SELÇUK ÜNİVERSİTESİ
TEKNOLOJİ FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

TASARIM DESENLERİ

[DESIGN PATTERNS]

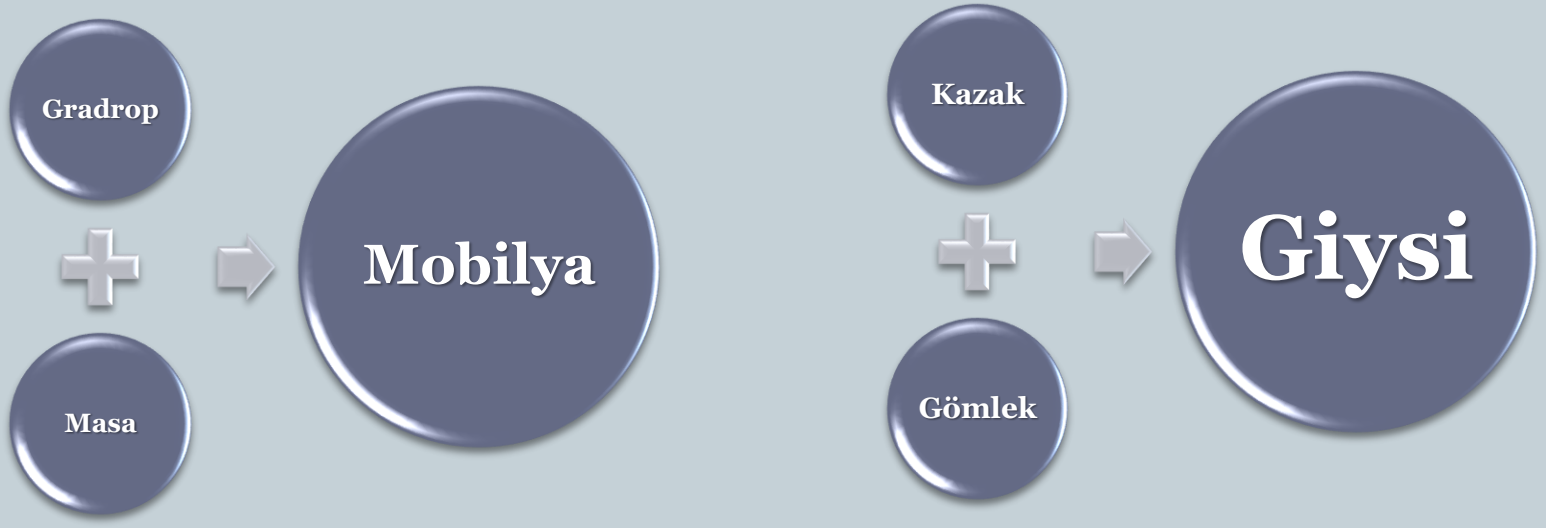
(ABSTRACT FACTORY – BUILDER – OBJECT POOL)

