REPOSİTORY PATTERN NEDİR?

Selam bu yazıma ulaşmış güzel okuyucu! Repository pattern ile karşınızdayım.

Tasarım desenleri, yazılımda karşılaşılan problemlere çözümler olarak geliştirilmiştir. Yazılımcı atalarımız zamanında yaşadıkları problemler canlarına tak etmiş, bu işler böyle gitmez demişler. Gang of Four diye adlandırılan abilerimiz kafa kafaya verip “Design Patterns, Elements of Reusable Object-Oriented Software” isimli meşhur kitabı yazmışlar.

Problemler doğrultusunda çıkan bu tasarım desenlerinde demek ki önce nasıl bir soruna çözüm ürettiğini bilmemiz gerekir. Böylece böyle bir problem ile karşılaşmamak için projenin en başında projemizi ona göre tasarlamalıyız. O zaman önce problem ne ona bakalım…

Repository Pattern’in çözdüğü problem ne?

Veri tabanı işlemleri yaparken plansız şekilde yapılan kodlama, bizleri kod tekrarına götürebilir. Özellikle CRUD işlemlerinde. Nedir CRUD işlemleri? Açılımı şu Create Read Update Delete. Yani temel veri tabanı fonksiyonları.

Şimdi varsayalım ki bizden ürün ekle, sil, güncelle, listele fonksiyonlarını gerçekleştirilen bir proje istediler. Biz de aşağıdaki gibi bir kod yazdık. Metotların içlerinin boş olmasını aldırmayalım maksat tasarımı anlamak.

```
public class ProductDal
{
    0 references
    public void Add(Product name)
    {
        throw new NotImplementedException();
    }

    0 references
    public void Delete(Product name)
    {
        throw new NotImplementedException();
    }

    0 references
    public List<Product> getAll()
    {
        throw new NotImplementedException();
    }

    0 references
    public void Update(Product name)
    {
        throw new NotImplementedException();
    }
}
```

Yazdığımız program yanlış mı hayır tabi ki de çalışıyorsa hiçbir problem ile karşılaşmazsın. Ama diyelim ki bizden böyle yazılım isteyen kişinin başka bir ihtiyacı daha doğdu ve dedi ki ben kullanıcı için ekle, sil, güncelle, listele işlemlerini yapan yazılımı eklemesini istedi. Biz yine aynı şekilde kodladığımı düşünelim. Program çalışır evet ama gün geçtikçe veri tabanı işlemlerinizi arttığı varsayalım. 30 tane tablonun ekle, sil, güncelle, listele fonksiyonlarınız yapmamız istendi. 30 tane tablo için 120 tane metot demek. Ne! Gerçekten hepsi için ayrı ayrı böyle kod yazacak mıyız? Her tablo için ayrı ayrı bu koddan yazmak hem kod tekrarına hem de kontrol edilebilirliği azaltmaya başlayacaktır. Yapma böyle bir şey akıl işi değil…

O zaman hadi gel bu kodu şöyle tasarlayalım.

```
1 reference
public interface IRepository<T>
{
    1 reference
    List<T> getAll();
    1 reference
    void Add(T name);
    1 reference
    void Update(T name);
    1 reference
    void Delete(T name);
}
```

Kodu inceleyim. Şimdi projemizde artık sadece tek bir tipte veri tabanı işlemleri yapılmayacak. Her biri için bir interface classı oluşturmak yerine IRepository ile her biri için bu classı tanımladık. <T> kısmı, bu interface kullanacak sınıfın tipini alacaktır. Hemen göstereyim.

```
2 references
public void Add(Product name)
{
    throw new NotImplementedException();
}

2 references
public void Delete(Product name)
{
    throw new NotImplementedException();
}

2 references
public List<Product> getAll()
{
    throw new NotImplementedException();
}

2 references
public void Update(Product name)
{
    throw new NotImplementedException();
}
```

Gördüğünüz gibi ProductDal sınıfımızı IRepository interface'ini implements ettiğimiz zaman bize "<>" sembollari içinde belirtilen tipi yani Product'ı metotlarda bu tipe bağlı olarak oluşturmuş olduk. Böylelikle herhangi bir veritabanı işlem fonksiyonu eklemek istediğimizde IRepository interface belirttiğimizde diğer implement ettiğimiz classlara eklenecektir. Böylelikle kontrol edilebilir, kod tekrarı olmadan, gelişime açık bir yapı elde ederiz.