -- only Order -> Customer remains
class Order {
    Customer customer;
}
class Customer {
    // orders field removed -- if needed, query from Order repository
```

C9. Replace Magic Number with Symbolic Constant (Sihirli Sayıyı Sembolik Sabit ile Değiştirme)

Nedir: Sabit bir sayıyı isimlendirilmiş bir sabitle değiştirin.

Problem: Kodunuz belirli bir anlama sahip bir sayı kullanıyor (örn. yerçekimi ivmesi için 9.81).

Çözüm: Bu sayıyı, anlamını açıklayan okunabilir bir isme sahip bir sabitle değiştirin.

Ne zaman kullanılır: Bir sayının anlamı bağlamdan anlaşılamıyorsa.

Ne zaman KULLANILMAZ: Sayı açıkça anlaşılır olduğunda (örn. basit bağlamlarda 0, 1).

```
// Before
double potentialEnergy(double mass, double height) {
    return mass * 9.81 * height;
}

// After
static final double GRAVITATIONAL_CONSTANT = 9.81;

double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
```

C10. Encapsulate Field (Alanı Kapsülleme)

Nedir: Genel bir alanı özel yapın ve erişim metotları sağlayın.

Problem: Dış sınıflar tarafından doğrudan erişilen genel bir alanınız var.

Çözüm: Alanı özel yapın ve getter ile setter metotları sağlayın.

Ne zaman kullanılır: Her zaman -- genel alanlar kapsüllemeyi ihlal eder.

Ne zaman KULLANILMAZ: Kapsüllemenin önemli olmadığı basit veri transfer nesnelerinde (yine de iyi bir uygulamadır).

```
// Before
class Person {
    public String name;
}
// Client: person.name = "John";

// After
class Person {
    private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
// Client: person.setName("John");
```

C11. Encapsulate Collection (Koleksiyonu Kapsülleme)

Nedir: Koleksiyon getter metodunun salt okunur bir görünüm döndürmesini sağlayın ve ayrı ekleme/çıkarma metotları sağlayın.

Problem: Bir metot doğrudan bir koleksiyon döndürüyor, bu da çağıranların sahip sınıf bilmeden değiştirmesine izin veriyor.

Çözüm: Getter metodunun koleksiyonun salt okunur görünümünü veya kopyasını döndürmesini sağlayın ve ayrı ekleme/çıkarma metotları sağlayın.

Ne zaman kullanılır: Dış kod, sahip sınıfın bilgisi olmadan bir koleksiyonu değiştirebildiğinde.

Ne zaman KULLANILMAZ: Koleksiyonun dış kod tarafından serbestce değiştirilmesi gerektiğinde (nadiren).

C11. Encapsulate Collection -- Önce / Sonra

```
// Before
class Course {}
class Person {
    private List<Course> courses = new ArrayList<>();
    List<Course> getCourses() { return courses; }
    void setCourses(List<Course> courses) { this.courses = courses; }
}
// Client: person.getCourses().add(new Course());  // bypasses Person
```

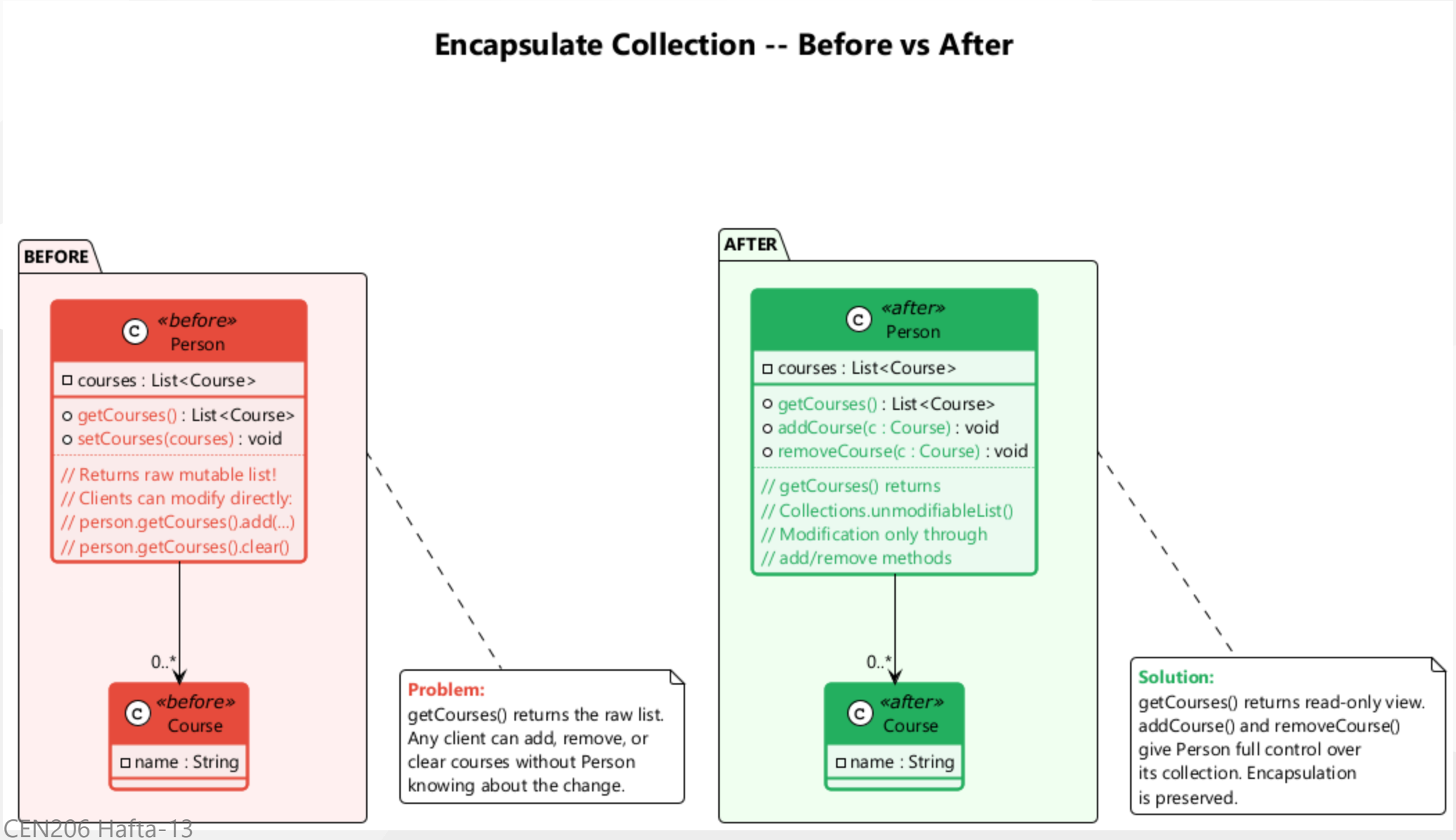
```
// After
class Person {
    private List<Course> courses = new ArrayList<>();

    List<Course> getCourses() {
        return Collections.unmodifiableList(courses);
    }

    void addCourse(Course course) {
        courses.add(course);
    }

    void removeCourse(Course course) {
        courses.remove(course);
    }
}
// Client: person.addCourse(new Course());  // Person controls its collection
```

Koleksiyonu Kapsülleme - Önce/Sonra Diyagramı



C12. Replace Type Code with Class (Tür Kodunu Sınıf ile Değiştirme)

Nedir: Bir tür kodunu (int veya String) bir sınıfla değiştirin.

Problem: Bir sınıf, tür kodu içeren bir alana sahip. Bu türün değerleri operatör koşullarında kullanılmıyor ve programın davranışını etkilemiyor.

Çözüm: Yeni bir sınıf oluşturun ve tür kodu değerleri yerine nesnelerini kullanın.

Ne zaman kullanılır: Tür kodları yalnızca veri için kullanıldığında (davranış bunlara bağlı değilken).

Ne zaman KULLANILMAZ: Tür kodu davranışı etkilediğinde -- bunun yerine Replace Type Code with Subclasses veya State/Strategy kullanın.

C12. Replace Type Code with Class -- Önce / Sonra

```
// Before
class Person {
    static final int O = 0;
    static final int A = 1;
    static final int B = 2;
    static final int AB = 3;
    private int bloodGroup;
    Person(int bloodGroup) { this.bloodGroup = bloodGroup; }
}
```

```
// After
class BloodGroup {
    public static final BloodGroup O = new BloodGroup(0);
    public static final BloodGroup A = new BloodGroup(1);
    public static final BloodGroup B = new BloodGroup(2);
    public static final BloodGroup AB = new BloodGroup(3);

    private final int code;
    private BloodGroup(int code) { this.code = code; }
    int getCode() { return code; }
}

class Person {
    private BloodGroup bloodGroup;
    Person(BloodGroup bloodGroup) { this.bloodGroup = bloodGroup; }
}
// Usage: new Person(BloodGroup.A);
```

C13. Replace Type Code with Subclasses (Tür Kodunu Alt Sınıflarla Değiştirme)

Nedir: Tür kodunun her değeri için alt sınıflar oluşturun.

Problem: Program davranışını doğrudan etkileyen kodlanmış bir türünüz var (örn. tür koduna göre değişen koşullar).

Çözüm: Kodlanmış türün her değeri için alt sınıflar oluşturun. İlgili davranışları orijinal sınıftan bu alt sınıflara çıkarın. Kontrol akışı kodunu polimorfizm ile değiştirin.

Ne zaman kullanılır: Tür kodu davranışı etkilediğinde ve tür oluşturulduktan sonra değişmediğinde.

Ne zaman KULLANILMAZ: Tür kodu çalışma zamanında değişebildiğinde (bunun yerine State/Strategy kullanın).

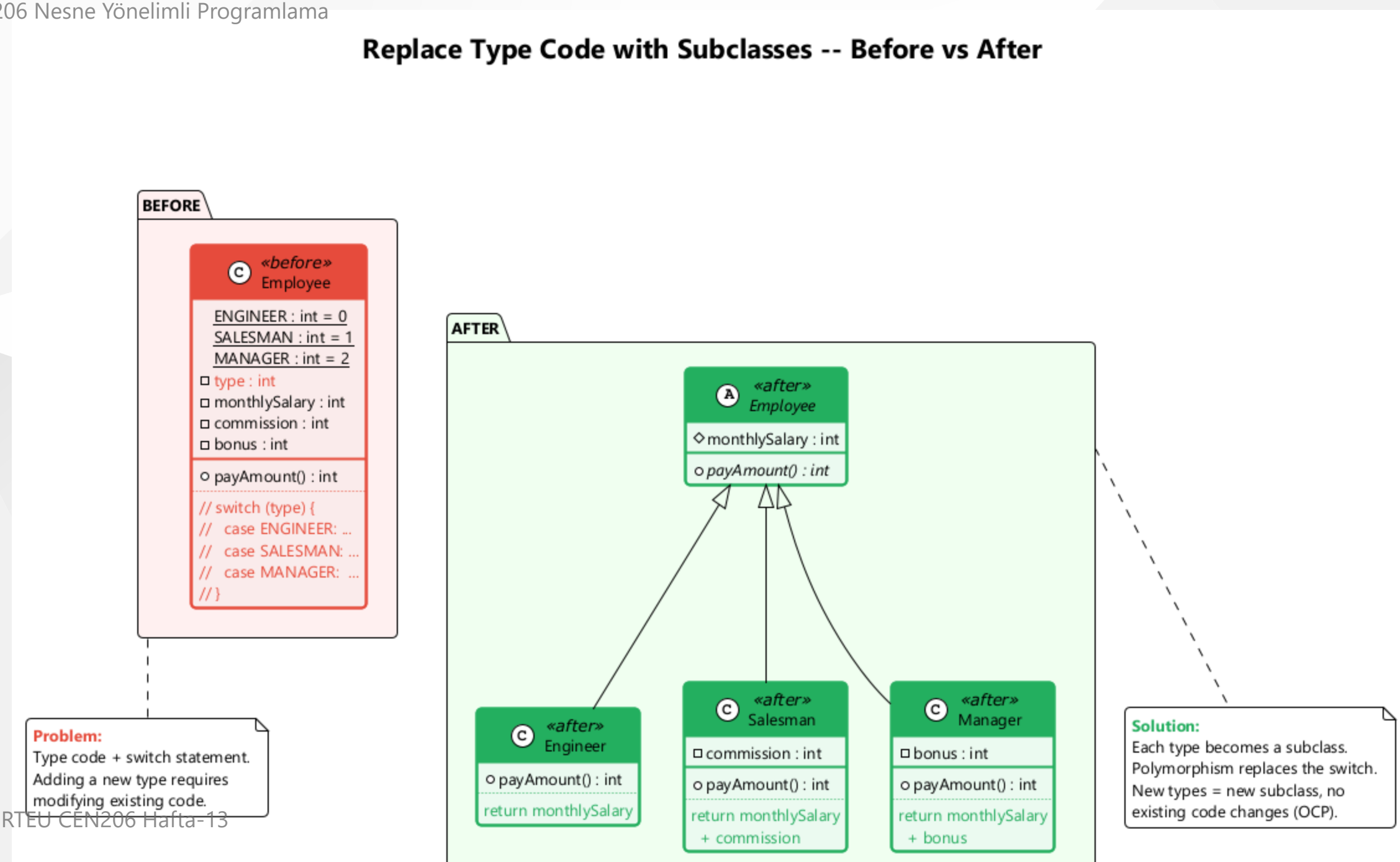
C13. Replace Type Code with Subclasses -- Önce / Sonra

