Nesneye Yönelimli Programlama – BLM 205

Hafta 12: Refactoring

Fenerbahçe Üniversitesi

Öğretim Elemanları

Öğretim Üyesi: Dr. Vecdi Emre Levent

Ofis: 311

Email: emre.levent@fbu.edu.tr

Asistan: Arş. Gör. Uğur Özbalkan

Ofis: 307

Email: ugur.ozbalkan@fbu.edu.tr

Asistan: Arş. Gör. Ecenur Alioğulları

Ofis: 307

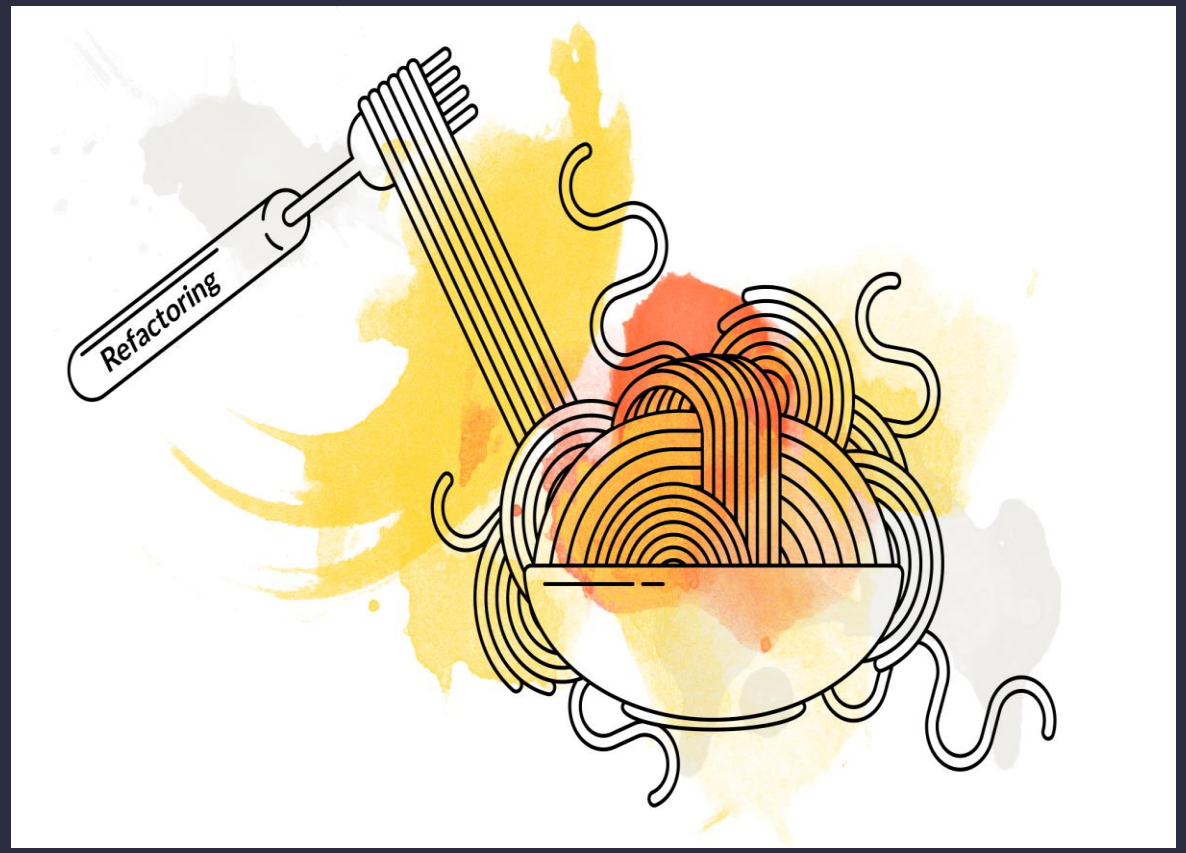
Email: ecenur.aliogullari@fbu.edu.tr

Ders Planı

- Refactoring (Yeniden Düzenleme)
  - Kirli kod
  - Temiz kod
  - Yeniden düzenleme süreci

Refactoring

Refactoring, sistematik bir biçimde koda yeni bir fonksiyonalite eklemeden, kodu daha temiz ve basit hale getirmektedir.



Refactoring

Kirli kod genellikle deneyimsizlik ve çok yakın teslim zamanları ile çalışılan projelerin sonucunda ortaya çıkmaktadır.

