Sürdürülebilirlik yazılımda çok önemli bir yere sahiptir.

**Peki sürdürülebilirlik ne demektir?**

Yazmış olduğumuz program kodları yıllar boyunca çalışacak ve bir sistem içerisinde yaşayacaktır. Kodlar kullanıldığı sürece yeni ihtiyaçlar da doğabilmektedir. Bu nedenle eskiden yazmış olduğumuz bu kodlara yeni kodlar eklememiz ve bakımlarını yapmamız gerekecektir. Eski kodlarımız her ne kadar anlaşılır, genişletilebilir ve test edilebilir ise biz yeni kodlarımızı o kadar hatasız ve temiz bir şekilde kodlayabiliriz. Sürdürülebilirlik de tam da burada önem arz etmektedir.

**Peki nasıl sürdürülebilir kod yazabiliriz?**

Öncelikle Nesne Yönelimli Programlama kavramlarını bilmemiz gerekmektedir. Daha sonra SOLID prensipleri ve Design Pattern’leri bilmemiz gerekmektedir.

Ayrıca yazılımda sürdürülebilirlikle ilgili olarak sıkça karşılaşılan sorunlar ve çözümlerini belirtmek gerekirse;

* **Sorun:** Sınıflar'a sorumluluğundan daha fazlasını yükleyerek sınıflar arasında stres oluşturmak. Yani sınıf ile ilgili olmayan fonksiyon ve özelliklerin sınıf içerisinde yer alması.
  + **Çözüm:** Doğru mimarinin uygulanması gerekir. Örneğin: Mvc, 3-tier, Layered, Mvp, Mvvp vb.
* **Sorun:** Sınıfları birbirine bağlı olmaya zorlamak (Tightly Coupled). Bir sınıf değiştiğinde bağlı tüm diğer sınıflarda da geliştirme ihtiyacının doğması.
  + **Çözüm:** SOLID prensiplerinin kullanılması gerekir.
* **Sorun:** Aynı kodların sürekli tekrar edilmesi.
  + **Çözüm:** uygun Design Patternlerin kullanılması gerekir.

**SOLID ne demektir?**

SOLID prensipleri Object Oriented Prensiplerinin nasıl kullanılacağını belirten prensiplerdir ve prensiplerin baş harflerinden oluşmaktadır.

* Single Responsible (SRP)
  + Bir nesne, fonksiyon veya bir metodun yalnızca bir sorumluluğunun olması gerekmektedir.
* Open / Closed (OSP)
  + Var olan bir sınıfa yeni davranışlar eklenebilmesi fakat sınıfın temel özelliklerinin değişmemesi anlamına gelir.
  + Örneğin uygulamamızda oluşan exceptionları loglamak için ExceptionLogger adında bir sınıfımız olsun. Bu sınıf oluşan exceptionlar için database’imize log atsın. Fakat database connection sağlanamadığı zaman exception durumları bir file loglansın veya bir servis üzerinden log atsın. Bu durumda görüldüğü üzere ihtiyaç ve talepler devamlı değişebilmektedir. Bu talepleri karşılamak için her seferinde if bloğu yazmamız bir süre sonra yazılımda sürdürülebilirliği azaltacaktır. Peki bunu nasıl önleriz? Bu durumda Open / Closed prensibine göre FileLogger, SqlLogger, ServiceLogger gibi sınıflar yazmak ve bu sınıfları ILogger interface üzerinden türetmemiz. Daha sonrada ExceptionLogger’a ILogger interface’ini parametre olarak vermemiz gerekir. Bu sayede hem nesneler arasında gevşek bağlamayı gerçekleştirmiş hem de yazılımımızı yeni davranışlara açık fakat ExceptionLogger nesnemizin temel özelliklerini değiştirmeyecek şekilde kurgulamış oluruz.
* Liskov Substitution (LSP)
  + Sırf birbirine benziyorlar diye 2 nesneyi birbirinin yerine kullanmamamız gerekmektedir.
  + Örneğin, bir baba doktor olsun fakat oğlu sporcu olsun, her ikisi de aynı aileden diye oğlunun babasının yerine doktorluk yapmasını bekleyemeyiz. Başka bir örnek verecek olursak; bir şirketin hem bireysel hem de tüzel müşterileri olabilir. Her ne kadar müşteri müşteridir deyip ikisini de tek bir nesnede tutma eğiliminde olsak da. Bu iki müşterinin birbirlerinden ayrıldıkları noktalar vardır. Tüzel müşteriler vknNo, Unvan gibi bilgileri taşırken bireysel müşteriler tckn veya isim, soyisim bilgilerini taşımaktadırlar. Liskov prensibine göre bu iki nesneyi ayrı ayrı tanımlamak ve ICustomer interface üzerinden kalıtım almak gerekmektedir. Bu sayede bu iki nesneyi kullanan business yapıları gerçekten iki nesnenin ayrımını yapabileceklerdir.
* Interface Segregation (ISP)
  + Birbiri ile ilişkisi olmayan fonksiyon veya özellikler aynı interface içerisinde yer almaz, bunun yerine anlamsal olarak farklı interface’lere bölünürler.
  + Örneğin; Maillerimizi atarken bazen attachment ekleyebiliyoruz veya cc, bcc ekleyebiliyoruz. Aynı zamanda bunların hiçbirini yapmayıp sadece to ve message yazıp mail atabiliyoruz. İş akışımıza göre bizim bu işleri birbirinde ayırt etmemiz gerekmektedir. Bu nedenle IEmail, IEmailWithAttachment, IEmailWithCc, IEmailWithBcc gibi interfacelere bu işleri bölmemiz bizim daha esnek ve yerinde geliştirme yapmamızı sağlamaktadır. Bunu bize söyleyen interface segregation prensibidir.
* Dependency Inversion (DIP)
  + Üst seviye nesnelerin alt seviye nesnelere sıkı sıkıya bağlı (Tightly Coupled) olamayıp gevşek (Loosly Coupled) olarak bağlanmasını gerektirmektedir. Yani her iki nesnenin de abstract yapılara bağlı olması gerekmektedir. Bu sayede üst seviyede veya alt seviyede yaptığımız değişiklikler bizim tüm iş akışımızı etkilemeyecek belki de tek bir yerde geliştirme yaparak yeni geliştirme talebimizi tamamen yerine getirmiş olacağız.
  + Örneğin; Open / Closed prensibinde belirtmiş olduğum ExceptionLogger sınıfı ILogger interface almıştır. Bu sayede ILogger’dan türeyen sınıflar ile ExceptionLogger arasında Loosly Coupled sağlanarak Dependency Inversion prensibine uyulmuştur. Peki bu neyi sağlamıştır. Yeni bir Log tipi yazılmak istendiğinde mesela EventLogger. Sadece EventLogger sınıfının oluşturulması ve ILogger’dan türetilmesi yeterli olacaktır.