Bölüm 4

Kalıtım

4.1 Giriş

Kalıtım (inheritance), nesne tabanlı programlamanın (OOP) temel prensiplerinden biridir ve bir sınıfın başka bir sınıfın özelliklerini ve davranışlarını miras almasını sağlar. Bu, kod yeniden kullanımını ve hiyerarşik tasarımı teşvik eder. Bu bölümde, kalıtımın tanımı, türleri, avantajları, zorlukları ve modern yazılım mühendisliğindeki rolü ele alınacaktır. Amaç, okuyucuların kalıtımın nasıl çalıştığını ve nasıl etkili bir şekilde kullanıldığını anlamasıdır.

4.2 Kalıtım Nedir?

Kalıtım, bir alt sınıfın (subclass) üst sınıfın (superclass) özelliklerini ve metodlarını miras alarak genişletmesini sağlar. Örneğin, bir *Araç* üst sınıfı, tüm araçların ortak özelliklerini (örneğin, hız, renk) ve davranışlarını (örneğin, hareket et) tanımlar. *Araba* veya *Kamyon* gibi alt sınıflar, bu özellikleri miras alır ve kendine özgü özellikler ekler (örneğin, Araba için kapı sayısı).

Kalıtımın temel kavramları:

- **Üst Sınıf (Superclass)**: Miras alınan sınıf. Bu sınıf, ortak özellikler ve davranışlar tanımlayarak alt sınıflar için temel oluşturur. Örneğin, bir *Hayvan* üst sınıfı, tüm hayvanların ortak özelliklerini (örneğin, yaş, tür) ve davranışlarını (örneğin, beslen) içerir ve alt sınıflar bu temeli kullanır.
- **Alt Sınıf (Subclass)**: Üst sınıfı miras alan ve genişleten sınıf. Alt sınıf, üst sınıfın özelliklerini otomatik olarak alır ve yeni özellikler veya metodlar ekleyebilir. Örneğin, *Kedi* sınıfı *Hayvan* sınıfını miras alarak kediye özgü özellikler (örneğin, tüy rengi) ekler.
- **Metod Geçersiz Kılma (Overriding)**: Alt sınıfın, üst sınıfın metodunu yeniden tanımlaması. Bu, alt sınıfın aynı isimli metodu farklı bir şekilde uygulamasına izin verir. Örneğin, *Hayvan* sınıfındaki *sesÇıkar* metodu "Ses çıkarıyor" derken, *Kedi* alt sınıfında bu metod "Miyav" olarak geçersiz kılınabilir.

- **super**: Üst sınıfın metodlarına veya yapıcısına erişim. Alt sınıfın yapıcısında veya metodunda üst sınıfın versiyonunu çağırmak için kullanılır. Örneğin, alt sınıfın yapıcısında *super(yapıcı parametreleri)* ile üst sınıfın yapıcısı çağrılır, böylece ortak başlatma işlemleri tekrarlanmaz.

4.3 Kalıtımın Türleri

Kalıtım, nesne tabanlı programlamada farklı biçimlerde uygulanabilir ve her biri belirli senaryolarda farklı avantajlar ve zorluklar sunar. Aşağıda, kalıtımın temel türleri detaylı bir şekilde açıklanmıştır:

4.3.1 Tekli Kalıtım

Bir alt sınıfın yalnızca tek bir üst sınıftan miras aldığı yapıdır. Bu model, sadeliği ve anlaşılırlığı nedeniyle birçok programlama dilinde (örneğin, Java, C#) yaygın olarak kullanılır. Örneğin, bir *Araba* sınıfı, *Araç* sınıfından miras alarak onun özelliklerini (örneğin, hız, renk) ve davranışlarını (örneğin, hareket et) alır ve kendine özgü özellikler (örneğin, kapı sayısı) ekler.

Avantajları:

— Kod yapısı sade ve yönetilebilir kalır.

— Hiyerarşi açık ve takip edilebilirdir, bu da bakım ve hata ayıklamayı kolaylaştırır.

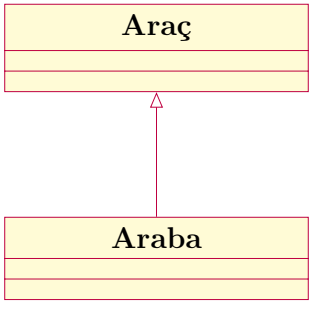
— Diamond problem gibi karmaşıklıklar ortaya çıkmaz.

