**SOLID – Single Responsibility Principle**

Nesne yönelimli programlama dillerini kullanan tüm yazılımcı arkadaşlar bu kelimeyi duymuşlardır. Peki nedir SOLID, bir de sen bize anlat derseniz önce Bob Amcamızı bir hatırlayalım. SOLID, yazılım prensipleri diye adlandırılan ve dünya üzerinde standart olarak kabul edilen 5 önemli tasarım ilkesidir. Bu 5 tasarım ilkesi, Robert C. Martin (Bob Amca) tarafından “Design Principles and Design Patterns” kitabında tanıtılmıştır.

Peki bu yazılım ilkelerine neden ihtiyaç duyarız, bu ilkelere uymak zorunda mıyız canım? Bu 5 prensibin aşağıdaki amaçlarından bir tanesini bile projenizde olsun istiyorsanız EVET uymak zorundasınız. Ama bana sadece çalışan bir kod yeter, gerisini zamanı geldiğinde düşünürüz diyorsanız e o zaman siz bilirsiniz.

1. Esnek, anlaşılabilir ve sürdürülebilir bir yazılım geliştirmek
2. Zaman içerisinde değişebilecek gereksinimlere en az değişimle yanıt verebilmek
3. Geliştirme üzerinde sürekli düzeltme yapılmasından kaynaklanan zaman kaybını minimuma indirmek

“İyi kod” yazabilmek için bu prensipleri uygulamamız gerekir. Gelin, tüm prensipleri özetleyelim ve bu yazının asıl amacı olan Single Responsibility Principle konusunu detaylandıralım.

**S – Single Responsibility Principle**

* Bir nesne, sadece bir iş yapıyor olmalıdır.
* Bir nesne içinde değişiklik yapmak için birden fazla sebebiniz varsa prensibe uymuyorsunuz demektir.

**O – Open Closed Principle**

* Bir nesne gelişime açık değişime kapalı olmalıdır.
* Bir uygulamaya yeni bir özellik eklemek istediğinizde var olan nesneyi değiştirmek zorunda kalıyorsanız bu prensibi ihlal ediyorsunuz demektir.

**L – Liskov Substitution Principle**

* Bir sınıftan miras alan iki sınıf birbirlerinin ve üst sınıftan oluşan nesnenin yerine istisnasız kullanılabilmelidir.
* Bir sınıftan miras alan her sınıfın nesnesine özel bir if ya da try catch yazıyorsanız bu prensibi ihlal ediyorsunuz demektir.

**I – Interface Segregation Principle**

* Her fonksiyon kendi amacına göre bir arayüze sahip olmalıdır.
* Bir interface implemente etmek zorundaysanız fakat bu interface içindeki bir method sizin için anlamsız ise o method o interface ile bağlı olmamalıdır.

**D – Interface Segregation Principle**

* Büyük sistemler küçük sistemlere değil, küçük sistemler büyük sistemlere bağlı olmalıdır.

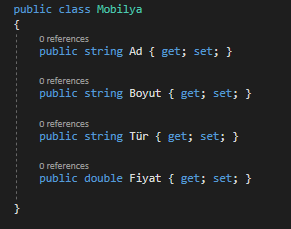
Şimdi, Single Responsibility Principle konusuna yakından bir göz atalım.

**S – Single Responsibility Principle** **(SRP),** daha iyi anlayabilmek için önce günlük ya da iş hayatımıza bir bakalım. Herkesin tek bir sorumluluğun olması her zaman bize cazip gelir öyle değil mi? Örneğin, ben ev işlerini aramızda bölüştürsek ben sadece yerleri temizlesem, annem yemek yapsa, kardeşim de toz alsa. Kimin hangi işi yapacağı belirli öyle değil mi, yemeğin pişme derecesini değiştirmek istediğinizde bunu hangimize söylemeniz gerektiğini biliyorsunuz.

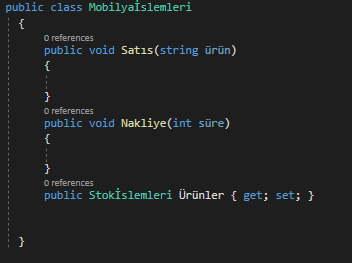
Tabi yazılımda bu sorumlukları belirlemek ve analiz etmek biraz zor olabilir. Başlangıçta anlayamasak da zaman içerisinde geliştirdiğimiz kodda yer alan sınıfın birden fazla amaca hizmet ettiğini fark edebiliriz. İşte o an kodda bazı düzenlemeler ve parçalamalar yapmamız gerekecektir. Bir sınıfa ne kadar az sorumluluk verirsek, o sınıf o kadar kolay değişime uyum sağlar.

Aşağıda bu prensibe uyan bir örnek yapalım. Bakın her sınıf yalnızca tek bir işten sorumlu olacak şekilde üç tane sınıfımız olsun.

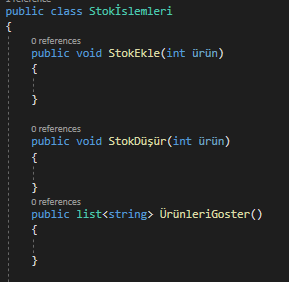
Mobilya sınıfı sadece nesnenin özelliklerini içeriyor.



Mobilyaİslemleri sınıfı, Satış, Nakliye ve Stok İslemleri ile ilgili işlemlerden sorumlu. Ama stok işlemleri, başka bir değişim sebebi olduğundan stok ekleme, düşürme ve stoktaki ürünleri gösterme işlerinin sorumluluğu bu sınıfa ait değil.



İşte bu sorumluluklar, Stokİslemleri sınıfına ait.



Her sınıf tek bir işten sorumlu oldu ve sorumlukları etkin bir şekilde dağıtmış olduk. Dolasıyla, güncelleme gerektiğinde bakımı ve geliştirilmesi çok daha kolay olacağı bir yapı oluşturmuş olduk.