Bölüm 6

Soyutlama

6.1 Giriş

Soyutlama (abstraction), nesne tabanlı programlamanın (OOP) temel prensiplerinden biridir ve karmaşık sistemleri basitleştirmek için yalnızca gerekli detayların gösterilmesini sağlar. Kullanıcı, bir nesnenin iç işleyişini bilmeden onun arayüzünü kullanabilir. Bu bölümde, soyutlamanın tanımı, uygulanması, avantajları, zorlukları ve modern yazılım mühendisliğindeki rolü ele alınacaktır.

6.2 Soyutlama Nedir?

Soyutlama, bir nesnenin veya sistemin yalnızca temel ve gerekli özelliklerini öne çıkararak gereksiz detayları gizler. Örneğin, bir uzaktan kumanda düşünün: Kullanıcı, televizyonu açmak için düğmeye basar, ancak içindeki elektronik devrelerin nasıl çalıştığını bilmesi gerekmez. OOP'de soyutlama, soyut sınıflar (abstract classes) ve arayüzler (interfaces) ile uygulanır.

Soyutlamanın temel kavramları, OOP tasarımında karmaşıklığı yönetmek için kritik öneme sahiptir.

Soyut sınıf, doğrudan örneklenemeyen (instantiate edilemeyen) bir sınıftır ve yalnızca miras alınmak üzere tasarlanır. Bu sınıflar, alt sınıfların paylaşması gereken ortak özellikleri ve davranışları tanımlar, ancak soyut metodlar aracılığıyla alt sınıfların bu davranışları zorunlu olarak uygulamalarını sağlar. Örneğin, bir soyut sınıf, temel bir yapı sağlar ancak tam bir işlevsellik sunmaz; bu, hiyerarşik tasarımlarda tutarlılık ve kod yeniden kullanımını teşvik eder.

Arayüz ise, yalnızca metod imzalarını (signature) içeren bir sözleşmedir; implementasyon detaylarını tamamen alt sınıflara bırakır. Bu, farklı sınıfların aynı arayüzü uygulamasına izin vererek, çoklu kalıtım benzeri esneklik sağlar ve gevşek bağlanmayı (loose coupling) destekler.

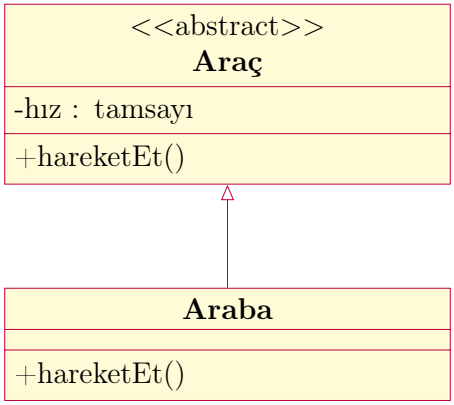
Veri gizleme kavramı, kapsülleme ile iç içedir ve soyutlamanın bir parçası olarak nesnenin iç durumunu (private fields) dış dünyadan saklar, yalnızca tanımlı fonksiyonlar

(public methods) üzerinden erişime izin verir. Bu üç kavram bir araya gelerek, karmaşık sistemlerin yönetilebilir hale gelmesini ve bakımın kolaylaşmasını sağlar.

6.2.1 Soyut Sınıf

Soyut sınıf, doğrudan örneklenemeyen bir sınıftır; yani, "new" anahtar kelimesi ile nesne oluşturulamaz. Bu, sınıfın tam bir implementasyon sağlamadığını ve yalnızca alt sınıflar tarafından genişletilmek üzere tasarlandığını belirtir. Instantiate edilememesi, sınıfın soyut metodlar içerdiği ve bu metodların alt sınıflarda uygulanması gerektiği anlamına gelir; aksi takdirde, sınıf eksik kalır ve nesne oluşturmak mantıksız olur. Bu yaklaşım, tasarımda tutarlılık sağlar ve alt sınıfların belirli davranışları zorunlu kılmasını garantiler.

