Single Responsibility Principle – SRP- Temel Metotlardan Çok Katmanlı Mimariye-2

Merhaba sevgili dostlar!

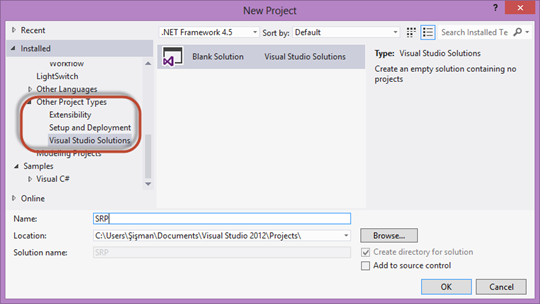
Bir önceki Single Responsibility Principle (SRP) makalemizde, prensibi çok temel olarak incelemiş ve örneği de basit tutmuştuk. Bu yazımda ise, işin biraz daha detayına inmeyi planlıyorum.

Aslında Single Responsibility Principle , iki farklı yapıya uygulanabilen bir prensip. Bir önceki yazımda “class” düzeyinde Single Responsibility Principle uygulamıştım. Bu nedenle örnek basit oldu doğal olarak. Bu kez “metot” düzeyinden “sınıf” düzeyine doğru hareket ederek bu prensibi biraz daha kavrayacağız. Elbette yine refactoring (yeniden yapılandırma) uygulayarak.

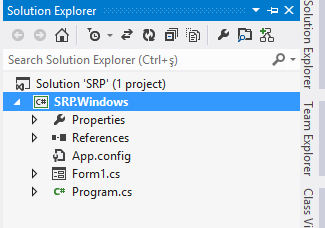
Haydi! Biz susalım, kodlarımız konuşsun.

Ha bu arada, bu kodlarda veri erişimi için eski usul ADO.NET mimarisini tercih edeceğim. Bunu tamamen Single Responsibility Principle’in anlaşılması için yapıyorum. Böylece, Entity Framework (EF)’ün neden önemli olduğunu da göreceksiniz (bir taşla iki kuş yani).

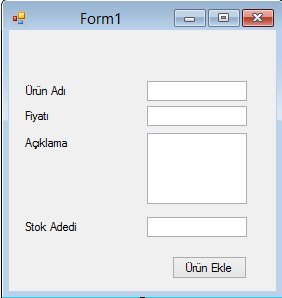
Mini uygulamamızda, tek katmanlı (monolitik) bir yapıdan çok katmanlı (n-tier) bir yapıya ilerleyeceğiz. Bu nedenle projenizi açarken; “Other Project Types” kategorisinden “Visual Studio Solutions” kategorisini ve “Blank Solution” şablonunu tercih edin. Hadi adını da ben vereyim bu “solution” un: SRP olsun.



Solution’a proje ekleyerek devam edeceğiz. İlk ekleyeceğimiz projemiz de Windows Forms Application türünde olsun. Elbette proje isimlendirme standartlarını da kullanalım. Her projemiz “SRP” ile başlayacak. Örneğin ilk projemizin adı “SRP.Windows” olabilir. Projenizi ekledikten sonra, Solution Explorer aşağıdaki gibi gözükecek:



Bayanlar ve baylar, şimdi de arayüzümüzü tasarlayalım. Minik uygulamamız, veri tabanına ürün ekleme amacını güdüyor. Haliyle arayüz de buna uygun olacak.



Tamamdır. Artık butonun Click olayına kodumuzu yazabiliriz:



Şimdi gelin hep beraber bu yapıyı inceleyelim. Öncelikle belirteyim ki yukarıda görmüş olduğunuz kod kirli bir kod. Kirli çünkü yeniden kullanılabilir bir yapı değil. Bununla birlikte kontrollere sıkı sıkıya bağlı bir kod bloğu. Üstelik değişiklik yapmak istediğinizde birçok yeri değiştirmeniz gerekebilir.

