****

**T.C**

**KTO KARATAY ÜNİVERSİTESİ**

**MÜHENDİSLİK FAKÜLTESİ**

**ELEKTRİK-BİLGİSAYAR MÜHENDİSLİĞİ TEZLİ YÜKSEK LİSANS**

Design Pattern

&

Anti-Pattern

**Hazırlayan**

**Hilal MÜSEVİTOĞLU - 21743563**

**Ders Sorumlusu**

**Dr. Öğr. Üyesi Ali ÖZTÜRK**

**İÇİNDEKİLER**

1.Builder Pattern Kullanımında Motivasyon………………………………………………………………………..…2

2.Builder Pattern Nedir?............................................................................................................3

3.Builder Pattern Örneği……………………………………………………………………………………………..….…….4

4.Builder Pattern Kullanan Örnek Program…………………………………………………………………………...7

5.Walking through a Minefield Antipattern……………………………………………………………………..….10

5.1.Walking through a Minefield Antipattern’i ortaya çıkaran kök neden (root cause)…………………………………………………………………………………………………………………..….10

5.2.Walking through a Minefield Antipattern’e verilen diğer isimler……………………..…10

5.3.Walking through a Minefield Antipattern yazılım tasarım-seviyesi modelinde nereye karşılık gelir?.....................................................................................................10

5.4.Walking through a Minefield Antipattern hangi problem sınıfına bir çözüm önermektedir?..............................................................................................................10

5.5.Walking through a Minefield Antipattern hakkında detaylı bilgi….........................10

6.Swiss Army Knife Antipattern……………………………………………………………………………………………11

6.1.Swiss Army Knife Antipattern’i ortaya çıkaran kök neden (root cause)……………...11

6.2. Swiss Army Knife Antipattern’e verilen diğer isimler…………………………………………11

6.3.Swiss Army Knife Antipattern yazılım tasarım-seviyesi modelinde nereye karşılık gelir?.............................................................................................................................11

6.4.Swiss Army Knife Antipattern hangi problem sınıfına bir çözüm önermektedir?..............................................................................................................11

6.5.Swiss Army Knife Antipattern hakkında detaylı bilgi………………………………..…………11

7. Throw It Over The Wall Antipattern……………………………………………………………………………..…12

7.1.Throw It Over The Wall Antipattern’i ortaya çıkaran kök neden (root cause)…… 12

7.2. Throw It Over The Wall Antipattern’e verilen diğer isimler………………………………12

7.3. Throw It Over The Wall Antipattern yazılım tasarım-seviyesi modelinde nereye karşılık gelir?.................................................................................................................12

7.4. Throw It Over The Wall Antipattern hangi problem sınıfına bir çözüm önermektedir?..............................................................................................................12

7.5. Throw It Over The Wall Antipattern hakkında detaylı bilgi………………………………..12

7.5.1.Anti Pattern Problem ve İlgili Çözümler...................................................12

1

**1.Builder Pattern Kullanımında Motivasyon**

Karmaşık olan bir uygulamadan daha karmaşığı, kullanılan sınıfların ve nesnelerin karmaşıklığıdır. Karmaşık nesneler, inşa edilirken özel dikkat gerektiren diğer nesneler tarafından üretilen parçalardan yapılır. Bir uygulama, nesneyi oluşturanlardan bağımsız olan karmaşık nesneler oluşturmak için bir mekanizmaya ihtiyaç duyabilir. Karşılaştığınız sorun bu ise, Builder (veya Adaptive Builder) tasarım desenini kullanmayı deneyebilirsiniz.

Bu kalıp, bir istemci nesnesinin, yalnızca onun türünü ve içeriğini belirterek, nesnenin temsiliyle ilgili ayrıntılardan korunan karmaşık bir nesne oluşturmasına izin verir. Bu şekilde inşaat süreci farklı temsiller oluşturmak için kullanılabilir. Bu işlemin mantığı, karmaşık nesnenin oluşturulmasında kullanılan asıl adımlar biçiminde izole edilir, böylece süreç, birincisi ile aynı basit nesne setinden farklı bir nesne formu oluşturmak için tekrar kullanılabilir.

2

**2.Builder Pattern Nedir?**

Builder (İnşaatçı) tasarım deseni creational grubuna ait, biden fazla parçadan oluşan nesnelerin üretilmesinden sorumlu bir tasarım desenidir. dofactory.com a göre kullanım oranı 40% larda olan builder tasarım deseni yapı olarak abstract factory desenine benzer.

