Yazılım Geliştirme Sürecinde SOLID Prensipleri

Yusuf Safa Köksal
Bilgisayar Mühendisliği
Yıldız Teknik Üniversitesi
İstanbul / Türkiye
safa.koksal@std.yildiz.edu.tr

***Abstract*—Bu çalışma, yazılım geliştirme sürecinde SOLID prensiplerinin uygulanmasının önemini ve yazılım tasarımına sağladığı avantajları ele almaktadır. SOLID prensipleri; Tek Sorumluluk Prensibi (Single Responsibility Principle), Açık-Kapalı Prensibi (Open-Closed Principle), Liskov'un Yerine Geçme Prensibi (Liskov Substitution Principle), Arayüz Ayrımı Prensibi (Interface Segregation Principle) ve Bağımlılıkları Tersine Çevirme Prensibi (Dependency Inversion Principle) olmak üzere beş temel prensipten oluşur. Bu prensipler, yazılım tasarımında esneklik, yeniden kullanılabilirlik ve sürdürülebilirliği artırmak amacıyla geliştiricilere bir rehberlik sağlar. Raporda, her bir prensip ayrıntılı olarak incelenmiş ve bu prensiplerin, yazılımın bakım ve geliştirme süreçlerinde nasıl kritik bir rol oynadığı vurgulanmıştır. Sunulan bilgiler, yazılım projelerinde daha sağlam ve ölçeklenebilir sistemler geliştirmek isteyen yazılımcılara yol göstermek amacıyla hazırlanmıştır.**

***Index Terms*—SOLID Prensipleri, Single Responsibility Principle, Open-Closed Principle, Liskov's Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle, Yazılım Tasarımı, Nesneye Yönelik Programlama, Clean Code**

I. Giriş

Yazılım geliştirme sürecinde, sistemlerin artan karmaşıklığı, yazılımcılar için ciddi zorluklar oluşturmuştur. Özellikle SOLID prensiplerinin tanımlanmasından önce, yazılımlar genellikle "karmaşık kod" olarak adlandırılan sorunlu yapılar içeriyordu. Bu tür kodlar; küçük bir değişikliğin tüm sistemi etkilemesi, tekrar eden kod parçaları, sıkı bağımlılıklar ve test veya bakım süreçlerinin zorlukları gibi çeşitli problemleri beraberinde getiriyordu. Bu durum, yazılımların sürdürülebilirliğini ve genişletilebilirliğini önemli ölçüde kısıtlıyordu.

Bu sorunlara çözüm getirmek amacıyla, yazılım mimarisi alanında önde gelen isimlerden Robert Cecil Martin (Uncle Bob), 2000 yılında yayınladığı "Design Principles and Design Patterns" başlıklı makalesinde SOLID prensiplerini ilk kez öne sürmüştür [1]. SOLID, yazılım geliştirme süreçlerinde esneklik, sürdürülebilirlik ve yeniden kullanılabilirlik sağlamak için geliştirilen beş temel prensipten oluşur:

- **Single Responsibility Principle** (Tek Sorumluluk Prensibi)
- **Open-Closed Principle** (Açık-Kapalı Prensibi)
- **Liskov Substitution Principle** (Liskov'un Yerine Geçme Prensibi)
- **Interface Segregation Principle** (Arayüz Ayrımı Prensibi)
- **Dependency Inversion Principle** (Bağımlılık Tersine Çevirme Prensibi)

Bu prensipler, yazılımın daha modüler bir yapıya sahip olmasını sağlayarak, değişikliklere hızlı ve etkili bir şekilde adapte olabilmesine olanak tanır. Ayrıca, SOLID prensipleri, "Clean Code" (Temiz Kod) felsefesiyle doğrudan bağlantılıdır. Clean Code, yazılımların anlaşılabilir, bakımı kolay ve geliştirilebilir olması gerektiğini savunurken, SOLID prensipleri bu hedeflere ulaşmayı sağlayan somut bir rehber sunar. Örneğin, bir sınıfın yalnızca tek bir sorumluluğu olması veya yazılım bileşenlerinin değişime açık ancak mevcut yapıya kapalı olması gerektiği gibi prensipler, temiz kod yazımını destekleyen anahtar unsurlardır [2].