```
// Before
class Employee {
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;
    private int type;

    int payAmount() {
        switch (type) {
            case ENGINEER: return monthlySalary;
            case SALESMAN: return monthlySalary + commission;
            case MANAGER:  return monthlySalary + bonus;
            default: throw new RuntimeException("Invalid type");
        }
    }
}
```

```
// After
abstract class Employee {
    abstract int payAmount();
}
class Engineer extends Employee {
    int payAmount() { return monthlySalary; }
}
class Salesman extends Employee {
    int payAmount() { return monthlySalary + commission; }
}
class Manager extends Employee {
    int payAmount() { return monthlySalary + bonus; }
}
```

Tür Kodunu Alt Sınıflara Değiştirme - Diyagram



C14. Replace Type Code with State/Strategy (Tür Kodunu State/Strategy ile Değiştirme)

Nedir: Tür kodunu State veya Strategy deseni kullanarak bir durum nesnesiyle değiştirin.

Problem: Davranışı etkileyen kodlanmış bir türünüz var, ancak tür çalışma zamanında değiştiği veya sınıf başka bir nedenle zaten alt sınıflanmış olduğu için alt sınıflama kullanamazsınız.

Çözüm: Tür kodunu bir durum nesnesiyle değiştirin. State veya Strategy desenini kullanın.

Ne zaman kullanılır: Tür kodu davranışı etkileyen VE tür nesnenin yaşam süresi boyunca değiştiğinde.

Ne zaman KULLANILMAZ: Tür oluşturulduktan sonra değişmez olduğunda (bunun yerine Alt Sınıflar kullanın).

C14. Replace Type Code with State/Strategy -- Örnek

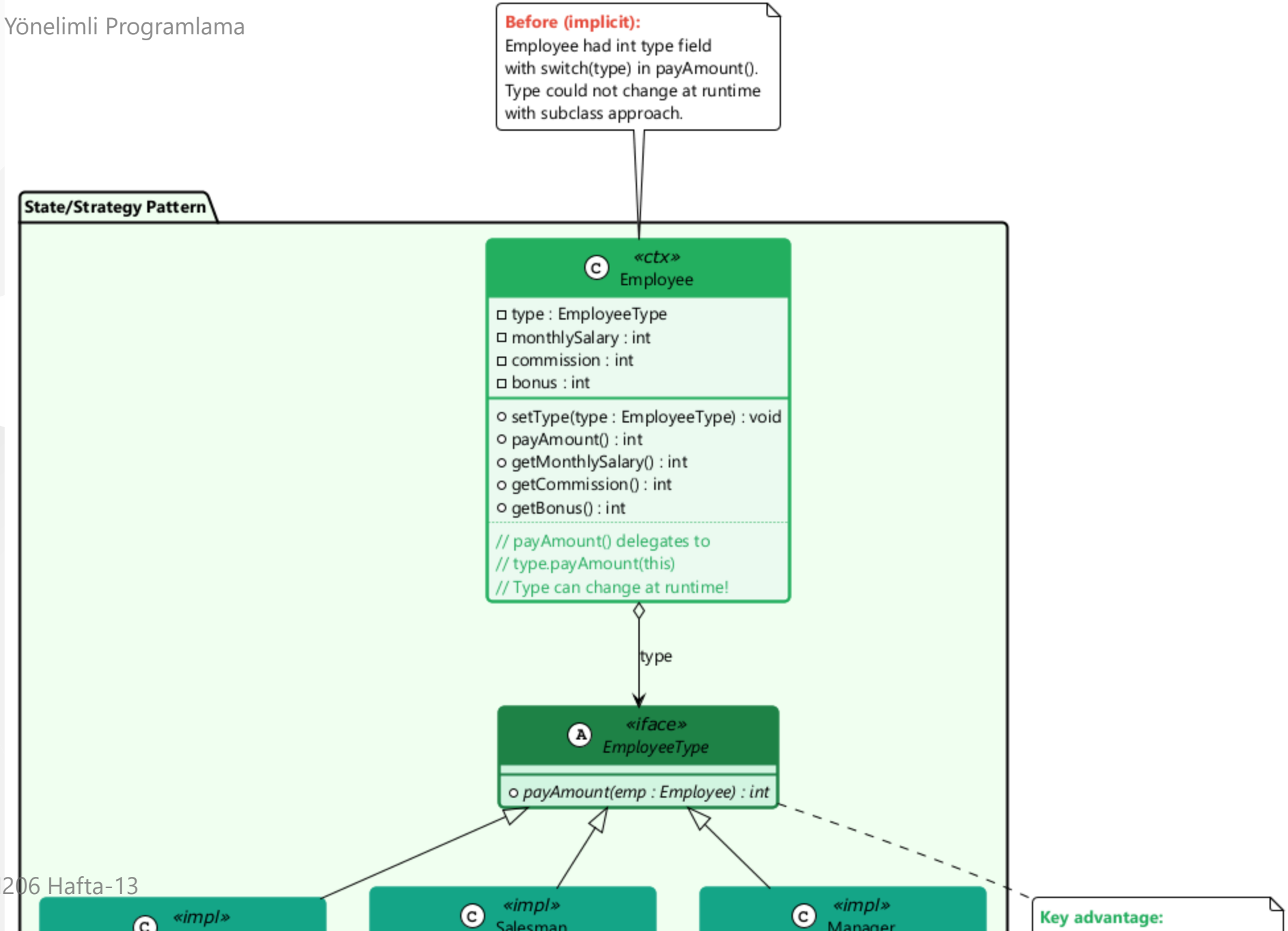
```
// After applying State/Strategy pattern
abstract class EmployeeType {
    abstract int payAmount(Employee employee);
}

class Engineer extends EmployeeType {
    int payAmount(Employee emp) { return emp.getMonthlySalary(); }
}
class Salesman extends EmployeeType {
    int payAmount(Employee emp) { return emp.getMonthlySalary() + emp.getCommission(); }
}
class Manager extends EmployeeType {
    int payAmount(Employee emp) { return emp.getMonthlySalary() + emp.getBonus(); }
}

class Employee {
    private EmployeeType type;

    void setType(EmployeeType type) { this.type = type; }

    int payAmount() { return type.payAmount(this); }
}
// Now the type can change at runtime: emp.setType(new Manager());
```


Before (implicit):
Employee had int type field
with switch(type) in payAmount().
Type could not change at runtime
with subclass approach.
State/Strategy Pattern
«ctx»
Employee
type : EmployeeType
monthlySalary : int
commission : int
bonus : int
setType(type : EmployeeType) : void
payAmount() : int
getMonthlySalary() : int
getCommission() : int
getBonus() : int
// payAmount() delegates to
// type.payAmount(this)
// Type can change at runtime!
type
«iface»
EmployeeType
payAmount(emp : Employee) : int
«impl»
Salesman
«impl»
Manager
Key advantage:

C15. Replace Subclass with Fields (Alt Sınıfı Alanlarla Değiştirme)

Nedir: Tek farkı sabit döndüren metotlar olan alt sınıfları, üst sınıftaki alanlarla değiştirin.

Problem: Yalnızca sabit veri döndüren metotlarında farklılaşan alt sınıflarınız var (örn. `getCode()` farklı değerler döndürür).

Çözüm: Metotları üst sınıftaki alanlarla değiştirin ve alt sınıfları ortadan kaldırın.

Ne zaman kullanılır: Alt sınıflar gerçek bir davranış eklemediğinde, yalnızca farklı sabit değerler döndürdüğünde.

Ne zaman KULLANILMAZ: Alt sınıfların sabit döndürmenin ötesinde anlamlı davranışsal farklılıkları olduğunda.

C15. Replace Subclass with Fields -- Önce / Sonra

```
// Before
abstract class Person {
    abstract boolean isMale();
    abstract char getCode();
}
class Male extends Person {
    boolean isMale() { return true; }
    char getCode() { return 'M'; }
}
class Female extends Person {
    boolean isMale() { return false; }
    char getCode() { return 'F'; }
}
```

```
// After
class Person {
    private final boolean male;
    private final char code;

    private Person(boolean male, char code) {
        this.male = male;
        this.code = code;
    }

    static Person createMale() { return new Person(true, 'M'); }
    static Person createFemale() { return new Person(false, 'F'); }

    boolean isMale() { return male; }
    char getCode() { return code; }
}
```

Modül C -- Özet

Organizing Data (Verileri Düzenleme): Temel Noktalar

#	Teknik	Amaç
1	Self Encapsulate Field	Alanlara dahili olarak bile getter/setter ile erişme
2	Replace Data Value with Object	İlkel türleri davranışa sahip nesnelere yükseltime
3	Change Value to Reference	Aynı nesneleri kayıt defteri aracılığıyla paylaşma
4	Change Reference to Value	Küçük, değişmez nesneleri sadeleştirme
5	Replace Array with Object	Dizi elemanlarını türlü alanlarla isimlendirme
6	Duplicate Observed Data	GUI verisini alan verisinden ayırma
7	Tek Yönlüden Çift Yönlüye	Karşılıklı gezinme için geri işaretçi ekleme
8	Çift Yönlüden Tek Yönlüye	Gereksiz ilişki yönünü kaldırma
9	Replace Magic Number	Netlik için isimlendirilmiş sabitler kullanma
10	Encapsulate Field	Genel alanları erişimçilerle özel yapma

Modül D: Simplifying Conditional Expressions (Koşulları Sadeleştirme)

Daha Temiz Koşullar için 8 Teknik

Simplifying Conditional Expressions (Koşulları Sadeleştirme)

Koşullar zamanla mantıklarında giderek daha karmaşık hale gelme eğilimindedir ve bununla mücadele etmek için daha fazla teknik vardır. Bu yeniden yapılandırmalar karmaşıklığı azaltır ve koşullu mantığın okunabilirliğini artırır.

1. Decompose Conditional (Koşulu Ayrıştırma)
2. Consolidate Conditional Expression (Koşullu İfadeyi Birleştirme)
3. Consolidate Duplicate Conditional Fragments (Tekrarlanan Koşul Parçalarını Birleştirme)
4. Remove Control Flag (Kontrol Bayrağını Kaldırma)
5. Replace Nested Conditional with Guard Clauses (İç İçe Koşulu Koruma Cümleleriyle Değiştirme)
6. Replace Conditional with Polymorphism (Koşulu Polimorfizm ile Değiştirme)
7. Introduce Null Object (Null Nesne Ekleme)

D1. Decompose Conditional (Koşulu Ayrıştırma)

Nedir: Karmaşık koşul parçalarını ayrı, iyi isimlendirilmiş metotlara çıkarın.

Problem: Karmaşık bir koşulunuz (if-then-else veya switch) var ve koşul, then-dalı ve else-dalı anlaşılması zor.

Çözüm: Koşulun karmaşık parçalarını ayrı metotlara ayırın: koşul, then-dalı ve else-dalı.

Ne zaman kullanılır: Koşullu mantık bir bakışta anlaşılması güç hale geldiğinde.

Ne zaman KULLANILMAZ: Koşul zaten basit ve kendi kendini açıklayan olduğunda.

D1. Decompose Conditional -- Önce / Sonra

