**SOLID Yazılım Prensipleri**

SOLID, yazılım geliştirme sürecinde daha okunabilir, sürdürülebilir, esnek ve hataya karşı dayanıklı kodlar yazabilmek için kullanılan beş temel prensibi ifade eder. Bu prensipler, özellikle nesne yönelimli programlama (OOP) yaklaşımıyla geliştirme yapılan projelerde büyük önem taşır. SOLID prensipleri ilk kez Robert C. Martin (Uncle Bob) tarafından ortaya atılmıştır.

SOLID kısaltması şu prensipleri temsil eder:

* **S** – Single Responsibility Principle (Tek Sorumluluk)
* **O** – Open/Closed Principle (Açık/Kapalı)
* **L** – Liskov Substitution Principle (Yerine Geçebilirlik)
* **I** – Interface Segregation Principle (Arayüz Ayrımı)
* **D** – Dependency Inversion Principle (Bağımlılıkların Ters Çevrilmesi)

**Avantajları**

1. **Bakımı Kolay Kod**: Her sınıfın görevi belli olduğu için karmaşıklık azalır.
2. **Test Edilebilirlik Artar**: Prensipler sayesinde birim testler daha etkili yazılır.
3. **Yeniden Kullanılabilirlik**: Modüler yapılar, aynı bileşeni farklı yerlerde kullanmayı kolaylaştırır.
4. **Esneklik ve Genişletilebilirlik**: Yeni özellikler eklerken mevcut kodu bozma riski azalır.
5. **Takım İçi Uyumluluk**: Temiz yapı, ekip çalışmalarını kolaylaştırır.

**Dezavantajları**

1. **Başlangıçta Fazladan Zaman Gerektirir**: Kodun mimari olarak doğru yapılandırılması zaman alabilir.
2. **Küçük Projelerde Aşırı Karmaşıklık**: Basit uygulamalarda SOLID uygulamak gereksiz soyutlamalara neden olabilir.
3. **Öğrenme Eğrisi**: SOLID prensiplerini anlamak ve uygulamak, yeni başlayanlar için zor olabilir.
4. **Fazla Sınıf ve Arayüz**: Gereksiz bölünmelerle dosya sayısı artabilir.

**Nerelerde Kullanılır?**

* **Kurumsal büyük ölçekli projeler**
* **Uzun vadeli bakımı yapılacak sistemlerde**
* **Takım olarak geliştirilen projelerde**
* **Modüler mimariler (Clean Architecture, Hexagonal Architecture)**
* **Test güdümlü geliştirme (TDD) yaklaşımında**

**Kullanılmazsa Ne Olur?**

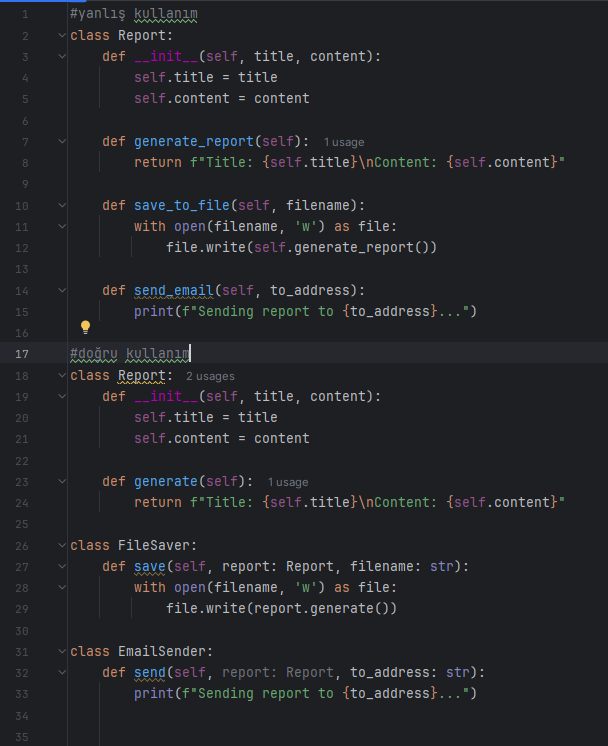
* Kod **dağınık ve karmaşık** hale gelir.
* Bir değişiklik tüm sistemi **bozabilir**.
* Test yazmak **zorlaşır**.
* Kod parçaları arasında **yüksek bağımlılık** oluşur.
* Ekip içi geliştirmeler **çakışabilir** ve hata oranı artar.
* Kodun **okunabilirliği düşer**, yeni geliştiriciler anlamakta zorlanır.