SOLID prensipleri, yazılım dünyasında, özellikle karmaşık sistemlerde karşılaşılan problemlere çözüm sunarak yazılımın uzun vadede daha sürdürülebilir ve yönetilebilir olmasını sağlar. Bu raporda, SOLID prensiplerinin her biri detaylı olarak ele alınacak ve yazılım geliştirme süreçlerine sağladığı katkılar kod örnekleriyle birlikte açıklanacaktır.

II. Single Responsibility Prensibi (Tek Sorumluluk Prensibi

Single Responsibility Prensibi (SRP), bir sınıfın, metodun ya da modülün yalnızca tek bir sorumluluğa sahip olması gerektiğini belirtir. Başka bir ifadeyle, bir sınıf veya metodun değişmesine yol açabilecek yalnızca bir nedeni olmalıdır. Bu prensibin temel amacı, yazılımı daha kolay anlaşılabilir, yönetilebilir ve bakım yapılabilir bir hale getirmektir.

A. Prensibin Detayları

1) Sorumlulukların Dağıtılması: Bir sınıfın birden fazla sorumluluk alması durumunda, bu sorumluluklar birbiriyle doğrudan ilişkili olmayabilir. Örneğin, bir sınıf hem veritabanı işlemleriyle hem de kullanıcı ara yüzüyle ilgilenirse, bu sınıfta bir değişiklik yapmak gerektiğinde tüm sistemi etkileyebilir. Bu durum, kodun esnekliğini ve sürdürülebilirliğini azaltır.

2) Değişim Sebeplerinin Azaltılması: Eğer bir sınıf yalnızca bir sorumluluğa sahipse, bu sınıfın değişmesi için yalnızca tek bir neden vardır. Bu da kodun test edilebilirliğini ve sürdürülebilirliğini artırır. Bir sınıfın birden fazla sorumluluğa sahip olması, her sorumluluk için farklı değişim nedenleri oluşturur ve karmaşıklığı artırır.

B. Kod Örneği (Prensibe Aykırı Tasarım)

```
public class Student {
    private String firstName;
    private String lastName;
    private int age;

    public void saveStudent() {
        //Öğrenci veritabanına kaydediliyor
    }

    public void printStudent() {
        //Öğrenci bilgileri yazdırılıyor
    }

    public float calculateResult() {
        //Öğrenci başarısı hesaplanıyor
        return 0.0f;
    }
}
```

Bu örnekte Student sınıfı hem veritabanı işlemlerini, hem öğrenci bilgilerinin yazdırılmasını, hem de başarı hesaplamalarını yapmaktadır. Bu, sınıfın birden fazla sorumluluğa sahip olduğu anlamına gelir ve prensibe aykırıdır.

C. Kod Örneği (Prensibe Uygun Tasarım)

```
public class Student {
    private String firstName;
    private String lastName;
    private int age;

    public Student(String firstName,
    String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    // Getter ve setter metotları
}
public class StudentRepository {
    public void saveStudent(Student s){
        // Öğrenci db'ye kaydediliyor
    }
}
public class StudentPrinter {
    public void printStudent(Student s){
        // Öğrenci bilgileri yazdırılıyor
    }
}
```

Bu tasarımda Student sınıfı yalnızca öğrenci bilgilerini tutma sorumluluğuna sahiptir. StudentRepository sınıfı, veritabanı işlemlerini gerçekleştirmekten sorumludur. StudentPrinter sınıfı, öğrenci bilgilerini yazdırma işlemini gerçekleştirir.