Örneğin, bir soyut *Araç* sınıfı ele alalım: Bu sınıf, tüm araçların ortak özelliklerini (örneğin, hız) tanımlar, ancak *hareketEt* gibi soyut bir metodu alt sınıflara bırakır. Doğrudan *Araç* nesnesi oluşturulamayacağı için, hata önlenir ve alt sınıflar (örneğin, Araba) bu metodu uygulamak zorunda kalır.

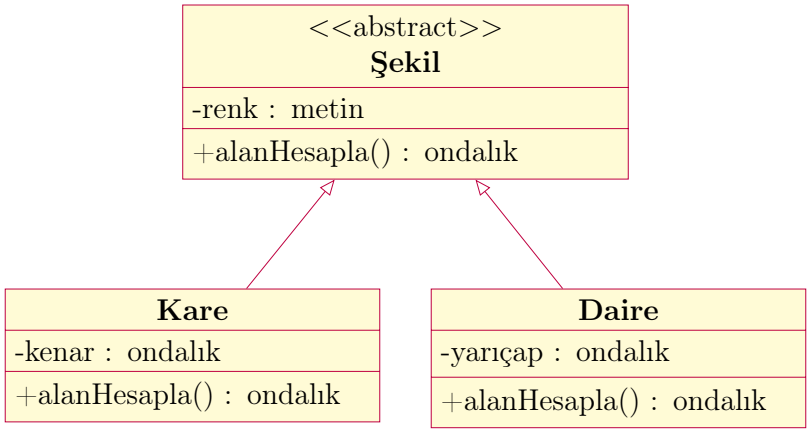


Şekil 6.1 – Soyut Araç Sınıfının UML Diyagramı

Sözde-kod 6.1 – Soyut Sınıf Örneği

```
soyut sınıf Araç {
 özel özellik hız: tamsayı

 yapıcı(hız: tamsayı) {
  bu.hız = hız
 }

 soyut genel metod hareketEt()
}

sınıf Araba miras_alır Araç {
 yapıcı(hız: tamsayı) {
  super(hız)
 }

 genel metod hareketEt() {
  yaz "Araba ", hız, " km/s ile hareket ediyor"
 }
}

ana_program {
 // araç = yeni Araç(100)  // Hata: Soyut sınıf örneklenemez
 araba = yeni Araba(100)
 araba.hareketEt()  // Çıktı: Araba 100 km/s ile hareket ediyor
}
```

Şekil sınıfından Kare ve Daire için soyutlamanın bir örneğinin UML şeması ve sözde-kodu aşağıda verilmiştir:



Şekil 6.2 – Şekil Hiyerarşisinin UML Diyagramı

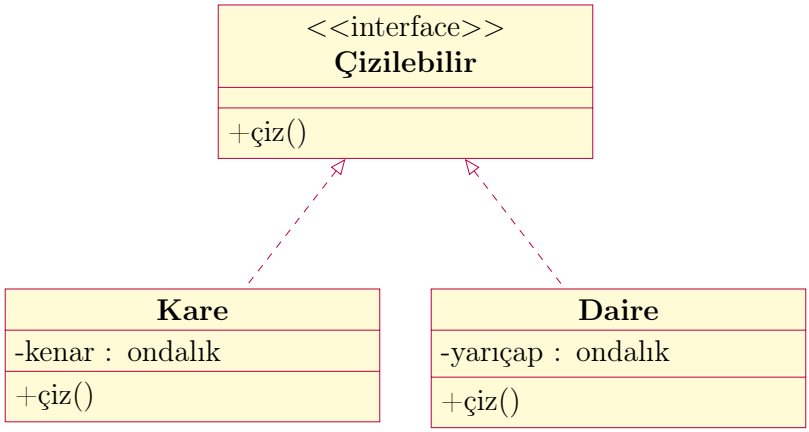
Sözde-kod 6.2 – Şekil Hiyerarşisi Örneği

```
soyut sınıf Şekil {
 özel özellik renk: metin

 yapıcı(renk: metin) {
  bu.renk = renk
 }

 soyut genel metod alanHesapla(): ondalık
}

sınıf Daire miras_alır Şekil {
 özel özellik yarıçap: ondalık

 yapıcı(renk: metin, yarıçap: ondalık) {
  super(renk)
  bu.yarıçap = yarıçap
 }

 genel metod alanHesapla(): ondalık {
  döndür 3.14 * yarıçap * yarıçap
 }
}

sınıf Kare miras_alır Şekil {
 özel özellik kenar: ondalık

 yapıcı(renk: metin, kenar: ondalık) {
  super(renk)
  bu.kenar = kenar
 }

 genel metod alanHesapla(): ondalık {
  döndür kenar * kenar
 }
}

ana_program {
 şekiller = [yeni Daire("Kırmızı", 5), yeni Kare("Mavi", 4)]
 her şekil için şekiller içinde {
  yaz "Alan: ", şekil.alanHesapla()
 }
 // Çıktı:
 // Alan: 78.5
 // Alan: 16
}
```

