EBMYL 512

Nesneye Yönelik Yazılım Mühendisliği Dersi Ödevi

2017-2018 Bahar Dönemi

Yrd. Doç. Dr. Ali ÖZTÜRK

TASARIM ŞABLONLARI (DESING PATTERNS)

ADAPTER PATTERN

&

ANTIPATTERNS

Continuous Obsolescence, Stovepipe Enterprise, Analysis Paralysis

Emrehan EKENTOK
29.06.2018

TASARIM ŞABLONLARI (DESING PATTERN) :

Yazılım geliştirme süreçlerinde yazılan kodların tekrar kullanılabilirliğini sağlama, ortaya çıkan yazılımsal problemleri çözüm sağlama ve bu çözümlerin standartlar haline getirip yeniden kullanılabilir şekilde kodlanması sonucu oluşan sisteme Design Pattern (tasarım şablonu) denir. Tasarım şablonları, yazılım geliştiricilerin karşılaştıkları sorunlar karşısında zaman içerisinde ürettikleri çözümler ele alınarak ortaya çıkartılan standart ve en uygun çözüm yöntemidir. Bu yöntemler tüm yazılım geliştiriciler tarafından kullanılabilirler.

Standart olarak kabul edilen tasarım şablonları ;

1- Creatinal Patterns (Kurucu Desenler): Nesnelerin oluşturulması ve yönetilmesi ile ilgili desenlerdir.

- **Singleton:** Uygulamanın yaşam süresince bir nesnenin bir kez oluşturulmasını sağlar.
- **Abstract Factory:** Birbirleri ile ilişkili sınıfların oluşturulmasını düzenler.
- **Builder:** Biden fazla parçadan oluşan nesnelerin üretilmesinden sorumludur.
- **Factory Method:** Aynı arayüzü kullanan nesnelerin üretiminden sorumludur.
- **Prototype:** Var olan nesnelerin kopyasının üretiminden sorumludur.

2- Behavioral Patterns (Davranışsal Desenler): Birden fazla sınıfın bir işi yerine getirirken nasıl davranacağını belirleyen desenlerdir.

- **Chain of responsibility:** Bir isteğin belli sınıflar içinde gezdirilerek ilgili sınıfın işlem yapmasını yönetir.
- **Command:** İşlemlerin nesne haline getirilip başka bir nesne(invoker) üzerinden tetiklendiği bir tasarım desenidir.
- **Interpreter:** İşlemlerin nesne haline getirilip başka bir nesne(invoker) üzerinden tetiklendiği bir tasarım desenidir.
- **Iterator:** Nesne koleksiyonlarının elemanlarını belirlenen kurallara göre elde edilmesini düzenler.
- **Mediator:** Çalışmaları birbirleri ile aynı arayüzden türeyen nesnelerin durumlarına bağlı olan nesnelerin davranışlarını düzenler.
- **Memento:** Bir nesnenin tamamının veya bazı özelliklerinin tutularak sonradan tekrar elde edilmesini sağlar.
- **Observer:** Bir nesnede meydana gelen değişikliklerde içinde bulundurduğu listede bulunan nesnelere haber gönderen tasarım desenidir.
- State: Nesnelerin farklı durumlarda farklı çalışmalarını sağlar.
- **Strategy:** Bir işlemin birden fazla şekilde gerçekleştirile bilineceği durumları düzenler.
- **Template method:** Bir algoritmanın adımlarının abstract sınıfta tanımlanarak farklı adımların concrete sınıflarında overwrite edilip çalıştırılmasını düzenler.
- **Visitor:** Uygulamada ki sınıflara yeni metotlar eklenmesini düzenler.

3- Structural Patterns (Yapısal Desenler): Nesnelerin birbirleri ile olan ilişkilerini düzenleyen desenlerdir.

- **Adapter:** Uygulamada ki bir yapıya dışarıdaki bir yapıyı uygulamayı düzenler.
- **Bridge:** Nesnelerin modelleme ve uygulanmasını ayrı sınıf hiyerarşilerinde tanımlanmasını düzenler.
- **Composite:** Ağaç yapısında ki nesne kalıplarının hiyerarşik olarak iç içe kullanılmasını düzenler.
- **Decorator:** Bir yapıya dinamik olarak yeni metotlar eklenmesini düzenler.
- **Facade:** Alt sistemlerin direkt olarak kullanılması yerine alt sistemdeki nesneleri kullanan başka bir nesne üzerinden kullanılmasını sağlar.
- **Flyweight:** Sık kullanılan nesnelerin bellek yönetimini kontrol etmek için kullanılan bir tasarım desenidir.
- **Proxy:** Oluşturulması karmaşık veya oluşturulması zaman alan işlemlerin kontrolünü sağlar.