```
// Before
double calculateCharge(Date date, int quantity) {
    double charge;
    if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
        charge = quantity * winterRate + winterServiceCharge;
    } else {
        charge = quantity * summerRate;
    }
    return charge;
}
```

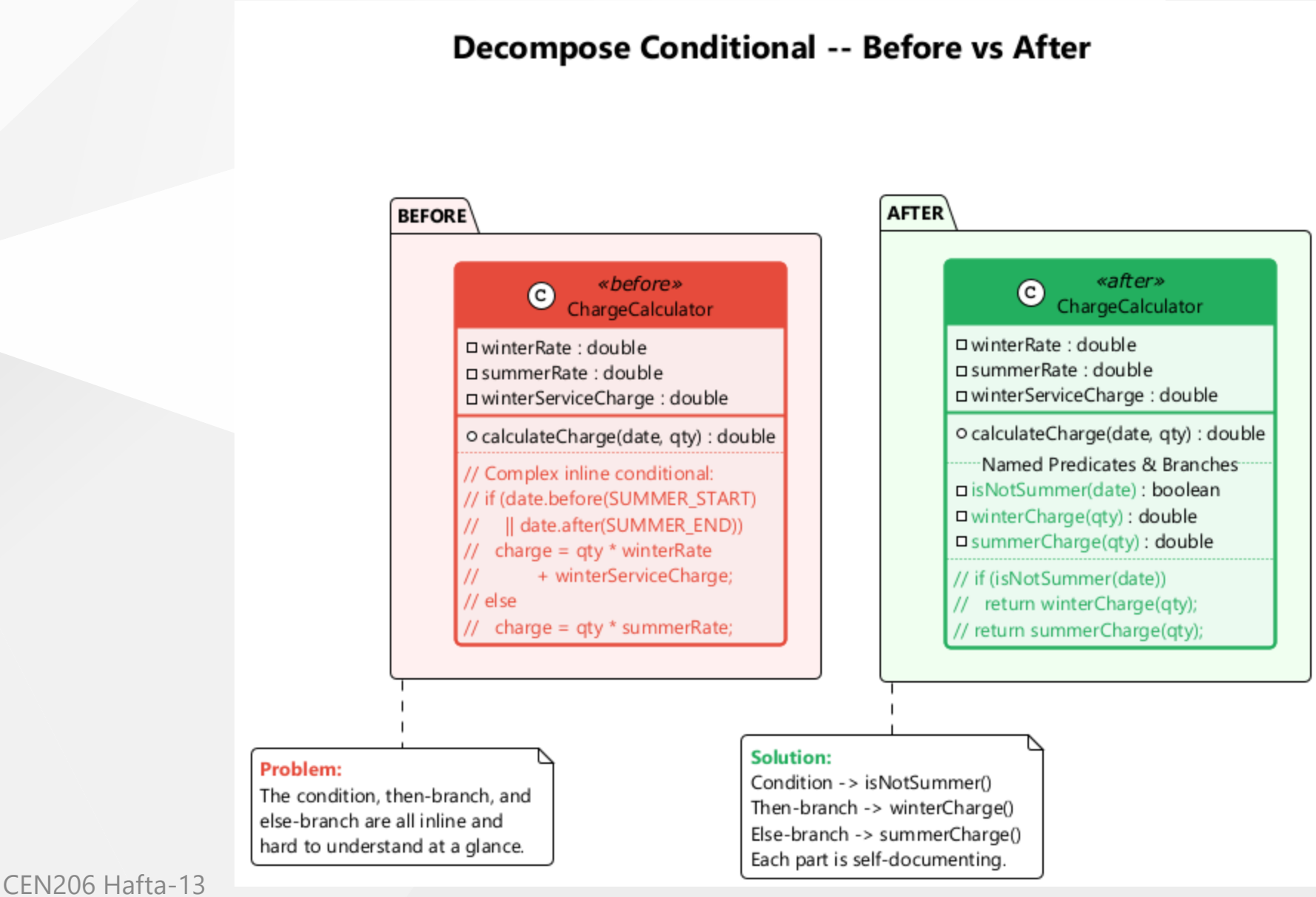
```
// After
double calculateCharge(Date date, int quantity) {
    if (isNotSummer(date)) {
        return winterCharge(quantity);
    }
    return summerCharge(quantity);
}

boolean isNotSummer(Date date) {
    return date.before(SUMMER_START) || date.after(SUMMER_END);
}

double winterCharge(int quantity) {
    return quantity * winterRate + winterServiceCharge;
}

double summerCharge(int quantity) {
    return quantity * summerRate;
}
```

Koşulu Ayrıştırma - Önce/Sonra Diyagramı



D2. Consolidate Conditional Expression (Koşullu İfadeyi Birleştirme)

Nedir: Aynı sonuca yol açan birden fazla koşulu tek bir ifadede birleştirin.

Problem: Aynı sonuca veya eyleme yol açan birden fazla koşulunuz var.

Çözüm: Tüm bu koşulları tek bir ifadede birleştirin, ardından birleştirilmiş koşulu iyi isimlendirilmiş bir metoda çıkarın.

Ne zaman kullanılır: Birden fazla koşul aynı sonuca yol açtığında ve mantıksal olarak birleştirilebildiğinde.

Ne zaman KULLANILMAZ: Koşullar gerçekten bağımsız kontroller olup yalnızca tesadüfen aynı sonucu döndürdüğünde.

D2. Consolidate Conditional Expression -- Önce / Sonra

```
// Before
double disabilityAmount() {
    if (seniority < 2) return 0;
    if (monthsDisabled > 12) return 0;
    if (isPartTime) return 0;
    // compute the disability amount
    return basePay * 0.5;
}
```

```
// After
double disabilityAmount() {
    if (isNotEligibleForDisability()) return 0;
    return basePay * 0.5;
}

boolean isNotEligibleForDisability() {
    return seniority < 2 || monthsDisabled > 12 || isPartTime;
}
```

Parçalarını Birleştirme)

Nedir: Tüm dallarda görünen aynı kodu koşulun dışına taşıyın.

Problem: Bir koşulun tüm dallarında aynı kod bulunuyor.

Çözüm: Tekrarlanan kodu koşulun dışına taşıyın.

Ne zaman kullanılır: Aynı ifade her dalın başında veya sonunda göründüğünde.

Ne zaman KULLANILMAZ: Kod yalnızca benzer görünüp her dalda anlamsal olarak farklı olduğunda.

```
// Before
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}

// After
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
```

D4. Remove Control Flag (Kontrol Bayrağını Kaldırma)

Nedir: Boolean kontrol bayrağı yerine `break`, `continue` veya `return` kullanın.

Problem: Birden fazla boolean ifade için kontrol bayrağı görevi gören bir boolean değişkeniniz var.

Çözüm: Bayrak değişkeni yerine `break`, `continue` ve `return` kullanın.

Ne zaman kullanılır: Boolean değişkenler yalnızca döngü veya koşullu yürütme akışını kontrol etmek için kullanıldığında.

Ne zaman KULLANILMAZ: Bayrak değişkenin akış kontrolünün ötesinde gerçek bir alan anlamı olduğunda.

D4. Remove Control Flag -- Önce / Sonra

```
// Before
boolean found = false;
for (Person person : people) {
    if (!found) {
        if (person.getName().equals("Don")) {
            sendAlert();
            found = true;
        }
        if (person.getName().equals("John")) {
            sendAlert();
            found = true;
        }
    }
}
```

```
// After
for (Person person : people) {
    if (person.getName().equals("Don") || person.getName().equals("John")) {
        sendAlert();
        break;
    }
}
```

D5. Replace Nested Conditional with Guard Clauses (İç İçe Koşulu Koruma Cümleleriyle Değiştirme)

Nedir: Özel durumlar için koruma cümleleri kullanın, normal yolu girintisiz bırakın.

Problem: Kod yürütmenin normal akışını belirlemeyi zorlaştıran iç içe koşullar grubunuz var.

Çözüm: Tüm özel kontrolleri ve sınır durumlarını ayrı cümlelere (koruma cümleleri) ayırın ve ana mantıktan önce yerleştirin. Her koruma cümlesinden erken dönüş yapın.

Ne zaman kullanılır: Derin iç içe koşullar kodu takip etmeyi zorlaştırdığında.

Ne zaman KULLANILMAZ: Koşul gerçekten iki eşit alternatifi temsil ettiğinde (if-else kullanın).

D5. Replace Nested Conditional with Guard Clauses -- Önce / Sonra

```
// Before
double getPayAmount() {
    double result;
    if (isDead) {
        result = deadAmount();
    } else {
        if (isSeparated) {
            result = separatedAmount();
        } else {
            if (isRetired) {
                result = retiredAmount();
            } else {
                result = normalPayAmount();
            }
        }
    }
    return result;
}
```

```
// After
double getPayAmount() {
    if (isDead) return deadAmount();
    if (isSeparated) return separatedAmount();
    if (isRetired) return retiredAmount();
    return normalPayAmount();
}
```

D6. Replace Conditional with Polymorphism (Koşulu Polimorfizm ile Değiştirme)

Nedir: Koşul dallarıyla eşleşen alt sınıflar oluşturun ve polimorfik metot çağrıları kullanın.

Problem: Nesne türüne veya özelliklerine bağlı olarak çeşitli eylemler gerçekleştiren bir koşulunuz var.

Çözüm: Her koşul dalıyla eşleşen alt sınıflar oluşturun. Polimorfizm aracılığıyla paylaşılan bir metot oluşturun ve her daldan ilgili kodu eşleşen alt sınıfa taşıyın. Orijinal metot soyut hale gelir.

Ne zaman kullanılır: Koşullar nesne türlerine bağlı olduğunda; aynı koşul yapısı tekrarlandığında.

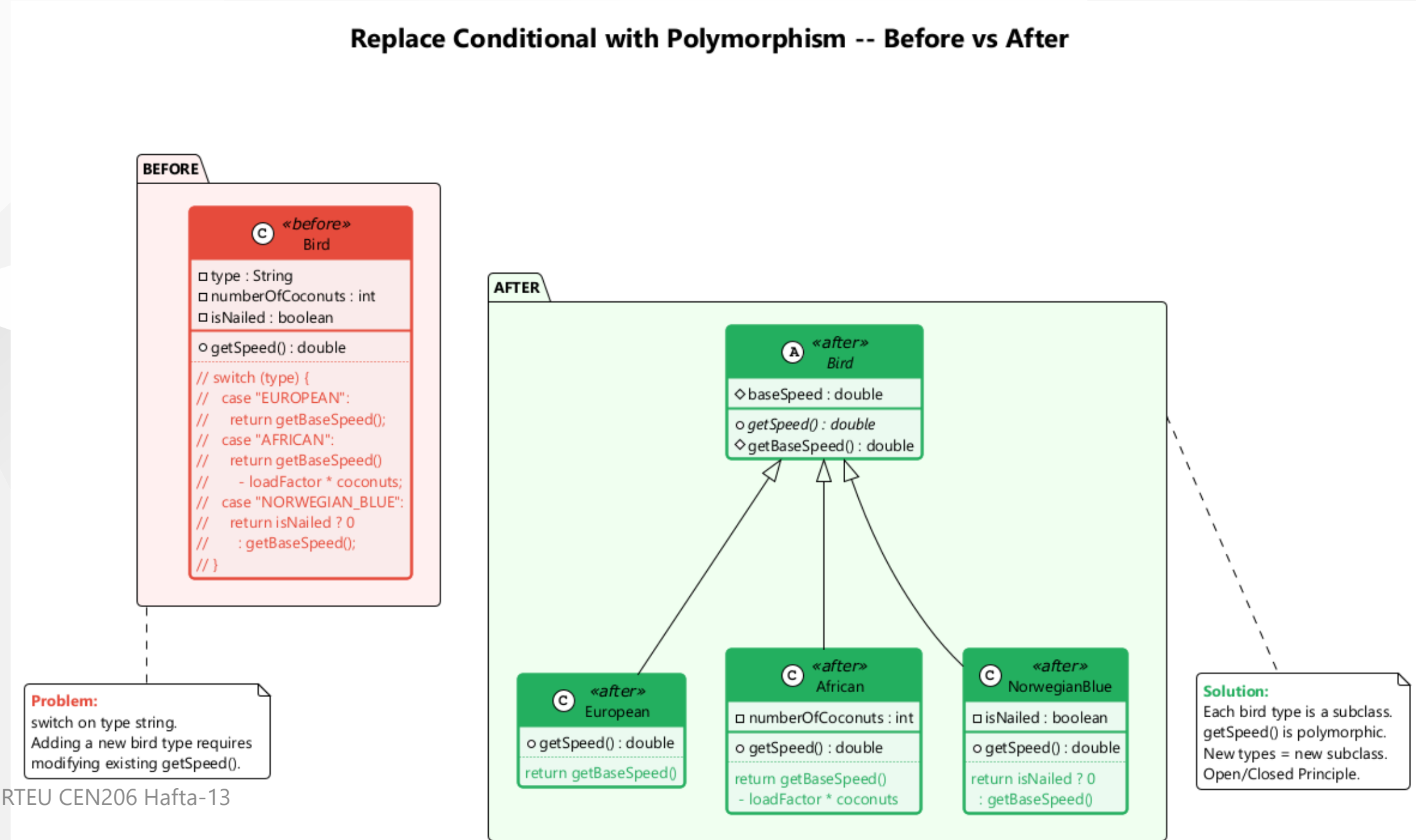
Ne zaman KULLANILMAZ: Koşul basit ve büyüme olasılığı düşük olduğunda; alt sınıf oluşturmak aşırı olduğunda.

D6. Replace Conditional with Polymorphism -- Önce / Sonra

```
// Before
class Bird {
    String type;
    double getSpeed() {
        switch (type) {
            case "EUROPEAN":  return getBaseSpeed();
            case "AFRICAN":   return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
            case "NORWEGIAN_BLUE": return (isNailed) ? 0 : getBaseSpeed();
            default: throw new RuntimeException("Unknown type");
        }
    }
}
```