Zorlukları:

— Esneklik sınırlıdır; bir sınıf birden fazla üst sınıftan özellik miras alamaz.

— Kompozisyon gibi alternatif yöntemler, bazı durumlarda daha uygun olabilir.

Örnek: Java'da, *String* sınıfı yalnızca *Object* sınıfından miras alır, böylece temel metodları (örneğin, *toString*) kullanabilir.



Şekil 4.1 – Tekli Kalıtım UML Diyagramı

4.3.2 Çoklu Kalıtım

Bir alt sınıfın birden fazla üst sınıftan miras aldığı yapıdır. Bu, özellikle C++ ve Python gibi dillerde desteklenir. Örneğin, bir *UçanAraba* sınıfı hem *Araba* hem de *Uçak*

sınıflarından miras alarak hem karada hareket etme hem de uçma yeteneklerini kazanabilir. Ancak, çoklu kalıtım, *diamond problem* gibi karmaşıklıklara yol açabilir; bu, iki üst sınıfın aynı metod veya özelliği tanımlaması durumunda hangi versiyonun kullanılacağının belirsizleşmesidir.

Avantajları:

— Daha fazla esneklik sağlar; birden fazla üst sınıfın özelliklerini birleştirme imkanı sunar.

— Gerçek dünya senaryolarını modellemek için güçlü bir araçtır (örneğin, bir cihazın hem yazıcı hem tarayıcı özelliklerini miras alması).

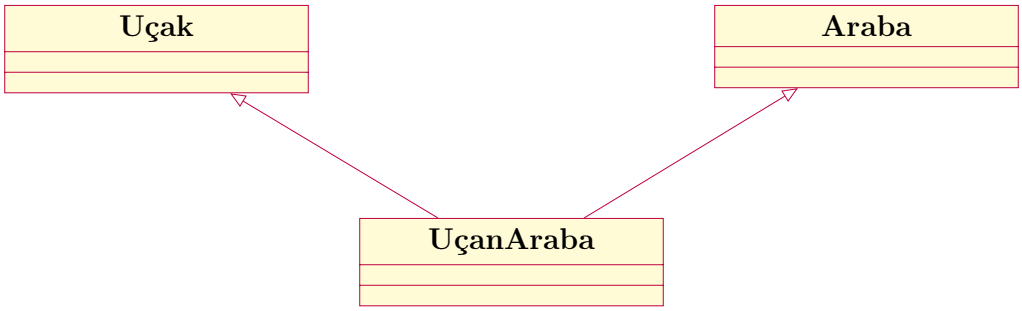
Zorlukları:

— Diamond problem nedeniyle çakışmalar meydana gelebilir. Örneğin, iki üst sınıf aynı isimde bir metod tanımlarsa, alt sınıf hangisini kullanacağını belirlemek için ek kurallara ihtiyaç duyar.

— Kod karmaşıklığı artar ve hata ayıklaması zorlaşabilir.

— Bazı diller (örneğin, Java) bu karmaşıklıkları önlemek için çoklu kalıtımı desteklemez ve yerine arayüzler (interfaces) kullanır.

Örnek: Python'da, bir sınıf *A* ve *B* sınıflarından miras alabilir. Diamond problemi çözmek için Python, Metod Çözümleme Sırası (Method Resolution Order - MRO) kullanır.



Şekil 4.2 – Çoklu Kalıtım UML Diyagramı

4.3.3 Hiyerarşik Kalıtım

Birden fazla alt sınıfın tek bir üst sınıftan miras aldığı yapıdır. Bu model, ortak bir temelin birden fazla özelleşmiş sınıf tarafından paylaşıldığı durumlarda kullanışlıdır. Örneğin, *Araç* sınıfından *Araba*, *Kamyon* ve *Motosiklet* sınıfları miras alabilir.

Avantajları:

— Ortak özelliklerin ve davranışların tek bir üst sınıfta toplanması, kod tekrarını azaltır.