ADAPTER PATTERN :

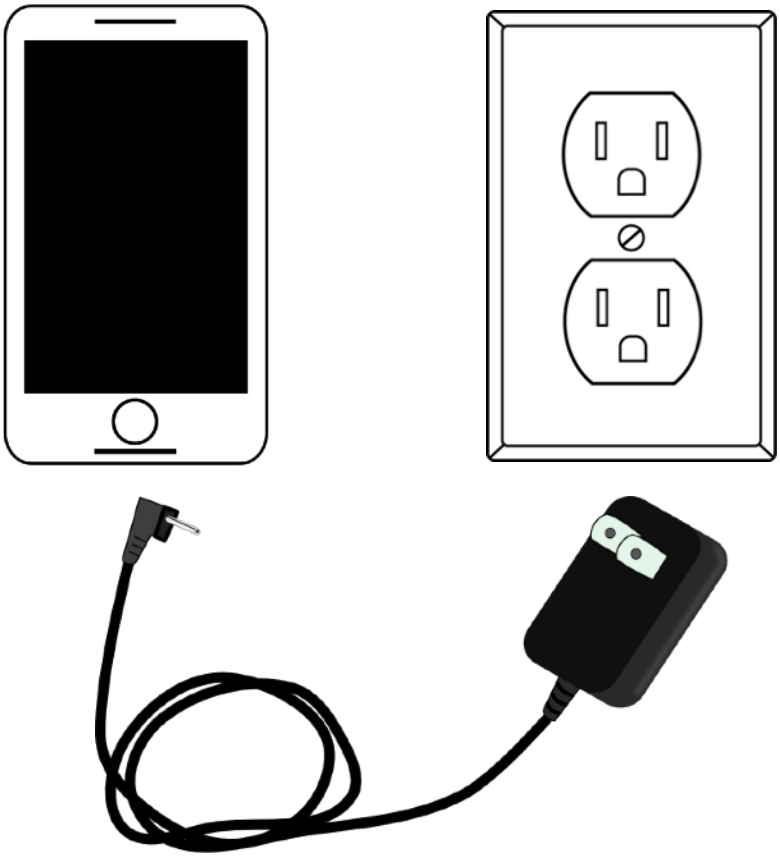
Adaptör Pattern, sadece bir sınıfa özel olan arayüzleri diğer sınıflarla uyumlu arayüzler haline getiren bir tasarım şablonudur. Adaptörler uyumlu olmayan arayüzler nedeniyle birbirleri ile çalışamayan sınıflara da birbirleri ile çalışma imkanı sunarlar.

Bazı durumlarda bir araç takımı (toolkit) ya da sınıf kütüphanesinin (framework) arayüzü uygulama için gerekli olan arayüz ile uyumsuz olduğu için kullanılamaz. Bu durumda kütüphane değiştirilmez, çünkü kaynak kodu bilinmiyor olabilir. Kaynak kodun bilindiği düşünülecek olsa bile her bir uygulama için tek tek kütüphaneyi değiştirmek doğru bir çözüm değildir. Adaptör kalıp bu gibi durumlarda devreye girmesi için geliştirilmiştir. Kısaca adaptör kalıp, kullanılmak istenen bir sınıfın kullanılacak arayüzün gereklilikleri ile eşleşmediği zamanlarda kullanılır.

Adaptör Pattern'in başlıca özellikleri şunlardır:

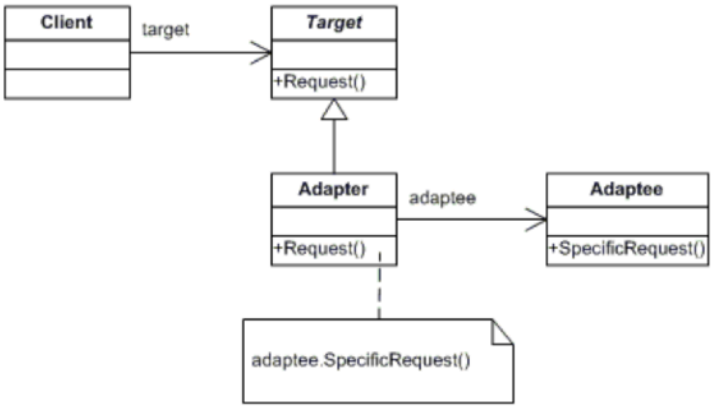
- Bir sınıfın arayüzünü istemcinin istediği yeni bir arayüze dönüştürme işlemi yapar.
- Birbiriyle uyumsuz arayüzleri olan sınıfların birbiriyle çalışmasını sağlar.
- Var olan sınıfı yeni bir arayüzle sarmalar.
- Eski bir bileşeni yeni oluşturulan sisteme entegre eder.

Adapter Pattern'in Günlük Hayat Örneklemesi :



Resimlerden de anlaşılacağı üzere, cep telefonunun şarj olması için gerekli elektrik voltajı DC 5 V dir. Ancak şehir şebekesindeki elektrik voltajı AC 220 V'dır. Bu neden ile cep telefonunu şarj edebilmemiz için bir dönüştürücüye gerek duyarız. Şehir şebekesinden gelen AC 220 V'u DC 5 V'a çeviren adaptör sayesinde telefonumuzu şarj edebiliriz. Bu Bağlamda günlük hayatta çoğu zaman çevirici/dönüştürücüleri sıklıkla kullanırız.