```
// After
abstract class Bird {
    abstract double getSpeed();
}
class European extends Bird {
    double getSpeed() { return getBaseSpeed(); }
}
class African extends Bird {
    double getSpeed() { return getBaseSpeed() - getLoadFactor() * numberOfCoconuts; }
}
class NorwegianBlue extends Bird {
    double getSpeed() { return (isNailed) ? 0 : getBaseSpeed(); }
}
```

Koşulu Polimorfizm ile Değiştirme - Diyagram



D7. Introduce Null Object (Null Nesne Ekleme)

Nedir: `null` döndürmek yerine, varsayılan davranış sağlayan özel bir null nesnesi döndürün.

Problem: Bazı metotlar gerçek nesneler yerine `null` döndürdüğü için, kodunuzda çok sayıda `null` kontrolü var.

Çözüm: `null` yerine, varsayılan davranış sergileyen bir null nesnesi döndürün.

Ne zaman kullanılır: Null kontrolleri kodu karmaşıklaştırdığında; null, bilinen bir varsayılan davranışla "hiçbir şey" anlamına geldiğinde.

Ne zaman KULLANILMAZ: Null gerçek bir anlama sahip olduğunda (örn. "bulunamadı" "varsayılanlarla bulundu"dan farklı işlenmelidir).

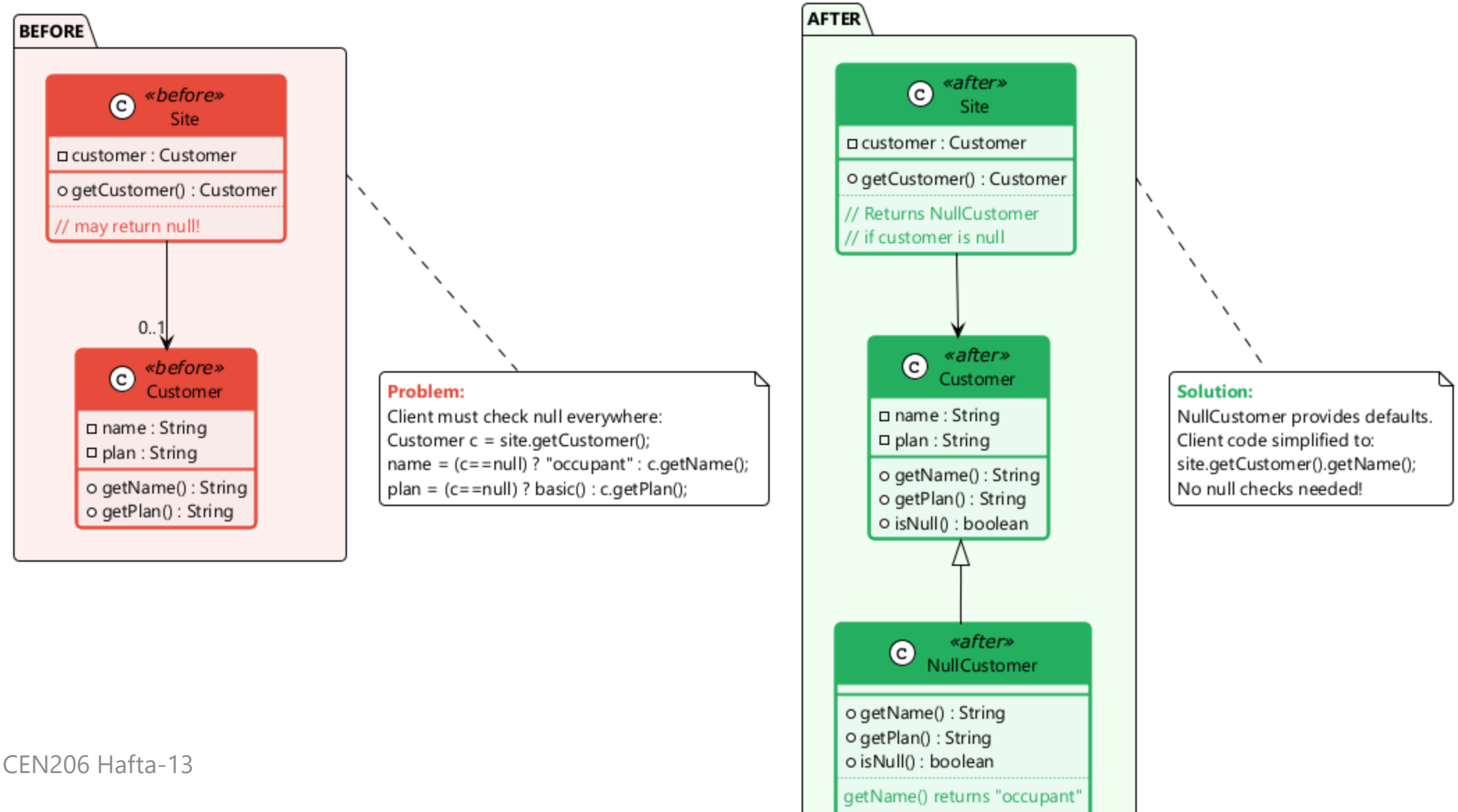
D7. Introduce Null Object -- Önce / Sonra

```
// Before
class Site {
    Customer getCustomer() {
        return customer; // may return null
    }
}
// Client:
Customer c = site.getCustomer();
String name = (c == null) ? "occupant" : c.getName();
String plan = (c == null) ? BillingPlan.basic() : c.getPlan();
```

```
// After
class NullCustomer extends Customer {
    @Override String getName() { return "occupant"; }
    @Override String getPlan() { return BillingPlan.basic(); }
    @Override boolean isNull() { return true; }
}

class Site {
    Customer getCustomer() {
        return (customer == null) ? new NullCustomer() : customer;
    }
}
// Client: no null checks needed
String name = site.getCustomer().getName();
String plan = site.getCustomer().getPlan();
```

Introduce Null Object -- Before vs After



D8. Introduce Assertion (Onaylama Ekleme)

Nedir: Program durumu hakkındaki varsayımları açık onaylama kontrolleriyle değiştirin.

Problem: Bir kod bölümü, nesnenin veya parametre değerlerinin mevcut durumu hakkında varsayım yapıyor ama bunu zorlamamış.

Çözüm: Bu varsayımları belirli onaylama kontrolleriyle değiştirin.

Ne zaman kullanılır: Gerekli ön koşulları belgelemek ve zorlamak gerektiğinde; örtük varsayımları açık hale getirmek için.

Ne zaman KULLANILMAZ: Genel API'lerde girdi doğrulama için onaylama kullanmayın (bunun yerine istisnalar kullanın). Onaylamalar programcı hataları içindir, kullanıcı hataları için değil.

D8. Introduce Assertion -- Önce / Sonra

```
// Before
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (expenseLimit != NULL_EXPENSE)
        ? expenseLimit
        : primaryProject.getMemberExpenseLimit();
}
```

```
// After
double getExpenseLimit() {
    assert (expenseLimit != NULL_EXPENSE || primaryProject != null)
        : "Must have either expense limit or primary project";

    return (expenseLimit != NULL_EXPENSE)
        ? expenseLimit
        : primaryProject.getMemberExpenseLimit();
}
```

Onaylamalar örtük varsayımları açık hale getirir ve programcı hatalarını erken yakalamaya yardımcı olur.

Modül D -- Özet

Simplifying Conditional Expressions (Koşulları Sadeleştirme): Temel Noktalar

#	Teknik	Amaç
1	Decompose Conditional (Koşulu Ayrıştırma)	Karmaşık koşul/then/else'i isimlendirilmiş metotlara çıkarma
2	Consolidate Conditional Expression	Aynı sonuca sahip koşulları birleştirme
3	Consolidate Duplicate Conditional Fragments	Aynı kodu tüm dallardan çıkarma
4	Remove Control Flag	Bayrakları break/continue/return ile değiştirme
5	Replace Nested Conditional with Guard Clauses	Derin iç içeliği erken dönüşlerle düzleştirme
6	Replace Conditional with Polymorphism	Tür değiştirme yerine alt sınıflar kullanma
7	Introduce Null Object	Varsayılan davranış nesneleriyle null kontrollerini ortadan kaldırma
8	Introduce Assertion	Örtük varsayımları açık hale getirme

Modül E: Simplifying Method Calls (Metot Çağrılarını Sadeleştirme)

Daha Temiz Arayüzler için 14 Teknik

Simplifying Method Calls (Metot Çağrılarını Sadeleştirme)

Bu teknikler metot arayüzlerini daha basit ve anlaşılır hale getirir. Metot adlandırma, parametre listeleri ve dönüş anlam bilgisini ele alırlar.

1. Rename Method (Metodu Yeniden Adlandırma)
2. Add Parameter (Parametre Ekleme)
3. Remove Parameter (Parametre Kaldırma)
4. Separate Query from Modifier (Sorguyu Değiştiriciden Ayırma)
5. Parameterize Method (Metodu Parametreleştirme)
6. Replace Parameter with Explicit Methods (Parametreyi Açık Metotlarla Değiştirme)
7. Preserve Whole Object (Nesnenin Tamamını Koruma)
8. Replace Parameter with Method Call (Parametreyi Metot Çağrısıyla Değiştirme)
9. Introduce Parameter Object (Parametre Nesnesi Ekleme)
10. Remove Setting Method (Setter Metodunu Kaldırma)
11. Hide Method (Metodu Gizleme)
12. Replace Constructor with Factory Method (Yapıcıyı Fabrika Metoduyla Değiştirme)
13. Replace Error Code with Exception (Hata Kodunu İstisna ile Değiştirme)
14. Replace Exception with Test (İstisna'yı Test ile Değiştirme)

E1. Rename Method (Metodu Yeniden Adlandırma)

Nedir: Bir metoda amacını ortaya koyan bir isim verin.

Problem: Bir metodun adı, metodun ne yaptığını açıklamıyor.

Çözüm: Metodu daha açıklayıcı bir isimle yeniden adlandırın.

Ne zaman kullanılır: Metot adı yanıltıcı, kısaltılmış veya belirsiz olduğunda.

Ne zaman KULLANILMAZ: Metot adı zaten açık ve iyi anlaşılır olduğunda.

```
// Before
class Customer {
    String getsnm() { return surname; }
}

// After
class Customer {
    String getSurname() { return surname; }
}
```

E2. Add Parameter (Parametre Ekleme)

Nedir: Ek veriye ihtiyacı olan bir metoda yeni bir parametre ekleyin.

Problem: Bir metot belirli eylemleri gerçekleştirmek için yeterli veriye sahip değil.

Çözüm: Gerekli veriyi geçirmek için yeni bir parametre oluşturun.

Ne zaman kullanılır: Bir metot işini tamamlamak için ek bilgiye ihtiyaç duyduğunda.

Ne zaman KULLANILMAZ: Veri mevcut bir parametreden veya nesnenin kendisinden elde edilebildiğinde. Çok fazla parametre ekliyorsanız Introduce Parameter Object düşünün.

```
// Before
class Customer {
    Contact getContact() { ... }
}

// After
class Customer {
    Contact getContact(Date date) { ... }
}
```

E3. Remove Parameter (Parametre Kaldırma)

Nedir: Kullanılmayan bir parametreyi metottan kaldırın.

Problem: Bir parametre metodun gövdesinde kullanılmıyor.

Çözüm: Kullanılmayan parametreyi kaldırın.

Ne zaman kullanılır: Yeniden yapılandırmadan sonra bir parametre artık gerekli olmadığında.

Ne zaman KULLANILMAZ: Parametre alt sınıflardaki geçersiz kılan metotlar tarafından kullanıldığında, temel sınıf kullanmasa bile.

```
// Before
class Customer {
    Contact getContact(Date date) {
        // date is never used
        return contact;
    }
}

// After
class Customer {
    Contact getContact() {
        return contact;
    }
}
```

E4. Separate Query from Modifier (Sorguyu Değiştiriciden Ayırma)

Nedir: Değer döndüren ve nesne durumunu değiştiren bir metodu iki metoda bölün.

Problem: Bir değer döndüren ama aynı zamanda nesne içinde bir şey değiştiren (yan etki) bir metodunuz var. Çağıranlar yan etkiyi beklemeyebilir.

Çözüm: Metodu iki ayrı metoda bölün. Biri değeri döndürür (sorgu), diğeri değişikliği yapar (komut).

Ne zaman kullanılır: Sorguları ve komutları karıştırmak beklenmeyen yan etkilere yol açtığında.

Ne zaman KULLANILMAZ: Birleşik işlem atomik olduğunda ve bölmek eşzamanlılık sorunlarına neden olacağında.

E4. Separate Query from Modifier -- Önce / Sonra

```
// Before
class SecurityService {
    String foundMiscreant(String[] people) {
        for (String person : people) {
            if (person.equals("Don")) {
                sendAlert();
                return "Don";
            }
            if (person.equals("John")) {
                sendAlert();
                return "John";
            }
        }
        return "";
    }
}
```

```
// After
class SecurityService {
    String foundMiscreant(String[] people) {
        for (String person : people) {
            if (person.equals("Don") || person.equals("John")) {
                return person;
            }
        }
        return "";
    }

    void alertIfMiscreant(String[] people) {
        if (!foundMiscreant(people).isEmpty()) {
            sendAlert();
        }
    }
}
```

E5. Parameterize Method (Metodu Parametreleştirme)

Nedir: Yalnızca iç değerlerde farklılaşan benzer metotları bir parametre ekleyerek birleştirin.

Problem: Yalnızca iç değerleri, sayıları veya işlemleri farklı olan benzer eylemler gerçekleştiren birden fazla metot.

Çözüm: Değişen değerler için bir parametre kullanarak bu metotları birleştirin.

Ne zaman kullanılır: Yinelenen metotlar yalnızca sabitler veya benzer değerlerde farklılaştığında.

Ne zaman KULLANILMAZ: Metotların farklılaşan değerlerin ötesinde önemli ölçüde farklı mantığı olduğunda.

E5. Parameterize Method -- Önce / Sonra

```
// Before
class Employee {
    void tenPercentRaise() {
        salary *= 1.1;
    }

    void fivePercentRaise() {
        salary *= 1.05;
    }
}
```

```
// After
class Employee {
    void raise(double factor) {
        salary *= (1 + factor);
    }
}
// Usage: employee.raise(0.10); employee.raise(0.05);
```

E6. Replace Parameter with Explicit Methods (Parametreyi Açık Metotlarla Değiştirme)

Nedir: Davranışı seçmek için parametre kullanan bir metodu ayrı metotlara bölün.

Problem: Bir metot parçalara bölünmüştür ve her biri bir parametrenin değerine bağlı olarak çalışır (örn. `setValue("height", 10)`).

Çözüm: Metodun ayrı parçalarını kendi metotlarına çıkarın.

Ne zaman kullanılır: Bir parametre yalnızca farklı davranışlar arasında seçim yapmak için kullanıldığında.

Ne zaman KULLANILMAZ: Parametre değerleri derleme zamanında bilinmediğinde.

