# **nterface Segregation Prensibi Nedir ? (Kod örneğiyle) — SOLID**

*Sorumlulukların hepsini tek bir arayüze toplamak yerine daha özelleştirilmiş birden fazla arayüz oluşturmayı tercih etmemizi söyleyen prensiptir.*



Interface Segregation Prensibi

Tek bir interface yerine kullanımlarına göre parçalanmış birden fazla interface ile işlemleri yürütmeliyiz. Yani her farklı sorumluluğun kendine özgü bir arayüzü olması gerekmektedir. Böylece interface’i kullanan kişide sadece ihtiyacı olanlarla ilgilenmiş olur. Birden fazla amaç için yalnızca bir arayüzümüz var ise buna gerektiğinden fazla method ya da özellik ekliyoruz demektir, bu da IS prensibine aykırı davrandığınız anlamına gelir..

*Nesneler asla ihtiyacı olmayan property/metot vs içeren interfaceleri implement etmeye zorlanmamalıdır*

Görüldüğü gibi single responsibility ve interface segregation prensipleri birbirine oldukça yakın ve aynı amaca hizmet eden prensiplerdir. Tek fark ise Interface segregation arayüz(interface)ler ile ilgilenirken, Single responsibility sınıflarla ilgilenmektedir.

Bir methodun IS prensibine uymayan interface’e eklenmesi veya bir method üzerindeki değişiklik, Bu interface’i kullanan(implement etmiş) tüm sınıfları değiştirmemizi gerektirir. Bir arayüz yazarken, farklı sorumlulukları ya da farklı davranış gruplarını her zaman farklı arayüzlere ayırmak önemlidir.

Interface segregation prensibine uymayan bir örnekle başlayabiliriz. *Animal* interface’ini ele alalım ve hayvanlara ait özellikleri barındırsın. Uçmak, koşmak, havlamak gibi özellikler olduğunu düşünelim. (Animal kullanılmayan bir yazı dizisi eksik olurdu diye düşünüyorum :) )

Animal interface olarak aşağıda yer alan üç özelliği birden barındıran bir nesne(hayvan) varsa onun için çok güzel bir interface oldu. Ancak diğer hayvanları da düşününce interface’imizin parçalanmaya ihtiyaç duyduğunu görebilirsiniz.

public interface Animal {

void fly(); void run(); void bark();

}

Kuşlar uçar ve koşar, ancak havlayamazlar diye varsayabiliriz . Havlamayı boşuna implement etmiş olduk.

public class Bird implements Animal {

public void bark() {

/\* do nothing \*/

} public void run() {

System.out.println("Koşan kuş");

} public void fly() {

System.out.println("Uçan kuş");

}

}

Kediler koşar, ama uçup, havlayamazlar. Havlamayı kedi için boşuna implement etmiş olduk.

public class Cat implements Animal {

public void fly() {

/\* do nothing \*/

} public void bark() {

/\* do nothing \*/

} public void run() {

System.out.println("Koşan kedi");

// logic

}

}

Köpekler koşar, havlar ama uçamazlar. Uçmayı köpek için boşuna implement etmiş olduk.



Köpekler uçamaz mı?

public class Dog implements Animal {

public void fly() {

/\* do nothing \*/

} public void bark() {

System.out.println("Havlayan köpek.");

// logic

} public void run() {

System.out.println("Koşan köpek.");

// logic

}

}

Böyle bir yapı okunabilirliği azaltır. Gerek duyulmayan metotları implement etmek zorunda kaldık ve bakımı zor bir hal almış oldu.

Hadi güzelleştirelim. Burada yapılması gereken; her temel özelliğe ayrı bir interface yaratarak Interface Segregation sınıfına uygun hale getirmektir.

Koşabilme, Uçabilme, Havlayabilme özelliklerini kendi metotlarını içeren şekilde ayrı interfaceler olarak tanımlayalım.

public interface Flyable {

void fly();

}public interface Runnable {

void run();

}public interface Barkable {

void bark();

}

Kuş için uçabilme ve koşabilme özelliklerini kazandırmak istediğimizde Flyable ve Runnable interfacelerini implement etmemiz yeterli olacaktır. Havlama özelliğini boşu boşuna implement etmek zorunda kalmadık.

public class Bird implements Flyable, Runnable {

public void run() {

System.out.println("Kuş,Koşuyorum");

//logic

} public void fly() {

System.out.println("Kuş, Uçuyorum.");

//logic

}

}

Kedi için koşabilme özelliğini kazandırmak istediğimizde Runnable interface’ini implement etmemiz yeterli olacaktır. Havlama ve uçabilme özelliklerini implement etmek zorunda kalmadık.

public class Cat implements Runnable {

public void run() {

System.out.println("Kedi,Koşuyorum");

//logic

}

}

Köpek için havlama ve koşabilme özelliklerini kazandırmak istediğimizde Barkable ve Runnable interfacelerini implement etmemiz yeterli olacaktır. Uçma özelliğini implement etmek zorunda kalmadık.

public class Dog implements Runnable, Barkable {

public void bark() {

System.out.println("Köpek,Havlıyorum.");

//logic

} public void run() {

System.out.println("Köpek,Koşuyorum.");

//logic

}

}

Böylece her yeni özellik için bir interface daha yazarak, özelliği barındıran sınıflara implement edebiliriz. Her sınıf, her hayvan kendi yapabildiği özelliği implement ederek, ihtiyaç duymadığı özelliklerden arındırılmış oldu. Kod okunurluğu artarken, esneklikte kazandırmış olduk. Yeni eklenecek olan herhangi bir class için, sadece ihtiyaç duydugu interfaceleri kullanabilmesini sağlamış olduk.

Interface segregation prensibine uygun olarak geliştirdiğimiz arayüzlerde implement eden sınıflar ihtiyaç duymadıkları metodları uygulamak(implement etmek) zorunda kalmaz böylece daha küçük arayüzler işleri kolaylaştırır. Bunun yanında birden fazla interface kodun okunabilirliğini artırırken, bakımını da kolaylaştırmaktadır. Böylece bir interface sadece ilgilendiği ve gerekli olan metodları barındırır.

SOLID prensiplerine devam etmek için son SOLID prensibi olan **Dependency Inversion** **Prensibi**ne göz atabilirsiniz;