III. Open-Closed Prensibi (Açık-Kapalı Prensibi)

Open-Closed Prensibi (OCP), bir sınıfın ya da modülün genişletmeye açık, ancak değişikliğe kapalı olması gerektiğini belirtir. Bu prensip, mevcut bir sınıfın kodunda değişiklik yapmadan, yeni işlevsellikler eklenmesine olanak tanıyan bir tasarımı teşvik eder. Yazılımın bu şekilde tasarlanması, kodun sürdürülebilirliğini artırır ve değişikliklerden kaynaklanan hata risklerini minimize eder.

A. Prensibin Detayları

1) Genişletmeye Açıklık: Bir sınıf, yeni özellikler ya da davranışlar eklenerek genişletilebilir olmalıdır. Ancak bu genişletme, mevcut kodun doğrudan değiştirilmesiyle değil, yeni sınıflar ya da metotlar aracılığıyla yapılmalıdır.

2) Değişikliğe Kapalılık: Var olan sınıfın ya da modülün kodu, yeni işlevsellikler eklemek için değiştirilemez olmalıdır. Bu, mevcut kodun güvenilirliğini korur ve yazılımın diğer bileşenlerini etkileyebilecek değişikliklerden kaçınılmasını sağlar.

B. Kod Örneği (Prensibe Aykırı Tasarım)

```
class PaymentProcessor {
    public void processPayment(String payment){
        if (payment.equals("CreditCard")){
          System.out.println("CreditCard ödemesi")
        } else if (payment.equals("PayPal")){
          System.out.println("PayPal ödemesi");
        }
    }
}
```

Bu tasarımda yeni bir ödeme yöntemi eklemek için mevcut sınıfa else if blokları eklenmesi gerekir. Bu durum, Open-Closed Prensibine aykırıdır, çünkü var olan sınıfın kodu değişikliğe maruz kalır.

C. Kod Örneği (Prensibe Uygun Tasarım)

```
// Ödeme işlemleri için bir interface
interface Payment {
    void processPayment();
}

// Kredi kartı ile ödeme sınıfı
class CreditCardPayment implements Payment {
    @Override
    public void processPayment() {
        System.out.println("Kredi Kartı ödemesi");
    }
}

// PayPal ile ödeme sınıfı
class PayPalPayment implements Payment {
    @Override
    public void processPayment() {
        System.out.println("PayPal ödemesi");
    }
```

```
}

class PaymentProcessor {
    public void processPayment(Payment p){
        payment.processPayment();
    }
}
```

Bu tasarımda, yeni bir ödeme yöntemi eklemek için yalnızca yeni bir sınıf oluşturmanız gerekir. Mevcut sınıfların kodunda herhangi bir değişiklik yapılmaz.

IV. Liskov's Substitution Prensibi (Liskov'un Yerine Geçme Prensibi)

Liskov's Substitution Prensibi (LSP), bir yazılımda alt sınıfların, türedikleri üst sınıflar yerine herhangi bir değişiklik yapılmadan kullanılabilir olması gerektiğini savunur. Barbara Liskov tarafından öne sürülen bu prensip, yazılım tasarımında polimorfizmi doğru bir şekilde kullanmayı teşvik eder. Alt sınıflar, üst sınıfın özelliklerini genişletebilir, ancak bu özelliklerin davranışını değiştiremez veya bozmaz.

A. Prensibin Detayları

1) Alt Sınıfın Uyumluluğu: Alt sınıfın, üst sınıfın davranışlarını değiştirmeden genişletilmesi gerekir. Eğer bir alt sınıf, üst sınıfın beklenen davranışını bozarsa, bu prensip ihlal edilmiş olur.

2) Kapsam ve Polimorfizm: Liskov's Substitution Prensibi, polimorfizmin doğru uygulanmasını sağlar. Bir üst sınıf nesnesi yerine, herhangi bir alt sınıf nesnesinin kullanılabilmesi hedeflenir.