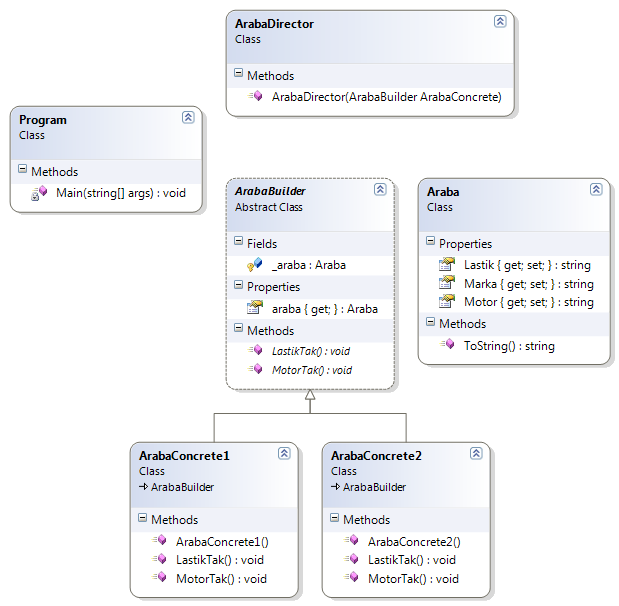
Bazı nesneler birden fazla nesnenin birleşmesinden(bazı işlemleri yapması sonucu) oluşabilir. Zamanla bu ana nesneyi oluşturan nesnelerin yapısı değişebilir, bu nesnelerin oluşturulması karışık bir hal alabilir veya bu nesnelere başka nesneler de eklenebilir. Builder tasarım deseni bu gibi durumlarda genişletilebilirliği sağlamak ve kod karmaşıklığını engellemek için kullanılır. Builder tasarım deseninde bu nesnelerin oluşturulması Builder denilen sınıfların sorumluluğundadır. Client sadece oluşturulacak nesne türünü belirterek ana nesneyi oluşturan nesnelerin oluşturulmasıyla ilgilenmez. Abstract factory tasarım kalıbı ile benzer bir yapısı vardır. Aralarındaki fark builder tasarım deseni birden fazla nesnenin birleşmesinden oluşan nesnelerin üretilmesinden sorumludur.

* Builder desenini oluşturan 4 yapı vardır.
* Product: Oluşturulan nesne.
* Builder: Product oluşturacak nesnelerin (Concrete Builder) uygulaması gereken arayüz.
* Concrete Builder: Product nesnesini oluşturan nesne veya özelliklerin oluşturulduğu sınıflar. Her concrete builder sınıfı aynı arayüzde farklı bir ürünün oluşturulmasını sağlar.
* Director: Verilen builder nesnesine göre product örneği oluşturur.

3

**3.Builder Pattern Örneği**

Örneğin Marka, Model ve Lastik özellikleri olan bir araba nesnemiz olsun. Bu araba nesnemizin özelliklerinin farklı değerler alması ile farklı özelliklerde araba nesnesi üretebiliriz. Builder tasarım deseni ile bu senaryoyu gerçekleştirelim. Uygulamamızın class diyagramı aşağıdadır.



4

public class Araba

    {

        public string Marka { get; set; }

        public string Motor { get; set; }

        public string Lastik { get; set; }

        public override string ToString()

        {

            return String.Format("Marka:{0},Motor:{1},Lastik:{2}", Marka, Motor, Lastik);

        }

    }

    // ArabaBuilder arayüzü builder desenindeki Builder yapısıdır.

    // Bir sınıfın Product ı oluşturan nesneleri oluşturması veya product ın özelliklerini setlemesi ile product ı oluşturan sınıflar bu arayüzü kullanmalıdır.

    public abstract class ArabaBuilder

    {

        protected Araba \_araba;

        public Araba araba

        {

            get { return \_araba; }

        }

        public abstract void MotorTak();

        public abstract void LastikTak();

    }

    // ArabaBuilder arayüzünden türeyen bütün sınıflar Builder desenindeki ConcreteBuilder yapısıdır.

