KTO Karatay Üniversitesi

Fen Bilimleri Enstitüsü

Elektrik ve Bilgisayar Mühendisliği A.B.D.

Nesne Yönelimli Yazılım Mühendisliği

Final Ödevi

Hatice TEKİŞ

21743387

29/06/2018

1. Proxy Pattern

1.1 Bu tasarım şablonunun kullanılmasındaki motivasyon nedir?

Proxy tasarım deseni, structural grubuna ait oluşturulması karmaşık veya oluşturulması zaman alan işlemlerin kontrolünü sağlar. Bir domain alanı modelindeki tüm nesnelerin, bir sistemin çeşitli sorumluluklarını yerine getirdiklerinde kullanacakları programların kullanılabilir olmasını isteriz. Aynı zamanda, birçok nesnenin aynı programın çalıştırılmasından çalışmaya devam etmesi de önemlidir.

Ancak, büyük bir sistemde, bir program başladığında tüm nesnelerin belleğe yüklenmesi pratik ve uygun olmaz. Bellek boyutu sınırlıdır; veritabanını belleğe yüklemek uzun zaman alır; ve bir programın herhangi bir belirli çalışması sırasında veritabanındaki nesnelerin sadece az bir kısmına ihtiyaç duyulacaktır. Tüm nesneleri bellekte tutmak, aynı zamanda birden çok programın aynı nesneleri paylaşmasını da zorlaştırır.

Tüm nesneler bellekte konumlanmışçasına uygulamayı programlayabilmek ideal olacaktır. Nesnelerin gerçekte nasıl depolandığına ve yükleneceğine dair ayrıntılar programcıya şeffaf olmalıdır.

Bunun yanında istemcinin (client) işlem yapan sınıfa doğrudan temasa geçmemesi için proxy pattern (vekil kalıbı) kullanılır. Böylece işlemin gerçekleşme performansında bir azalma olmaması sağlanır. Bu tasarım patterni fazla yük getiren "HeighWeigh" işlemlerde performans sorununu ortadan kaldırmak için kullanılır.

Bu pattern, pahalı hesaplamaları azaltarak, yalnızca gerekli olduğunda belleği kullanarak veya bir nesneyi belleğe yüklemeden önce erişimi kontrol ederek bir sistemin performansını veya güvenliğini artırmak amacıyla kullanılmaktadır.

1.2 Bu tasarım şablonu hakkında detaylı bilgi veriniz.

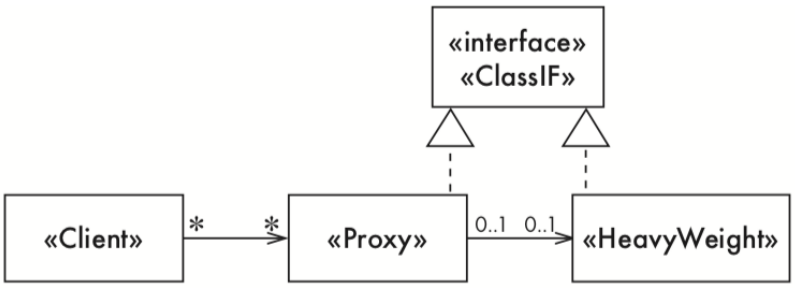
Bir proxy, en genel haliyle, başka bir şeye arabirim olarak çalışan bir sınıftır. Proxy, herhangi bir şeye (ağ bağlantısına, bellekte büyük bir nesneye, dosya veya çoğaltılması pahalı veya imkansız olan başka bir kaynağa) arabirim kurabilir. Kısaca, bir proxy patterni, istemci tarafından erişilebilen, bir objeyi başka bir sınıfın işlevselliğini temsil

eden başka bir tasarım objesi olarak adlandırabiliriz.

Proxy modelinde, işlevselliğini dış dünyaya bağlamak için orijinal nesneye sahip nesne yaratılır. Proxy kullanımı sadece gerçek nesneye yönlendirilebilir veya ek mantık sağlayabilir. Proxy'de, örneğin, gerçek nesneler üzerindeki işlemler kaynak yoğun olduğunda önbelleğe alma veya gerçek nesneler üzerindeki işlemlerden önce ön koşulların kontrol edilmesi gibi ekstra işlevler sağlanabilir. İstemci için, bir proxy nesnesinin kullanımı gerçek nesneyi kullanmaya benzer, çünkü her ikisi de aynı arabirimi uygular.