ADAPTER PATTERN'İN ÇALIŞMA MANTIĞI :



Adapter tasarım şablonunda 4 temel yapı vardır. Bunlar;

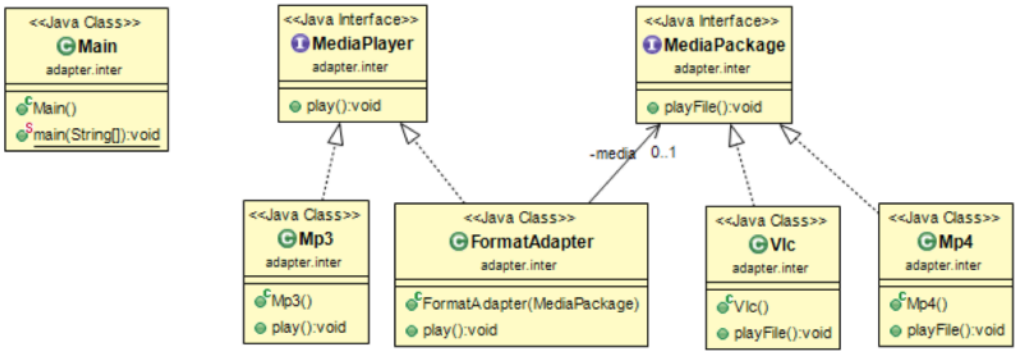
1- Adaptee : Mevcut sisteme uygun hale getirilmek istenen nesne.

2- Adapter : Mevcut sisteme göre uygunlaştırma işlemi yapan nesne.

3- Target : İhtiyaç duyulan class veya interface.

4- Client: İşlemleri gerçekleştirdiğimiz ortak sınıf. İstemci , uygulama.

Adaptör için özetle **mevcut bir sınıfı veya arayüz sınıfını, eldeki farklı bir arayüz sınıfına uygun hale getirerek tekrar kullanmak amacıyla uygulanır.** Genellikle işe yarayacağını düşündüğümüz mevcut bir sınıfı kendi sistemimizde tekrar kullanmak isteriz. Ne yazıkki mevcut sınıf, ihtiyaçlarımıza tam uymayabilir. Bu durumda araya bir tane adaptör kod yazarak, mevcut sınıfı kendi sistemimize uygun hale getirebiliriz. Böylece adapte edilen nesnede kod değişikliği olmadan benzer bir arayüzü destekler hale getiririz. Ayrıca adaptasyon işlemi sırasında, adapte edilen nesnenin desteklemediği özellikler de adaptör tarafından gerçekleştirilebilir diyebiliriz. Özetle, bir bağdaştırıcı iki uyumsuz arabirimin birlikte çalışmasına yardımcı olur. Adaptör Tasarım Kalıbını Java'da uygulamak için, aşağıdaki örneği seçiyoruz:



Burada iki uyumlu olmayan arayüzümüz var: MediaPlayer ve MediaPackage. MP3 sınıfı, MediaPlayer arabiriminin bir uygulamasıdır ve MediaPackage arabiriminin uygulamaları olarak VLC ve MP4'e sahibiz. MediaPackage uygulamalarını MediaPlayer örnekleri olarak kullanmak istiyoruz. Bu yüzden, iki uyumsuz sınıfla çalışmamıza yardımcı olacak bir bağdaştırıcı oluşturmamız gerekiyor.

Bağdaştırıcının adı **FormatAdapter**'tir ve MediaPlayer arabiriminde uygulamalıdır. Ayrıca, uyumsuz arabirim olan MediaPackage öğesine FormatAdapter sınıfının bağlantısı olması gerekir.

İki arayüzün kodu:

MediaPlayer.java

```
public interface MediaPlayer {
 void play(String filename);
}
```

MediaPackage.java

```
public interface MediaPackage {
 void playFile(String filename);
}
```

Uygulama Sınıflarının Oluşturulması :

MP3.java

```
public class MP3 implements MediaPlayer {
 @Override
 public void play(String filename) {
    System.out.println("Playing MP3 File " + filename);
 }
}
```

MP4.java

```
public class MP4 implements MediaPackage {
 @Override
 public void playFile(String filename) {
    System.out.println("Playing MP4 File " + filename);
 }
}
```

VLC.java

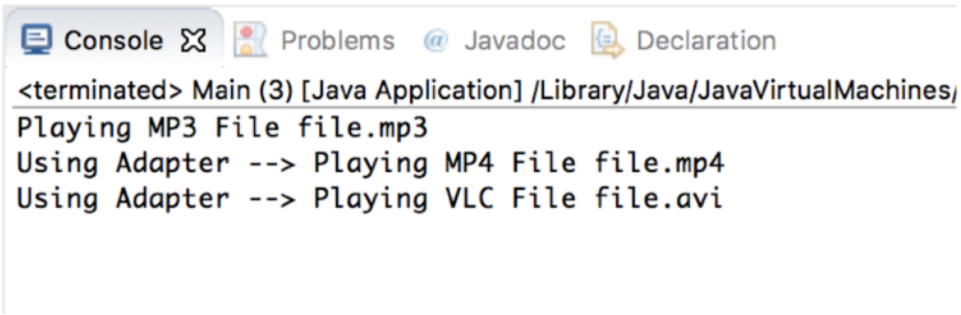
```
public class VLC implements MediaPackage {
 @Override
 public void playFile(String filename) {
    System.out.println("Playing VLC File " + filename);
 }
}
```

Bu noktada, VLC ve MP4 örneklerini MediaPlayer örnekleri olarak kullanmak istiyoruz. Yani, adaptör kullanmaya ihtiyacımız var. FormatAdapter'ın kodu aşağıdaki gibidir ;

```
public class FormatAdapter implements MediaPlayer {
 private MediaPackage media;
 public FormatAdapter(MediaPackage m) {
    media = m;
 }
 @Override
 public void play(String filename) {
   System.out.print("Using Adapter --> ");
   media.playFile(filename);
 }
}
```

Adaptör kullanmak için bir parçaları bir araya getirilme aşaması :

```
public class Main {
 public static void main(String[] args) {
    MediaPlayer player = new MP3();
    player.play("file.mp3");
    player = new FormatAdapter(new MP4());
    player.play("file.mp4");
    player = new FormatAdapter(new VLC());
    player.play("file.avi");
 }
}
```



Programın çalıştırılmasından sonra MP4 ve VLC' yi Adaptör aracılığıyla MediaPlayer örnekleri olarak kullanılabilmekteyiz.