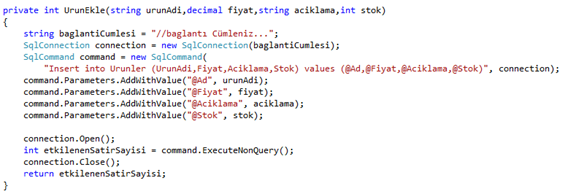
Bir yazılımcı, her zaman geniş düşünebilmelidir. Bir yazılım projesi de, her zaman gelişime ve değişime açık yani dinamik bir yapıda olmalıdır. Diyelim ki yukarıdaki mini projeyi piyasaya sürdükten bir süre sonra, başka bir RDBMS ile (Mesela Oracle ya da MySQL) çalışmaya karar verdiniz. Projenin tamamında nasıl bir değişiklik yapacaksınız? Her butonun Click olayına gidip tek tek değiştirmek pek mantıklı değil öyle değil mi?

Bu kodlar form üzerinde yer alan kontrollere (“ürün ekle” butonu ve diğer textbox kontrolleri) yani arayüze bağlı. Böyle bir durumda, veritabanına eklemiş olduğunuz ürünü tekrar kullanmak istediğinizde ne edeceksiniz?

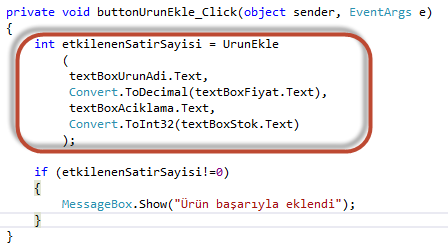
Eğer bu kod bloğunun neden problemli olduğunu anlatabildiysem ne mutlu bana  Öyleyse düzenlememize başlayalım:

İlk Refactoring

İlk olarak kodumuzu, Button ve TextBox kontrollerinden bağımsız hale getirelim. Bu hedefe ulaşmak için veritabanına ürün eklemekten sorumlu bir metot oluşturacağız. Haliyle bu metot geriye “etkilenen satır sayısı”nı döndürecek (int). Parametre olarak da TextBox’lardan gelen verileri alması gerekiyor. İşte o metot geliyor:

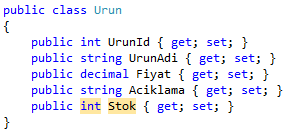


Şimdi Ürün Ekle butonumuzun Click anında çalışacak olan kodlarımızı da düzenlememiz gerekiyor. İşte sonuç:

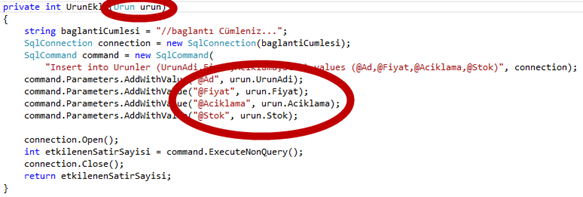


İkinci Refactoring

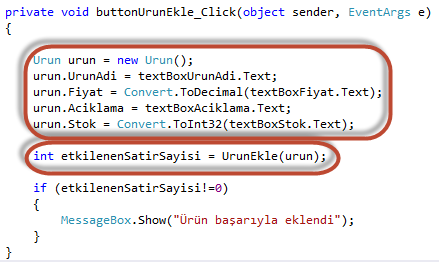
Evet. Artık biraz daha temiz bir kodumuz var. Fakat yeniden kullanılabilirlik sıkıntısını henüz çözmüş değiliz. En basit anlamda, veritabanına eklemiş olduğunuz ürün ile başka bir işlem yapmanız gerektiğinde bu ihtiyacınızı karşılayacak bir yapıya sahip değilsiniz. Öyleyse, bu amaç için tasarlanmış bir sınıfa ihtiyacım var. Aynı projeye bir class ekleyerek yolumuza devam edelim:



Şimdi, veritabanına ürün ekleme işlemini bu urun sınıfının bir nesnesini kullanarak yapabiliriz. O halde, az önce yazmış olduğumuz UrunEkle isimli metodumuzu buna göre tekrar düzenleyelim.



Doğal olarak buttonUrunEkle\_Click metodumuzda da değişikliğe gideceğiz.