Bu model, bir sistemin mimarisinin yönlerinin nasıl uygulanacağını gösteren sınıf diyagramlarında bulunur. Çoğu zaman, bir programda bir sınıfın objesine (instance) erişmek zaman alıcı ve karmaşıktır. Bu tür sınıflara *"HeavyWeight"* diyoruz. *HeavyWeight* sınıfının örnekleri, örneğin, her zaman bir veritabanında bulunabilir. Bir programdaki örnekleri kullanmak için, bir constructor bunları veritabanından veriyle yüklemelidir. Benzer şekilde, *HeavyWeight* bir nesne sadece bir sunucuda bulunabilir: nesneyi kullanmadan önce, bir istemcinin sorgu göndermesi gerekir; Müşteri daha sonra nesnenin gelmesini beklemek zorundadır.

Her iki durumda da, nesnenin belleğe oluşturulmasında yer alan bir zaman gecikmesi ve karmaşık bir mekanizma vardır. Yine de, sistemdeki diğer birçok nesne *HeavyWeight* sınıflarına başvurmak veya kullanmak isteyebilir.



Şekil 1: Proxy tasarım patternin UML diyagramı

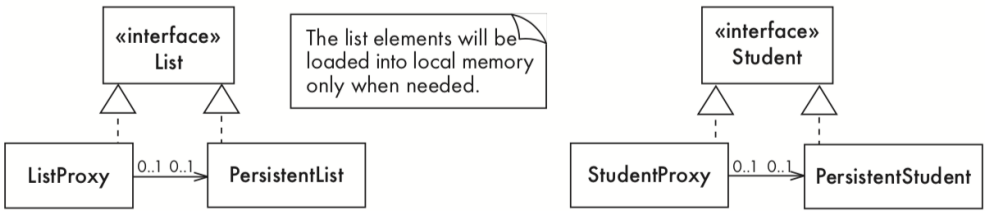
Yukarıdaki diyagramda *Client*, *HeavyWeight* içindeki verilere erişmek isterler. Fakat

performansın daha iyi olması ve daha güvenilir olması adına verilere erişmek için *HeavyWeight*'den *Proxy* nesnesi oluşturulur. *Proxy* nesnesi, normalde yalnızca bir yer tutucu görevi görür. *HeavyWeight* ve *Proxy*'in aralarında iletişimi *interface ClassIF* ile sağlanır. Oluşan *HeavyWeight* kopyaları "call by value" olarak metotlarda kulllanılabilir ve sorgulanan veriye *Proxy* ile erişilir.

Proxy üzerinde bir işlem yapmak için herhangi bir girişimde bulunulursa, o zaman *Proxy*, işlemi *HeavyWeight* a devreder. Gerektiğinde, *Proxy*, gerçek *HeavyWeight* nesnesini elde eden pahalı görevi üstlenir. Vekil sadece bir kez *HeavyWeight* almak zorundadır; daha sonra bellekte kullanılabilir ve erişim bu nedenle hızlıdır.

Bazı proxy'ler,*HeavyWeight* yükleme çabası olmaksızın gerçekleştirilebilecek sınırlı sayıda işlem gerçekleştirebilir. Bazı sistemlerde, domain alanı modeli tarafından manipüle edilen değişkenlerin çoğu aslında "Proxy" sınıflarının örneklerini içerir.

1.3 Bir örnekle kullanımını açıklayınız.

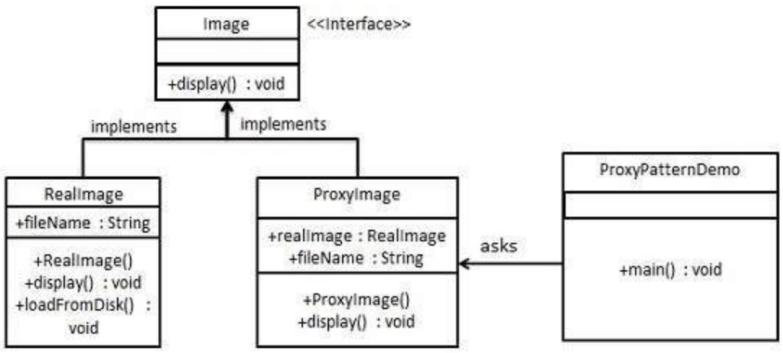


Şekil 2: Proxy tasarım patterni ile öğrenci verisine erişim örneğinin UML diyagramları

