KTO KARATAY
ÜNİVERSİTESİ

T.C.
KTO KARATAY ÜNİVERSİTESİ FEN

BİLİMLERİ ENSTİTÜSÜ

ELEKTRİK BİLGİSAYAR MÜHENDİSLİĞİ TEZLİ YÜKSEK LİSANS BÖLÜMÜ NESNE TABANLI PROGRAMLAMA (OBJECT ORIENTED PROGRAMMING)

DERSİ FİNAL ÖDEVİ

ÖDEV KONUSU

Strategy Design Pattern
Spaghetti Code, Warm Bodies, Smoke and Mirrors Anti Pattern

ÖĞRETİM ÜYESİ

Doktor Öğretim Üyesi Ali ÖZTÜRK

ÖĞRENCİ

Recep Ali AY

21743543

Konya, 2018

Birinci Kısım Design Pattern : Strategy Pattern

a) Bu tasarım şablonunun kullanılmasındaki motivasyon nedir?

Strategy pattern nesne yapılı yani object oriented programlama dillerinin en temel pattern örneklerinden biri olarak ön plana çıkmaktadır.

Yapılması istenilen bir işimiz var ve bu işi birden farklı yollarla yapma ihtiyacımız var. Bu gibi durumlarda var olan işi ilgili sınıfı sürekli refactor ederek if-else blokları ile yapmak yerine, yeni bir sınıf daha ekleyerek istenilen durumda ilgili işi ilgili sınıfta yapmamıza olanak sağlar. Böylece var olan sınıfımız üzerinde değişiklik yapmadan sistemimizi geliştirmiş olacağız. Burada en önemli tasarım prensiplerinden birisini olan Open-Closed prensibi söz konusudur.

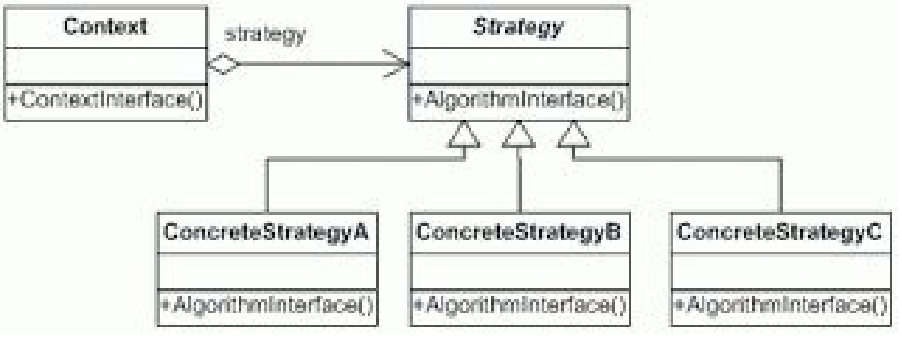
b) Bu tasarım şablonu hakkında detaylı bilgi veriniz.

Kullanıcı sınıfların, kullanılan yöntemler hakkında bilgi sahibi olmamaları gerektiği durumlarda Strategy tasarım şablonu kullanılır. Bir işlemi birden fazla yöntem (algoritma) ile implemente etmek için Strategy tasarım şablonu kullanılır. Sistem gereksinimleri doğrultusunda en uygun yöntem seçilerek, işlemin gerçekleştirilmesi sağlanmaktadır. Kullanılan yerlerde sınıfın hangi yöntemin kullanıldığını bilemez. Böylece kullanıcı ile sistem arasında gizli bir duvar örerek, sistemin nasıl çalıştığını kullanıcıdan saklama yeteneğine kavuşmuş oluyor.

Şekil 1'de UML Diyagrama göz atıldığında **Context** ile **Strategy** sıfını arasında **aggregation**(bu tür bir ilişkide ilgili nesnelerin yaşam döngüleri birbirlerinden ayrıdır) türünde bir ilişki olduğunu görmekteyiz.

Strategy: Bir arayüz(Interface) tasarlayarak ortak olan tüm algoritmalar burada toplanmaktadır.

ConcreteStrategy: İlgili algoritmayı gerçekleştirilen yerdir.