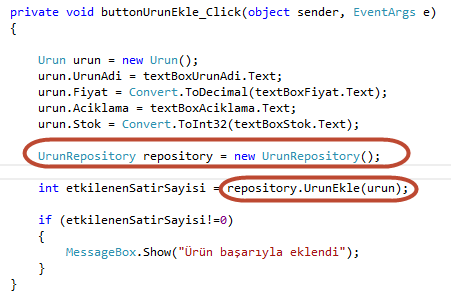
Üçüncü Refactoring

Sevgili dostlar; umarım buraya kadar her şey yolundadır. Bu noktada SRP’nin anahtar cümlesini hatırlayalım:”**Bir nesneyi değiştirmek için tek bir neden olmalıdır**“. Dolayısı ile bu nesne diğer nesnelerden mümkün olduğunca bağımsız olmalıdır da diyebiliriz. O halde burada bu kurala uymayan bir durum görüyoruz. Form1 sınıfı, kullanıcıya arayüz oluşturma sorumluğu olan bir sınıf. Öyleyse “UrunEkle” metodunun bu sınıfta ne işi var?

Bu ve benzeri metotları barındırmak için ayrı bir sınıf tanımlayalım şimdi. Genellikle bu işlemleri yapmak için kullanılan sınıflara, İngilizcede depo anlamına gelen “Repository” kelimesiyle biten isimler verilir. Biz de bu kurala uyalım ve projemize “UrunRepository” isminde bir sınıf ekleyelim. Daha sonra, Form1 içindeki UrunEkle metodunu bu sınıfa taşıyalım:



E tabii akabinde Form1 içinde yer alan buttonUrunEkle\_Click metodunu tekrar değiştiriyoruz:



Dördüncü Refactoring

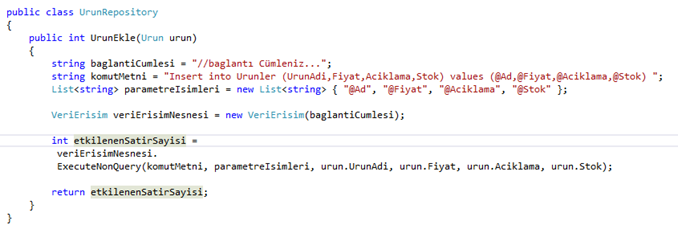
Şu anda Form1 sınıfımız çok daha ferah görünüyor. Ancak aynı şeyi UrunRepository sınıfı için söyleyemeyiz. Biraz geniş düşünürsek; veritabanı üzerinde yapılacak tüm işlemler (ekleme, silme, güncelleme ve veri alma işlemleri) bu sınıfta yapılacak. Bu işlemlerin her birinde de ADO.NET kodları kullanılacak. Hem de tekrar tekrar. Unutmamak gerekir ki bir kodu birden fazla metotta kullanıyorsanız orada bir tasarım hatası var demektir. Ne yapacağız o zaman? Hadi UrunRepository sınıfı içinde yer alan UrunEkle metodundaki işlemleri inceleyelim.

1. SqlConnection nesnesi oluşturuluyor. Tüm veritabanı işlemlerinin olmazsa olmazı.
2. SqlCommand nesnesi oluşturuluyor. Komutsuz veri tabanı işlemi mi olur? Bu da tüm işlemlerde ortak.
3. SqlCommand nesnesine parametre ekleniyor. Sadece bu metoda özel olduğunu düşünen yoktur sanırım.
4. Connection nesnesinin bağlantısını açıp komutu çalıştırdıktan sonra bağlantıyı kapatıyor. Yorum yazmaya bile gerek yok.

Madem bu kadar çok ortak nokta var o zaman bunları da ayrı bir sınıfa alalım. Aynı projeye VeriErisim isminde bir sınıf ekleyerek devam edelim… Tabii bu sınıf sadece benim mini örneğim için gerekli olan işlemleri yapacak. Bu tarz sınıfları sıklıkla yazan profesyonel dostlarım, bazı kısımları hiç beğenmeyerek bana kızabilirler. Hemen amacımın yalnızca SRP’yi anlatmak olduğunu belirterek kıvırayım ben de.

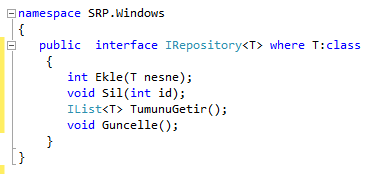


Şimdi hiç üşenmeden UrunRepository sınıfımı güncelliyorum:



Beşinci Refactoring

“Ne zaman bitecek bu refactoring” dediğinizi duyar gibiyim. Hemen söyleyeyim: Siz istediğiniz zaman! Şimdi bu projede muhtemelen sadece ürün eklenmeyecektir değil mi? Yani aynı işlemleri kategori, sipariş, müşteri gibi varlıklar için de icra edeceksiniz. Yani her birinin Repository sınıfı olacak. Bu sınıfları standart hale getirmek ve düşük bağımlılık için bir Interface kullanmaya ne dersiniz? Aynı projeye Interface ekleyerek devam edelim.