Şekil 6.13'te, bir yazılım tasarımcısı bir değişkenin bir liste içereceğini beyan edebilir. Bununla birlikte, bu değişken aslında bir *ListProxy* içerecektir, çünkü nesnelerin bütün bir listesini belleğe yüklemek pahalı olacaktır ve bütün liste aslında gerekli olmayabilir. Ancak, bir işlem listeye erişir erişmez, ListProxy bu noktada bellekteki PersistentList örneğini oluşturabilir. Öte yandan, *ListProxy*, *PersistentList*'i yükleme çabasına girmeden listedeki eleman sayısı gibi belirli soruları yanıtlayabilir.

Şimdi, Persistent List'in aslında bir öğrenci listesi olduğunu düşünelim. PersistentList'te öğrenci ile ilgili her detayı burada tutuyoruz. Öğrencinin almış olduğu dersler,

hangi okuldan transfer olduğu, hangi liseden geldiği, hangi dersleri aldı veya drop ettiği dersleri, öğrencinin adres, telefon gibi yani tüm bilgilerini Persistent Listte tutuyoruz. Bu listenin sürekli olarak kopyalanmasını istemiyoruz ve bu listeye erişmek için bir StudentProxy nesnesi oluşturuyoruz. Daha sonra aradaki erişim için Student interface'nı tanımlıyoruz.



Şekil 3: Proxy tasarım patterni ile veritabanındaki resimlerin gösterilmesi örneğinin UML diyagramları

Bu örnekte gerçekte gösterilecek büyük boyutta resimlerin olduğunu ve kullanıcıya resmi gösterirken resimler yüklenene kadar ön izleme resmini gösteriyor olalım. *Image* arayüzünü ve *Image* arayüzünü uygulayan somut dersler oluşturacağız. *ProxyImage*, *RealImage* nesne yüklemesinin bellek ayak izini azaltmak için bir proxy sınıfıdır.

ProxyPatternDemo, bizim demo sınıfmızdır ve resmi yüklemek ve görüntülemek için Image nesnesi oluşturmak gerekir. Bu nesneyi oluşturmak için ProxyImage kullanacaktır.

```
1 public interface Image {
2    void display();
```

Kod dosyası 1: Image.java

```
1 public class RealImage implements Image {
2
3    private String fileName;
4
```

```
5     public RealImage(String fileName){
6        this.fileName = fileName;
7        loadFromDisk(fileName);
8     }
9
10    @Override
11    public void display() {
12       System.out.println("Displaying " + fileName);
13    }
14
15    private void loadFromDisk(String fileName){
16       System.out.println("Loading " + fileName);
17    }
18 }
```

Kod dosyası 2: RealImage.java

```
1  public class ProxyImage implements Image{
2
3     private RealImage realImage;
4     private String fileName;
5
6     public ProxyImage(String fileName){
7        this.fileName = fileName;
8     }
9
10    @Override
11    public void display() {
12       if(realImage == null){
13          realImage = new RealImage(fileName);
14       }
15       realImage.display();
16    }
17 }
```

Kod dosyası 3: ProxyImage.java

```java
public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      //image will be loaded from disk
      image.display();
      System.out.println("");

      //image will not be loaded from disk
      image.display();
   }
}
```

Kod dosyası 4: ProxyPatternDemo.java

1.4 Bu tasarım şablonunu kullanan örnek bir program veriniz.

Resimleri yükleme ve gösteren programlarda, film-dizi sitelerinde; Netflix (Mesela; bir film sitesinden film izlenirken, filmin indirilmesi beklenmez. Arka tarafta proxy tasarım kalıbı oluşturularak parça parça işlem yapılır ve zaman kaybı önlenmiş olur.), üniveris,telerin öğrenci veri tabanı programlarında kullanılır. Proxy tasarım deseni çalışma maliyeti yüksek işlemlerin olduğu yapılarda, web servisi kullanılan yapılarda, remoting uygulamalarında, operasyonun gerçekleştirilmesinden önce hazırlık yapılması veya ön işlem yapılması durumlarında kullanılır.

Proxy, ağ bağlantısında, bellekte büyük bir nesne içeren programlarda, dosya veya çoğaltılması pahalı veya imkansız olan başka bir kaynaklarda kullanılır.

