**<”Ömer Faruk Akyapak”>**

**SOLID :**

SOLID yazılım prensipleri; geliştirilen yazılımın esnek, yeniden kullanılabilir, sürdürülebilir ve anlaşılır olmasını sağlayan, kod tekrarını önleyen ve Robert C. Martin tarafından öne sürülen prensipler bütünüdür.

Kısaltması Michael Feathers tarafından tanımlanan bu prensiplerin amacı;

Geliştirdiğimiz yazılımın gelecekte gereksinimlere kolayca adapte olması,

Yeni özellikleri kodda bir değişikliğe gerek kalmadan kolayca ekleyebileceğimiz

Yeni gereksinimlere karşın kodun üzerinde en az değişimi sağlaması,

Kod üzerinde sürekli düzeltme hatta yeniden yazma gibi sorunların yol açtığı zaman kaybını da minimuma indirmektir.

Bu prensipler uygulanarak uygulamalarımızın büyürken, karmaşıklığın da büyümesinin önüne geçmiş oluruz. “İyi kod” yazmak için bu prensiplere uygun yazılım geliştirmelisiniz.

S — Single-Responsibility Principle

Bir sınıf (nesne) yalnızca bir amaç uğruna değiştirilebilir, o da o sınıfa yüklenen sorumluluktur, yani bir sınıfın(fonksiyona da indirgenebilir) yapması gereken yalnızca bir işi olması gerekir.

O — Open-Closed Principle

Bir sınıf ya da fonksiyon halihazırda var olan özellikleri korumalı ve değişikliğe izin vermemelidir. Yani davranışını değiştirmiyor olmalı ve yeni özellikler kazanabiliyor olmalıdır.

L — Liskov Substitution Principle

Kodlarımızda herhangi bir değişiklik yapmaya gerek duymadan alt sınıfları, türedikleri(üst) sınıfların yerine kullanabilmeliyiz.

I — Interface Segregation Principle

Sorumlulukların hepsini tek bir arayüze toplamak yerine daha özelleştirilmiş birden fazla arayüz oluşturmalıyız.

D — Dependency Inversion Principle

Sınıflar arası bağımlılıklar olabildiğince az olmalıdır özellikle üst seviye sınıflar alt seviye sınıflara bağımlı olmamalıdır.

**DRY :**

Dont repeat yourself! DRY prensibi kodun tekrarlanmasına (code duplication) düşmandır. Aynı işi yapan kodların (genellikle kopyala-yapıştır ile) ihtiyaç duyulduğu her yerde tekrarlanarak projenin çöp olması; yönetilebilirlik, haliyle geliştirilebilirlik ve sürdürülebilirlik imkanlarından gittikçe uzaklaşmasını engellemeye odaklanır.

Eğer daha önce yazılmış bir kod ya da kod bloğunu tekrar kullanmaya ihtiyacımız olursa; mevcut kodu bir fonksiyona/sınıfa/metoda dönüştürerek (mümkünse refactor ederek) eski kullanan yeri bu noktaya bağlayıp, sonrasında yeni yazacağımız kodlarda da bu merkezileştirdiğimiz nokta üzerinden kullanmalıyız.

Bu prensip ile biz yazılımcıların çok hoşuna giden “çalışıyorsa kurcalama” atasözüne ters davranacağız. Başlangıçta konfor alanımızdan çıkmamıza zorlayacak, hoşumuza gitmeyecek fakat zamanla alışacak ve faydalarını tecrübe ettikçe kendinizi geçmişte yazdığınız tekrarlayan kodlarınız için pişmanlık hissederken bulacaksınız.

Avantajları : Maintainablity, Readability, Reuse, Testing

Refactoring, kodun işlevselliğini değiştirmeden, kodun kalitesinin artırılması sürecidir. Evet refactoring bir süreçtir. Her ne kadar amaç/niyet her zaman en baştan temiz kod yazmak olsa da, (her ne kadar yazarken en temiz şekilde yazılsa da) değişen ve gelişen kodların zaman içerisinde sürekli optimize edilmesi gerekecektir.

**API :**

Application Programming Interface, bir uygulamanın işlevlerine dışarıdan veya uzaktan erişilip bu işlevlerin kullanılmasını sağlayan arayüzdür. API, iki uygulamanın birbiriyle iletişime geçmesini sağlayan bir yazılım aracıdır da diyebiliriz. API herhangi bir sunucunun response dönmesine ve sunucunun üzerindeki uygulamaya farklı platformlardan ulaşılmasına olanak sağlar. Farklı platformlarda denildiğinden de anlaşılacağı üzere platform bağımsızdır.

API İşleminde Olan 3 Aşama;

Request: Bir şeyin yapılması için istek

Program: Bu isteğin tamamlanması için bir programın çalıştırılması

Response: Programın bir yanıt göndermesi