B. Kod Örneği (Prensibe Aykırı Tasarım)

```
class Dikdortgen {
    protected int genislik;
    protected int yukseklik;

    public void setGenislik(int genislik){
        this.genislik = genislik;
    }

    public void setYukseklik(int yukseklik){
        this.yukseklik = yukseklik;
    }

    public int getAlan() {
        return genislik * yukseklik;
    }
}

class Kare extends Dikdortgen {
    @Override
    public void setGenislik(int genislik) {
        this.genislik = genislik;
        this.yukseklik = genislik;
    }
    @Override
    public void setYukseklik(int yukseklik) {
        this.genislik = yukseklik;
        this.yukseklik = yukseklik;
    }
}
```

Bu örnekte kare sınıfı, dikdörtgen sınıfından türetilmiştir. Ancak karede genişlik ve yükseklik her zaman eşit olmak zorundadır. Bu nedenle, kare sınıfının setGenislik ve setYukseklik metotları, üst sınıfın beklenen davranışını bozarak prensibe aykırı bir duruma neden olur.

V. Interface Segregation Prensibi (Arayüz Ayrımı Prensibi)

Interface Segregation Prensibi (ISP), yazılımda geniş ve şişkin arayüzler yerine, daha küçük ve özelleştirilmiş arayüzlerin tercih edilmesini savunur. Bir sınıf, yalnızca ihtiyaç duyduğu metotları içeren bir arayüzü uygulamalıdır. Bu prensip, nesnelerin gereksiz metotlarla dolu bir arayüzü implement etmek zorunda bırakılmamasını amaçlar.

A. Prensibin Detayları

1) Kapsam ve Uyumluluk: Bu prensip, bir sınıfın yalnızca ihtiyacı olan işlevsellikleri sağlayan arayüzleri implement etmesi gerektiğini belirtir. Eğer bir sınıf, gereksiz metotlar içeren büyük bir arayüzü implement etmek zorunda bırakılırsa, bu durum gereksiz kod karmaşıklığına ve gelecekte bakım sorunlarına yol açabilir.

2) Single Responsibility ile Bağlantısı: Interface Segregation Prensibi, Single Responsibility Prensibi'nin arayüz düzeyindeki uygulaması olarak düşünülebilir. Single Responsibility Prensibi sınıflar için, Interface Segregation Prensibi ise arayüzler için aynı düşünceyi ifade eder.

B. Kod Örneği (Prensibe Aykırı Tasarım)

```
public interface Animal {
    void fly();
    void run();
    void bark();
}
public class Cat implements Animal {
    @Override
    public void fly() {
        // Kediler uçamaz, bu metot gereksiz.
    }

    @Override
    public void bark() {
        // Kediler havlamaz, bu metot gereksiz.
    }

    @Override
    public void run() {
        System.out.println("Koşan kedi");
    }
```

```
}
```

Bu tasarımda Cat sınıfı, Animal arayüzündeki fly() ve bark() metotlarını implement etmek zorunda bırakılmıştır. Ancak bu metotlar, Cat sınıfının işlevselliğiyle ilgili değildir ve bu durum prensibe aykırıdır.

C. Kod Örneği (Prensibe Uygun Tasarım)

```
public interface Flyable {
    void fly();
}

public interface Runnable {
    void run();
}

public interface Barkable {
    void bark();
}

public class Cat implements Runnable {
    @Override
    public void run() {
        System.out.println("Kedi koşuyor");
    }
}
```

Bu tasarımda ,Flyable, Runnable ve Barkable olmak üzere üç özelleştirilmiş arayüz oluşturulmuştur. Cat sınıfı yalnızca ihtiyaç duyduğu Runnable arayüzünü implement etmektedir.

VI. Dependency Inversion Prensibi (Bağımlılık Tersine Çevirme Prensibi)

Dependency Inversion Prensibi (DIP), yüksek seviyeli modüllerin (iş mantığını barındıran sınıflar) düşük seviyeli modüllere (uygulama detaylarını barındıran sınıflar) bağımlı olmaması gerektiğini belirtir. Her iki modül de soyutlamalara (interfaceler ya da soyut sınıflar) bağımlı olmalıdır. Başka bir ifadeyle, yazılımda bağımlılık detaylara değil, soyutlamalara olmalıdır.