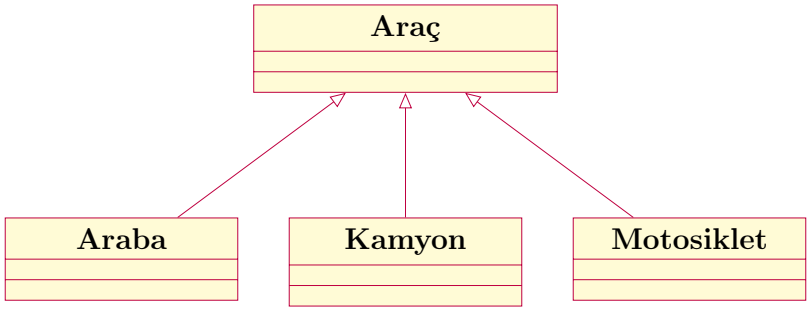
— Gerçek dünya hiyerarşilerini modellemek için idealdir (örneğin, biyolojideki tür sınıflandırmaları).

Zorlukları:

— Üst sınıfta yapılan değişiklikler, tüm alt sınıfları etkileyebilir, bu da bağımlılık sorunlarına yol açabilir.

— Çok derin hiyerarşiler, kodun anlaşılmasını ve bakımını zorlaştırabilir.

Örnek: Java'da, *java.util* paketindeki *Collection* sınıfından *List*, *Set* ve *Queue* gibi alt sınıflar miras alır.



Şekil 4.3 – Hiyerarşik Kalıtım UML Diyagramı

4.3.4 Çok Seviyeli Kalıtım

Bir alt sınıfın başka bir alt sınıftan miras aldığı, zincirleme bir hiyerarşi yapısıdır. Örneğin, *Araç* sınıfından *Araba* miras alır, *Araba* sınıfından ise *SporAraba* miras alır.

Avantajları:

— Daha spesifik sınıflar oluşturmak için hiyerarşik bir yapı sağlar.

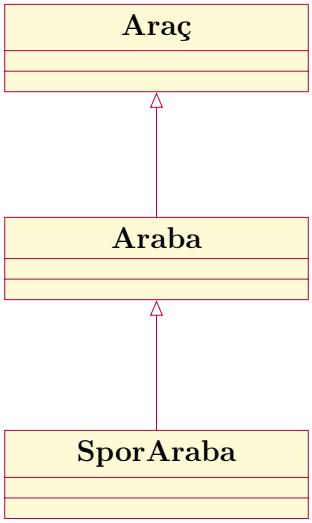
— Kod yeniden kullanımı, her seviyede daha fazla özelleşme ile artırılır.

Zorlukları:

— Hiyerarşi derinleştikçe, bağımlılıklar ve karmaşıklık artar.

— Üst sınıflardaki değişiklikler, tüm alt sınıfları etkileyebilir.

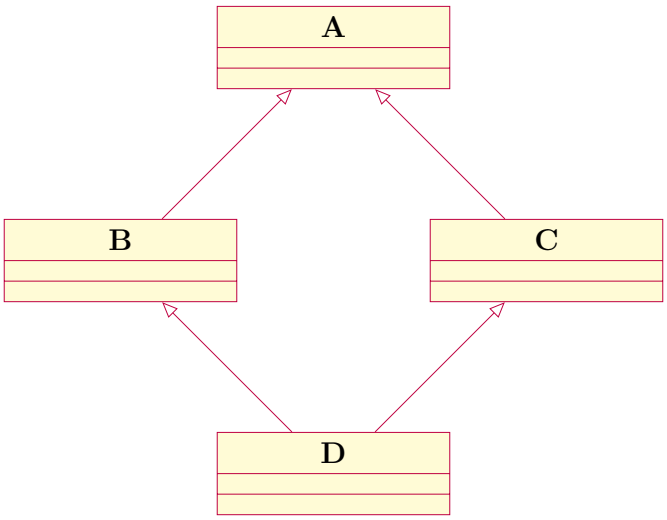
Örnek: C++'ta, bir *SporAraba* sınıfı *Araba* sınıfından, o da *Araç* sınıfından miras alabilir.



Şekil 4.4 – Çok Seviyeli Kalıtım UML Diyagramı

NOT.

Diamond Problemi: Diamond problemi, çoklu kalıtımda ortaya çıkan bir belirsizlik (ambiguity) sorunudur. Bu sorun, bir alt sınıfın iki veya daha fazla üst sınıftan miras alması ve bu üst sınıfların ortak bir atadan (ancestor) miras alması durumunda meydana gelir. Hiyerarşi, elmas (diamond) şekline benzediği için bu isimle anılır. Örneğin, A sınıfı temel sınıf olsun; B ve C sınıfları A'dan miras alsın; D sınıfı ise B ve C'den miras alsın. D sınıfında A'nın bir metodu çağrıldığında, bu metodun B üzerinden mi yoksa C üzerinden mi geleceği belirsizleşir. Bu, derleme zamanı hatalarına veya beklenmedik davranışlara yol açabilir. Çözümler arasında C++'taki sanal kalıtım (virtual inheritance) veya Java'daki arayüzler gibi mekanizmalar yer alır.