ANTIPATTERN'LER

İlk kez 1995 yılında **Andrew Koenig** tarafından kullanılan Anti-Pattern kavramı, anti madde kavramı gibi biraz soyuttur. Aslında anti-pattern'ler de birer pattern'dir. Üzerine yazılan bir kitapla iyice popüler hale gelen bu kavram; yazılımsal bir problemi bilinen ve doğru çözüm diye kabul edilmiş bir patern'i kullanmadan ve özgün bir yöntemle çözmek anlamında kullanılmaktadır. Şüphesiz anti-pattern kavramının bu *diplomatik* tanımının dışında pratikte kötü çözüm, kötü fikir gibi algılanan bir yönü de vardır. Bu pejoratif algının sebebi; çoğu zaman ilk bakışta mükemmel gibi görünen bir çözümün sonradan çokça baş ağrıtacak olma olasılığıdır. Dolayısıyla ilk bakışta doğruymuş gibi gelen birtakım anti-pattern çözümlerden mümkün olduğunca kaçınılması gerekir.

Bir yazılım mühendisliği kavramı olan anti-pattern, kullanışsız çoğu zaman hatalı sonuçlar doğuran, yanlış geliştirilmiş çözümlere denir. Genellikle doğru olduğu düşünülerek gerçekleştirilen ve tecrübe eksikliğinin temel neden olduğu bu hatalar, projelerin başarısızlıkla sonuçlanmasına yol açar. Anti-pattern'lerin en büyük tehlikesi, davranışın ya da çözümün ilk etapta doğru olduğunun düşünülmesidir, bu durum sorunların uzun vadede ortaya çıkmasına neden olur.

CONTINUOUS OBSOLESCENCE (Mini Antipattern) :
Antipattern Adı : Continuous Obsolescence (Sürekli Eskime)
En Sık Ölçek : Development

Teknoloji o kadar hızlı değişiyor ki, geliştiriciler mevcut yazılım sürümlerini takip etmekte zorlanıyorlar ve birlikte çalışan ürün sürümleri kombinasyonlarını buluyorlar.

Her ticari ürün hattının yeni ürün sürümleriyle geliştiği göz önünde bulundurulduğunda, bu durumun geliştiriciler için başa çıkması gittikçe zorlaşıyor.Başarılı bir şekilde birlikte çalışan ürünlerin uyumlu sürümlerini bulmak daha da zor.

Java, bu fenomenin iyi bilinen bir örneğidir ve yeni versiyonlar birkaç ayda bir ortaya çıkar. Örneğin, Java 1.X'deki bir çalışmaya başlandığında, yeni bir Java Geliştirme Seti çalışmayı engeller. Sadece Java değil; Diğer birçok teknoloji de sürekli eskimektedir.

En temel örnekler, Product98 gibi sürüm tarihini marka isimlerine dahil eden ürünlerdir. Bu şekilde, bu ürünler eskimiş hissiyatını açığa çıkarır. Başka bir örnek, Microsoft dinamik teknolojilerin ilerlemesidir:

- DDE
- OLE 1.0
- OLE 2.0
- COM
- ActiveX
- DCOM
- COM?

Pazarlamacıların bakış açısına göre, iki temel faktör vardır: tüketici bilinci ve pazar paylaşımı. Hızlı inovasyon, tüketicilerin en son ürün özellikleri, duyurular ve terminoloji ile güncel kalmaları için özel ilgi gerektirir.

Teknolojiyi izleyenler için, hızlı inovasyon, tüketici bilinci sağlamaya katkıda bulunur; Diğer bir deyişle, teknoloji X hakkında her zaman yeni haberler vardır. Baskın bir pazar payı elde edildikten sonra, tedarikçilerin birincil gelirleri, eski ürün bültenlerinin eskimesi ve değiştirilmesiyle olur. Teknolojiler daha hızlı eskimez (ya da modası geçmiş gibi algılanır), gelir artar.

Hızlı eskimelerin avantajı teknolojinin geçişli bir sürece sahip olmasıdır. Mimarlar ve geliştiriciler, kararlı ya da kontrol ettikleri arayüzlere bağlı olmalıdır. Açık sistem standartları, kaotik bir teknoloji pazarına istikrarın bir ölçüsünü verir.

Elden Geçirilmiş (Refactored) Çözüm :

Teknoloji pazarında önemli bir dengeleyici faktör açık sistem standartlarıdır. Bir konsorsiyum standardı, zaman ve yatırım gerektiren bir endüstri konsensüsünün ürünüdür.