Temiz kod ise; okuması, anlaması ve yeniden düzenlemesi kolay olan koddur. Üzerine yeni geliştirmeler kolayca yapılabilir.

Refactoring

Temiz kod'un birkaç özelliği:

- Diğer programcılar kodu devraldıklarında, kolayca adapte olabilirler
- Aynı kod parçacığının birden çok yerde kopyası olmaz. Kod oldukça modülerdir
- Oldukça yalındır, minimum sayıda sınıf ve fonksiyon barındırır
- Yazılım test teknikleri uygulanmış ve testlerden geçmiştir

Refactoring

Sıfırdan kod geliştirmek, geliştirme sürecinde kodun yeniden kullanılabilir olmasına önem vermeden geliştirme yapmak başta bir miktar zaman kazandırabilir.

Ancak kod büyüdükçe günden güne ortaya çıkacak olan sorunların büyümesiyle, zaman kaybı daha çok olacaktır.

Refactoring

Zaman kayıplarına neden olan faktörler:

- Zaman baskısı, ürünü hemen çıkartma. Geliştirme süreci bitmeden, bazı kısımların üzerinin örtülmesi.

- Kodun yalın olmasının öneminin anlaşılamamış olması, özellikle yönetim ekibinin geliştiricilerin kodunun refactoring işlemini için ayıracağı zamanı gereksiz görmeleri. Uzun vadede çok zaman kaybetirir.

Refactoring

Zaman kayıplarına neden olan faktörler:

- Test yapmama
- Doküman hazırlamama, yeni geliştiricilere mevcut uygulamanın durumunu anlatan bir açıklama dokümanının olmaması
- Geliştirme takımı içi iletişim sorunları
- Proje yöneticisinin iş takip sorunları
- Geliştirici deneyimsizlikleri

Refactoring

Bir uygulamada, aynı veya benzer işlevi olan kod parçacıkları gerekli olduğu zaman bunların kopyalarının oluşması yerine, refactoring yapılması doğru olacaktır.

Bir başkasının kodunu devir aldığınızda, kodun nasıl çalıştığını anlamak için refactoring yaklaşımı kullanılabilir. Kodu daha basit hale getirerek anlaşılması kolaylaştırılabilir.

Refactoring

Refactoring sonunda;

- Kod daha okunabilir
- Kodun mevcut davranışı değişmemiş
- Daha önceki yapılan yazılım testlerinden başarılı geçti ise, refactoring sonrasında da aynı testlerden başarı ile geçmelidir

Refactoring

Refactoring Teknikleri

Fonksiyon üretme: Bir arada gruplanabilecek kod parçacıkları bulunuyor.

Örnek Kod Parçacığı

```
def kisiBilgileriniBastir():
    self.isimBastir()

    # Detaylari bastir
    print("Dogum yeri", dYeri)
    print("Geliri:", gelir)
```

Dönüşüm

```
def kisiBilgileriniBastir():
    self.isimBastir()
    detaylariBastir()

def detaylariBastir():
    print("Dogum yeri", dYeri)
    print("Geliri:", gelir)
```

Refactoring

Refactoring Teknikleri

Fonksiyon üretme: Bir arada gruplanabilecek kod parçacıkları bulunuyor.

Bir fonksiyonda çok kod parçacıklarının bulunması, o fonksiyonun ne iş yaptığının anlaşılmasını zorlaştırmaktadır.

Bu yüzden fonksiyonun bir kısmı başka bir fonksiyona dönüştürülerek, fonksiyon yalınlaştırılabilir.

Refactoring

Refactoring Teknikleri

Değişken üretme: Bir ifadedeki kontroller iç içe ve anlaşılmaz olabilir. Bunlar ayrılarak daha anlaşılır hale getirilebilir.

Örnek Kod Parçacığı

```
def renderBanner(self):
    if (self.platform.toUpperCase().indexOf("MAC") > -1)
and \
       (self.browser.toUpperCase().indexOf("IE") > -1)
and \
       self.wasInitialized() and (self.resize > 0):
        # do something
```

Dönüşüm

```
def renderBanner(self):
    isMacOs = self.platform.toUpperCase().indexOf("MAC")
> -1
    isIE = self.browser.toUpperCase().indexOf("IE") > -1
    wasResized = self.resize > 0

    if isMacOs and isIE and self.wasInitialized() and
wasResized:
        # do something
```

Refactoring

Refactoring Teknikleri

Gereksiz kullanılan değişkenlerin yok edilmesi

Örnek Kod Parçacığı

```
def indirimKontrol(siparis):
    anaFiyat = siparis.anaFiyat()
    return anaFiyat > 1000
```

Dönüşüm

```
def indirimKontrol(siparis):
    return siparis.anaFiyat() > 1000
```

Refactoring

Refactoring Teknikleri

Geçici değişkenleri ayırma, kod daha okunabilir olacaktır. Bir birine bağımlı işlemler azalacağından, ayırmak istendiğinde rahat ayırılabilecektir.