Şekil 4.5 – Diamond Problemi UML Diyagramı

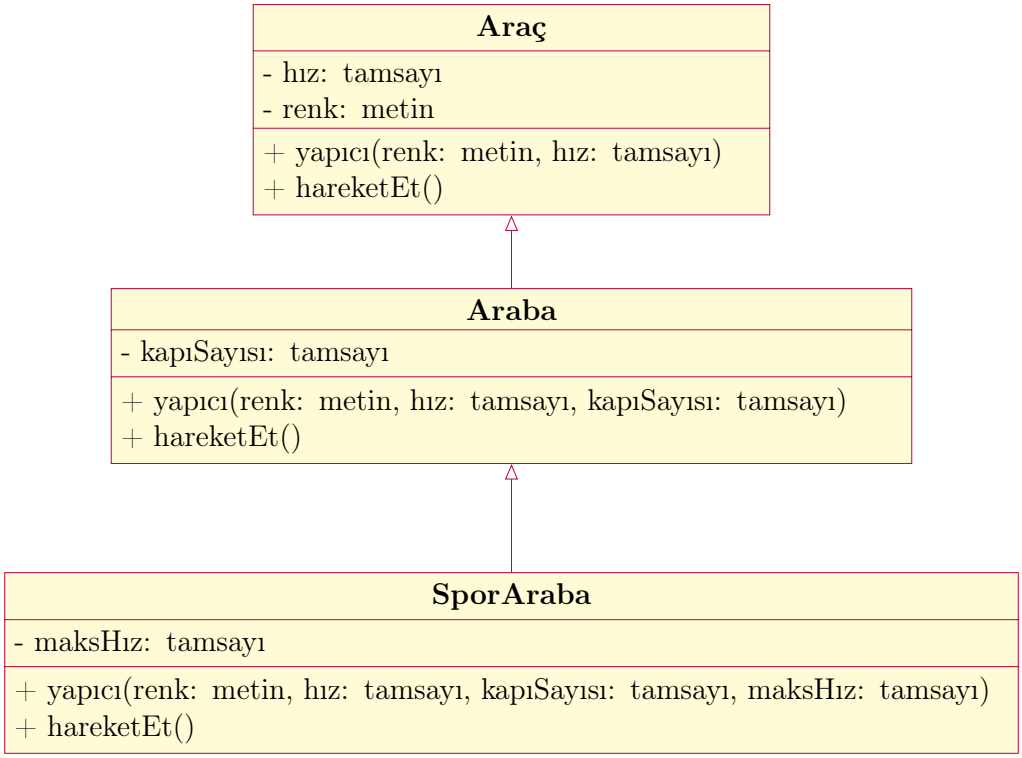
4.4 Kalıtımın Uygulanması

Kalıtım, kod yeniden kullanımını sağlar ve metod geçersiz kılma ile esneklik sunar.

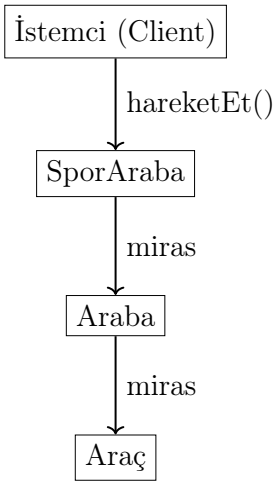
Örnek 1: Araç sınıfı için kalıtımı gösteren bir sözde-kod örneği verilmiştir:

Sözde-kod 4.1 – Kalıtım ile Araç Hiyerarşisi

```
sınıf Araç {
 özel özellik hız: tamsayı
 özel özellik renk: metin

 yapıcı(renk: metin, hız: tamsayı) {
  bu.renk = renk
  bu.hız = hız
 }

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

sınıf Araba miras_alır Araç {
 özel özellik kapıSayısı: tamsayı

 yapıcı(renk: metin, hız: tamsayı, kapıSayısı: tamsayı) {
  super(renk, hız)
  bu.kapıSayısı = kapıSayısı
 }

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

sınıf SporAraba miras_alır Araba {
 özel özellik maksHız: tamsayı

 yapıcı(renk: metin, hız: tamsayı, kapıSayısı: tamsayı, maksHız:
    tamsayı) {
  super(renk, hız, kapıSayısı)
  bu.maksHız = maksHız
 }

 genel metod hareketEt() {
  yaz "Spor araba maksimum ", maksHız, " hızıyla hareket ediyor"
 }
}

ana_program {
 araba = yeni Araba("Kırmızı", 100, 4)
 sporAraba = yeni SporAraba("Mavi", 150, 2, 300)
 araba.hareketEt()      // Çıktı: Araba 4 kapıyla hareket ediyor,
    hız: 100
 sporAraba.hareketEt()  // Çıktı: Spor araba maksimum 300 hızıyla
    hareket ediyor
}
```



Şekil 4.6 – Araç Hiyerarşisi UML Diyagramı



Şekil 4.7 – Kalıtım: Metod Çağrı Zinciri

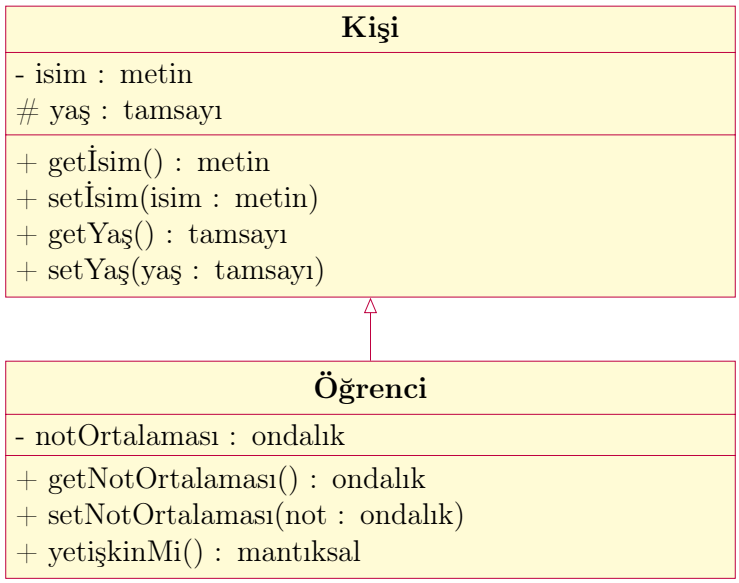
Örnek 2. Kapsülleme ve Kalıtım özelliklerinin kullanıldığı örneği verelim.
Adımlar:

- **Sınıf Yapısının Belirlenmesi**: Temel sınıf (*Kişi*) ve miras alan alt sınıf (*Öğrenci*) tanımlanır. Bu, kalıtım mekanizmasını kullanarak protected erişimin faydasını gösterir.
- **Özelliklerin Erişim Seviyelerinin Atanması**: Hassas veriler (örneğin, isim) private olarak tanımlanır, böylece yalnızca sınıf içi erişime izin verilir. Kalıtım için paylaşılması gereken veriler (örneğin, yaş) protected olarak tanımlanır, böylece alt

sınıflar erişebilir. Alt sınıfa özgü veriler (örneğin, not ortalaması) private tutulur.

- **Metodların Tanımlanması**: Public getter ve setter metodları ile kontrollü erişim sağlanır. Bu metodlarda doğrulama kuralları (örneğin, yaşın negatif olmaması) eklenir.
- **Kalıtım ve Protected Erişim**: Alt sınıfta protected özelliklere doğrudan erişilerek (örneğin, yaş kullanarak yetişkinlik kontrolü) kapsüllemenin esnekliği gösterilir.
- **Veri Gizleme ve Kontrol**: Doğrudan erişim engellenerek, veri tutarlılığı korunur.

Aşağıda, bu örneğe ait UML diyagramı ve sözde-kod verilmiştir:



Şekil 4.8 – Kişi ve Öğrenci Sınıflarının Kapsülleme ile UML Diyagramı

Sözde-kod 4.2 – Kapsülleme ile Kişi ve Öğrenci Sınıfları

```
sınıf Kişi {
 özel özellik isim: metin
 korumalı özellik yaş: tamsayı
 yapıcı(isim: metin, yaş: tamsayı) {
  bu.isim = isim
  bu.yaş = yaş
 }

 genel metod getİsim(): metin {
  döndür isim
 }

 genel metod setİsim(yeniİsim: metin) {
  isim = yeniİsim
 }

 genel metod getYaş(): tamsayı {
  döndür yaş
 }

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

sınıf Öğrenci miras_al Kişi {
 özel özellik notOrtalaması: ondalık
 yapıcı(isim: metin, yaş: tamsayı, notOrtalaması: ondalık) {
  üst.yapıcı(isim, yaş)
  bu.notOrtalaması = notOrtalaması
 }

 genel metod getNotOrtalaması(): ondalık {
  döndür notOrtalaması
 }

 genel metod setNotOrtalaması(yeniNot: ondalık) {
  eğer yeniNot >= 0 ve yeniNot <= 100 ise
  notOrtalaması = yeniNot
  değilse
  hata "Geçersiz not değeri"
 }

 genel metod yetişkinMi(): mantıksal {
  döndür yaş >= 18  // Protected yaş'a alt sınıftan erişim
 }
}

ana_program {
 ogrenci = yeni Öğrenci("Ali", 20, 85.5)
 yaz "İsim: ", ogrenci.getİsim()
 yaz "Yaş: ", ogrenci.getYaş()
 yaz "Not Ortalaması: ", ogrenci.getNotOrtalaması()
 yaz "Yetişkin mi? ", ogrenci.yetişkinMi()
}
```

4.5 Kalıtımın Avantajları ve Zorlukları

Kalıtımın avantajları ve zorlukları, nesne tabanlı programlamada tasarım kararlarını etkileyen önemli faktörlerdir.

4.5.1 Avantajlar

- **Kod Yeniden Kullanımı**: Kalıtım, üst sınıfın özelliklerini ve metodlarını alt sınıfların tekrar yazmadan kullanmasını sağlar, böylece kod tekrarını azaltır ve geliştirme sürecini hızlandırır. Örneğin, bir *Hayvan* üst sınıfında tanımlı *beslen* metodu, *Kedi* ve *Köpek* alt sınıfları tarafından doğrudan kullanılabilir, böylece her alt sınıf için bu metodu yeniden yazmaya gerek kalmaz. Bu, özellikle büyük projelerde, ortak işlevselliğin merkezi bir yerde toplanmasını sağlayarak kod tabanını daha düzenli hale getirir. Ayrıca, üst sınıfta yapılan bir güncelleme, tüm alt sınıflara otomatik olarak yansır, bu da bakım süreçlerini kolaylaştırır. Örneğin, Java'daki *java.util* paketinde *Collection* sınıfı, *List* ve *Set* gibi alt sınıflara ortak metodlar sağlar.
- **Hiyerarşik Tasarım**: Kalıtım, gerçek dünya nesneleri arasındaki ilişkileri modellemek için güçlü bir araçtır. Nesneler arasındaki "is-a" (bir türdür) ilişkisini doğal bir şekilde temsil eder. Örneğin, bir *Araba* sınıfı, *Araç* sınıfının bir türü olarak modellenebilir ve bu hiyerarşi, gerçek dünyadaki araç türlerini (araba, kamyon, motosiklet) yansıtır. Bu, kodun daha sezgisel ve anlaşılır olmasını sağlar. Ayrıca, hiyerarşik tasarım, biyoloji gibi alanlarda sınıflandırma sistemlerini (örneğin, memeliler → kedigiller → kedi) modellemek için idealdir. Bu yapı, geliştiricilerin karmaşık sistemleri daha organize bir şekilde tasarlamasına olanak tanır ve yazılımın okunabilirliğini artırır.
- **Esneklik**: Kalıtım, metod geçersiz kılma (overriding) ve polimorfizm aracılığıyla alt sınıfların üst sınıf davranışlarını özelleştirmesine olanak tanır. Örneğin, *Araç* sınıfındaki *hareketEt* metodu, *Araba* sınıfında kapı sayısına bağlı olarak, *SporAraba* sınıfında ise maksimum hıza bağlı olarak farklı şekilde uygulanabilir. Bu, aynı arayüzü korurken farklı davranışlar tanımlamayı mümkün kılar. Polimorfizm sayesinde, bir üst sınıf referansı kullanılarak farklı alt sınıf nesneleri üzerinde aynı metod çağrılabilir, bu da kodun genelleştirilmesini ve yeniden kullanılabilirliğini artırır. Örneğin, bir *Hayvan* listesi, *Kedi* ve *Köpek* nesnelerini tutabilir ve her biri kendi *sesÇıkar* metodunu çağırabilir.