Ortak pazarlama inisiyatifleri, teknolojilerin ana akım haline gelmesiyle kullanıcı farkındalığını ve kabulünü oluşturuyor. Bu süreçte tüketiciye yarar sağlayan doğal bir etki vardır, Ürün belirli bir standarda uyumlu olduğu için, satıcının özelliklerini değiştirmesi mümkün değildir.

Varyasyonlar :

Wolf Ticket Mini-AntiPattern, tüketicilerin ürün yönünü daha iyi ürün kalitesine doğru etkilemek için kullanabilecekleri çeşitli yaklaşımları tanımlar.

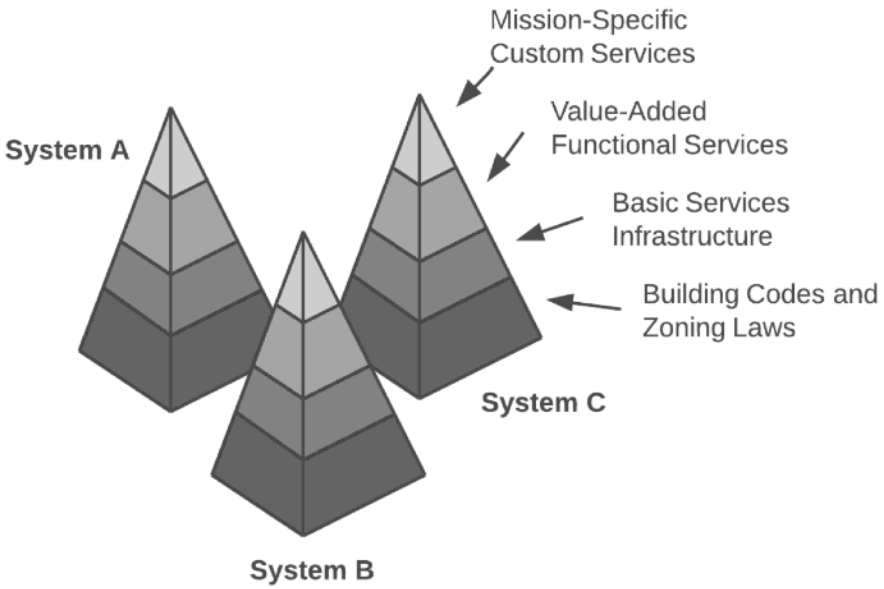
STOVEPIPE ENTERPRISE :

Antipattern Adı : Stovepipe Enterprise
Verilen Diğer İsimler : Island of Automation
En Sık Ölçek : Enterprise
Elden Geçirilmiş (Refactored) Çözüm Adı : Enterprise Architecture Planning
Elden Geçirilmiş (Refactored) Çözüm Tipi : Proses
Kök Nedenleri (Root Cause) : Haste, Apathy, Narrow–Mindedness
Dengesiz Kuvvetler : Management of Change, Resources, Technology Transfer

Bir işletmedeki, çoklu sistemler her düzeyde bağımsız olarak tasarlanmıştır. Ortak çalışmama nedeniyle, sistemler arasında birlikte çalışılamaz, sistemin yeniden kullanımını önler ve maliyetini artırır; Ayrıca, yeniden yapılandırılmış sistem mimarisi ve hizmetleri, uyumluluğunu destekleyen kaliteli yapıdan yoksundur.

En düşük seviyede standartlar ve kurallar vardır. Bunlar, kurumsal sistemler arasında mimari bina kodları ve imar yasaları gibi çalışır. Hiyerarşide bir sonraki seviye, altyapı ve nesne hizmetlerini içeren çalışma ortamıdır.

En üstteki iki katman, katma değerli fonksiyonel hizmetleri ve göreve özel hizmetleri içerir. Bu teknolojilerin tümünü bağımsız olarak seçerek ve tanımlayarak, Stovepipe Enterprises "kurumun geri kalanından izole edilmiş" otomasyon adaları yaratır.



Şekil 1: Soba Borusu İşletmeleri, her bir koordinasyon düzeyinde izole teknoloji kararlarından kaynaklanmaktadır.

Belirti ve Sonuçlar :

- Kurumsal sistemler arasındaki uyumsuz terminoloji, yaklaşımlar ve teknoloji.
- Kırılgan, monolitik sistem mimarileri ve mimarilerde eksik dökümantasyon.
- İş gereksinimlerini desteklemek için sistemleri genişletememe.
- Bir teknoloji standardının yanlış kullanımı.
- Yazılım eksikliği, kurumsal sistemler arasında yeniden kullanılır.
- Kurumsal sistemler arasında birlikte çalışabilirlik olmaması.
- Sistemlerin aynı standartları kullanırken bile birlikte çalışamaması.
- Değişen iş gereksinimleri nedeniyle aşırı bakım maliyetleri; Sistemi yeni ürünler ve teknolojilerle birleştirmek için ihtiyaç duyulmaktadır.
- Personel eksikliği projenin süreksizliğini ve bakım problemlerini ortaya çıkarır.

Tipik Nedenleri :

- Özellikle kurumsal teknoloji stratejisinin eksikliği:
  - Standart bir referans modelinin eksikliği
  - Sistem profillerinin eksikliği
- Sistem gelişmeleri genelinde işbirliği için teşvik eksikliği
- Sistem geliştirme projeleri arasında iletişim eksikliği.
- Kullanılan teknoloji standardının bilinmemesi.
- Sistem entegrasyon çözümlerinde yatay arayüzlerin olmaması.