6.2.2 Arayüz

Arayüz (interface), bir sınıfın uygulaması gereken metodların sözleşmesini tanımlayan bir yapıdır. Arayüzler, herhangi bir implementasyon içermez; yalnızca metod imzalarını (ad, parametreler, dönüş tipi) belirtir. Bu, sınıfların belirli bir davranış setini garanti etmesini sağlar ve çoklu arayüz uygulama sayesinde, sınıflar farklı sözleşmeleri aynı anda karşılayabilir (çoklu kalıtım benzeri). Arayüzler, tasarımda esneklik ve modülerlik sağlar; örneğin, farklı sınıflar aynı arayüzü uygulayarak, polimorfizm ile kullanılabilir. OOP'de arayüzler, SOLID prensiplerinden "Interface Segregation"ı destekler, çünkü küçük ve odaklanmış arayüzler oluşturmayı teşvik eder.

Örneğin, bir *Çizilebilir* arayüzü, farklı şekillerin (Daire, Kare) çizim davranışını zorunlu kılar, ancak nasıl çizileceğini alt sınıflara bırakır.



Şekil 6.3 – Çizilebilir Arayüzünün UML Diyagramı

Sözde-kod 6.3 – Arayüz Örneği

```
arayüz Çizilebilir {
 metod çiz()
}

sınıf Daire uygular Çizilebilir {
 özel özellik yarıçap: ondalık

 yapıcı(yarıçap: ondalık) {
  bu.yarıçap = yarıçap
 }

 genel metod çiz() {
  yaz "Daire çiziliyor, yarıçap: ", yarıçap
 }
}

sınıf Kare uygular Çizilebilir {
 özel özellik kenar: ondalık

 yapıcı(kenar: ondalık) {
  bu.kenar = kenar
 }

 genel metod çiz() {
  yaz "Kare çiziliyor, kenar: ", kenar
 }
}

ana_program {
 çizilebilirler = [yeni Daire(5), yeni Kare(4)]
 her nesne için çizilebilirler içinde {
  nesne.çiz()
 }
 // Çıktı:
 // Daire çiziliyor, yarıçap: 5
 // Kare çiziliyor, kenar: 4
}
```

6.2.3 Veri Gizleme

Veri gizleme (data hiding), bir nesnenin iç verilerini dış dünyadan saklayarak, yalnızca tanımlı metodlar üzerinden erişime izin veren bir mekanizmadır. Bu, kapsülleme prensibinin bir parçasıdır ve erişim belirleyiciler (private, protected, public) ile uygulanır. Veri gizleme, veri tutarlılığını korur, yetkisiz değişiklikleri önler ve sınıfın iç yapısını değiştirirken dış arayüzü etkilemez. Örneğin, private bir değişken, doğrudan erişilemez; getter/setter metodları ile kontrollü erişim sağlanır, böylece doğrulama kuralları (örneğin, negatif değer kabul etmeme) uygulanabilir.

Örneğin, bir *Kişi* sınıfında yaş özelliği private tutulur ve setter metodu ile negatif değerler engellenir.