4.5.2 Zorluklar

- **Karmaşıklık**: Çoklu kalıtım, özellikle diamond problem gibi sorunlara yol açabilir ve kodun anlaşılmasını zorlaştırabilir. Diamond problemi, bir alt sınıfın birden fazla üst sınıftan miras alması ve bu üst sınıfların ortak bir atadan gelmesi durumunda

ortaya çıkar (örneğin, Şekil 4.5). Bu durumda, hangi üst sınıfın metodunun veya özelliğinin kullanılacağı belirsizleşebilir. Örneğin, bir *UçanAraba* sınıfı hem *Araba* hem de *Uçak* sınıflarından miras alırsa ve her ikisi de aynı isimde bir *hareketEt* metodu tanımlıyorsa, bu çakışma kodun davranışını öngörülemez hale getirebilir. Bu tür karmaşıklıklar, özellikle büyük projelerde hata ayıklamayı ve bakımı zorlaştırır. Çözüm olarak, C++ sanal kalıtım sunarken, Java gibi diller arayüzler kullanarak bu sorunu önler.

- **Bağımlılık**: Kalıtım, alt sınıfları üst sınıfa sıkı sıkıya bağlar, bu da bağımlılık sorunlarına yol açabilir. Üst sınıfta yapılan bir değişiklik, tüm alt sınıfları etkileyebilir ve bu, özellikle büyük ve karmaşık hiyerarşilerde sorun yaratabilir. Örneğin, *Araç* sınıfındaki *hız* özelliğinin veri türü tamsayıdan ondalıklı sayıya değiştirilirse, *Araba* ve *SporAraba* sınıflarında bu değişikliğe uyum sağlamak için ek düzenlemeler gerekebilir. Bu durum, kodun bakımını zorlaştırabilir ve "kırılgan taban sınıf" (fragile base class) sorununa yol açabilir. Kompozisyon, bu tür bağımlılıkları azaltmak için sıklıkla tercih edilir, çünkü nesneler arasındaki ilişkiler daha gevşek bağlanır.
- **Kötü Kullanım**: Kalıtımın yanlış veya aşırı kullanımı, gereksiz derin hiyerarşilere ve karmaşık kod yapılarına yol açabilir. Örneğin, bir sınıf hiyerarşisi çok fazla seviye içeriyorsa kodun anlaşılması ve bakımı zorlaşır:

  (örneğin, *Araç* → *Araba* → *SporAraba* → *YarışArabası* → *Formula1Arabası*).

  Ayrıca, kalıtımın gereksiz yere kullanılması, kompozisyon gibi daha uygun alternatiflerin göz ardı edilmesine neden olabilir. Örneğin, bir *UçanAraba* sınıfı, *Araba* ve *Uçak* sınıflarından miras almak yerine, bu sınıfların davranışlarını kompozisyon yoluyla birleştirebilir. Kötü kullanım, kodun esnekliğini azaltabilir ve gelecekteki değişiklikleri zorlaştırabilir.

4.6 Modern Bağlamda Kalıtım

Kalıtım, modern yazılım mühendisliğinde tasarım desenlerinde sıkça kullanılır:

- Template Method: Üst sınıf, bir algoritmanın iskeletini tanımlar; alt sınıflar detayları özelleştirir.
- Strategy: Kalıtım yerine kompozisyon tercih edilse de, hiyerarşik davranışlar için kalıtım kullanılabilir.
- Kütüphaneler ve Çerçeveler: Örneğin, Java'daki Swing veya .NET'teki WPF, kalıtımı yoğun bir şekilde kullanır.

4.7 Alıştırmalar

1. Kalıtımın tanımını ve avantajlarını kendi kelimelerinizle açıklayın.
2. Bir *Çalışan* üst sınıfı ve *Yönetici* alt sınıfı için sözde-kod yazın. Yönetici sınıfına maaş bonusu gibi bir özellik ekleyin ve *maaşHesapla* metodunu geçersiz kılın.
3. Şekil 4.3'deki UML diyagramını inceleyin ve başka bir hiyerarşi (örneğin, Hayvan → Kedi → Kaplan) için benzer bir UML diyagramı çizin.
4. Çoklu kalıtımın avantajlarını ve diamond problem gibi zorluklarını tartışın.
5. Kalıtımın bir tasarım deseninde (örneğin, Template Method) nasıl kullanıldığını araştırın ve bir örnekle açıklayın.