    // ConcreteBuilder yapısı değişik product nesnelerinin oluşturulmasını sağlamaktır.

    public class ArabaConcrete1 : ArabaBuilder

    {

        public ArabaConcrete1()

        {

            \_araba = new Araba { Marka = "Concrete1" };

        }

        public override void MotorTak()

        {

            \_araba.Motor = "1.4 LPG";

        }

        public override void LastikTak()

        {

            \_araba.Lastik = "15' Çelik jant";

        }

    }

    public class ArabaConcrete2 : ArabaBuilder

    {

        public ArabaConcrete2()

        {

            \_araba = new Araba { Marka = "Concrete2" };

        }

        public override void MotorTak()

        {

            \_araba.Motor = "1.8 Dizel";

        }

        public override void LastikTak()

        {

            \_araba.Lastik = "17' Bor alaşımlı çelik jant";

        }

    }

5

    // ArabaBuilder arayüzündeki metodları çalıştırarak productın oluşturulmasını sağlar.

    // Builder desenindeki Director yapısıdır.

    public class ArabaDirector

    {

        public ArabaDirector(ArabaBuilder ArabaConcrete)

        {

            ArabaConcrete.MotorTak();

            ArabaConcrete.LastikTak();

        }

    }

Desenimizi aşağıdaki şekilde kullanabiliriz.

class Program

{

    static void Main(string[] args)

    {

        ArabaBuilder araba\_builder = new ArabaConcrete1();

        ArabaDirector araba\_olusutucu = new ArabaDirector(araba\_builder);

        Console.WriteLine(araba\_builder.araba.ToString());

        araba\_builder = new ArabaConcrete2();

        araba\_olusutucu = new ArabaDirector(araba\_builder);

        Console.WriteLine(araba\_builder.araba.ToString());

        Console.ReadKey();

    }

}

6

**4.Builder Pattern Kullanan Örnek Program**

|  |
| --- |
| #include <iostream> |
|  | #include <string> |
|  |  |
|  | /\* Car parts \*/ |
|  | class Wheel |
|  | { |
|  | public: |
|  | int size; |
|  | }; |
|  |  |
|  | class Engine |
|  | { |
|  | public: |
|  | int horsepower; |
|  | }; |
|  |  |
|  | class Body |
|  | { |
|  | public: |
|  | std::string shape; |
|  | }; |
|  |  |
|  | /\* Final product -- a car \*/ |
|  | class Car |
|  | { |
|  | public: |
|  | Wheel\* wheels[4]; |
|  | Engine\* engine; |
|  | Body\* body; |
|  |  |
|  | void specifications() |
|  | { |
|  | std::cout << "body:" << body->shape << std::endl; |
|  | std::cout << "engine horsepower:" << engine->horsepower << std::endl; |
|  | std::cout << "tire size:" << wheels[0]->size << "'" << std::endl; |
|  | } |
|  | }; |
|  |  |
|  | /\* Builder is responsible for constructing the smaller parts \*/ |
|  | class Builder |
|  | { |
|  | public: |
|  | virtual Wheel\* getWheel() = 0; |
|  | virtual Engine\* getEngine() = 0; |
|  | virtual Body\* getBody() = 0; |
|  | };  7 |
|  |  |
|  | /\* Director is responsible for the whole process \*/ |
|  | class Director |
|  | { |
|  | Builder\* builder; |
|  |  |
|  | public: |
|  | void setBuilder(Builder\* newBuilder) |
|  | { |
|  | builder = newBuilder; |
|  | } |
|  |  |
|  | Car\* getCar() |
|  | { |
|  | Car\* car = new Car(); |
|  |  |
|  | car->body = builder->getBody(); |
|  |  |
|  | car->engine = builder->getEngine(); |
|  |  |
|  | car->wheels[0] = builder->getWheel(); |
|  | car->wheels[1] = builder->getWheel(); |
|  | car->wheels[2] = builder->getWheel(); |
|  | car->wheels[3] = builder->getWheel(); |
|  |  |
|  | return car; |
|  | } |
|  | }; |
|  |  |
|  | /\* Concrete Builder for Jeep SUV cars \*/ |
|  | class JeepBuilder : public Builder |
|  | { |
|  | public: |
|  | Wheel\* getWheel() |
|  | { |
|  | Wheel\* wheel = new Wheel(); |
|  | wheel->size = 22; |
|  | return wheel; |
|  | } |
|  |  |
|  | Engine\* getEngine() |
|  | { |
|  | Engine\* engine = new Engine(); |
|  | engine->horsepower = 400; |
|  | return engine; |
|  | } |
|  |  |
|  | Body\* getBody() |
|  | { |
|  | Body\* body = new Body(); |
|  | body->shape = "SUV"; |
|  | } |
|  | }; |
|  | 8 |
|  | class NissanBuilder : public Builder |
|  | { |
|  | public: |
|  | Wheel\* getWheel() |
|  | { |
|  | Wheel\* wheel = new Wheel(); |
|  | wheel->size = 16; |
|  | return wheel; |
|  | } |
|  |  |
|  | Engine\* getEngine() |
|  | { |
|  | Engine\* engine = new Engine(); |
|  | engine->horsepower = 85; |
|  | return engine; |
|  | } |
|  |  |
|  | Body\* getBody() |
|  | { |
|  | Body\* body = new Body(); |
|  | body->shape = "hatchback"; |
|  | } |
|  | }; |
|  |  |
|  |  |
|  | int main() |
|  | { |
|  | Car\* car; // Final product |
|  |  |
|  | /\* A director who controls the process \*/ |
|  | Director director; |
|  |  |
|  | /\* Concrete builders \*/ |
|  | JeepBuilder jeepBuilder; |
|  | NissanBuilder nissanBuilder; |
|  |  |
|  | /\* Build a Jeep \*/ |
|  | std::cout << "Jeep" << std::endl; |
|  | director.setBuilder(&jeepBuilder); // using JeepBuilder instance |
|  | car = director.getCar(); |
|  | car->specifications(); |
|  |  |
|  | std::cout << std::endl; |
|  |  |
|  | /\* Build a Nissan \*/ |
|  | std::cout << "Nissan" << std::endl; |
|  | director.setBuilder(&nissanBuilder); // using NissanBuilder instance |
|  | car = director.getCar(); |
|  | car->specifications(); |
|  |  |
|  | return 0; |
|  | } |