Şekil 1 UML Diyagramı

```
interface ILogStrategy

{

        void InsertLog(string LogValue);

}
//ConcreteStrategy1
class LogFile:ILogStrategy
{
        public void InsertLog(string LogValue)
        {
        Console.WriteLine("File bazlı log yazıldı.");
        }
}
//ConcreteStrategy2
class LogDb : ILogStrategy
{
        public void InsertLog(string LogValue)
        {
        Console.WriteLine("Veri tabanı bazlı log yazıldı.");
        }
}
//Context yapısı
class LogWriter
{
        ILogStrategy logstrategy;
        public LogWriter(ILogStrategy LogStrategy)
        {
        logstrategy = LogStrategy;
        }
        public void LogInsert(string LogValue)
        {
        logstrategy.InsertLog(LogValue);
        }
}
class Program
{
        static void Main(string[] args)
        {
        //LogDb veya LogFile nesnelerini direk kullanmak yerine LogWrite üzerinden erişim sağlanmaktadır.
        LogWriter lw = new LogWriter(new LogFile());
        lw.LogInsert("ins1");
        lw = new LogWriter(new LogDb());
        lw.LogInsert("ins2");
        Console.ReadKey();
        }
}
```

Daha sonra uygulamada registry bazlı log tutmasını gerek yapılması gereken yeni bir ConcreteStrategy yapısı oluşturmak olacaktır. Tabiki bütün loglar aynı şekilde tutulacak ise Constructor a parametre vermek yerine LogWriter Constructor da direkt istediğimiz Concrete türünü oluşturabiliriz.

c) Bir örnekle kullanımını açıklayınız.

Örnekte bir e-ticaret sitesindeki ödeme kütüphanesini ele alınmaktadır. Sistem birden fazla yöntemle ve banka ile çalışmaktadır. İster banka transferi ile ister mail order yöntemi ile veya sanal pos yöntemi ile ödeme yapılmak isteniyor olabilir.

Öncelikle yukarıda bahsedildiği gibi ortak olan algoritma örnek olarak **MakePayment** metotu olsun ve ilgili arayüzü tasarlayalım.

```
/// <summary>
   /// Strategy - Tüm ödeme strategy'lerimiz bu interface'den türeyecek.
   /// </summary>
   interface IPayment
   {
      void MakePayment();
   }
```

İlgili arayüzü tasarladığımıza ve **MakePayment** metotunu tanımladığımıza göre gerçekleşecek olan ilgili **ConcreteStrategy**'lerimizi oluşturmaya başlayabiliriz.

```
/// <summary>
   /// ConcreteStrategy - Mail order yöntemi ile ödeme strategy'miz.
   /// </summary>
   class MailOrderStrategy : IPayment
   {
      public void MakePayment()
      {
         Console.WriteLine("Mail order yöntemi ile ödeme yapıldı.");
      }
   }
```

Mail order ödeme yöntemini sistemimize uyarladık ve ilgili **MakePayment** metotunu bir mesaj yazdırarak doldurduk. Sistem her zaman gelişime açık olmalıdır. Birde sistem kullanıcıları sürekli yeni bir şeyler eklenmesini istemektedirler. Müşterimiz bizden bu seferde **banka transferi** yöntemi ile ödeme tipinin de entegre edilmesini istedi.

```
/// <summary>
   /// ConcreteStrategy - Havale yöntemi ile ödeme strategy'miz.
   /// </summary>
   class BankTransferStrategy : IPayment
   {
      public void MakePayment()
      {
         Console.WriteLine("Havale yöntemi ile ödeme yapıldı.");
      }
   }
```

IPayment arayüzünü implemente ederek sistemimizi genişletmiş olduk.Bu seferde sanal pos yöntemi ile ödeme çekilmesini istediler.

```
/// <summary>
/// ConcreteStrategy - Kredi kartı ile ödeme strategy'miz.
/// </summary>
class CreditCardStrategy : IPayment
{
    public void MakePayment()
    {
        Console.WriteLine("Kredi kartı yöntemi ile ödeme yapıldı.");
    }
}
```

Sınıflarımızı istekler doğrultusunda oluşturduk.