Örnek Kod Parçacığı

```
temp = 2 * (en + boy)
print(temp)
temp = en * boy
print(temp)
```

→

Dönüşüm

```
cevre = 2 * (en + boy)
print(cevre)
alan = en * boy
print(alan)
```

Refactoring

Refactoring Teknikleri

Fonksiyondan sınıf üretme, fonksiyondaki değişkenler o kadar iç içe girmiş durumdadır ki, fonksiyon bölünerek yeni bir fonksiyon elde edilmesi çok zordur. Bu durumda fonksiyondan sınıf'a dönüştürülebilir.

Örnek Kod Parçacığı

```
class Order:
    # ...
    def price(self):
        primaryBasePrice = 0
        secondaryBasePrice = 0
        tertiaryBasePrice = 0
        # Perform long computation.
```

→

Dönüşüm

```
class Order:
    # ...
    def price(self):
        return PriceCalculator(self).compute()


class PriceCalculator:
    def __init__(self, order):
        self._primaryBasePrice = 0
        self._secondaryBasePrice = 0
        self._tertiaryBasePrice = 0
        # Copy relevant information from the
        # order object.

    def compute(self):
        # Perform long computation.
```

Refactoring

Refactoring Teknikleri

Algoritma değiştirmek, fonksiyonun işlevini değiştirerek yapılabilir.

Örnek Kod Parçacığı

```
def foundPerson(people):
    for i in range(len(people)):
        if people[i] == "Don":
            return "Don"
        if people[i] == "John":
            return "John"
        if people[i] == "Kent":
            return "Kent"
    return ""
```

→

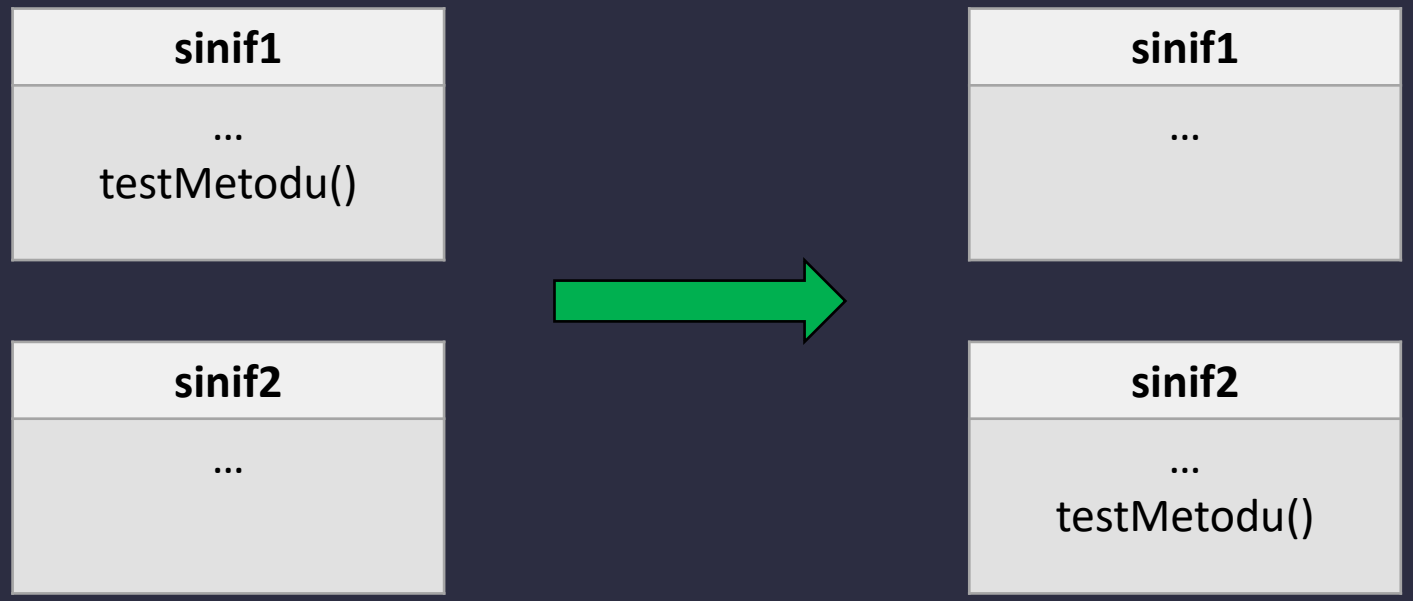
Dönüşüm

```
def foundPerson(people):
    candidates = ["Don", "John", "Kent"]
    for i in range(len(people)):
        if people[i] in candidates:
            return people[i]
    return ""
```