[**http://wiki.c2.com/?AntiPatternsCatalog**](http://wiki.c2.com/?AntiPatternsCatalog)

9

**Antipatterns**

**5.Walking through a Minefield Antipattern**

**5.1.Walking through a Minefield Antipattern’i ortaya çıkaran kök neden (root cause)**

Ignorance, Sloth

**5.2.Walking through a Minefield Antipattern’e verilen diğer isimler**

* Nothing Works
* Do You Believe in Magic?

**5.3.Walking through a Minefield Antipattern yazılım tasarım-seviyesi modelinde nereye karşılık gelir?**

Test aşamasına karşılık gelir.

**5.4.Walking through a Minefield Antipattern hangi problem sınıfına bir çözüm önermektedir?**

Software development antipattern

**5.5.Walking through a Minefield Antipattern hakkında detaylı bilgi**

Günümüzün yazılım teknolojisini kullanmak, yüksek teknolojili bir maden sahasında yürümeye benzer. Yayımlanan yazılım ürünlerinde çok sayıda hata oluşur. Aslında uzmanlar, orijinal kaynak kodunun kod satırı başına iki ila beş hata içerdiğini tahmin eder. Bu, kodun tüm hataları gidermek için satır başına iki veya daha fazla değişiklik gerektireceği anlamına gelir. Soru sormadan, operasyonel sistemleri desteklemeye hazır olmadan önce birçok ürün piyasaya sürülüyor.

Yazılım kusurlarının yeri ve sonuçları, görünür nedenleriyle ilgisizdir ve küçük bir hata bile felaket olabilir. Örneğin işletim sistemleri, (UNIX, Windows, vb.) saldırılara karşı savunmasız hale getiren birçok bilinen ve bilinmeyen güvenlik kusuru içerir. Son kullanıcılar sık ​​sık yazılım hatalarıyla karşılaşır.

Ticari yazılım testinin amacı; riskleri, özellikle de destek maliyetlerini sınırlamaktır.

Geçmişin daha basit sistemleriyle şanslıydık. Günümüzün sistemleri, bilgisayar kontrollü yolcu trenleri ve uzay kontrol sistemleri dahil olmak üzere, küçük bir hatanın sonucu bile felaket olabilir. Zaten finansal kayıpların 100 milyon doları aştığı yarım düzine büyük yazılım hatası vardır.

<https://sourcemaking.com/antipatterns/walking-through-minefield>

10

**6. Swiss Army Knife Antipattern**

**6.1.Swiss Army Knife Antipattern’i ortaya çıkaran kök neden (root cause)**

Pride

**6.2.Swiss Army Knife Antipattern’e verilen diğer isimler**

Kitchen Sink

**6.3.Swiss Army Knife Antipattern yazılım tasarım-seviyesi modelinde nereye karşılık gelir?**

Tasarım(Design) aşamasına karşılık gelir.