A. Prensibin Detayları

1) Yüksek ve Düşük Seviye Modüller: Yüksek seviyeli sınıflar, iş mantığını kontrol eder ve daha geneldir. Düşük seviyeli sınıflar ise bu mantığı uygulamak için detayları içerir. DIP, bu iki katman arasında gevşek bir bağımlılık sağlar.

2) Soyutlamaların Kullanımı: Her iki modül de bir interface veya soyut sınıfa bağımlı olmalıdır. Bu, modüllerin birbirine bağımlılığını azaltır ve değişikliklere karşı daha dayanıklı bir yapı oluşturur.

B. Kod Örneği (Prensibe Aykırı Tasarım)

```
public class Email {
    public void sendEmail() {
        // E-posta gönderiliyor
    }
}

public class SMS {
    public void sendSMS() {
        // SMS gönderiliyor
    }
}

public class Notification {
    private Email email = new Email();
    private SMS sms = new SMS();

    public void sender() {
        email.sendEmail();
        sms.sendSMS();
    }
}
```

Bu tasarımda Notification sınıfı, doğrudan Email ve SMS sınıflarına bağımlıdır. Yeni bir bildirim türü eklendiğinde, Notification sınıfında değişiklik yapılması gerekir, bu da prensibe aykırıdır.

C. Kod Örneği (Prensibe Uygun Tasarım)

```
// Bildirim gönderme işlemleri için bir interface
public interface Message {
    void sendMessage();
}

// E-posta gönderme sınıfı
public class Email implements Message {
    @Override
    public void sendMessage() {
        sendEmail();
    }

    private void sendEmail() {
        // E-posta gönderiliyor
    }
}

// SMS gönderme sınıfı
public class SMS implements Message {
    @Override
    public void sendMessage() {
        sendSMS();
    }

    private void sendSMS() {
        // SMS gönderiliyor
    }
}

// Bildirim sınıfı
public class Notification {
    private List<Message> messages;
```

```
    public Notification(List<Message> m) {
        this.messages = m;
    }

    public void sender() {
        for (Message message : messages) {
            message.sendMessage();
        }
    }
}
```

Bu tasarımda, Notification sınıfı, Message interface'ine bağımlıdır, düşük seviyeli modüllere (Email, SMS) doğrudan bağımlı değildir. Yeni bir bildirim türü eklemek (örneğin Push Notification), mevcut sınıflarda herhangi bir değişiklik gerektirmez.

VII. Özet

Özetle, SOLID prensipleri yazılım geliştirme sürecinde güçlü bir rehber işlevi görerek, yazılımın kalitesinin ve sürdürülebilirliğinin önemli ölçüde artmasını sağlar. Bu prensipler, yazılımın daha modüler, esnek ve test edilebilir olmasını sağlayarak, yazılımın zaman içinde kolayca bakımını yapılabilir ve genişletilebilir hale getirir. SOLID prensipleri, aynı zamanda kodun yeniden kullanılabilirliğini teşvik ederek, geliştirme sürecini daha verimli ve maliyet etkin hale getirir. Bu sayede, yazılım projelerinin karmaşıklığı kontrol altında tutulur, hatalar erken aşamalarda tespit edilir ve yazılımın uzun vadede daha sağlam bir temel üzerine inşa edilmesi sağlanır. Sonuç olarak, SOLID prensiplerine dayalı bir yaklaşım, yazılım mühendisliğinde kaliteyi, güvenilirliği ve sürdürülebilirliği en üst düzeye çıkaran bir metodolojidir ve yazılım projelerinin başarısını büyük ölçüde destekler.

References

[1] Martin, Robert C. "Design principles and design patterns." Object Mentor 1.34 (2000): 597.

[2] Singh, Harmeet, and Syed Imtiyaz Hassan. "Effect of solid design principles on quality of software: An empirical assessment." International Journal of Scientific & Engineering Research 6.4 (2015): 1321-1324.