Bilinen İstisnalar :

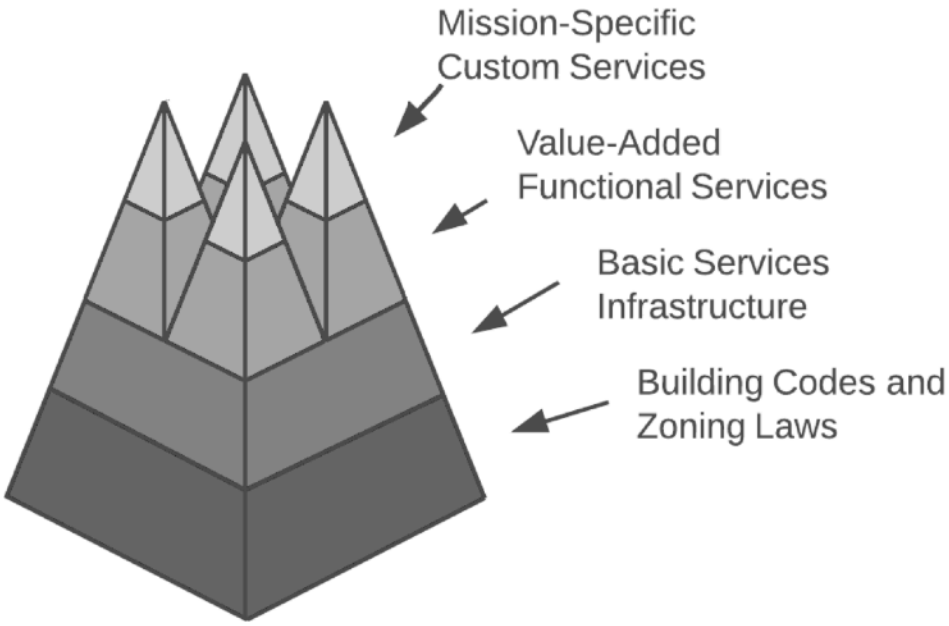
Stovepipe Enterprise AntiPatterni, özellikle çoğu şirket ticari sistemlerini genişletme gereksinimi ile karşı karşıya kaldığında, yeni sistemler ufak ölçekli şirketler için kabul edilemez. Ancak, şirketler devralma ya da birleşme ile büyüdüklerinde, Stovepipe AntiPattern'in ortaya çıkması muhtemeldir; Bu durumda, bazı sistemleri sarmalamak bir geçici çözüm olabilir.

Diğer bir istisna, ortak sistemler arasında kurumsal bir hizmet katmanının uygulanmasıdır. Bu genellikle satıcıya bağımlılık durumunda ortaya çıkar. Bu sistemler ortak bir yatay bileşene sahiptir; Örneğin, bankacılıkta bu genellikle DB2 ve Oracle gibi veritabanlarında geçerlidir.

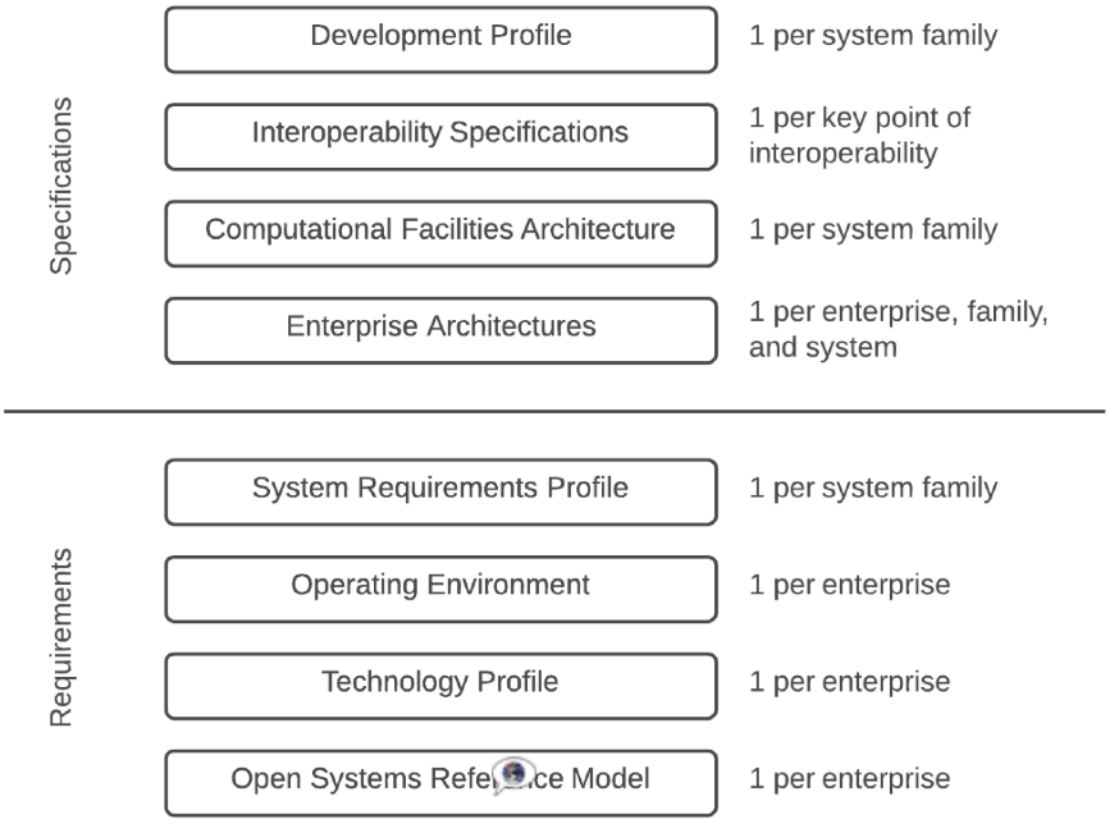
Elden Geçirilmiş (Refactored) Çözüm :