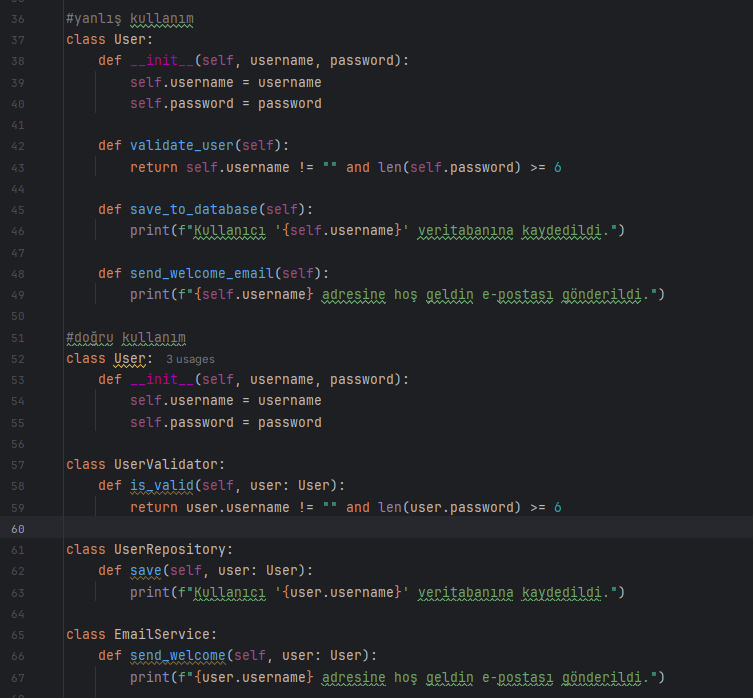
**Her Prensibin Uygulama Etkisi**

| **Prensip** | **Kullanılmazsa Ne Olur** | **Sağladığı Faydalar** |
| --- | --- | --- |
| **SRP** | Sınıf karmaşıklaşır | Kod okunabilirliği ve değiştirilebilirlik artar |
| **OCP** | Her yeni özellik kodu bozar | Genişlemeye açık hale gelir |
| **LSP** | Kalıtım yanlış işler, hata çıkar | Kod yeniden kullanılabilir olur |
| **ISP** | Gereksiz metotlar zorunlu olur | Arayüzler temiz ve işlevseldir |
| **DIP** | Sınıflar birbirine bağımlı olur | Soyutlama ile esneklik sağlanır |