Refactoring

Refactoring Teknikleri

Bir sınıfa ait olan bir metod veya değişken, başka bir sınıf tarafından çok daha fazla çağırılıyorsa, o metodu veya değişkeni, çok çağıran sınıfa taşınmalıdır.



Refactoring

Refactoring Teknikleri

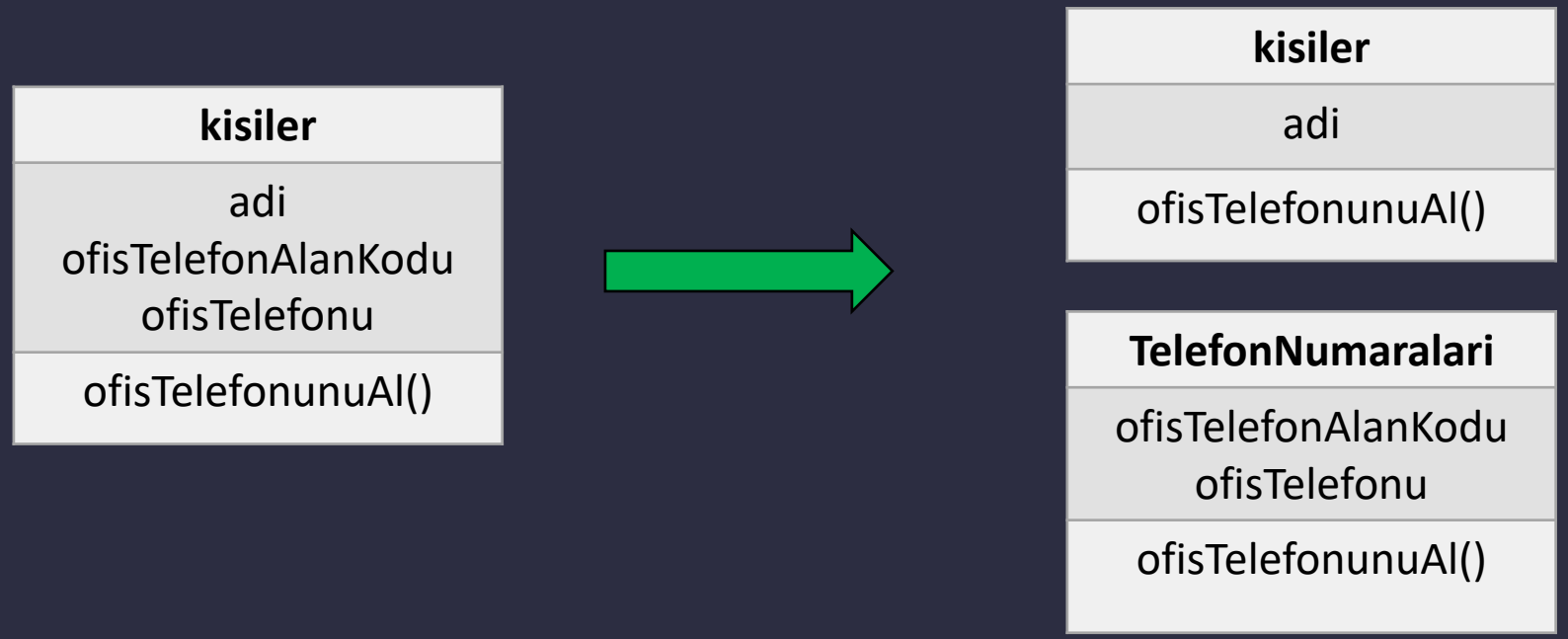
Bu yaklaşım;

- Sınıflar arasındaki bağımlılığı azaltacaktır.
- Sınıfları daha tutarlı hale getirir.