Şekil 6.4 – Veri Gizleme ile Kişi Sınıfının UML Diyagramı

Sözde-kod 6.4 – Veri Gizleme Örneği

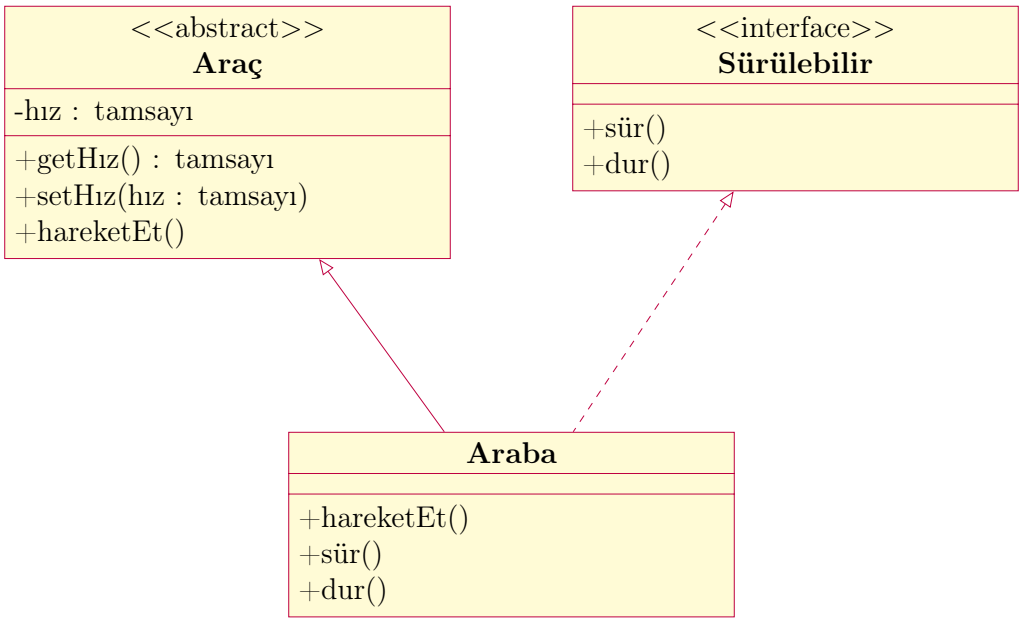
```
sınıf Kişi {
 özel özellik yaş: tamsayı

 yapıcı(yaş: tamsayı) {
  setYaş(yaş)
 }

 genel metod getYaş(): tamsayı {
  döndür yaş
 }

 genel metod setYaş(yeniYaş: tamsayı) {
  eğer yeniYaş >= 0 ise
  yaş = yeniYaş
  değilse
  hata "Geçersiz yaş değeri"
 }
}

ana_program {
 kişi = yeni Kişi(25)
 yaz "Yaş: ", kişi.getYaş()  // Çıktı: Yaş: 25
 kişi.setYaş(-5)  // Hata: Geçersiz yaş değeri
 // kişi.yaş = -5  // Hata: Özel özellik erişilemez
}
```

6.3 Soyutlamanın Uygulanması

Soyutlama tüm kavramlarını (soyut sınıf, arayüz, veri gizleme) bir arada kullanmak, karmaşık sistemlerde güçlü bir tasarım sağlar.

Aşağıda, bir araç simülasyonu örneği verilmiştir: Soyut *Araç* sınıfı (veri gizleme ile private hız özelliği ve soyut hareketEt metodu), *Sürülebilir* arayüzü (sür ve dur metodları) ve somut *Araba* sınıfı (soyut sınıfı miras alır ve arayüzü uygular) bir araya getirilir. Bu, kalıtım, implementasyon ve kontrollü erişimi entegre eder; hız özelliği gizlenirken, sürüş davranışları zorunlu kılınır ve farklı araç türleri polimorfik olarak işlenir.

Bu örnekte, soyut sınıf veri gizleme sağlar (hız özelliği kontrollü erişimle yönetilir), arayüz sürüş davranışlarını zorunlu kılar ve soyut metodu alt sınıf tamamlar. Bu birleşim, esnek ve güvenli bir tasarım sunar.



