Bölüm 5

Çok Biçimlilik

5.1 Giriş

Çok biçimlilik (polymorphism), nesne tabanlı programlamanın (OOP) temel prensiplerinden biridir ve aynı metodun farklı sınıflarda farklı davranışlar sergilemesini sağlar. Bu, kodun esnekliğini ve genelleştirilmesini artırır. Bu bölümde, çok biçimliliğin tanımı, türleri, avantajları, zorlukları ve modern yazılım mühendisliğindeki rolü ele alınacaktır.

5.2 Çok Biçimlilik Nedir?

Çok biçimlilik, bir nesnenin farklı bağlamlarda farklı davranışlar sergileyebilmesini ifade eder. Temel olarak, aynı arayüzün (metod adı) farklı implementasyonlara (davranışlara) sahip olmasını sağlar. Örneğin, bir *Hayvan* sınıfının *sesÇıkar* metodu, *Kedi* sınıfında "miyav" üretirken, *Köpek* sınıfında "hav" üretir. Bu, nesnelerin "birçok biçime" (poly-morph) girebilmesini sağlar ve OOP'nin soyutlama prensibiyle yakından ilişkilidir. Çok biçimlilik, kalıtım veya arayüzler (interfaces) aracılığıyla uygulanır.

OOP açısından çok biçimlilik neden gereklidir? OOP'de, kodun bakımını kolaylaştırmak ve genişletilebilirliği artırmak için SOLID prensiplerinden "Açık-Kapalı Prensibi" (Open-Closed Principle) kritik öneme sahiptir: Yazılım, yeni özelliklere açık olmalı ancak mevcut koda kapalı kalmalıdır. Çok biçimlilik, mevcut kodun değiştirilmeden yeni sınıflarla genişletilmesini sağlar. Örneğin, bir hayvan simülasyonu sisteminde yeni bir hayvan türü eklemek için yalnızca yeni bir alt sınıf oluşturmak yeterlidir; ana döngü veya istemci kodu değişmez. Bu, kodun esnekliğini artırır, hata riskini azaltır ve büyük ölçekli sistemlerde bakım maliyetini düşürür. Ayrıca, soyutlama sayesinde kodun okunabilirliği ve test edilebilirliği iyileşir; istemciler, somut sınıflardan ziyade soyut arayüzlerle çalışır.

OOP'da çok biçimliliği nasıl kullanmalıyız?

— **Kalıtım ve Overriding ile**: Üst sınıfı soyut tutun ve alt sınıflarda metodları geçersiz kılın. Bu, hiyerarşik ilişkilerde idealdir.

— **Arayüzler/İnterface'ler ile**: Farklı hiyerarşilerden sınıfları birleştirin (örneğin, "Uçabilen" arayüzü hem kuş hem uçak için). Bu, gevşek bağlanmayı (loose coup-

ling) teşvik eder.
— **Overloading ile**: Aynı sınıf içinde parametre çeşitliliği sağlayın, ancak aşırı kullanmayın ki kod karmaşıklaşmasın.
— **İyi Pratiğe Uygun Kullanım**: Her zaman soyut referanslar (üst sınıf veya arayüz) kullanın, somut sınıflara doğrudan bağımlı olmayın. Bu, Dependency Inversion Prensibi'ni destekler.

5.3 Çok biçimliliğin türleri

5.3.1 Statik Polimorfizm Nedir?

Statik polimorfizm, bir programın derleme zamanında (compile-time) hangi metodun çağrılacağının belirlendiği bir polimorfizm türüdür. Bu mekanizma, genellikle metod aşırı yüklenmesi (overloading) ile ilişkilendirilir. Statik polimorfizmde, metod seçimi derleyici tarafından metodun adı ve parametre listesine göre yapılır. Örneğin, aynı isme sahip ancak farklı parametre türleri veya sayıları alan metodlar tanımlandığında, derleyici hangi metodun kullanılacağına karar verir. Bu yaklaşım, kodun esnekliğini artırır ve aynı metod adını farklı bağlamlarda kullanma imkanı sağlar. Statik polimorfizm, performans açısından avantajlıdır çünkü çalışma zamanında ek bir karar alma sürecine gerek kalmaz. Ancak, bu yöntem miras veya dinamik bağlamlar olmadan yalnızca aynı sınıf içinde çalışır.