**1. Single Responsibility Principle (Tek Sorumluluk Prensibi)**

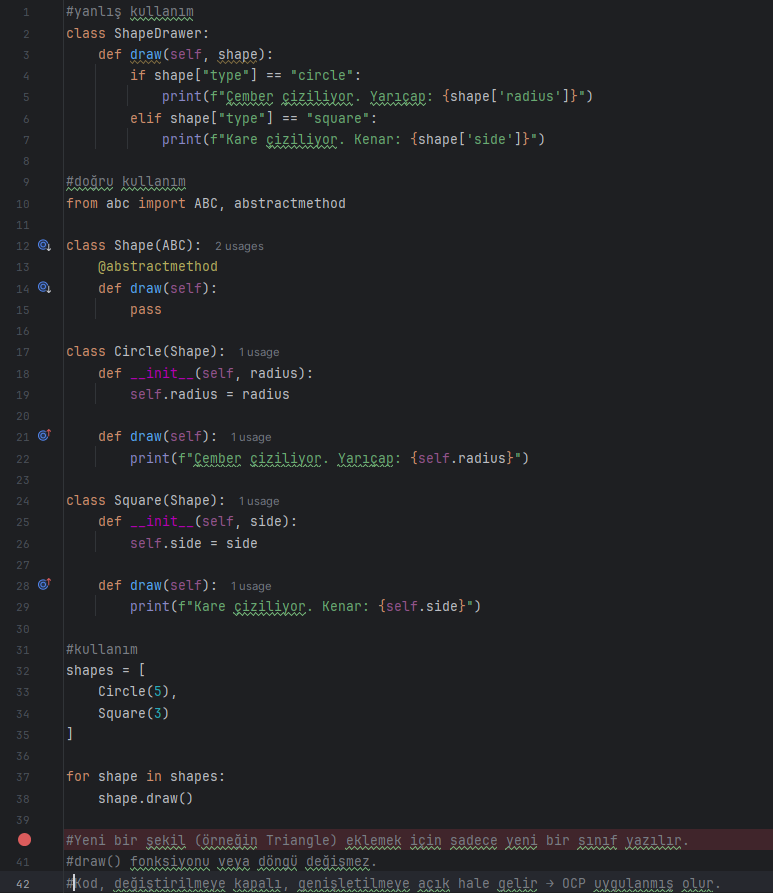
Bir sınıf sadece tek bir işi yapmalı ve yalnızca bir nedenle değiştirilebilir olmalıdır.  
Bu prensip sayesinde sınıfların karmaşıklığı azalır, test edilebilirlik artar.