Hemen **Context**'imizi yani örneğimiz gereği **PaymentOperation** sınıfını hazırlayalım. Bir nevi ilgili ödeme işlemini sarmalayıp developerlara daha basic bir kullanım sunan **wrapper** sınıfımız.

```
/// <summary>
/// Context'imiz. IPayment strategy'mizin içeriğindeki metotları sarmalar.
/// </summary>
class PaymentOperation
{
    private IPayment _odeme;
    public PaymentOperation(IPayment _odeme)
    {
        this._odeme = _odeme;
    }

    public void MakePayment()
    {
        this._odeme.MakePayment();
    }
}
```

Sarmalayıcı sınıfıda **constructor injetion** yaparak hazırladığımıza göre hemen örneğimizin kullanımına bir göz atalım:

```
    static void Main(string[] args)
    {
        PaymentOperation paymentOperation = null;

        // Client gelecek olan değere göre runtime'da istediği gibi ödeme tipini seçebilir.
        string paymentType = "BankTransfer";

        // If-Else bloklarını ise gerektiğinde bir kaç satır Reflection kodu ile aşabiliriz.
        // Fakat gerekmedikçe over architectur'ada kaçınılmaması gerekmektedir.
        // Attığımız taş, ürküttüğümüz kurbağaya değecek mi? Buna karar vererek. :)

        if (paymentType == "BankTransfer")
        {
            paymentOperation = new PaymentOperation(new BankTransferStrategy());
        }
```

```
    else if (paymentType == "CreditCard")
    {
        paymentOperation = new PaymentOperation(new CreditCardStrategy());
    }
    else if (paymentType == "MailOrder")
    {
        paymentOperation = new PaymentOperation(new MailOrderStrategy());
    }

    paymentOperation.MakePayment();

    Console.ReadLine();
}
```

d) Bu tasarım şablonunu kullanan örnek bir program veriniz.

Yukarıdaki örnekler üzerinden Strategy Pattern nasıl kullanıldığını anlatılmıştır. Strategy Pattern sayesinde sürekli bir genişlemeye sahip, güvenli ve kullanışlı bir sınıf oluşmaktadır.

İkinci Kısım Anti Pattern : Spaghetti Code, Warm Bodies, Smoke and Mirrors

a) Bu Antipattern'I ortaya çıkaran kök neden (root cause) nedir?

Spaghetti Code
Bilgisayar kodlamasında, bir kodun okunabilirliğinin düşük olması, yani kod takibinin zor olması durumunda, koda verilen isimdir. Bu tip komutların kullanılması durumunda, kodun hem diğer programcılar tarafından okunabilirliği düşer, hem de kodun karmaşıklığı kestirilemez bir hâle gelebilir. Genellikle yapısal programlama dillerinde (structured programming languages) fonksiyonların bulunması ile birlikte GOTO veya JMP gibi, kodun içerisinde bir yerden başka bir yere atlayan komutların kaldırılması mümkündür.

Warm Bodies
Temel bir sorun, teknoloji tüketicilerinin genellikle açıklığın bazı faydalar ile geldiğini varsaydıklarıdır. Gerçekte, standartlar, standartların kullanıcılara sunabileceği herhangi bir faydadan ziyade, marka tanıma için teknoloji tedarikçileri için daha önemlidir.

Smoke and Mirrors
Bir ürün oluştururken yanıltıcı araçlar kullanmaktır. Ürünün geleceği açısından büyük sorunlar çıkaracaktır.

b) Bu Antipattern'e verilen başka isimler varsa nelerdir?

Spaghetti Code : -

Warm Bodies : Ayrıca Deadwood, Seat Warmers, Mythical Man Month isimleri kullanılmaktadır.

Smoke and Mirrors : Ayrıca Vaporware ismide kullanılmaktadır.

c) Bu Antipattern yazılım tasarım-seviyesi modelinde nereye karşılık gelir?

Spaghetti Code : yazılım tasarım seviyesi modelinde Application (uygulama) karşılık gelmektedir. Refactored çözüm türü ise software olarak bilinir. Bu antipattern Yazılım Geliştirme antipattern'leri arasında yer almaktadır.

Warm Bodies : yazılım tasarım seviyesi modelinde Software Architecture AntiPattern yani yazılım mimari antipattern'leri arasında yer almaktadır.

Smoke and Mirrors : yazılım tasarım seviyesi modelinde Project Management AntiPattern yani proje yönetimi antipattern'leri arasında yer almaktadır.

d) Bu Antipattern hangi problem sınıfına bir çözüm önermektedir?

Spaghetti Code
Spagetti kodlama çok az yazılım mimari yapısı içeren sistemler olarak görünür. Kodlama ve progresif uzantılar yazılım yapısından yazılımın yaşam süresi boyunca uzak kalmış bir geliştiriciye bile netlikten uzak tutar.