Şekil 6.5 – Araç Simülasyonu: Soyut Sınıf, Arayüz ve Veri Gizleme Birleşimi

Sözde-kod 6.5 – Tüm Kavramların Birleşik Kullanımı Örneği

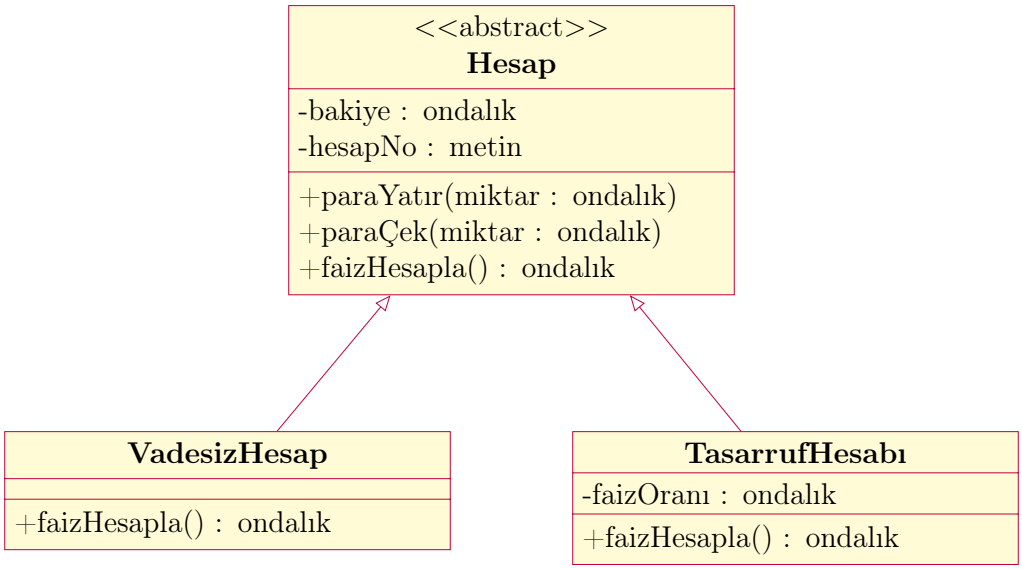
```
soyut sınıf Araç {
 özel özellik hız: tamsayı

 yapıcı(başlangıçHız: tamsayı) {
  setHız(başlangıçHız)
 }

 genel metod getHız(): tamsayı {
  döndür hız
 }

 genel metod setHız(yeniHız: tamsayı) {
  eğer yeniHız >= 0 ise
  hız = yeniHız
  değilse
  hata "Geçersiz hız değeri"
 }

 soyut genel metod hareketEt()
}

arayüz Sürülebilir {
 metod sür()
 metod dur()
}

sınıf Araba miras_alır Araç uygular Sürülebilir {
 yapıcı(başlangıçHız: tamsayı) {
  super(başlangıçHız)
 }

 genel metod hareketEt() {
  yaz "Araba hareket ediyor, hız: ", getHız()
 }

 genel metod sür() {
  setHız(getHız() + 10)
  yaz "Araba hızlanıyor, yeni hız: ", getHız()
 }

 genel metod dur() {
  setHız(0)
  yaz "Araba durdu"
 }
}

ana_program {
 araba = yeni Araba(50)
 araba.sür()      // Çıktı: Araba hızlanıyor, yeni hız: 60
 araba.hareketEt() // Çıktı: Araba hareket ediyor, hız: 60
 araba.dur()      // Çıktı: Araba durdu
 // araba.hız = -10  // Hata: Veri gizleme (private)
}
```

6.3.1 Soyut Sınıfın İleri Örnekleri

Soyut sınıfların ileri düzey kullanımı, karmaşık hiyerarşilerde ortak mantığı paylaşırken alt sınıflara özgü davranışları zorunlu kılmak için idealdir. Bu, SOLID prensiplerinden "Interface Segregation" ve "Dependency Inversion" ile uyumludur, çünkü soyut sınıflar hem ortak implementasyon hem de soyut kontrat sağlar.