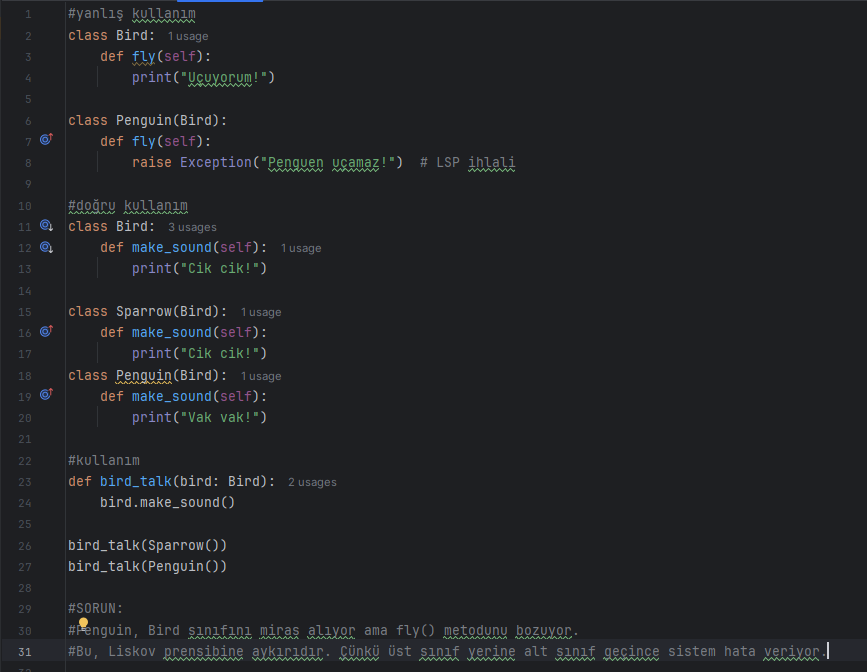
**2. Open/Closed Principle (OCP)**

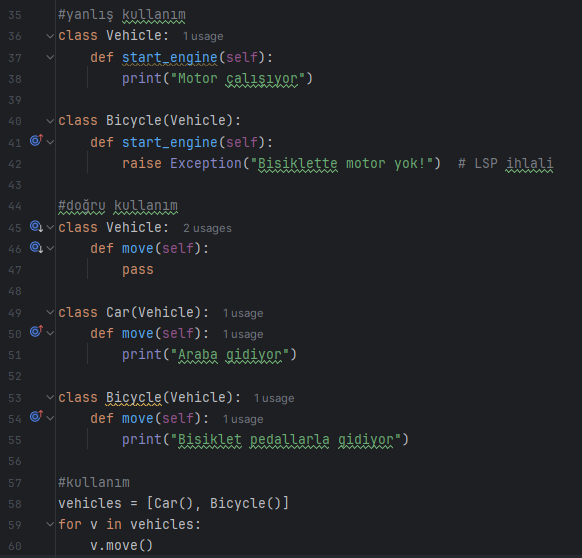
Açık/Kapalı Prensibi**,** yazılım varlıklarının (sınıflar, modüller, fonksiyonlar vb.) genişletmeye açık, ancak değişime kapalı olması gerektiğini belirtir. Bu prensip, yeni davranışların eklenebilmesi için kodun değiştirilmesi yerine genişletilmesi gerektiğini savunur. Yani bir sınıfa yeni özellik eklemek için mevcut kodu değiştirmemeliyiz, onun yerine yeni sınıflar veya alt sınıflar yazarak genişletmeliyiz.



**3. Liskov Substitution Principle (LSP)**

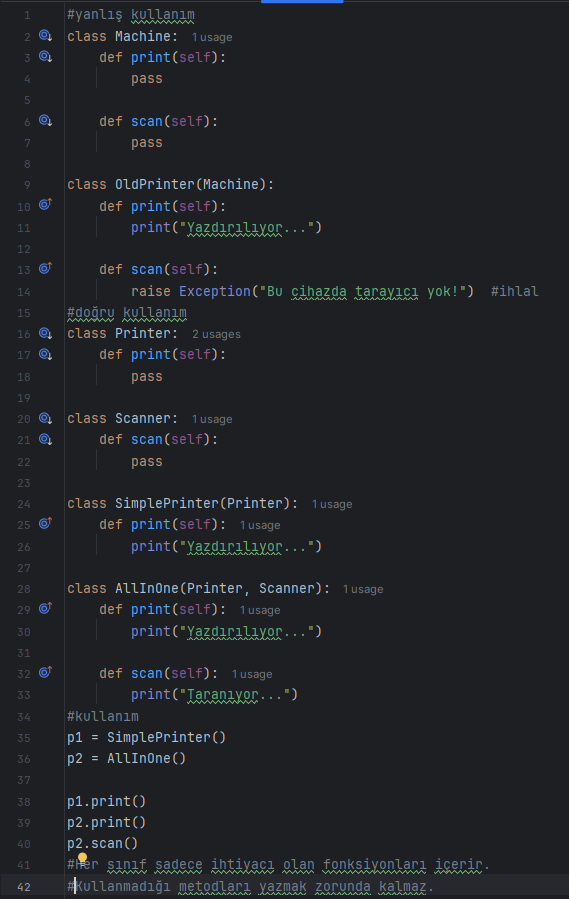
Liskov Yerine Geçme Prensibi, türetilmiş sınıfların, temel sınıfların yerine kullanılabilmesi gerektiğini belirtir. Bu prensip, alt sınıfların, üst sınıfların yerine geçecek şekilde tasarlanmasını ve bu şekilde kullanılabilmesini sağlar.





**4. Interface Segregation Principle (ISP)**

Arayüz Ayırma Prensibi, bir sınıfın, kullanmadığı metotları içeren arayüzleri implemente etmemesi gerektiğini belirtir. Bu prensip, büyük ve genel arayüzler yerine, daha küçük ve spesifik arayüzlerin kullanılmasını savunur.



5. **Dependency Inversion Principle (DIP)**

Bağımlılıkların Tersine Çevrilmesi Prensibi, üst seviye modüllerin, alt seviye modüllere bağımlı olmaması gerektiğini belirtir. Bunun yerine, her iki tür modül de soyutlamalara bağımlı olmalıdır. Bu prensip, bağımlılıkların daha esnek ve sürdürülebilir olmasını sağlar.