```
// Before
void setValue(String name, int value) {
    if (name.equals("height")) { height = value; }
    else if (name.equals("width")) { width = value; }
}

// After
void setHeight(int value) { height = value; }
void setWidth(int value) { width = value; }
```

E7. Preserve Whole Object (Nesnenin Tamamını Koruma)

Nedir: Birkaç değeri çıkarıp tek tek geçirmek yerine nesnenin tamamını geçin.

Problem: Bir nesneden birkaç değer alıp bunları parametre olarak bir metoda geçiyorsunuz.

Çözüm: Bunun yerine nesnenin tamamını geçmeyi deneyin.

Ne zaman kullanılır: Birkaç parametre aynı nesneden geldiğinde; gelecekteki değişiklikler aynı nesneden daha fazla veri gerektirebileceğinde.

Ne zaman KULLANILMAZ: Çağrılan metot nesnenin tamamına bağımlı olmaması gerektiğinde (bağımlılıktan kaçınmak için); yalnızca bir değer gerektiğinde.

E7. Preserve Whole Object -- Önce / Sonra

```
// Before
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);
```

```
// After
boolean withinPlan = plan.withinRange(daysTempRange);

// In HeatingPlan class:
class HeatingPlan {
    boolean withinRange(TempRange daysTempRange) {
        return daysTempRange.getLow() >= rangeLow
            && daysTempRange.getHigh() <= rangeHigh;
    }
}
```

Nesnenin tamamını geçmek parametre sayısını azaltır ve metodu gelecekteki değişikliklere karşı daha dayanıklı kılar.

Değiştirme)

Nedir: Hesaplanmış bir değeri parametre olarak geçirmek yerine, metodun kendisinin hesaplamasına izin verin.

Problem: Bir sorgu metodu çağırıp sonuçlarını başka bir metoda parametre olarak geçiyorsunuz, oysa o metot sorguyu doğrudan çağırabilir.

Çözüm: Parametreyi kaldırın ve alıcının metodu doğrudan çağırmasına izin verin.

Ne zaman kullanılır: Çağrılan metot değeri kendi başına kolayca elde edebildiğinde.

Ne zaman KULLANILMAZ: Çağrılan metot değerin kaynağını bilmemesi veya buna bağımlı olmaması gerektiğinde.

```
// Before
int basePrice = quantity * itemPrice;
double discount = getDiscount();
double finalPrice = discountedPrice(basePrice, discount);

// After
double finalPrice = discountedPrice(basePrice);

double discountedPrice(int basePrice) {
    return basePrice - getDiscount();
```

E9. Introduce Parameter Object (Parametre Nesnesi Ekleme)

Nedir: Tekrarlanan bir parametre grubunu bir nesneyle değiştirin.

Problem: Metotlarınız tekrarlanan bir parametre grubu içeriyor (örn. başlangıç ve bitiş tarihleri birden fazla metotta görünüyor).

Çözüm: Bu parametreleri bir nesneyle değiştirin.

Ne zaman kullanılır: Aynı parametre grubu birden fazla metotta birlikte dolaştığında.

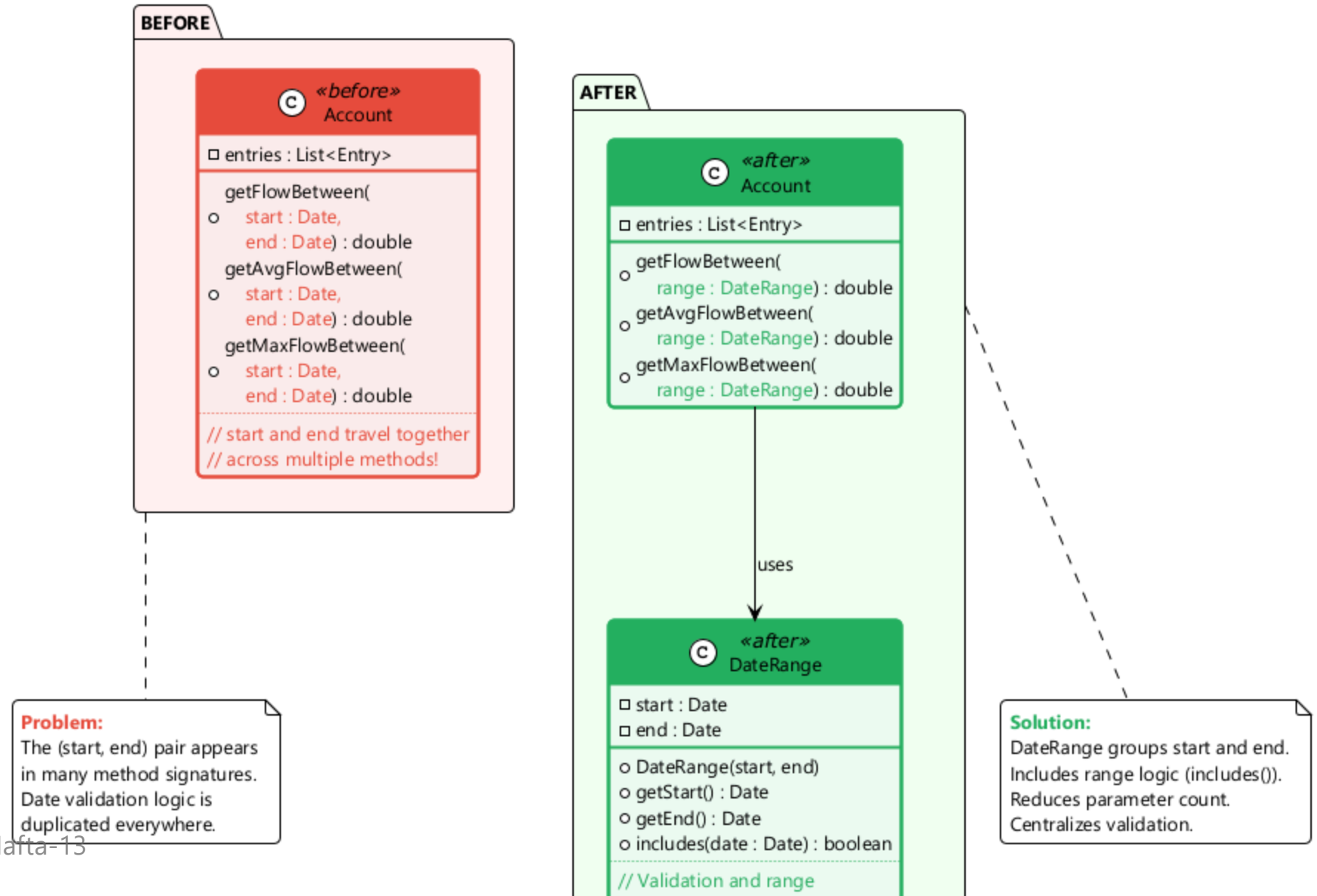
Ne zaman KULLANILMAZ: Parametre grubu tesadüfi olup alan anlamı taşımadığında.

```
// Before
class Account {
    double getFlowBetween(Date start, Date end) {
        double result = 0;
        for (Entry e : entries) {
            if (e.getDate().after(start) && e.getDate().before(end)) {
                result += e.getValue();
            }
        }
        return result;
    }
}
```

```
// After
class DateRange {
    private final Date start;
    private final Date end;
    DateRange(Date start, Date end) { this.start = start; this.end = end; }
    Date getStart() { return start; }
    Date getEnd() { return end; }
    boolean includes(Date date) { return date.after(start) && date.before(end); }
}

class Account {
    double getFlowBetween(DateRange range) {
        double result = 0;
        for (Entry e : entries) {
            if (range.includes(e.getDate())) {
                result += e.getValue();
            }
        }
        return result;
    }
}
```

Introduce Parameter Object -- Before vs After



E10. Remove Setting Method (Setter Metodunu Kaldırma)

Nedir: Yalnızca oluşturma zamanında ayarlanması gereken bir alan için setter metodunu kaldırın.

Problem: Bir alanın değeri yalnızca oluşturulduğunda ayarlanmalı ve bundan sonra hiçbir zaman değişmemeli.

Çözüm: Setter metodunu kaldırın. Alanı yalnızca yapıcıda ayarlayın.

Ne zaman kullanılır: Bir alan oluşturulduktan sonra değişmez olması gerektiğinde.

Ne zaman KULLANILMAZ: Alanın oluşturulduktan sonra meşru olarak değiştirilmesi gerektiğinde.

```
// Before
class Employee {
    private String id;
    void setId(String id) { this.id = id; }
    String getId() { return id; }
}

// After
class Employee {
    private final String id;
    Employee(String id) { this.id = id; }
    String getId() { return id; }
}
```

Nedir: Başka sınıflar tarafından kullanılmayan bir metodu private veya protected yapın.

Problem: Bir metot başka hiçbir sınıf tarafından kullanılmıyor veya yalnızca kendi sınıf hiyerarşisi içinde kullanılıyor.

Çözüm: Metodu private (veya protected) yapın.

Ne zaman kullanılır: Bir metodun görünürlüğü gerekenden geniş olduğunda.

Ne zaman KULLANILMAZ: Metot, dış istemcilerin bağımlı olduğu genel bir API'nin parçası olduğunda.

```
// Before
class Employee {
    public int calculateBonusPercentage() {
        // only used internally
        return yearsOfService > 10 ? 15 : 5;
    }
    public int getBonus() {
        return salary * calculateBonusPercentage() / 100;
    }
}

// After
class Employee {
    private int calculateBonusPercentage() {
        return yearsOfService > 10 ? 15 : 5;
    }
    public int getBonus() {
```

E12. Replace Constructor with Factory Method (Yapıcıyı Fabrika Metoduyla Değiştirme)

Nedir: Bir yapıcıyı fabrika metodu ile değiştirin.

Problem: Nesne alanlarında parametre değerlerini ayarlamaktan daha fazlasını yapan karmaşık bir yapıcınız var. Veya farklı alt sınıf örnekleri döndürmeniz gerekiyor.

Çözüm: Bir fabrika metodu oluşturun ve yapıcı yerine onu kullanın.

Ne zaman kullanılır: Yapıcılar basit atamanın ötesinde mantık içerdiğinde; farklı türler döndürmeniz gerektiğinde.

Ne zaman KULLANILMAZ: Oluşturma basit ve doğrudan olduğunda.

E12. Replace Constructor with Factory Method -- Önce / Sonra

```
// Before
class Employee {
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;
    private int type;

    Employee(int type) {
        this.type = type;
    }
}
// Usage: Employee e = new Employee(Employee.ENGINEER);
```

```
// After
abstract class Employee {
    static Employee create(int type) {
        switch (type) {
            case ENGINEER:  return new Engineer();
            case SALESMAN:  return new Salesman();
            case MANAGER:   return new Manager();
            default: throw new IllegalArgumentException("Invalid type: " + type);
        }
    }
}
// Usage: Employee e = Employee.create(Employee.ENGINEER);
```

E13. Replace Error Code with Exception (Hata Kodunu İstisna ile Değiştirme)

Nedir: Özel bir hata kodu değeri döndürmek yerine istisna fırlatın.

Problem: Bir metot, bir hatayı belirten özel bir değer döndürüyor (-1 veya null gibi).

Çözüm: Bunun yerine bir istisna fırlatın.

Ne zaman kullanılır: Hata koşulları normal akışı kesmeli ve çağıranlar tarafından açıkça işlenmelidir.

Ne zaman KULLANILMAZ: "Hata" aslında normal, beklenen bir sonuç olduğunda (dönüş değeri veya Optional kullanın).

```
// Before
int withdraw(int amount) {
    if (amount > balance) return -1;
    balance -= amount;
    return 0;
}

// After
void withdraw(int amount) {
    if (amount > balance) {
        throw new InsufficientFundsException();
    }
    balance -= amount;
}
```

E14. Replace Exception with Test (İstisna'yı Test ile Değiştirme)

Nedir: Bir istisnayı koşullu bir testle değiştirin.

Problem: Basit bir testin işe yaracağı bir yerde istisna fırlatıyorsunuz. İstisnalar beklenen kontrol akışı için kullanılıyor.

Çözüm: İstisna'yı koşul testiyle değiştirin.

Ne zaman kullanılır: İstisnalar tahmin edilebilir, istisnai olmayan durumlar için kullanıldığında.

Ne zaman KULLANILMAZ: Koşul gerçekten istisnai olup önceden kontrol edilemediğinde.