UrunRepository sınıfını da bu değişikliğe göre tekrar düzenleyelim:



Tamamdır. Artık her nesne kendi işinden sorumlu diyebiliriz. Ancak şimdiye dek oluşturduğunuz tüm sınıflar tek bir proje içerisinde. Yani monolitik bir yapıda. Bu sınıflardan birini başka bir projede kullanmak istediğinizde, koca bir Windows Projesini referans olarak eklemek zorunda kalacaksınız. Aynı zamanda, hareketliliği kısıtlı olduğundan daha sonra işleyiş bakımından yeni bir özellik eklenmeye çalışıldığında can sıkıcı bazı durumlarla karşılaşabilirsiniz.

O zaman sınıfları katmanlara ayırmanın vakti geldi demektir.

Altıncı Refactoring

Bu sınıfların yaptıkları işlere göre ayıralım önce:

1. Model: Her varlığı temsil eden bir sınıf var (Urun, Kategori vb.)
2. DA (Data Access) : Veri erişim sınıfı
3. İş (Business) Sınıfları: Veritabanında yapılacak işleri bir araya toplayan (Repository) sınıflar.

O zaman Solution Explorer üzerinden yeni projeler ekleyelim ve katmanlara göre düzenleyelim.

Model Katmanı

Solution’a yeni bir Class Library ekleyin ve adı SRP.Model olsun. Projenizde kullandığınız Urun sınıfını buraya taşıyın. Taşıdıktan sonra namespace’ini değiştirmeyi unutmayın. Ardından Windows Form uygulamanızın referanslarına SRP.Model’i ekleyin. Urun nesnesini kullandığınız kod bloklarında using anahtar kelimesi ile SRP.Model namespace’ini tanımlayın.

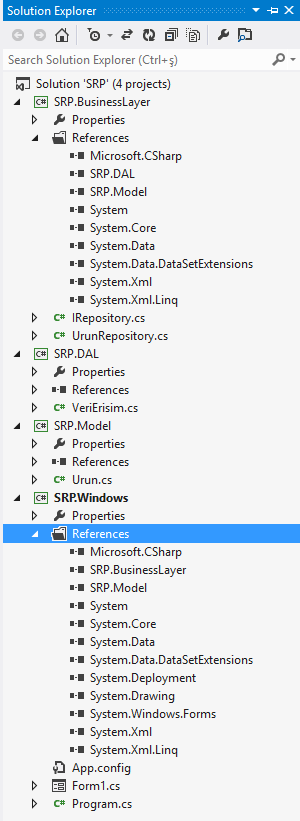
İş Katmanı

Solution’a bu kez SRP.BusinessLayer isimli bir Class Library ekleyin. IRepository interface’i ve UrunRepository sınıfını buraya taşıyın. Yine namespace’e dikkat edin. Bu katmanın referansına, SRP.Model projesini ekleyin. Ardından Windows Forms uygulamanızın referanslarına SRP.BusinessLayer dll’ini ekleyin. Gerekli yerlerde using SRP.BusinessLayer tanımını yapmayı unutmayın.

Veri Erişim Katmanı

Son kez solution’a SRP.DAL isminde bir Class Library daha ekleyin. VeriErisim sınıfını buraya alın. Taşıdığınız dosyanın namespace’ini düzelttikten sonra SRP.BusinessLayer projesine SRP.DAL referansını ekleyin. Yine using anahtar kelimesini kullanarak SRP.DAL namespace’ini tanıtmayı unutmayın.

Tüm bu değişikliklerden sonra Solution Explorer aşağıdaki gibi olacaktır:



İşte değerli dostlarım! SRP (Tekil Sorumluluk) Prensibini kullanarak bakımı daha kolay ve temiz bir kod yazdık gördüğünüz gibi. Elbette bu projede SRP uygulanabilecek birçok yer daha bulunabilir.

Umarım bu prensibi daha iyi kavramanıza yardımcı olabilmişimdir.

Sağlıcakla kalın. Hepinize tertemiz kodlar diliyorum.