Warm Bodies
Bir kuruma bağlı çalışan programcılar, Warm Bodies mini-AntiPattern'den kaynaklanan proje arızaları ve taşmalara kıyasla makul oranlarda önemli yazılım ürünleri üretebilirler.

Smoke and Mirrors
Expections yöntemi ile ilgilidir. Son kullanıcı yeteneklerinin en azından önceden belirlenmiş işlevsellik, bütçe ve program dahilinde olmadıklarında teslim edilebileceğine inanmaktadır.

e) Bu Antipattern hakkında detaylı bilgi veriniz.

Spaghetti Code
Spagetti Kod AntiPattern klasik ve en ünlü AntiPattern; programlama dillerinin icadından beri bir biçimde veya başka bir şekilde var olmuştur. Nesnesiz yönelimli diller bu AntiPattern'e daha duyarlı görünmektedir, ancak nesne yöneliminin altında yatan gelişmiş kavramları henüz tam olarak ustaca bilmeyen geliştiriciler arasında oldukça yaygındır.

Warm Bodies
Uygunluğu sağlamak için mekanizmalar olduğundan çok daha fazla bilgi sistemi standardı vardır. Bilgi sistemleri standartlarının sadece yüzde 6'sı test süitlerine sahiptir. Test edilebilir standartların çoğu dil derleyicileri - FORTRAN, COBOL, Ada, vb.

Bir Wolf Ticket, uygulanabilir bir anlamı olmayan standartlara açıklık ve uygunluk iddia eden bir üründür. Ürünler yayınlanan standarttan önemli ölçüde değişebilen patentli arayüzlerle teslim edilir. Temel bir sorun, teknoloji tüketicilerinin genellikle açıklığın bazı faydalar ile geldiğini varsaydıklarıdır. Gerçekte, standartlar, standartların kullanıcılara sunabileceği herhangi bir faydadan ziyade, marka tanıma için teknoloji tedarikçileri için daha önemlidir.

Standartlar, teknoloji geçiş maliyetlerini düşürür ve teknoloji istikrarını artırır, ancak standartların uygulanmasındaki farklılıklar, genellikle, birden çok kuruluşla birlikte çalışabilirlik ve yazılım taşınabilirliği gibi varsayılan faydalarını ortadan kaldırır.

Ayrıca, birçok standart özellikleri birlikte çalışabilirliği ve taşınabilirliği sağlamak için çok esnektir; Diğer standartlar aşırı derecede karmaşıktır ve bunlar ürünlerde eksik ve tutarsız olarak uygulanır. Sıklıkla, standardın farklı alt kümeleri çeşitli satıcılar tarafından uygulanır.

Wolf Biletleri, fiili standartlar için (popüler kullanım veya piyasaya maruz kalma yoluyla oluşturulan gayri resmi bir standart) önemli bir sorundur. Ne yazık ki, bazı fiili standartların etkili bir özelliği yoktur; Örneğin,

her satıcısına özgü, çoklu tescilli arayüzlerle ticari olarak temin edilebilen, yeni bir veri tabanı teknolojisi, fiili bir standart haline gelmiştir.

Smoke and Mirrors :

Gösteri sistemleri, genellikle son kullanıcılar tarafından üretim kalitesi özelliklerinin temsili olarak yorumlandığı için önemli satış araçlarıdır. Yeni iş için istekli bir yönetim ekibi, bazen (yanlışlıkla) bu yanlış algılamaları cesaretlendirir ve organizasyonun operasyonel teknolojiyi sunma yeteneklerinin ötesinde taahhütler yapar.

Bu, geliştiricileri zor bir duruma sokar, çünkü daha sonra söz verilen yetenekleri sunmak için baskı yapılır. Nihai kaybeden, söz verilen maliyet ve zamanda beklenen kapasiteyi almayan son kullanıcıdır; veya sistem teslim edildiğinde, son kullanıcı genellikle gösteri ortamını andıran aceleyle hazırlanmış bir Stovepipe Sistemi alır, ancak bakımdan yoksundur.

Bu AntiPattern'in, Vaporware olarak da adlandırılan başlıca nedeni, expections yönetimiyle ilgilidir. Son kullanıcı, yeteneklerin, en azından önceden belirlenmiş işlevsellik, bütçe ve program dahilinde olmadıklarında teslim edilebileceğine inanmaktadır. Geliştirme projesi teslim edilemediğinde, proje başarısız kabul edilir ve geliştirme kuruluşu güvenilirliğini kaybeder.