```
// Before
double getValueForPeriod(int periodNumber) {
    try {
        return values[periodNumber];
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}

// After
double getValueForPeriod(int periodNumber) {
    if (periodNumber >= values.length) return 0;
    return values[periodNumber];
}
```

Modül E -- Özet

Simplifying Method Calls (Metot Çağrılarını Sadeleştirme): Temel Noktalar

#	Teknik	Amaç
1	Rename Method (Yeniden Adlandırma)	Metotlara amaçlarını ortaya koyan isimler verme
2	Add Parameter (Parametre Ekleme)	Metoda eksik veriyi sağlama
3	Remove Parameter (Parametre Kaldırma)	Kullanılmayan parametreleri silme
4	Separate Query from Modifier	Yan etkili metotları sorgu + komut olarak bölme
5	Parameterize Method	Yinelenen metotları parametre ile birleştirme
6	Replace Parameter with Explicit Methods	Bayrak parametrelerini isimlendirilmiş metotlarla değiştirme
7	Preserve Whole Object	Çıkarılmış değerler yerine nesneyi geçme
8	Replace Parameter with Method Call	Alıcının değeri hesaplamasına izin verme
9	Introduce Parameter Object	Tekrarlanan parametreleri nesne olarak gruplama
10	Remove Setting Method	Değişmezliği zorlama
11	Hide Method	Genel yüzey alanını azaltma

Modül F: Dealing with Generalization (Genelleştirme ile Başa Çıkma)

Sınıf Hiyerarşilerini Yönetmek için 12 Teknik

Modül F Ana Hatları

Dealing with Generalization (Genelleştirme ile Başa Çıkma)

Soyutlamanın kendi yeniden yapılandırma teknikleri grubu vardır; bunlar öncelikle işlevleri sınıf kalıtım hiyerarşisi boyunca taşıma, yeni sınıflar ve arayüzler oluşturma ve kalıtımı delegasyon ile değiştirme ve tam tersini içerir.

1. Pull Up Field (Alanı Yukarı Çekme)
2. Pull Up Method (Metodu Yukarı Çekme)
3. Pull Up Constructor Body (Yapıcı Gövdesini Yukarı Çekme)
4. Push Down Method (Metodu Aşağı İtme)
5. Push Down Field (Alanı Aşağı İtme)
6. Extract Subclass (Alt Sınıf Çıkarma)
7. Extract Superclass (Üst Sınıf Çıkarma)
8. Extract Interface (Arayüz Çıkarma)
9. Collapse Hierarchy (Hiyerarşiyi Daraltma)
10. Form Template Method (Şablon Metot Oluşturma)
11. Replace Inheritance with Delegation (Kalıtımı Delegasyon ile Değiştirme)
12. Replace Delegation with Inheritance (Delegasyonu Kalıtım ile Değiştirme)

F1. Pull Up Field (Alanı Yukarı Çekme)

Nedir: Bir alanı alt sınıflardan üst sınıfa taşıyın.

Problem: İki veya daha fazla alt sınıf aynı alana sahip.

Çözüm: Alanı alt sınıflardan kaldırın ve üst sınıfa taşıyın.

Ne zaman kullanılır: Birden fazla alt sınıf aynı alanı bildirdiğinde.

Ne zaman KULLANILMAZ: Alanlar yalnızca benzer görünüp her alt sınıfta farklı kavramları temsil ettiğinde.

```
// Before
class Salesman extends Employee {
    private String name;
}
class Engineer extends Employee {
    private String name;
}

// After
class Employee {
    protected String name;
}
class Salesman extends Employee { }
class Engineer extends Employee { }
```

F2. Pull Up Method (Metodu Yukarı Çekme)

Nedir: Aynı metotları alt sınıflardan üst sınıfa taşıyın.

Problem: Alt sınıflarınız benzer iş yapan metotlara sahip.

Çözüm: Metotları aynı hale getirin ve ardından ilgili üst sınıfa taşıyın.

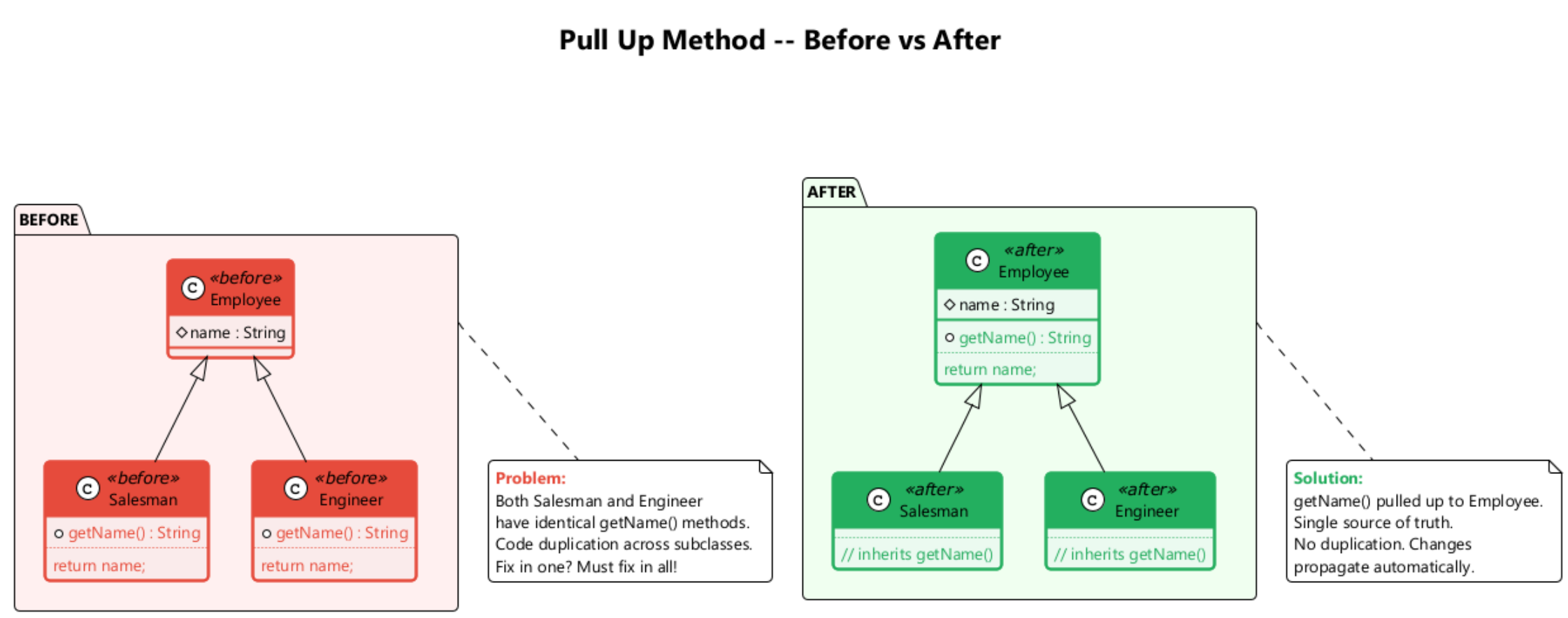
Ne zaman kullanılır: Alt sınıflar aynı veya neredeyse aynı metotları uyguladığında.

Ne zaman KULLANILMAZ: Metotlar benzer adlara rağmen gerçekten farklı olduğunda.

```
// Before
class Salesman extends Employee {
    String getName() { return name; }
}
class Engineer extends Employee {
    String getName() { return name; }
}

// After
class Employee {
    String getName() { return name; }
}
class Salesman extends Employee { }
class Engineer extends Employee { }
```

Metodu Yukarı Çekme - Önce/Sonra Diyagramı



F3. Pull Up Constructor Body (Yapıcı Gövdesini Yukarı Çekme)

Nedir: Ortak yapıcı kodunu üst sınıf yapıcısına taşıyın.

Problem: Alt sınıflarınızın çoğunlukla aynı olan kodlarla yapıcıları var.

Çözüm: Bir üst sınıf yapıcısı oluşturun ve paylaşılan kodu içine taşıyın. Alt sınıf yapıcılarında üst sınıf yapıcısını çağırın.

Ne zaman kullanılır: Yapıcı mantığı hiyerarşi genelinde tekrarlandığında.

Ne zaman KULLANILMAZ: Yapıcılar arasında yalnızca yüzeysel bir benzerlik olduğunda.

```
// Before
class Manager extends Employee {
    private int grade;
    Manager(String name, String id, int grade) {
        this.name = name;
        this.id = id;
        this.grade = grade;
    }
}
class Engineer extends Employee {
    Engineer(String name, String id) {
        this.name = name;
        this.id = id;
    }
}
```

```
// After
class Employee {
    protected String name;
    protected String id;
    Employee(String name, String id) {
        this.name = name;
        this.id = id;
    }
}
class Manager extends Employee {
    private int grade;
    Manager(String name, String id, int grade) {
        super(name, id);
        this.grade = grade;
    }
}
class Engineer extends Employee {
    Engineer(String name, String id) {
        super(name, id);
```

F4: Push Down Method (Metodu Aşağı İtme)

Nedir: Bir metodu üst sınıftan, ona ihtiyacı olan belirli alt sınıflara taşıyın.

Problem: Üst sınıfta uygulanan davranış yalnızca bir (veya birkaç) alt sınıf tarafından kullanılıyor.

Çözüm: Metodu bu belirli alt sınıflara taşıyın.

Ne zaman kullanılır: Bir metot yalnızca alt sınıfların bir alt kümesiyle ilgili olduğunda.

Ne zaman KULLANILMAZ: Metot çoğu veya tüm alt sınıflar tarafından kullanıldığında.

```
// Before
class Employee {
    int getQuota() { ... } // only relevant for Salesman
}
class Engineer extends Employee { }
class Salesman extends Employee { }

// After
class Employee { }
class Engineer extends Employee { }
class Salesman extends Employee {
    int getQuota() { ... }
}
```

F5: Push Down Field (Alanı Aşağı İtme)

Nedir: Bir alanı üst sınıftan, ona ihtiyacı olan belirli alt sınıflara taşıyın.

Problem: Bir alan yalnızca birkaç alt sınıfta kullanılıyor, tüm hiyerarşi genelinde değil.

Çözüm: Alanı bu belirli alt sınıflara taşıyın.

Ne zaman kullanılır: Tüm alt sınıflar aynı veriye ihtiyaç duymadığında.

Ne zaman KULLANILMAZ: Alan çoğu alt sınıf tarafından kullanıldığında.

```
// Before
class Employee {
    protected int quota; // only used by Salesman
}
class Engineer extends Employee { }
class Salesman extends Employee { }

// After
class Employee { }
class Engineer extends Employee { }
class Salesman extends Employee {
    private int quota;
}
```

F6. Extract Subclass (Alt Sınıf Çıkarma)

Nedir: Yalnızca belirli durumlarda kullanılan özellikler için bir alt sınıf oluşturun.

Problem: Bir sınıfın yalnızca belirli durumlarda kullanılan özellikleri var.

Çözüm: Bu özelliklerden bir alt sınıf oluşturun.

Ne zaman kullanılır: Bir sınıfın yalnızca bazı örnekler için geçerli davranışı olduğunda; örnek davranışını farklılaştıran tür kodları veya koşullar gördüğünüzde.

Ne zaman KULLANILMAZ: Varyasyon delegasyon veya strateji ile daha iyi yönetildiğinde.

F6. Extract Subclass -- Önce / Sonra

```
// Before
class JobItem {
    private int unitPrice;
    private int quantity;
    private boolean isLabor;
    private Employee employee;

    int getTotalPrice() {
        return getUnitPrice() * quantity;
    }

    int getUnitPrice() {
        return isLabor ? employee.getRate() : unitPrice;
    }
}
```

```
// After
class JobItem {
    protected int unitPrice;
    protected int quantity;

    int getTotalPrice() { return getUnitPrice() * quantity; }
    int getUnitPrice() { return unitPrice; }
}

class LaborItem extends JobItem {
    private Employee employee;

    @Override
    int getUnitPrice() { return employee.getRate(); }
}
```

F7. Extract Superclass (Üst Sınıf Çıkarma)

Nedir: Ortak alanlar ve metotlara sahip iki sınıf için paylaşılan bir üst sınıf oluşturun.

Problem: Ortak alanlar ve metotlara sahip iki sınıfınız var.

Çözüm: Onlar için bir üst sınıf oluşturun ve tüm ortak alanları ve metotları üst sınıfa taşıyın.

Ne zaman kullanılır: Birden fazla sınıf önemli işlevleri paylaştığında.

Ne zaman KULLANILMAZ: Benzerlik tesadüfi olup gerçek bir "şu-dur" ilişkisini temsil etmediğinde.

F7. Extract Superclass -- Önce / Sonra

```
// Before
class Department {
    String name;
    int getAnnualCost() { /* sum of staff costs */ }
    int getHeadCount() { /* number of staff */ }
    String getName() { return name; }
}
class Employee {
    String name;
    int getAnnualCost() { return salary; }
    int getId() { return id; }
    String getName() { return name; }
}
```

```
// After
abstract class Party {
    protected String name;
    String getName() { return name; }
    abstract int getAnnualCost();
}
class Department extends Party {
    int getAnnualCost() { /* sum of staff costs */ }
    int getHeadCount() { /* number of staff */ }
}
class Employee extends Party {
    int getAnnualCost() { return salary; }
    int getId() { return id; }
}
```

F8. Extract Interface (Arayüz Çıkarma)

Nedir: Bir sınıf arayüzünün paylaşılan bölümünü kendi arayüzüne taşıyın.

Problem: Birden fazla istemci bir sınıfın aynı arayüz alt kümesini kullanıyor. Veya iki sınıfın arayüzlerinin ortak bir bölümü var.

Çözüm: Bu aynı bölümü kendi arayüzüne taşıyın.

Ne zaman kullanılır: İstemcileri belirli uygulamalardan ayırmak istediğinizde; sınıflar bir davranışsal sözleşmeyi paylaştığında.

Ne zaman KULLANILMAZ: Arayüz yalnızca bir metot içereceğinde (soyutlamaya değmeyebilir).

F8. Extract Interface -- Önce / Sonra

```
// Before
class Employee {
    int getRate() { return salary / 12; }
    boolean hasSpecialSkill() { ... }
    String getName() { return name; }
    String getDepartment() { return department; }
}
// Client only cares about getRate() and hasSpecialSkill()
```

```
// After
interface Billable {
    int getRate();
    boolean hasSpecialSkill();
}

class Employee implements Billable {
    public int getRate() { return salary / 12; }
    public boolean hasSpecialSkill() { ... }
    String getName() { return name; }
    String getDepartment() { return department; }
}
// Client depends on Billable interface, not Employee
```

F9: Collapse Hierarchy (Hiyerarşiyi Daraltma)

Nedir: Pratik olarak aynı olan bir üst sınıf ve alt sınıfı birleştirin.

Problem: Bir alt sınıfın üst sınıfıyla pratik olarak aynı olduğu bir sınıf hiyerarşiniz var.

Çözüm: Alt sınıfı ve üst sınıfı birleştirin.

Ne zaman kullanılır: Hiyerarşi artık değer katmadığında; alt sınıfın ebeveyninden önemli farklılıkları olmadığında.

Ne zaman KULLANILMAZ: Sınıflar arasındaki ayrım gelecekteki genişletilebilirlik için anlamlı olduğunda.