Proxy (Vekil) tasarım deseni;

- Oluşturulması zaman alan bir nesne yaratılması gerektiğinde,
- Uzaktan erişilerek bir nesne yaratılması gerektiğinde,
- Uzaktan erişilerek bir nesne yaratılması gerektiğinde Nesneye erişmeden önce bazı kontroller yapılması gerektiğinde,

- Nesneye erişimin kısıtlı olduğunda yararlı olabilir.

2. Functional Decomposition

- Bu antipatterni ortaya çıkaran kök neden (root cause) nedir?

  Functional decompostion antipattern'ini ortaya çıkaran kök nedenleri (root causes) şunlardır: *Avarice, Greed, Sloth*

- Bu Antipattern'e verilen başka isimler varsa nelerdir?

  No Object Oriented AntiPattern "No OO"

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

  Most Frequent Scale: Uygulama (Application) leveline karşılık gelir.
  Bu antipattern *(Software Development Antipattern)* yazılım geliştirme antipattern'i grubundadır.

- Bu Antipattern hangi problem sınıfına bir çözüm önermektedir?

  Refactored Çözüm Tipi: Process'tir. Process sınıfına çözümm önermektedir.

- Bu Antipattern hakkında detaylı bilgi veriniz.

  Bu AntiPattern, nesne yönelimli bir dilde bir uygulama tasarlayan ve uygulayan deneyimli, nesnel olmayan yönlendirici geliştiricilerin çıktısıdır. Ortaya çıkan kod, sınıf yapısında bir yapısal dile (Pascal, FORTRAN) benzemektedir. Akıllı prosedür geliştiricileri, zamana göre test edilmiş yöntemleri nesne odaklı mimaride çoğaltmak için çok "akıllı" yollar tasarlarken inanılmaz derecede karmaşık olabilir.

  İşlevsel Ayrıştırma, prosedürel bir programlama ortamında iyidir. Daha büyük ölçekli bir uygulamanın modüler yapısını anlamak için bile kullanışlıdır. Maalesef, doğrudan bir sınıf hiyerarşisine dönüştürülmez ve sorunun başladığı yer burasıdır.

3. Cover Your Assets

- Bu antipatterni ortaya çıkaran kök neden (root cause) nedir?

  Bu antipatterni ortaya çıkaran kök nedenler yoktur.

- Bu Antipattern'e verilen başka isimler varsa nelerdir?

  Bu antipatterne başka isimler verilmemiştir.

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

  Most Frequent Scale: Architecture System leveline karşılık gelir.

  Bu antipattern *(Software Architecture Antipattern)* yazılım mimari antipattern'i grubundadır.

- Bu Antipattern hakkında detaylı bilgi veriniz.

  Belgelere dayalı yazılım süreçleri, genellikle yararlı gerekliliklerden ve özellikleri daha az üretirler, çünkü yazarlar önemli kararlar vermekten kaçınırlar. Bir hata yapmamak için, yazarlar daha güvenli bir yol izler ve alternatifler üzerinde dururlar. Elde edilen belgeler hacimli ve bir gizem haline geldi; yazarların niyetini taşıyan içeriklerin yararlı bir soyutlaması yoktur. Metne bağımlı sözleşme yükümlülüklerine sahip olan talihsiz okuyucular, zihin uyuşma ayrıntılarına dikkat kesilirler. Karar verilmediğinde ve öncelikler belirlenmediğinde, belgelerin değeri sınırlıdır. Aynı derecede önemli veya zorunlu olan yüzlerce sayfa gereksinimine sahip olmak mantıksızdır. Geliştiriciler, hangi öncelikte ne yapılacağı konusunda pek yararlı olmayan rehberler bırakmaktadır.