Örnek: Bir *HesapMakinesi* sınıfında *topla* metodu, hem tamsayılar hem ondalık sayılar için ayrı overload'lar ile tanımlanır. Bu, istemcinin veri tipine göre manuel seçim yapmasını önler ve tip güvenliğini artırır.

5.3.2 Dinamik Polimorfizm Nedir?

Dinamik polimorfizm, bir programın çalışma zamanında (run-time) hangi metodun çağrılacağının belirlendiği bir polimorfizm türüdür. Bu, genellikle metod geçersiz kılma (overriding) ile uygulanır ve nesne yönelimli programlamanın miras (inheritance) özelliğine dayanır. Dinamik polimorfizmde, bir üst sınıf tipindeki bir referans, alt sınıf nesnelerine işaret edebilir ve çalışma zamanında alt sınıfın metodları çağrılır. Örneğin, bir üst sınıfta tanımlı bir metod, alt sınıfta yeniden tanımlanabilir ve bu metodun hangi versiyonunun çalışacağı, nesnenin gerçek tipine bağlı olarak çalışma zamanında belirlenir. Bu, kodun daha esnek ve genelleştirilebilir olmasını sağlar, ancak çalışma zamanında metod çözümlenmesi nedeniyle bir miktar performans maliyeti olabilir.

Örnek: Bir *Hayvan* dizisinde farklı hayvan nesneleri varsa, *sesÇıkar* çağrısı her nesnenin kendi alt sınıf metodunu çalıştırır. Bu, dinamik davranışlar için esneklik sağlar ancak hafif bir performans maliyeti getirir.

5.4 OOP'ta Overloading ve Overriding Arasındaki Farklar

Nesne Yönelimli Programlama (OOP) kavramlarında **overloading** (aşırı yükleme) ve **overriding** (geçersiz kılma), farklı amaçlarla kullanılan iki önemli mekanizmadır.

1. Overloading (Aşırı Yükleme)

Aynı sınıf içinde, aynı isme sahip ancak farklı parametre listelerine (farklı sayıda veya türde parametre) sahip birden fazla metodun tanımlanmasıdır.

- **Amaç**: Aynı işlevi farklı parametrelerle gerçekleştirmek için esneklik sağlamak.
- **Zaman**: Derleme zamanında (compile-time) çözümlenir (statik polimorfizm).
- **Özellikler**:
  - Metodun adı aynı olmalıdır.
  - Parametrelerin sayısı, türü veya sırası farklı olmalıdır.
  - Dönüş türü farklı olabilir, ancak yalnızca dönüş türü farklıysa overloading geçerli değildir.
  - Genellikle aynı sınıf içinde tanımlanır.

2. Overriding (Geçersiz Kılma)

Bir alt sınıfın (subclass), üst sınıftan (superclass) miras aldığı bir metodu yeniden tanımlayarak kendi davranışını uygulamasıdır.

- **Amaç**: Üst sınıftaki bir metodu alt sınıfta özelleştirmek veya değiştirmek.
- **Zaman**: Çalışma zamanında (run-time) çözümlenir (dinamik polimorfizm).
- **Özellikler**:
  - Metodun adı, parametre listesi ve dönüş türü üst sınıftakiyle aynı olmalıdır.
  - Genellikle `@Override` anotasyonuyla işaretlenir (örneğin, Java'da).
  - Sadece miras alma (inheritance) durumunda gerçekleşir.
  - Üst sınıftaki metodun erişim seviyesi kısıtlanamaz.

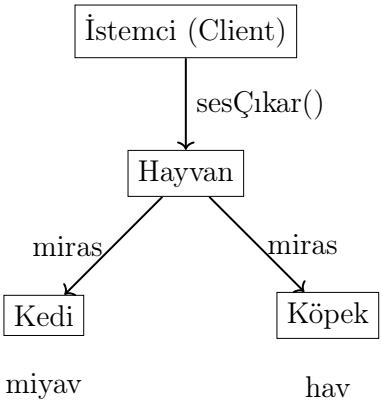
Özellik	**Overloading**	**Overriding**
Tanım	Aynı isme sahip farklı parametrelerle metodlar tanımlama	Üst sınıftaki metodu alt sınıfta yeniden tanımlama
Yer	Aynı sınıf içinde	Alt sınıf ve üst sınıf arasında
Zaman	Derleme zamanı (compile-time)	Çalışma zamanı (run-time)
Polimorfizm	Statik polimorfizm	Dinamik polimorfizm
Parametreler	Farklı olmalı	Aynı olmalı
Dönüş Türü	Farklı olabilir	Aynı veya kovaryant olmalı
Miras	Gerekmez	Gerekir