Refactoring

Refactoring Teknikleri

Bir sınıf yapması gereken işten fazlasını yapıyorsa, o sınıfı birden çok sınıfa bölün



Refactoring

Refactoring Teknikleri

Bir sınıf neredeyse hiçbir iş yapmıyorsa, o sınıfı, onu kullanan bir sınıf ile birleştirin

kisiler
adi
ofisTelefonunuAl()

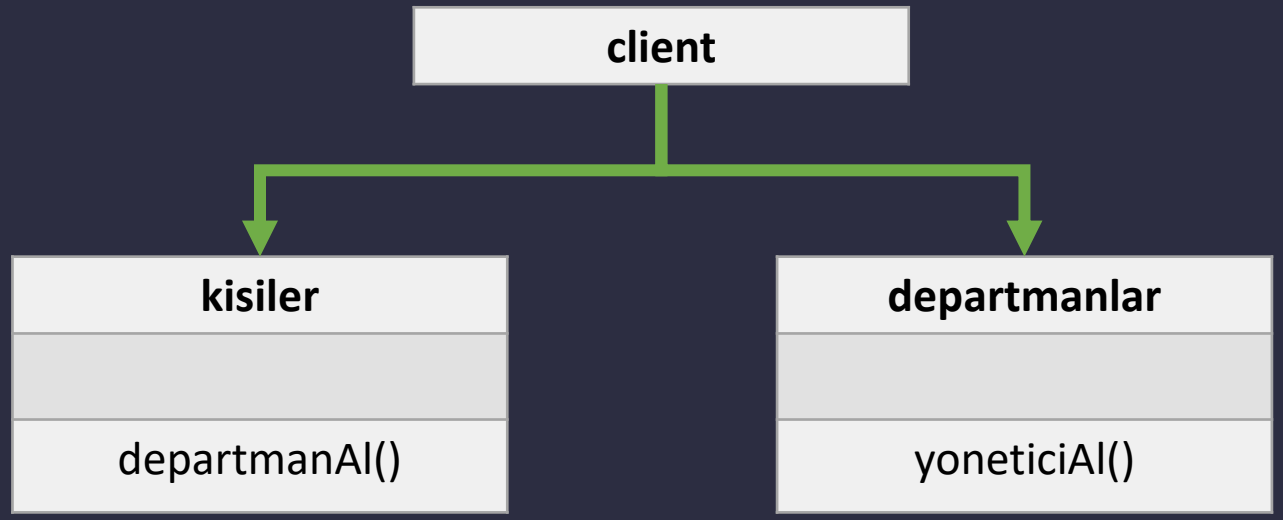
TelefonNumaralari
ofisTelefonAlanKodu ofisTelefonu
ofisTelefonunuAl()

kisiler
adi ofisTelefonAlanKodu ofisTelefonu
ofisTelefonunuAl()

Refactoring

Refactoring Teknikleri

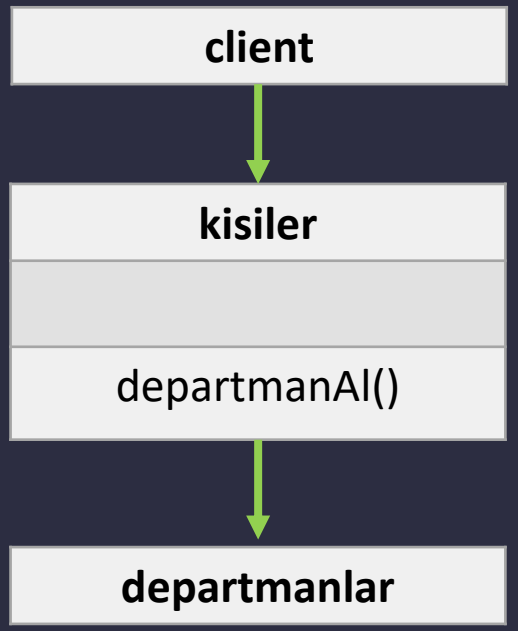
Client kodunda, bir sınıftan obje yaratılıp, o obje'den alınan bir başka sınıf çağrılması yapılması yerine, yaratılan obje'nin başka sınıfı çağırması sağlanabilir.



Refactoring

Refactoring Teknikleri

Client kodunda, bir sınıftan obje yaratılıp, o obje'den alınan bir başka sınıf çağrılması yapılması yerine, yaratılan obje'nin başka sınıfı çağırması sağlanabilir.



Refactoring

Refactoring Teknikleri

Anlamı olan sayıları, anlamlı isimler vererek değişken yapımı, daha sonrasında sabit sayılar birden çok yerde kullanılıyorsa, tek bir yerden değiştirilebilir.

