# Singleton

**Amaç:** Bir nesne örneğinden sadece bir defa üretilip her zaman kullanılmasını amaçlar. Singleton olarak oluşturulan nesne static olduğu için bellekte her zaman yer kaplamaktadır. Bu nedenle sürekli olarak tüketilmesi gereken ve birçok kişi tarafından tüketilmesi gereken durumlarda singleton kullanılmalıdır. Nesne içeriği sürekli değişmeyen durumlarda singleton kullanılmalıdır çünkü static olduğundan dolayı güncellenmesi gereken durumlarda IIS restart edilmesi gerekecektir. Multithread işlemlerde farklı kullanıcılar aynı anda nesne create etmek isterse nesne birden fazla kez oluşmuş olacaktır. Bu nedenle nesne create edilmesi süreci lock edilmelidir.

**Kullanımı:** var object = Object.CreateAsSingleton()

# Factory

**Amaç:** Yazılımda değişimi kontrol altına almak için kullanılmaktadır. Veri erişim katmanında kullanılan ORM den, catch ve loglama sistemlerine, hatta iş (business) sistemlerine kadar değişkenlik gösterebilecek durumları kontrol altına alabilmemizi sağlayacak bir yapı sunmaktadır.

**Kullanımı:** var object = ObjectFactory.Create()

**Client Kullanımı:** var customerManager = new CustomerManager(new LoggerFactoryLog4Net());

# Abstract Factory

**Amaç:** Factory design patern’e ek olarak, toplu nesne kullanımı ihtiyaçlarında yönetilebilir bir yapı sunmak amaçlanmaktadır. Örneğin, Loglama veya cache yapıları farklı şekillerde yapılabilmektedir. Loglama yaparken veritabanına, dosyaya veya consola loglama yapabiliriz. Aynı şekilde cache için de MemoryCache, MemCache, RedisCache gibi farklı yöntemler kullanabiliriz. İş ihtiyaçları için de farklı yöntemler izleyebiliriz. Örneğin evrak takip sisteminde kimi kullanıcılar için mail atabilirken kimileri için sms gönderebiliriz, yada sadece uyarı çıkarabiliriz. Bu gibi çoklu nesnelerin oluşturulmasının yönetiminde abstract factory patern kullanabiliriz.

**Kullanımı:**

public abstract class CrossCuttingConcernFactory

    {

        public abstract Logging CreateLogger();

        public abstract Caching CreateCaching();

    }

**Client Kullanımı:**

var productManager = new ProductManager(new Factory2());

# Prototype

**Amaç:** Nesne üretim maliyetlerini minimize etmek için kullanılır. Nesne clone yapmak için kullanılır. Özellikle çok property sahip bir nesnenin sadece birkaç property içeriğini değiştirerek yeni bir nesne oluşturmak istediğimizde bize büyük bir avantaj sağlar.

**Kullanımı:**

public override Person Clone()

        {

            return (Person)MemberwiseClone();

        }

**Client Kullanımı:**

var customer2 = (Customer)customer.Clone();

# Builder

**Amaç:** Bir nesne örneği çıkarmak için kullanılır. Nesne örneği birbiri ardına atılacak olan adımların sonrasında ortaya çıkmaktadır. İş katmanında if blokları ile yönetmek yerine Builder deseni kullanılabilir. Gerçek yaşam örneği olarak hamburger üretim süreçlerine benzetim yapılabilir. Hamburger bir nesne ise vejeteryan için ayrı, vejeteryan olmayanlar için ayrı bir süreç sonucu hamberger nesnesi üretilebilmektedir. Özellikle front-end development da kullanılan bir design pattern dir. Komplex nesnelerin oluşturulmasında büyük rol oynar.

**Kullanımı:**

abstract class ProductBuilder

    {

        public abstract void GetProductData();

        public abstract void ApplyDiscount();

        public abstract ProductViewModel GetViewModel();

    }

class ProductDirector

{

    private ProductBuilder productBuilder;

    public ProductDirector(ProductBuilder productBuilder)

    {

        this.productBuilder = productBuilder;

    }

    public void GenerateProduct()

    {

        productBuilder.GetProductData();

        productBuilder.ApplyDiscount();

    }

    public ProductViewModel GetProduct()

    {

        return productBuilder.GetViewModel();

    }

}

**Client Kullanımı:**

var productBuilder = new OldCustomerProductBuilder();

var productDirector = new ProductDirector(productBuilder);

var product = productDirector.GetProduct();

# Facade

**Amaç:** Ortak amaçlar için kullanılan birçok sınıf olabilir. Sınıfların yönetimini kolaylaştırmak için tüm ortaklaşan sınıfları tek bir cephe de toplayıp oradan kullanılmasını sağlayan bir yapı sunmaktadır. Özellikle Uygulamayı dikine kesen CrossCuttingConcern (Logging, Caching, Authorization) durumlarında kullanılmasının yönetimsel açıdan avantajlarını sunmaktadır.