**6.4.Swiss Army Knife Antipattern hangi problem sınıfına bir çözüm önermektedir?**

Software Architecture antipattern

**6.5.Swiss Army Knife Antipattern hakkında detaylı bilgi**

Kitchen Sink olarak da bilinen Swiss Army Knife, aşırı karmaşık bir ara yüzdür. Tasarımcı, sınıfın tüm olası kullanımlarını sağlamaya çalışır. Swiss Army Knife’ın gerçek dünya örnekleri, tek bir sınıf için düzinelerce, binlerce metot imzasını içerir. Swiss Army Knife, satıcıların ürünlerini mümkün olan tüm uygulamalara uygun hale getirmeye çalıştıkları ticari yazılım ara yüzlerinde yaygındır.

****

**Şekil 6.1 Swiss Army Knife Antipattern Antipattern**

11

**7. Throw It Over The Wall** **Antipattern**

**7.1.Throw It Over The Wall** **Antipattern’i ortaya çıkaran kök neden (root cause)**

Responsibility (the universal cause)

**7.2.Throw It Over The Wall Antipattern’e verilen diğer isimler**

The Code is finished (no testing, no documentation)

**7.3.Throw It Over The Wall Antipattern yazılım tasarım-seviyesi modelinde nereye karşılık gelir?**

Dokümantasyon, eğitim aşamalarına karşılık gelir.

**7.4.Throw It Over The Wall Antipattern hangi problem sınıfına bir çözüm önermektedir?**

Software Project Management antipattern

**7.5.Throw It Over The Wall Antipattern hakkında detaylı bilgi**

Nadiren, belgeler kendini açıklayıcı niteliktedir, ancak yazarların vizyonunu ve kavrayışını anlamak belgelerin anlaşılmasının önemli bir parçasıdır. Bu, özellikle bağımsız karar verme konusunda örtük bir varsayımın olduğu kılavuz belgeler için geçerlidir. Doğası gereği, tüm insan bilgisi kişiseldir. En seçkin bilim insanı spekülasyonları bile, keşiflerini yeni bilgi edinmeye yönlendiren kişisel bir bakış açısıdır.

**7.5.1.Anti Pattern Problem ve İlgili Çözümler**

Nesne yönelimli yöntemler, tasarım örüntüleri ve esnek kılavuzlar olarak tasarlanan uygulama planları, genellikle alt yöneticiler ve OO geliştiricileri tarafından tam anlamıyla alınır. Esnek yönergelerin bu türden yorumlanması, istenmeyen sonuçlara yol açabilir. Kararlar, sadece dikkatli analizlere ve yerel olarak optimize edilmiş karar verme mekanizmalarına ilham vermeyi amaçlayan kılavuzlara dayanarak yapılabilir. Örneğin, işe yaramaz analizler ve dokümantasyon üzerinde çaba harcanabilir çünkü geliştirme ekibinde hiç kimse amacını anlayamasa da, zorunludur. Bu fenomen hem büyük hem de küçük organizasyonlarda gerçekleşir ve gelişim aşamaları arasındaki iletişimsizlikten kaynaklanabilir. Diğer bir önemli neden ise, sistemin son kullanıcılarının ihtiyaçları yerine yönetimin belirgin beklentilerini karşılama arzusudur.

12

Teknik dokümantasyonun amaçlanan şekilde yorumlanması ve uygulanması için, materyallerin birkaç önemli teknikle iletilmesi gerekir. Biri bilgiyi bir öğretici aracılığıyla sunmaktır. Yeni politikalar ve kurallar her kurulduğunda, bilgiyi duyurmak ve motivasyonları iletmek için karşılık gelen bir bilgi aktarımı olmalıdır. Bir günlük eğitim oturumu genellikle 100 sayfaya kadar geliştirme özellikleri için yeterlidir. Bununla birlikte, bu oturumu iki bölüm halinde yürütmek yararlı olabilir: bir yönetim tanıtımı ve geliştiriciler için teknik bir brifing, çünkü bu iki farklı kitle büyük ölçüde farklı ihtiyaçlara sahiptir. Bu grupları birleştirirseniz, yönetim tartışması genellikle teknik ayrıntıların karşılanması için gereken süreye uzanır. Telefon ve elektronik posta kontakları da dahil olmak üzere takip desteğini almak, başarılı teknoloji transferini sağlamak için faydalıdır.

13