Aşağıda, bir banka sistemi örneği verilmiştir: Soyut *Hesap* sınıfı, tüm hesap türlerinin ortak özelliklerini (örneğin, bakiye yönetimi) tanımlar, ancak faiz hesaplama gibi özgü davranışları alt sınıflara bırakır. Bu yaklaşım, yeni hesap türleri eklerken (örneğin, Yatırım Hesabı) ana sistemin değişmemesini sağlar ve veri tutarlılığını korur.

UML diyagramı ve sözde-kod verilen bu ileri soyutlama örneğinde, soyut *faizHesapla* metodu, her hesap türünün kendine özgü faiz mantığını zorunlu kılar, ancak ortak işlemler olan *paraYatır* ve *paraÇekme* soyut sınıfta paylaşılır. Bu, bakım kolaylığı sağlar ve yeni hesap türleri için yalnızca alt sınıf eklemek yeterlidir.



Şekil 6.6 – Hesap Hiyerarşisinin UML Diyagramı (İleri Soyutlama Örneği)

Sözde-kod 6.6 – Soyut Hesap Sınıfının İleri Örneği

```
soyut sınıf Hesap {
 özel özellik bakiye: ondalık
 özel özellik hesapNo: metin

 yapıcı(hesapNo: metin, başlangıçBakiye: ondalık) {
  bu.hesapNo = hesapNo
  bu.bakiye = başlangıçBakiye
 }

 genel metod paraYatır(miktar: ondalık) {
  eğer miktar > 0 ise
  bakiye = bakiye + miktar
  değilse
  hata "Geçersiz miktar"
 }

 genel metod paraÇek(miktar: ondalık) {
  eğer miktar > 0 ve bakiye >= miktar ise
  bakiye = bakiye - miktar
  değilse
  hata "Yetersiz bakiye"
 }

 soyut genel metod faizHesapla(): ondalık  // Alt sınıflar için
}

sınıf TasarrufHesabı miras_alır Hesap {
 özel özellik faizOranı: ondalık

 yapıcı(hesapNo: metin, başlangıçBakiye: ondalık, faizOranı:
    ondalık) {
  super(hesapNo, başlangıçBakiye)
  bu.faizOranı = faizOranı
 }

 genel metod faizHesapla(): ondalık {
  döndür bakiye * faizOranı / 100  // Yıllık faiz hesabı
 }
}

sınıf VadesizHesap miras_alır Hesap {
 yapıcı(hesapNo: metin, başlangıçBakiye: ondalık) {
  super(hesapNo, başlangıçBakiye)
 }

 genel metod faizHesapla(): ondalık {
  döndür 0  // Vadesiz hesapta faiz yok
 }
}

ana_program {
 hesaplar = [yeni TasarrufHesabı("TR123", 1000, 5.0), yeni
    VadesizHesap("TR456", 2000)]
 her hesap için hesaplar içinde {
  hesap.paraYatır(100)
  yaz "Hesap: ", hesap.hesapNo, " - Faiz: ", hesap.faizHesapla()
 }
 // Hesap: TR123 - Faiz: 55.0
 // Hesap: TR456 - Faiz: 0
}
```

6.4 Soyutlamanın Avantajları ve Zorlukları

Soyutlama, yazılım tasarımında önemli avantajlar sunarken, yanlış kullanıldığında bazı zorluklara yol açabilir. Aşağıda, bu avantajlar ve zorluklar detaylı olarak ele alınacaktır.

Avantajları:

— **Basitleştirme**: Soyutlama, kullanıcının veya geliştiricinin yalnızca gerekli arayüzle ilgilenmesini sağlayarak bilişsel yükü azaltır. İç detaylar gizlendiği için, karmaşık sistemler daha yönetilebilir hale gelir. Örneğin, bir veritabanı bağlantı sınıfında soyut bir *bağlan* metodu tanımlanırsa, istemci yalnızca bu metodu çağırır; veritabanı sağlayıcısının (MySQL, PostgreSQL) detayları gizlenir. Bu, odaklanmayı artırır ve hata riskini düşürür, çünkü geliştiriciler soyut arayüzle çalışarak iç implementasyon değişikliklerinden etkilenmez.