Tablo 5.1 – Overloading ve Overriding Arasındaki Farklar

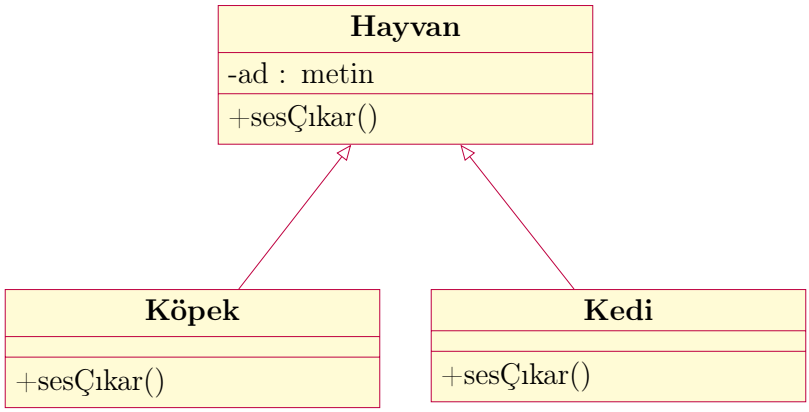
5.5 Çok Biçimliliğin Uygulanması

Çok biçimlilik, kalıtım veya arayüzler aracılığıyla uygulanır.

Çalışma zamanı çok biçimliliği, genellikle kalıtım ve metod geçersiz kılma ile sağlanır. Aşağıda, çok biçimliliği gösteren bir örneğin UML şeması ve sözde-kod örneği verilmiştir:



Şekil 5.1 – Çok Biçimlilik: Metod Çağrı Akışı

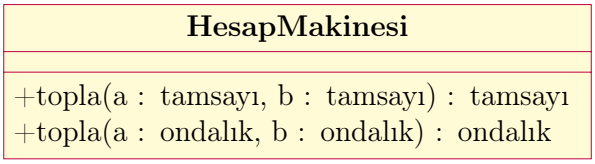


Şekil 5.2 – Hayvan Hiyerarşisinin UML Diyagramı (Metodu Geçersiz Kılma)

Sözde-kod 5.1 – Çok Biçimlilik ile Hayvan Hiyerarşisi

```
sınıf Hayvan {
 özel özellik ad: metin

 yapıcı(ad: metin) {
  bu.ad = ad
 }

 genel metod sesÇıkar() {
  yaz "Genel hayvan sesi"
 }
}

sınıf Kedi miras_alır Hayvan {
 yapıcı(ad: metin) {
  super(ad)
 }

 genel metod sesÇıkar() {
  yaz ad, " miyav diyor"
 }
}

sınıf Köpek miras_alır Hayvan {
 yapıcı(ad: metin) {
  super(ad)
 }

 genel metod sesÇıkar() {
  yaz ad, " hav diyor"
 }
}

ana_program {
 hayvanlar = [yeni Kedi("Minnoş"), yeni Köpek("Karabaş")]
 her hayvan için hayvanlar içinde {
  hayvan.sesÇıkar()
 }
 // Çıktı:
 // Minnoş miyav diyor
 // Karabaş hav diyor
}
```

Derleme zamanı çok biçimliliği için bir örneğin UML şeması ve sözde-kodu verilmiştir:



Şekil 5.3 – HesapMakinesi Sınıfının UML Diyagramı (Metod Aşırı Yükleme)

Sözde-kod 5.2 – Metod Aşırı Yükleme

```
sınıf HesapMakinesi {
 genel metod topla(a: tamsayı, b: tamsayı) {
  döndür a + b
 }

 genel metod topla(a: ondalık, b: ondalık) {
  döndür a + b
 }
}

ana_program {
 hesap = yeni HesapMakinesi()
 yaz hesap.topla(5, 3)      // Çıktı: 8
 yaz hesap.topla(5.5, 3.2)  // Çıktı: 8.7
}
```