```
// Before
class Employee { ... }
class Salesman extends Employee {
    // no meaningful additions
}

// After -- merge Salesman into Employee
class Employee {
    // all functionality here
}
```

F10. Form Template Method (Şablon Metot Oluşturma)

Nedir: Paylaşılan algoritma yapısını üst sınıfa taşıyın, değişen adımları alt sınıflarda tutun.

Problem: Alt sınıflarınız aynı sırada benzer adımlar içeren algoritmalar uyguluyor, ancak bireysel adımlar farklı.

Çözüm: Algoritma yapısını bir üst sınıf metoduna (şablon metot) taşıyın ve değişen adımları alt sınıflarda soyut metotlar olarak tutun.

Ne zaman kullanılır: Alt sınıflar bir algoritma yapısını paylaşıp ayrıntılarda farklılaştığında.

Ne zaman KULLANILMAZ: Algoritmalar temelden farklı yapılara sahip olduğunda.

P10: Form Template Method -- Once / Sonra

```
// Before
class TextStatement {
    String value(Customer c) {
        String result = "Statement for " + c.getName() + "\n";
        for (Rental r : c.getRentals()) {
            result += r.getMovie().getTitle() + ": " + r.getCharge() + "\n";
        }
        result += "Total: " + c.getTotalCharge();
        return result;
    }
}
class HtmlStatement {
    String value(Customer c) {
        String result = "<h1>Statement for " + c.getName() + "</h1>";
        for (Rental r : c.getRentals()) {
            result += "<p>" + r.getMovie().getTitle() + ": " + r.getCharge() + "</p>";
        }
        result += "<p>Total: " + c.getTotalCharge() + "</p>";
        return result;
    }
}
```

```
// After -- Template Method pattern
abstract class Statement {
    String value(Customer c) {  // template method
        String result = headerString(c);
        for (Rental r : c.getRentals()) {
            result += rentalString(r);
        }
        result += footerString(c);
        return result;
    }
    abstract String headerString(Customer c);
    abstract String rentalString(Rental r);
    abstract String footerString(Customer c);
}
```

F11. Replace Inheritance with Delegation (Kalıtımı Delegasyon ile Değiştirme)

Nedir: Üst sınıf için bir alan oluşturun, metotları ona devrdin ve kalıtımı kaldırın.

Problem: Bir alt sınıf, üst sınıfının metotlarının yalnızca bir bölümünü kullanıyor (veya üst sınıfın verisini miras almak mümkün değil). Kalıtım gerçek bir "şu-dur" ilişkisini temsil etmiyor.

Çözüm: Bir alan oluşturun ve içine bir üst sınıf nesnesi yerleştirin, metotları üst sınıf nesnesine devredin ve kalıtımdan kurtulun.

Ne zaman kullanılır: Bir alt sınıf Liskov Yerine Geçme Prensibini ihlal ettiğinde; yalnızca bazı miras alınan metotları kullandığında.

Ne zaman KULLANILMAZ: Kalıtım gerçekten bir "şu-dur" ilişkisini temsil ettiğinde.

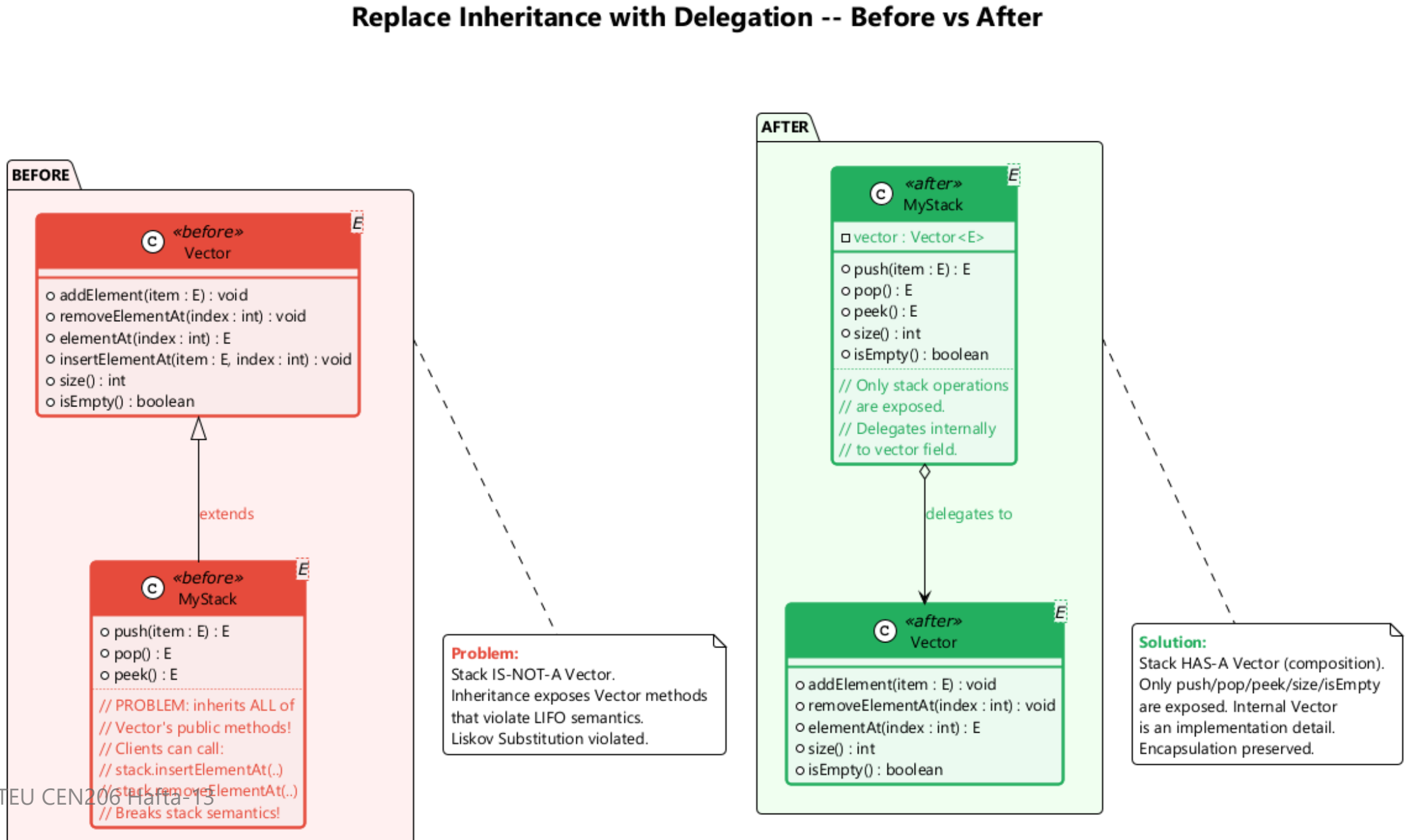
FTI: Replace Inheritance with Delegation -- Once / Sonra

```
// Before -- MyStack inherits from Vector but is not a true Vector
class MyStack<E> extends Vector<E> {
    public E push(E item) {
        addElement(item);
        return item;
    }
    public E pop() {
        E obj = peek();
        removeElementAt(size() - 1);
        return obj;
    }
    public E peek() {
        return elementAt(size() - 1);
    }
    // Problem: clients can call insertElementAt(), removeElementAt() etc.
}
```

```
// After -- delegation instead of inheritance
class MyStack<E> {
    private Vector<E> vector = new Vector<>();

    public E push(E item) {
        vector.addElement(item);
        return item;
    }
    public E pop() {
        E obj = peek();
        vector.removeElementAt(vector.size() - 1);
        return obj;
    }
    public E peek() {
        return vector.elementAt(vector.size() - 1);
    }
    public int size() { return vector.size(); }
    public boolean isEmpty() { return vector.isEmpty(); }
}
```

Kalıtımı Delegasyon ile Değiştirme - Diyagram



F12. Replace Delegation with Inheritance (Delegasyonu Kalıtım ile Değiştirme)

Nedir: Delege eden sınıfı, temsilciden miras almasını sağlayarak kalıp delegasyonu ortadan kaldırın.

Problem: Bir sınıf, başka bir sınıfın tüm metotlarına delege eden birçok basit metot içeriyor.

Çözüm: Sınıfı temsilcinin miras alıcısı yapın, bu da delege eden metotları gereksiz kılar.

Ne zaman kullanılır: Sınıf metotların çoğunu başka bir sınıfa devrettiğinde ve ilişki gerçekten "şu-dur" olduğunda.

Ne zaman KULLANILMAZ: Temsilci birden fazla sınıf tarafından kullanıldığında (paylaşılan durum); temsilci değişken olup yalnızca kısmi davranış istediğinizde.

F12. Replace Delegation with Inheritance -- Önce / Sonra

```
// Before -- excessive delegation
class Employee {
    private Person person = new Person();

    String getName() { return person.getName(); }
    void setName(String name) { person.setName(name); }
    String getAddress() { return person.getAddress(); }
    void setAddress(String addr) { person.setAddress(addr); }
    String getPhone() { return person.getPhone(); }
    void setPhone(String phone) { person.setPhone(phone); }
    // ... many more delegations
}
```

```
// After -- inheritance eliminates boilerplate
class Employee extends Person {
    // getName(), setName(), getAddress(), etc.
    // are all inherited automatically
    private String employeeId;
    private double salary;