Bir Stovepipe Enterprise'ı önlemek için çeşitli düzeylerde teknolojilerin koordinasyonu şarttır. Başlangıçta, standartların seçimi bir standart referans modelinin tanımıyla koordine edilebilir.

Standart referans modeli, kurumsal sistemler için ortak standartları ve bir geçiş yönünü tanımlar. Ortak bir çalışma ortamının kurulması, ürün seçimini koordine eder ve ürün sürümlerinin yapılandırmasını kontrol eder. Ürünlerin ve standartların kullanımını koordine eden sistem profillerini tanımlamak, standartların faydalarını, yeniden kullanımını ve birlikte çalışabilirliğini sağlamak için şarttır. En az bir sistem profili, sistemler arasında kullanım kurallarını tanımlamalıdır.



Büyük deneyimler sayesinde, büyük şirketler, birçok organizasyona uygulanabilecek nesne yönelimli mimarilerin tanımı için bazı faydalı sözleşmeler hazırladılar. Büyük ölçekli mimarinin önemli bir zorluğu, teknoloji stratejisini ve gereksinimlerini ele alırken sistemler arasında ayrıntılı birlikte çalışabilirlik sözleşmelerini tanımlamaktır. Çok büyük işletmeler için, deneyimler birlikte çalışabilirlik zorluklarını doğru bir şekilde belirlemek ve çözmek için dört gereksinim modelinin ve dört şartname modelinin gerekli olduğunu göstermiştir.



Gereksinimleri modelleri şunlardır:
Açık Sistem Referans Modeli.
Teknoloji Profili.
Çalışma ortamı.
Sistem Gereksinimleri Profili.

Şartname modelleri şunlardır:
Kurumsal Mimariler.
Hesaplamalı Tesisler Mimarisi.
Birlikte Çalışabilirlik Özellikleri.
Geliştirme Profili.

ANALYSIS PARALYSIS ANTIPATTERN :

Antipattern Adı : Analysis - Paralysis (Organizational)
Verilen Diğer İsimler : Waterfall, Process Mismatch
En Sık Ölçek : System
Elden Geçirilmiş (Refactored) Çözüm Adı : Iterative - Incremental Development
Elden Geçirilmiş (Refactored) Çözüm Tipi : Software
Kök Nedenleri (Root Cause) : Pride, Narrow-Mindedness
Dengesiz Kuvvetler : Management of Complexity

Analiz Felci olarak bilinir. Bazı projelerin analiz safhası hem uzun sürer hem de kullanılan kaynakların(eleman sayısı gibi) maliyeti yüksek ve fazla olur. Çoğunlukla Waterfall adı verilen yazılım geliştirme metodolojisinde karşımıza çıkar. Analizin uzamasının belli nedenleri vardır. Bunlardan birisi geresiz olabilecek detaylara çok fazla girilmesidir.

Mükemmel bir analiz olmadan tasarım yapılamayacağı ve ilerlenemeyeceği varsayılır. Bu AntiPattern ayrıca günümüzün popüler **çevik*(Scrum gibi)*** süreçlerinin daha çok tercih edilmesinde de rol almış olabilir. Nitekim çevik süreçler periyodik iterasyonlar sonucunda işe yarar bir ürünün veya paketin müşteriye sunulmasına odaklanırlar.

Bir proje üzerinde yapılan analizlere gereğinden fazla zaman ve enerji ayrılmasıyla ortaya çıkar. Temel nedeni bilgi yetersizliği ve tecrübesizliktir. Bir karar vermeden ya da eyleme geçmeden önce aşırıya kaçan analizler, iş sürecinin duraksamasına neden olur.

Analysis - Paralysis Antipattern'inin, amacı, analiz aşamasının mükemmelliğini ve eksiksizliğini sağlamak olduğunda ortaya çıkar. Bu AntiPattern, devir, modellerin revizyonu ve aşağı akış süreçlerine yararlı olandan daha az ayrıntılı modellerin oluşturulması ile karakterize edilir.

Nesne yönelimli yöntemlere yeni birçok geliştirici, çok fazla ön analiz ve tasarım yapar. Bazen, analiz modellemesini problem alanında rahat hissetmek için bir egzersiz olarak kullanırlar. Nesne yönelimli yöntemlerin faydalarından biri, alan uzmanlarının katılımıyla analiz modelleri geliştirmektir. Aksi takdirde, hedef kapsamlı bir model oluşturmak olduğunda analiz sürecinde hataya düşmek kolaydır.