— **Modülerlik**: Soyut sınıflar ve arayüzler, sistemi bağımsız bileşenlere ayırarak ayrımcılık prensibini (separation of concerns) teşvik eder. Bu, kodun test edilebilirliğini ve bakımını kolaylaştırır. Örneğin, mikro hizmet mimarisinde her hizmet bir arayüz üzerinden tanımlanırsa, bir hizmetin iç yapısı değişse bile diğer hizmetler etkilenmez. Bu yaklaşım, büyük ölçekli projelerde işbirliğini artırır ve bileşenlerin yeniden kullanılabilirliğini sağlar.

— **Esneklik**: Soyutlama, yeni alt sınıflar veya implementasyonlar eklemeyi kolaylaştırır ve SOLID prensiplerinden "Açık-Kapalı Prensibi"ni (Open-Closed Principle) destekler: Yazılım, genişletmeye açık ancak mevcut koda kapalıdır. Örneğin, bir grafik kütüphanesinde soyut *Şekil* sınıfı varsa, yeni bir *Üçgen* sınıfı eklemek için yalnızca miras alma yeterli olur; ana kod değişmez. Bu, sistemin evrimini hızlandırır ve gelecekteki gereksinimlere uyum sağlamayı kolaylaştırır.

Zorlukları:

— **Aşırı Soyutlama**: Gereksiz soyut sınıflar veya arayüzler oluşturmak, "YAGNI" (You Ain't Gonna Need It) prensibini ihlal ederek kodu karmaşıklaştırabilir ve geliştirme süresini uzatabilir. Örneğin, basit bir sayaç sınıfı için birden fazla arayüz tanımlamak, gereksiz katmanlar yaratır ve yeni geliştiricilerin anlamasını zorlaştırır. Bu durum, "over-engineering" olarak bilinir ve bakım maliyetini artırır; bu nedenle, soyutlama seviyesi gereksinimlere göre dengelenmelidir.

— **Tasarım Zorluğu**: Doğru soyutlama seviyesini belirlemek, deneyim ve alan bilgisi gerektirir. Yanlış bir soyutlama, Liskov Değiştirme Prensibi'ni (Liskov Substitution Principle) ihlal ederek beklenmedik davranışlara yol açabilir. Örneğin, bir soyut sınıfın metodunu alt sınıflarda yanlış uygulamak, sistemin tutarlılığını bozar. Bu, öğrenme eğrisini artırır ve başlangıçta hatalı tasarımlara neden olabilir; bu yüzden, prototip geliştirme ve refactoring teknikleri önerilir.

6.5 Modern Bağlamda Soyutlama

Soyutlama, modern yazılım mühendisliğinde kritik bir rol oynar:

— API Tasarımı: REST veya GraphQL API'leri, soyut arayüzler sunarak iç detayları gizler.

— Kütüphaneler ve Çerçeveler: Örneğin, Java'daki 'List' arayüzü, farklı uygulamaları (ArrayList, LinkedList) soyutlar.

— Oyun Motorları: Unity veya Unreal Engine, soyut sınıflarla nesne davranışlarını genelleştirir.

6.6 Alıştırmalar

1. Soyutlamanın tanımını ve avantajlarını kendi kelimelerinizle açıklayın.
2. Bir *Araç* soyut sınıfı ve *Bisiklet*, *Motosiklet* alt sınıfları için sözde-kod yazın. Her sınıf, *hızHesapla* metodunu uygulasın.
3. Şekil 6.2'deki UML diyagramını inceleyin ve başka bir hiyerarşi (örneğin, Cihaz → Telefon, Bilgisayar) için benzer bir UML diyagramı çizin.
4. Soyut sınıflar ve arayüzler arasındaki farkları tartışın ve birer örnek verin.
5. Soyutlamanın bir API tasarımında nasıl kullanıldığını araştırın ve bir örnekle açıklayın.