    String getEmployeeId() { return employeeId; }
    double getSalary() { return salary; }
}
```

Modül F -- Özet

Dealing with Generalization (Genelleştirme ile Başa Çıkma): Temel Noktalar

#	Teknik	Amaç
1	Pull Up Field (Alanı Yukarı Çekme)	Alt sınıflardaki tekrarlanan alanları ortadan kaldırma
2	Pull Up Method (Metodu Yukarı Çekme)	Alt sınıflardaki tekrarlanan metotları ortadan kaldırma
3	Pull Up Constructor Body	Yapıcı başlatmayı üst sınıfta paylaşma
4	Push Down Method (Metodu Aşağı İtme)	Üst sınıf metotlarını belirli alt sınıflara taşıma
5	Push Down Field (Alanı Aşağı İtme)	Üst sınıf alanlarını belirli alt sınıflara taşıma
6	Extract Subclass (Alt Sınıf Çıkarma)	Duruma özel özellikler için alt sınıf oluşturma
7	Extract Superclass (Üst Sınıf Çıkarma)	Ortak işlevsellik için paylaşılan üst sınıf oluşturma
8	Extract Interface (Arayüz Çıkarma)	Davranışsal sözleşmeleri ayrı olarak tanımlama
9	Collapse Hierarchy (Hiyerarşiyi Daraltma)	Gereksiz sınıf seviyelerini birleştirme
10	Form Template Method (Şablon Metot)	Algoritma yapısını paylaşma, adımları çeşitlendirme
11	Replace Inheritance with Delegation	Uygunsuz "şu-dur" yerine "sahip-dir" kullanma

Tam Özet

Tüm 66 Yeniden Yapılandırma Tekniği

Tüm 66 Teknik Bir Bakışta (1/4)

Metot Oluşturma (A1--A9) & Özellikleri Taşıma (B1--B8)

#	Teknik	Kategori	Tek Satırlık Amaç
A1	Extract Method	Metot Oluşturma	Uzun metotları isimlendirilmiş parçalara bölme
A2	Inline Method	Metot Oluşturma	Önemsiz dolaylılığı kaldırma
A3	Extract Variable	Metot Oluşturma	Karmaşık alt ifadeleri isimlendirme
A4	Inline Temp	Metot Oluşturma	Engelleyen geçici değişkenleri kaldırma
A5	Replace Temp with Query	Metot Oluşturma	Geçici değişkenleri yeniden kullanılabilir sorgu metotlarına dönüştürme
A6	Split Temporary Variable	Metot Oluşturma	Her değişken için tek amaç
A7	Remove Assignments to Parameters	Metot Oluşturma	Parametre anlam bilgisini koruma
A8	Replace Method with Method Object	Metot Oluşturma	Karmaşık değişken bağımlılıklarını yönetme
A9	Substitute Algorithm	Metot Oluşturma	Daha açık algoritma ile değiştirme
B1	Move Method	Özellikleri Taşıma	Metodu verisinin yanına yerleştirme
B2	Move Field	Özellikleri Taşıma	Alanı kullanıcılarının yanına yerleştirme
B3	Extract Class	Özellikleri Taşıma	Aşırı yüklenmiş sınıfları bölme
B4	Inline Class	Özellikleri Taşıma	Yetersiz sınıfları birleştirme

Tüm 66 Teknik Bir Bakışta (2/4)

Verileri Düzenleme (C1--C15)

#	Teknik	Kategori	Tek Satırlık Amaç
C1	Self Encapsulate Field	Verileri Düzenleme	Alanlara dahili olarak getter/setter ile erişme
C2	Replace Data Value with Object	Verileri Düzenleme	İlkel türleri nesnelere yükseltime
C3	Change Value to Reference	Verileri Düzenleme	Aynı nesneleri kayıt defteri aracılığıyla paylaşma
C4	Change Reference to Value	Verileri Düzenleme	Küçük, değişmez nesneleri sadeleştirme
C5	Replace Array with Object	Verileri Düzenleme	Dizi elemanlarını türlü alanlarla isimlendirme
C6	Duplicate Observed Data	Verileri Düzenleme	GUI verisini alan verisinden ayırma
C7	Tek Yönlüden Çift Yönlüye	Verileri Düzenleme	Karşılıklı gezinme için geri işaretçi ekleme
C8	Çift Yönlüden Tek Yönlüye	Verileri Düzenleme	Gereksiz ilişki yönünü kaldırma
C9	Replace Magic Number	Verileri Düzenleme	İsimlendirilmiş sabitler kullanma
C10	Encapsulate Field	Verileri Düzenleme	Genel alanları erişimçilerle özel yapma
C11	Encapsulate Collection	Verileri Düzenleme	Salt okunur görünüm döndürme, ekleme/çıkarma sağlama
C12	Replace Type Code with Class	Verileri Düzenleme	Int kodları yerine sınıf nesneleri kullanma
C13	Type Code with Subclasses	Verileri Düzenleme	Türe bağlı davranış için polimorfizm

Tüm 66 Teknik Bir Bakışta (3/4)

Koşulları Sadeleştirme (D1--D8) & Metot Çağrılarını Sadeleştirme (E1--E14)

#	Teknik	Kategori	Tek Satırlık Amaç
D1	Decompose Conditional	Koşulları Sadeleştirme	Koşul/then/else'i metotlara çıkarma
D2	Consolidate Conditional Expression	Koşulları Sadeleştirme	Aynı sonuca sahip koşulları birleştirme
D3	Consolidate Duplicate Fragments	Koşulları Sadeleştirme	Aynı kodu dallardan çıkarma
D4	Remove Control Flag	Koşulları Sadeleştirme	break/continue/return kullanma
D5	Guard Clauses	Koşulları Sadeleştirme	Derin iç içeliği düzleştirme
D6	Conditional with Polymorphism	Koşulları Sadeleştirme	Switch yerine alt sınıflar kullanma
D7	Introduce Null Object	Koşulları Sadeleştirme	Null kontrollerini ortadan kaldırma
D8	Introduce Assertion	Koşulları Sadeleştirme	Varsayımları açık hale getirme
E1	Rename Method	Metot Çağrıları	Metot amacını ortaya koyma
E2	Add Parameter	Metot Çağrıları	Eksik veriyi sağlama
E3	Remove Parameter	Metot Çağrıları	Kullanılmayan parametreleri silme
E4	Separate Query from Modifier	Metot Çağrıları	Yan etkili metotları bölme
E5	Parameterize Method	Metot Çağrıları	Metotları parametre ile birleştirme

Tüm 66 Teknik Bir Bakışta (4/4)

Genelleştirme ile Başa Çıkma (F1--F12)

#	Teknik	Kategori	Tek Satırlık Amaç
F1	Pull Up Field	Genelleştirme	Alt sınıflardaki tekrarlanan alanları ortadan kaldırma
F2	Pull Up Method	Genelleştirme	Alt sınıflardaki tekrarlanan metotları ortadan kaldırma
F3	Pull Up Constructor Body	Genelleştirme	Yapıcı başlatmayı paylaşma
F4	Push Down Method	Genelleştirme	Metotları belirli alt sınıflara taşıma
F5	Push Down Field	Genelleştirme	Alanları belirli alt sınıflara taşıma
F6	Extract Subclass	Genelleştirme	Özel özellikler için alt sınıf oluşturma
F7	Extract Superclass	Genelleştirme	Paylaşılan üst sınıf oluşturma
F8	Extract Interface	Genelleştirme	Davranışsal sözleşmeleri tanımlama
F9	Collapse Hierarchy	Genelleştirme	Gereksiz sınıf seviyelerini birleştirme
F10	Form Template Method	Genelleştirme	Algoritma yapısını paylaşma
F11	Replace Inheritance with Delegation	Genelleştirme	Uygunsuz "şu-dur" yerine "sahip-dir" kullanma

Toplam: 6 Kategori Genelinde 66 Yeniden Yapılandırma Tekniği

Kod Kokusu ve Yeniden Yapılandırma: Şişkinlikler (Bloaters)

Kod Kokusu	Açıklama	Önerilen Yeniden Yapılandırmalar
Long Method (Uzun Metot)	Bir metot çok fazla satır kod içeriyor (genellikle 10 satırdan fazla)	Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, Replace Method with Method Object, Decompose Conditional
Large Class (Büyük Sınıf)	Bir sınıf çok fazla alan/metot/kod satırı içeriyor	Extract Class, Extract Subclass, Extract Interface, Duplicate Observed Data
Primitive Obsession (İlkel Tür Takıntısı)	Basit görevler için küçük nesneler yerine ilkel türlerin kullanılması (para birimi, aralıklar, telefon numaraları)	Replace Data Value with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Array with Object, Introduce Parameter Object
Long Parameter List (Uzun Parametre Listesi)	Bir metot için üç veya dörtten fazla parametre	Replace Parameter with Method Call, Preserve Whole Object, Introduce Parameter Object
Data Clumps (Veri Kümeleri)	Birden fazla yerde birlikte geçirilen değişken grupları	Extract Class, Introduce Parameter Object, Preserve Whole Object

Kod Kokusu ve Yeniden Yapılandırma: Nesne Yönelim Kötüye Kullanıcıları

Kod Kokusu	Açıklama	Önerilen Yeniden Yapılandırmalar
Switch Statements (Switch İfadeleri)	Karmaşık switch operatörleri veya if-else zincirleri	Decompose Conditional, Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object
Temporary Field (Geçici Alan)	Değerlerini yalnızca belirli koşullar altında alan geçici alanlar	Extract Class, Introduce Null Object
Refused Bequest (Reddedilen Miras)	Bir alt sınıf, ebeveynlerinden miras alınan metot ve özelliklerin yalnızca bir kısmını kullanıyor	Replace Inheritance with Delegation, Extract Subclass
Alternative Classes with Different Interfaces (Farklı Arayüzlere Sahip Alternatif Sınıflar)	İki sınıf aynı işlevleri gerçekleştiriyor ancak farklı metot adlarına sahip	Rename Method, Extract Superclass, Move Method

Kod Kokusu ve Yeniden Yapılandırma: Değişiklik Engelleyiciler

Kod Kokusu	Açıklama	Önerilen Yeniden Yapılandırmalar
Divergent Change (İraksak Değişiklik)	Bir sınıfta değişiklik yaptığınızda birbirleriyle ilgisiz birçok metodu değiştirmek zorunda kalıyorsunuz	Extract Class
Shotgun Surgery (Saçma Ameliyat)	Herhangi bir değişiklik yapmak birçok farklı sınıfta birçok küçük değişiklik gerektiriyor	Move Method, Move Field, Inline Class
Parallel Inheritance Hierarchies (Paralel Kalıtım Hiyerarşileri)	Bir sınıf için alt sınıf oluşturduğunuzda, başka bir sınıf için de alt sınıf oluşturmanız gerekiyor	Move Method, Move Field

Kod Kokusu ve Yeniden Yapılandırma: Vazgeçilebilirler

Kod Kokusu	Açıklama	Önerilen Yeniden Yapılandırmalar
Comments (Yorumlar)	Bir metot açıklayıcı yorumlarla dolu	Extract Variable, Extract Method, Rename Method, Introduce Assertion
Duplicate Code (Tekrarlanan Kod)	İki kod parçası neredeyse aynı görünüyor	Extract Method, Pull Up Method, Form Template Method, Extract Class
Lazy Class (Tembel Sınıf)	Bir sınıf varlığını haklı çıkarmak için çok az şey yapıyor	Inline Class, Collapse Hierarchy
Data Class (Veri Sınıfı)	Bir sınıf yalnızca alanlar ve basit erişimçiler içeriyor	Encapsulate Field, Encapsulate Collection, Move Method, Extract Method, Remove Setting Method, Hide Method
Dead Code (Ölü Kod)	Hiçbir yerde kullanılmayan bir değişken, parametre, alan, metot veya sınıf	Kullanılmayan kodu silin (Inline Class, Collapse Hierarchy, Remove Parameter)
Speculative Generality (Spekulatif Genelleme)	Gelecekte kullanılmak üzere "her ihtimale karşı" oluşturulmuş kullanılmayan sınıf, metot, alan veya parametre	Collapse Hierarchy, Inline Class, Inline Method, Remove Parameter

Kod Kokusu ve Yeniden Yapılandırma: Bağlayıcılar (Couplers)

Kod Kokusu	Açıklama	Önerilen Yeniden Yapılandırmalar
Feature Envy (Özellik Kıskançlığı)	Bir metot, kendi verisinden çok başka bir nesnenin verisine erişiyor	Move Method, Extract Method
Inappropriate Intimacy (Uygunsuz Yakınlık)	Bir sınıf başka bir sınıfın iç alanlarını ve metotlarını kullanıyor	Move Method, Move Field, Extract Class, Hide Delegate, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation
Message Chains (Mesaj Zincirleri)	`a.getB().getC().getD()` gibi bir dizi çağrı görüyorsunuz	Hide Delegate, Extract Method, Move Method
Middle Man (Aracı)	Bir sınıf başka bir sınıfa devretmekten başka hiçbir şey yapmıyor	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
Incomplete Library Class (Eksik Kütüphane Sınıfı)	Bir kütüphane sınıfı ihtiyacınız olan bir metoda sahip değil	Introduce Foreign Method, Introduce Local Extension

Yeniden Yapılandırma En İyi Uygulamalar

1. **Refactoring öncesinde her zaman testleriniz olsun** -- testsiz refactoring risklidir
2. **Küçük adımlar** -- her refactoring küçük, doğrulanabilir bir değişiklik olmalıdır
3. **Sık commit yapın** -- her başarılı refactoring adımından sonra
4. **Refactoring yaparken işlevsellik eklemeyin** -- endişeleri ayırın
5. **Burnunuzu takip edin** -- kod kokuyorsa, araştırın ve refactoring yapın
6. **IDE desteği kullanın** -- otomatik refactoring araçları manuel hataları azaltır (IntelliJ IDEA, Eclipse, VS Code)
7. **Sonucu gözden geçirin** -- yeniden yapılandırılmış kodun gerçekten daha iyi olduğundan emin olun
8. **Sürekli refactoring yapın** -- tek seferlik bir olay olarak değil, günlük geliştirmenin parçası olarak

Kaynaklar

- Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd Edition). Addison-Wesley Professional.
- RefactoringGuru. *Refactoring Techniques*. https://refactoring.guru/refactoring/techniques
- RefactoringGuru. *Composing Methods*. https://refactoring.guru/refactoring/techniques/composing-methods
- RefactoringGuru. *Moving Features between Objects*. https://refactoring.guru/refactoring/techniques/moving-features-between-objects
- RefactoringGuru. *Organizing Data*. https://refactoring.guru/refactoring/techniques/organizing-data

Kaynaklar (devamı)

- RefactoringGuru. *Simplifying Conditional Expressions*. https://refactoring.guru/refactoring/techniques/simplifying-conditional-expressions
- RefactoringGuru. *Simplifying Method Calls*. https://refactoring.guru/refactoring/techniques/simplifying-method-calls
- RefactoringGuru. *Dealing with Generalization*. https://refactoring.guru/refactoring/techniques/dealing-with-generalization
- RefactoringGuru. *Code Smells*. https://refactoring.guru/refactoring/smells
- Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley Professional.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

$$End - Of - Week - 13 - Module$$