5.6 Çok Biçimliliğin Avantajları ve Zorlukları

Avantajları:

— **Esneklik**: Aynı arayüz, farklı davranışlar için kullanılabilir. Bu, sistemin genişletilebilirliğini artırır; örneğin, bir grafik kütüphanesinde *çiz* metodu, farklı şekiller (daire, kare) için farklı implementasyonlar sunar ve yeni şekiller eklemek istemci kodunu etkilemez. Bu, plugin mimarileri için idealdir ve yazılımın evrimini kolaylaştırır.

— **Genelleştirme**: Kod, üst sınıf referanslarıyla farklı alt sınıfları işleyebilir. Bu, Liskov Değiştirme Prensibi'ni (Liskov Substitution Principle) destekler; bir üst sınıf referansı, alt sınıf nesneleriyle değiştirilebilir. Örneğin, bir *Hayvan* dizisi, farklı hayvan türlerini tutabilir ve aynı döngüde hepsini işleyebilir, böylece kod tekrarını önler ve soyut düşünmeyi teşvik eder.

— **Kod Yeniden Kullanımı**: Ortak davranışlar üst sınıfta tanımlanır ve alt sınıflarda özelleştirilir. Bu, kalıtımla birleştiğinde, ortak kod bloklarının tekrar yazılmasını engeller. Örneğin, bir araç simülasyonunda *hareketEt* metodu üst sınıfta temel mantığı içerirken, alt sınıflar (araba, bisiklet) bunu genişletir, böylece bakım tek bir yerden yapılır.

Zorlukları:

— **Karmaşıklık**: Çalışma zamanı çok biçimliliği, hata ayıklamayı zorlaştırabilir. Nesne türü runtime'da belirlendiği için, derleme zamanı hataları yerine runtime hataları oluşabilir. Örneğin, bir metod çağrısı beklenmedik bir alt sınıf implementasyonuna yönlendirilirse, beklenmedik davranışlar ortaya çıkabilir ve bu, debug sürecini uzatır.

— **Performans**: Sanal metod çağrıları (virtual methods) ek yük getirebilir. vtable lookup işlemi, doğrudan metod çağrılarından daha yavaştır; özellikle sık çağrılan metodlarda (örneğin, oyun döngülerinde) bu birikerek performansı etkileyebilir.

Modern derleyiciler optimizasyon yapsa da, yüksek performanslı sistemlerde dikkatli kullanılmalıdır.

— **Kötü Kullanım**: Gereksiz çok biçimlilik, kodu karmaşıklaştırabilir. Her şeyi soyutlaştırmak, gereksiz hiyerarşiler yaratır ve "over-engineering"e yol açar. Örneğin, basit bir sayaç sınıfı için bile kalıtım ve overriding kullanmak, kodu şişirir ve yeni geliştiricilerin anlamasını zorlaştırır; bu durumda kompozisyon (composition) daha uygun olabilir.

5.7 Modern Bağlamda Çok Biçimlilik

Çok biçimlilik, modern yazılım mühendisliğinde tasarım desenlerinde ve dinamik sistemlerde yaygın olarak kullanılır:

— Strategy Deseni: Algoritmalar, çok biçimlilikle değiştirilebilir.

— Factory Deseni: Nesne oluşturma, çok biçimlilikle genelleştirilir.

— Dinamik Sistemler: Örneğin, oyun motorlarında (Unity), farklı nesneler aynı arayüzü kullanarak farklı davranışlar sergiler.

5.8 Alıştırmalar

1. Çok biçimliliğin tanımını ve türlerini kendi kelimelerinizle açıklayın.
2. Bir *Şekil* üst sınıfı ve *Daire*, *Dikdörtgen* alt sınıfları için sözde-kod yazın. Her sınıf, *alanHesapla* metodunu geçersiz kılsın.
3. Şekil 5.2 deki UML diyagramını inceleyin ve başka bir hiyerarşi (örneğin, Araç → Bisiklet, Motosiklet) için benzer bir UML diyagramı çizin.
4. Derleme zamanı ve çalışma zamanı çok biçimliliği arasındaki farkları tartışın ve birer örnek verin.