Dependency Inversion Principle – DIP- (Bağlılığı Tersine Çevirme Prensibi)



Merhaba sevgili dostlar!

SOLID tasarım prensiplerini anlattığım makale serisinin sonuna gelmiş bulunuyoruz. Dependency Inversion Principle (Bağlılığı tersine çevirme – bu arada itiraf ediyorum; bu prensiplerin Türkçesini yazınca çok tuhaf oluyorum) prensibi… Dilerseniz yine bu prensibin temel cümlesini ele alarak başlayalım.

1. Yüksek seviye modüller, düşük seviye modüllere bağlı olmamalı. Her ikisi de soyut (abstract) kavramlara bağlı olmalı.
2. Soyut kavramlar detaylara bağlı olmamalı. Detaylar soyut kavramlara bağlı olmalı.

Elbette bu kuralları öncelikle günlük hayata uygulayacağız. Hep söyledim yine söylüyorum, yazılım mimarisinde uygulanan tasarım prensipleri, hayatın ta kendisinden ilham almıştır. O halde şöyle hayatımıza bakalım ve kavraması zormuş gibi gözüken şu iki maddenin aslında ne kadar kolay olduğunu görelim.

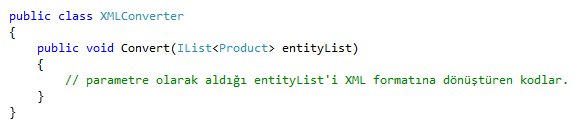
Şimdi size, mobilyaları zemine sabitlenmiş bir ev olduğunu söylesem; bu evin efektif olduğunu düşünür müydünüz? Hiçbir eşyasını yerinden dahi oynatamadığınız bir ev ne kadar mantıklı olabilir ki? Burada tasarımsal bir problemin olduğu aşikâr öyle değil mi?

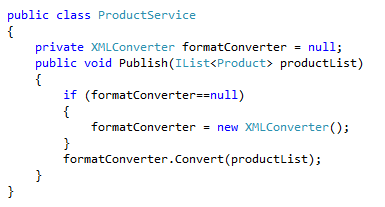
Hadi benzer bir örnekle devam edelim. Evinizdeki ampul patladığında koskoca elektrik tesisatını değiştirdiğinizi bir düşünsenize! Yani büyük modül (elektrik tesisatı) küçük modüle (ampul) bağlı olmamalı! Her ikisi de soyut kavrama (lamba –duy ve ampul-) bağlı olmalı. Üstelik burada, ampulün kaç Watt olduğu (detay), lambanın duy kısmını (soyut) ilgilendirir mi? Hiç sanmıyorum.

Buna benzer yüzlerce örnek verilebilir. Ancak biraz daha teknik bir örnek vermem gerektiğini düşünüyorum. Günümüzde; aynı altyapıyı kullanan fakat birden fazla arayüze (web, mobile gibi) sahip olan birçok uygulama var. Burada da altyapı üzerinde çalışan ve iş süreçlerini yöneten katman(BLL) bizim “yüksek seviye” modülümüzdür. Bu açıdan bakıldığında kullanıcı arayüzünün (UI) düşük seviye olduğunu söylememiz mümkün.

Tamam. Bu kadar sözel örnek yeterli sanırım. Artık bu prensibi kodlarımızda nasıl uygulayacağımıza odaklanabiliriz.

Bu örnekte kullanacağımız senaryoya göre; hali hazırda bir web servisiniz var ve bu servis, veri paylaşımı için XML formatını kullanıyor. Öncelikle kötü tasarlanmış bir yapıyı görelim sonrada bunun neden kötü olduğunu tartışalım.

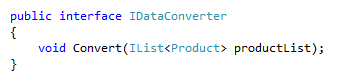




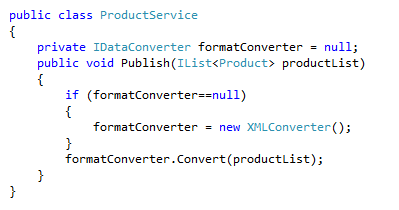
Kodumuz gayet sıkıntısız. Buraya kadar sorun yok. Ancak, yıllar sonra bir gün JSON (JavaScript Object Notation) diye yeni bir veri paylaşım formatı çıkar ve sizde tüm altyapıyı değiştirmek zorunda kalırsınız. Peki bunu yapmak, bu mimaride kolay olur muydu sizce?

Bu mimarideki yüksek modül, **ProductService**sınıfıdır. Düşük seviyede bulunan modül ise doğal olarak **XMLConverter** sınıfıdır. Bu noktada, **ProductService**sınıfının **XMLConverter**sınıfına bağımlı olduğu çok net ortada. Çünkü artık veri paylaşımını JSON formatıyla yapmaya karar verdiğinizde, değiştirmeniz gereken sınıf kesinlikle **ProductService**olacaktır. Evet, ampulü değiştirmek için koskoca tesisatı değiştirmek zorunda kalacaksınız.

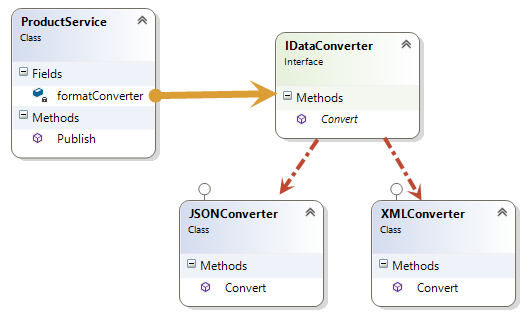
Öyleyse en başta mimariyi nasıl düzenlemeliydik? Burada, büyük modülünüzün **Publish**metodunun ana unsur olduğunun farkında olduğunuzu düşünüyorum. O halde yapmanız gereken şey Loosely Coupling yaklaşımı, SRP ilkesini ve ISP ilkesini bir araya getirerek **Convert**metodunu ayrı bir Interface içine almak olacaktır.



Bu değişimi sağladıktan sonra büyük modülünüzü de buna göre değiştirmelisiniz.



Bu değişiklik sayesinde, ampul patladığında sadece yeni bir ampul alabileceksiniz. Hatta dilerseniz floresan bile takabilirsiniz!



Peki, bu mimaride “istediğimiz zaman istediğimiz sınıfla (**JSONConverter** veya **XMLConverter**) çalışmak istersek” ne yapacağız? İşte bu Dependency Injection yani başka bir makale konusu…

Evet, sevgili dostlarım. Bu yazımız ile SOLID tasarım prensiplerinin sonuna gelmiş bulunuyoruz. Umarım faydalı olmuş ve nesne yönelimli mimari konusunda sizleri bir adım daha öteye taşımıştır.

Mutlu günler dileklerimle.