Analysis - Paralysis Antipattern'i genellikle waterfall varsayımlarını içerir:

- Detaylı analiz, kodlamadan önce başarıyla tamamlanabilir.
- Sorunla ilgili her şey önceden bilinebilir.
- Analiz modelleri geliştirme sırasında genişletilemez veya yeniden gözden geçirilmez. Nesne yönelimli gelişme, şelale analizi, tasarım ve uygulama süreçlerine yetersiz şekilde eşleştirilmiştir. Etkili nesne-yönelimli gelişme, artan ve yinelenen analiz sonuçlarının tasarım ve uygulama yoluyla doğrulandığı ve daha sonraki sistem analizlerine geri bildirim olarak kullanıldığı, artan bir süreçtir.

Belirti ve Sonuçlar :

- Personel değişiklikleri veya proje yönündeki değişiklikler nedeniyle birden fazla proje yeniden başlayabilir ve yeniden modelleme yapılmaktadır.
- Tasarım aşamasında, tasarım ve uygulama konuları sürekli olarak yeniden ele alınmaktadır.
- Analiz maliyeti öngörülebilir bir bitiş noktası olmaksızın beklentiyi aşmaktadır.
- Analiz aşaması artık kullanıcı etkileşimini içermemektedir. Yapılan analizlerin çoğu spekülatif kalmaktadır.
- Analiz modellerinin karmaşıklığı, karmaşık uygulamaları beraberinde getirmekte, sistemi geliştirmeyi, belgelemeyi ve test etmeyi zorlaştırmaktadır.
- Dört aşamalı tasarım desenlerinde kullanılanlar gibi tasarım ve uygulama kararları analiz aşamasında yapılmaktadır.

Tipik Nedenleri :

- Yönetim süreci aşamaları bir waterfall ilerlemesini varsayar. Gerçekte, neredeyse tüm sistemler, resmi süreçte kabul edilmese bile aşamalı olarak üretilir.
- Yönetim, problemi analiz etme ve çözme yeteneklerini tasarlama ve uygulamadan daha fazla güvenir.
- Yönetim, tasarım aşaması başlamadan önce tüm analizleri tamamlama konusunda ısrar eder.
- Analiz aşamasındaki hedefler iyi tanımlanmamıştır.
- İyi planlama veya liderlik Analiz aşamasının önüne geçer.
- Yönetim, alanın bölümlerinin yeterince açıklandığı zaman hakkında kesin kararlar vermek istememektedir.
- Proje vizyonu ve müşteriye ulaşma / hedefe odaklanma yaygındır. Analiz, anlamlı değer sağlamanın ötesine geçer.

Elden Geçirilmiş (Refactored) Çözüm :

Nesne yönelimli gelişimin başarısının anahtarı, aşamalı bir gelişmedir. Bir Waterfall süreci, problem hakkında önceden bilgi sahibi olurken, ek geliştirme süreçleri, problemin ve çözümün detaylarının gelişim süreci boyunca öğrenileceğini varsayar.

Artan geliştirmede, nesne yönelimli sürecin tüm aşamaları, hep bir yineleme (analiz, tasarım, kodlama, test ve doğrulama) ile gerçekleşir. İlk analiz, sistemin üst düzey bir incelemesini içerir, böylece sistemin hedefleri ve genel işlevselliği kullanıcılarla doğrulanabilir. Her bir artış, sistemin bir bölümünü tamamen detaylandırır.

İki çeşit artış vardır: iç ve dış. Dahili bir artış, uygulamanın altyapısı için gerekli olan yazılımı oluşturur. Örneğin, üçüncü düzey bir veritabanı ve veri erişim katmanı dahili bir artış içerecektir. Dahili artışlar, çoklu kullanım durumları tarafından kullanılan ortak bir altyapı oluşturur. Genel olarak, dahili artışlar yeniden çalışmayı en aza indirir. Harici bir artış, kullanıcı tarafından görülebilen işlevselliği içerir.

İlerleme göstererek proje için siyasi mutabakat kazanmak için dışsal artışlar şarttır. Kullanıcı doğrulama için harici artışlar da gereklidir. Altyapı ve arka uç katmanlarının eksik kısımlarını simüle etmek için genellikle bir kaç tane kod dizisi içerirler. Proje yöneticisinin ayrıcalığı, kullanıcı doğrulama ve minimum maliyet güçlerini dengeleyen artışları seçmek için. Riskle ilgili artışların zamanlamasını planlamaya ilişkin Proje Yanlışlığı AntiPattern'e bakın.

Sıklıkla, nesne yönelimli analiz yapılırken, analize devam etmekten ve yazılımın tasarımına geçmekten daha kolaydır. Analiz tekniklerinin birçoğu aynı zamanda tasarıma uygulandığı için, genel tasarımın özelliklerini yönlendirmek için analiz aşamasını kullanmak kolaydır.

Bazen, tasarımla eş zamanlı olarak en az analize ihtiyaç duyulduğunu ve tasarımın yeniden başlatılabileceğini düşünerek ortaya çıkar. Tipik olarak, bu, bir kullanıcının anlayabileceği bir etki alanı modeline benzemeyen bir ürüne veya uygulanan bir sistem için arzu edilen bir tasarıma neden olur.

Analysis - Paralysis Antipattern'i mimari düzeyde de uygulanabilir ve geliştirme düzeyindeki AntiPattern'e benzer bir biçim alır. Yapısal mimari, mimarların mimarlık ilkelerine ve yapıya bağlı kalmaları için gerekli olandan çok daha fazla belirtilebilir.