  Mimari planlar, kullanıcılar ve geliştiriciler arasındaki gereksinimlerin ve teknik planların iletişimini kolaylaştıran bilgi sistemlerinin soyutlamalarıdır [Blueprint 97]. Mimari bir plan, mevcut ve gelecekteki bilgi sistemlerinin operasyonel, teknik ve sistem mimarisini ileten küçük bir dizi şema ve çizelgedir [C4ISR 96]. Tipik bir blueprint, bir düzine diyagramdan ve tablodan daha fazlasını içermez ve bir görüş sunumu olarak bir saat veya daha kısa bir sürede sunulabilir. Mimari planlar, çoklu bilgi sistemlerine sahip bir kuruluşta özellikle kullanışlıdır. Her sistem, mimari plan-

larını oluşturabilir, daha sonra kuruluş, sisteme özgü detaylara dayanarak girişimsel planları derleyebilir. Blueprints hem mevcut sistemleri hem de planlanan uzantıları karakterize etmelidir. Uzantılar, çoklu sistemlerde mimari planlamayı koordine etmek için kullanılabilir. Mimari planlar, birden fazla projenin kendi teknolojilerini tasvir etmesine izin verdiğinden, birlikte çalışabilirlik ve yeniden kullanım için fırsatlar geliştirilmiştir.

4. Fear of Success

- Bu antipatterni ortaya çıkaran kök neden (root cause) nedir?

  Bu antipatterni ortaya çıkaran kök nedenler yoktur.

- Bu Antipattern'e verilen başka isimler varsa nelerdir?

  Bu antipatterne başka isimler verilmemiştir.

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

  Bu antipattern *(Project Management Antipattern)* proje yönetimi antipattern'i grubundadır.

- Bu Antipattern hakkında detaylı bilgi veriniz

  Bir proje başarılı bir şekilde tamamlandığında insanlar (yazılım geliştiriciler dahil) çılgınca şeyler yaparlar. Proje tamamlandığında, açık bir başarı beyanı hazırlanmalıdır.

  İlginç bir fenomen, insanlar ve projeler başarının eşiğinde olduğunda sıklıkla ortaya çıkar. Bazı insanlar yanlış gidebilecek şeyler hakkında takıntılı bir şekilde endişelenmeye başlar. Mesleki yeterlilik konusunda güvensizlikler gün ışığına çıkıyor. Açıkça tartışıldığında, bu endişeler ve güvensizlikler proje ekibi üyelerinin zihnini işgal edebilir. Mantıksız kararlar alınabilir ve bu endişeleri gidermek için uygun olmayan önlemler alınabilir. Örneğin, bu tartışmalar, proje ekibinin dışında, teslim edilebilirliğin nasıl algılandığını etkileyen olumsuz bir sonuç üretebilir ve sonuç olarak proje sonucu üzerinde yıkıcı bir etkiye sahip olabilir.

Başarının Korkusu, son verme sorunları ile ilgilidir. Genel olarak grup dinamiği, hem bir hafta süren projeler hem de uzun süreli çabalar için fark edilebilen bir dizi aşamadan ile ilerliyor. İlk aşama grup kabul konularını ele alır. İkinci aşamada, ilişkiler kuruldukça, bireyler, hem grup içindeki resmi rolleri hem de resmi olmayan kendi belirledikleri rolleri içeren grupta çeşitli roller üstlenirler. Bu ekip oluşturmada önemli bir faktördür. Roller oluşturulduktan sonra, iş başarılır (üçüncü aşama). Bu aşamada birçok kişilik sorunu ortaya çıkabilir. Projenin tamamlanması, grubun dağılmasıyla sonuçlanabileceğinden, bu sorunlar genellikle proje sonlandırma yaklaşımları olarak ortaya çıkar (dördüncü aşama). Fesih aşamasında, projenin sonuçları, gelecekteki yaşam döngüsü ve grubun sonraki faaliyetleri ile ilgili kaygılar genellikle dolaylı yollarla ifade edilir. Başka bir deyişle, insanlar çılgın şeyler yapar.

Bir projenin bitiminde yönetimin alabileceği önemli bir eylem, başarıyı beyan etmektir. Fiili sonuç belirsiz olsa bile, projenin başarılı sonucunu destekleyen yönetim beyanlarına ihtiyaç vardır. Proje ekibinin başarılarının ve öğrenilen derslerin önemini kabul etmesine yardımcı olmak önemlidir. Başarı beyan etmek aynı zamanda sonlandırma sorunlarını azaltmaya, ekibin organizasyona olan bağlılığını sürdürmeye ve gelecekteki proje aktivitelerine zemin hazırlamaya yardımcı olur. Sertifika veya plakların verilmesiyle birlikte verilen ödül töreni, bu amaç için uygun bir harekettir. Profesyonel tanıma ucuz ve alıcılar tarafından çok değerlidir.