**Kullanımı:**

class CrossCuttingConcernFacade

    {

        public ILogging Logging;

        public ICaching Caching;

        public IAuthorize Authorize;

        public CrossCuttingConcernFacade()

        {

            Logging = new Logging();

            Caching = new Caching();

            Authorize = new Authorize();

        }

    }

**Client Kullanımı:**

CrossCuttingConcernFacade concerns;

       public CustomerManager()

       {

           concerns = new CrossCuttingConcernFacade();

       }

       public void Save()

       {

           concerns.Logging.Log();

           concerns.Caching.Cache();

           concerns.Authorize.Authorize();

           Console.WriteLine("Saved");

       }

# Adapter

**Amaç:** Farklı sistemleri kendi sistemimize entegre ederken, kendi sistemimiz bozulmadan farklı sistemlerin kullanımının sağlanmasını amaçlamaktadır. Nesnel programlama ve test edilebilirliğin sağlanmasında önemli rol oynamaktadır.

**Kullanımı:**

class Log4netAdapter : ILogger

{

    public void Log(string message)

    {

        new Log4Net().Log(message,"");

    }

}

**Client Kullanımı:**

var productManager = new ProductManager(new Log4netAdapter());

# Composite

**Amaç:** Hiyerarşik nesnelerin oluşturulması için kullanılır. Örneğin bir kurumdaki roller ve o rollerin hiyerarşik yapısının (Organizasyon Şeması) kurgulanması için kullanılabilir.

**Kullanımı:**

class Employee : IPerson, IEnumerable<IPerson>

    {

        List<IPerson> subordinates = new List<IPerson>();

public void AddSubordinates(IPerson person)

        {

            subordinates.Add(person);

        }

        public void RemoveSubordinates(IPerson person)

        {

            subordinates.Remove(person);

        }

        public IPerson GetSubordinate(int index)

        {

            return subordinates[index];

        }

}

**Client Kullanımı:**

manager.AddSubordinates(employee);

# Proxy

**Amaç:** Cache’leme sistemine benzetebiliriz. İlk kez çağrıldığında üzerine düşen işlemi yapar ama ikinci kez tekrar çağrılırsa ilk çağrılanı kullanmayı amaçlar. Bir cache mekanizması olarak tanımlanabilir.

**Kullanımı:**

abstract class CustomerBase

    {

        public abstract int Calculate();

    }

    class CustomerManager : CustomerBase

    {

        public override int Calculate()

        {

            int result = 1;

            for (int i = 1; i <= 5; i++)

            {

                result \*= i;

                Thread.Sleep(1000);

            }

            return result;

        }

    }

    class CustomerManagerProxy : CustomerBase

    {

        private CustomerBase \_customerManager;

        private int \_result;

        public override int Calculate()

        {

            if(\_customerManager == null)

            {

                \_customerManager = new CustomerManager();

                \_result = \_customerManager.Calculate();

            }

            return \_result;

        }

    }

**Client Kullanımı:**

CustomerBase mng = new CustomerManagerProxy();

Console.WriteLine(mng.Calculate());

# Decorator

**Amaç:** Runtime da bir nesneye yeni özellikler eklemek istenirse kullanılır. Farklı zamanlarda farklı kullanıcılara farklı şekillerde ürünü sunmak istediğimizde decorator design patern kullanırız.

**Kullanımı:**

class CarDecorator : CarBase

    {

        private CarBase carBase;

        public CarDecorator(CarBase carBase)

        {

            this.carBase = carBase;

        }

        public override decimal HirePrice { get; set; }

    }

    class SpecialOffer : CarDecorator

    {

        public int DiscountPercentage { get; set; }

        private CarBase carBase;

        public SpecialOffer(CarBase carBase) : base(carBase)

        {

            this.carBase = carBase;

        }

        public override decimal HirePrice { get { return carBase.HirePrice - (carBase.HirePrice \* DiscountPercentage / 100);  } set { } }

    }

**Client Kullanımı:**

CarBase personelCar = new PersonalCar() { Make = "Citroen", Model = "C3", HirePrice = 100 };

            SpecialOffer specialOffer = new SpecialOffer(personelCar);

            specialOffer.DiscountPercentage = 10;

            Console.WriteLine("Special Offer: Make: {0}, Model:{1}, HirePrice: {2}", specialOffer.Make, specialOffer.Model, specialOffer.HirePrice);

# Bridge

**Amaç:** Bir nesnenin içerisinde soyutlanabilir nesneler varsa onları yönetmeye yarar.

**Kullanımı:**

public class MailSender : MessageSenderBase

    {

        public override void Send(Body body)

        {

            Console.WriteLine("Message send by MailSender " + body.Title);

        }

    }

    public class SmsSender : MessageSenderBase

    {

        public override void Send(Body body)

        {

            Console.WriteLine("Message send by SmsSender " + body.Title);

        }

    }

**Client Kullanımı:**

public class CustomerManager

   {

       public MessageSenderBase MessageSenderBase { get; set; }

       public void Update()

       {

           MessageSenderBase.Send(new Body { Title = "Title" });

           Console.WriteLine("Customer updated");

       }

   }