Örnek Kod Parçacığı

```
def potansiyelEnerji(kutle, yukseklik):
    return kutle * yukseklik * 9.81
```

Dönüşüm

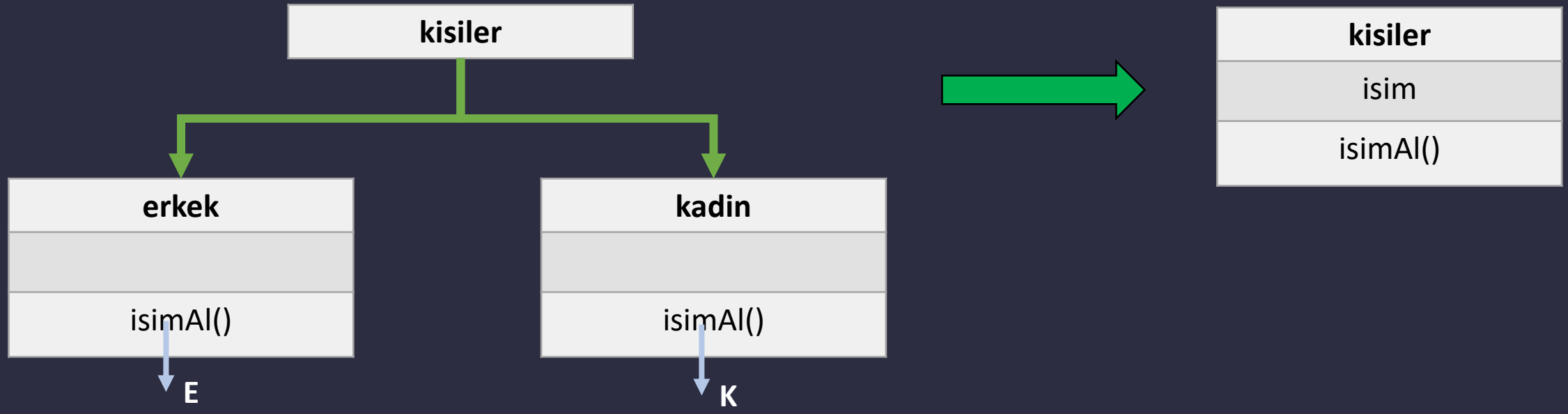
```
YERCEKIMI_SABITI = 9.81

def potansiyelEnerji(kutle, yukseklik):
    return kutle * yukseklik * YERCEKIMI_SABITI
```

Refactoring

Refactoring Teknikleri

Bir sınıftan türetilen, altsınıflardaki fonksiyonlar sabit değerler döndürüyorlarsa, o fonksiyon üst sınıfa taşınabilir.



Refactoring

Refactoring Teknikleri

Birden çok kontrol mekanizmalarının (if-else) bulunduğu durumlarda, aynı sonucu döndüren durumlar birleştirilebilir.

Örnek Kod Parçacığı

```
def denemeFunc():
    if x < 2:
        return 0
    elif y > 12:
        return 0
    elif z == 1:
        return 0
    elif t == 0:
        return 1
```

Dönüşüm

```
def denemeFunc():
    if kontrolGrubu() == 1:
        return 0
    elif t == 0:
        return 1
```

Bu fonksiyonda x, y ve z'nin değerleri kontrol ediliyor

Refactoring

Refactoring Teknikleri

Tekrarlanan kod parçacıklarını temizleme

Örnek Kod Parçacığı

```
if x == 0:
    toplam = fiyat * 0.95
    gonder(toplam)
else:
    toplam = fiyat * 0.98
    gonder(toplam)
```

Dönüşüm

```
if x == 0:
    toplam = fiyat * 0.95
else:
    toplam = fiyat * 0.98

gonder(toplam)
```

Refactoring

Refactoring Teknikleri

Gereksiz iç içe geçmiş koşulları temizleme

Örnek Kod Parçacığı

```
def odemeMiktariniAl(self):
    if x == 0:
        sonuc = 123
    else:
        if y == 0:
            sonuc = 456
        else:
            if z == 0:
                sonuc = 789
            else:
                sonuc = 1000
    return sonuc
```

Dönüşüm

```
def odemeMiktariniAl(self):
    if x == 0:
        return 123
    if y == 0:
        return  456
    if z == 0:
        return 789
    return 1000
```

Refactoring

Refactoring Teknikleri

Metod yeniden isimlendirme, işlevi hakkında bir isim verilmelidir.