YRD.DOÇ.DR. TAHİR SAĞ

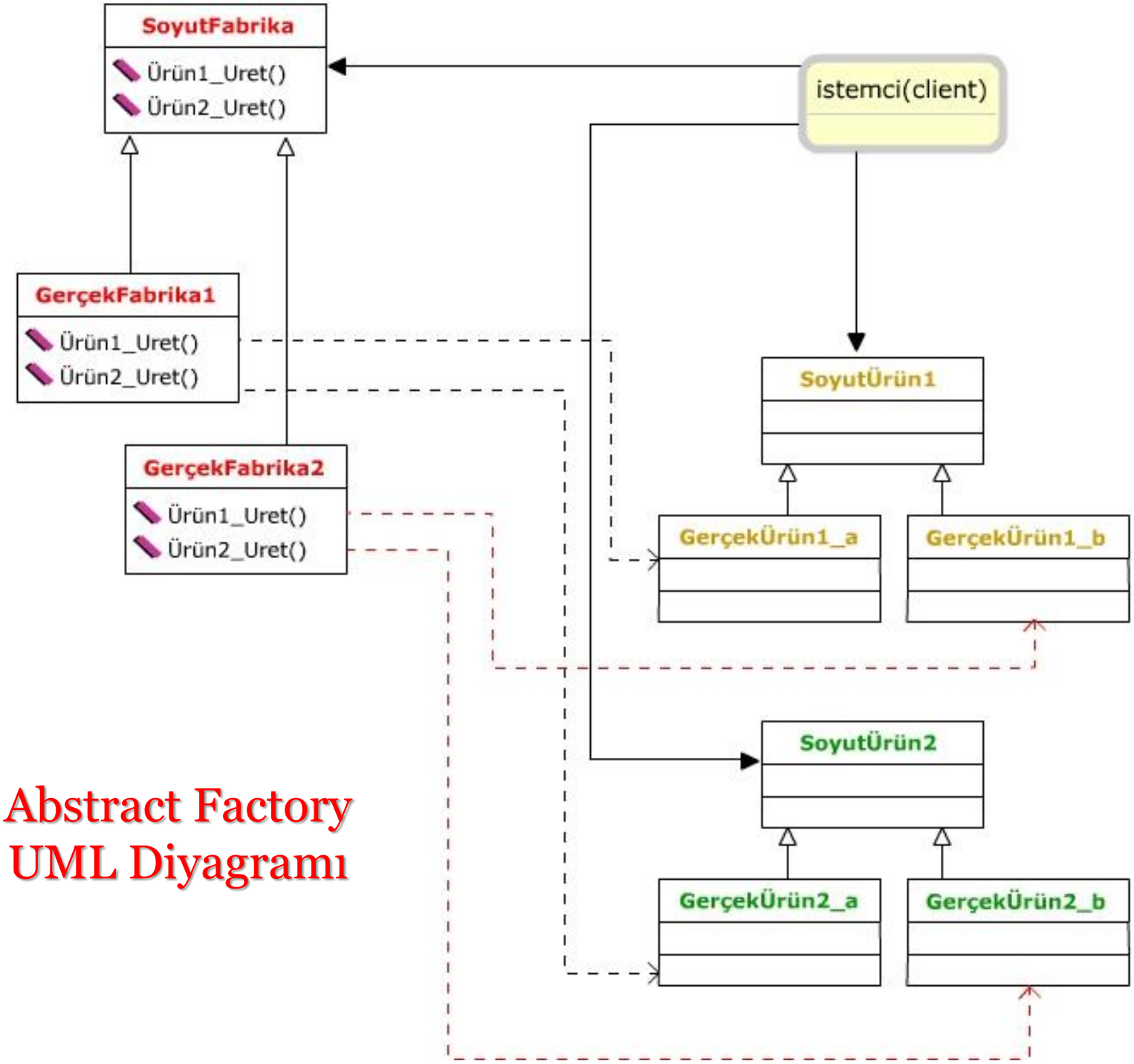
KONYA, 2017

Abstract Factory Tasarım Deseni

- Gerçek hayatta varlıkların çoğu zaman ilişkili olduğu gruplar veya aileler vardır.



- Aynı aileyi üreten çeşitli fabrikalar vardır.
- Kullanıcılar, bu varlıkların üretiminden fabrikalar sayesinde habersizdir; soyutlanmışlardır.



Abstract Factory UML Diyagramı

Abstract Factory Tasarım Deseni

Ne zaman kullanılır?

1. Birden fazla ürün ailesi ile çalışmak durumunda kaldığımızda, kullanılacak ürün ailesi ile istemci tarafı soyutlamak için...

2. Fabrika Deseni kullanıldığında oluşturucu sınıf içinde if-else blokları yazılır ve hangi ürün nesnesinin oluşturulacağına karar verilir. Kontrol bloklarını yazmadan nesneleri oluşturmak için...

Abstract Factory: Temel Özellikleri

- «Abstract Factory», nesneleri oluşturan bir sınıftır. Oluşturulan bu nesneler birbirleriyle ilişkili olan nesnelerdir. Diğer bir deyişle aynı arayüzü uygulamış olan nesnelerdir.
- Üretilen nesnelerin kendisiyle ilgilenilmez. İlgilenilen nokta oluşturulacak nesnelerin sağladığı arayüzlerdir. Dolayısıyla aynı arayüzü uygulayan yeni nesneleri desene eklemek çok kolay ve esnektir.
- Bu desende üretilecek nesnelerin birbirleriyle ilişkili olması beklenir.

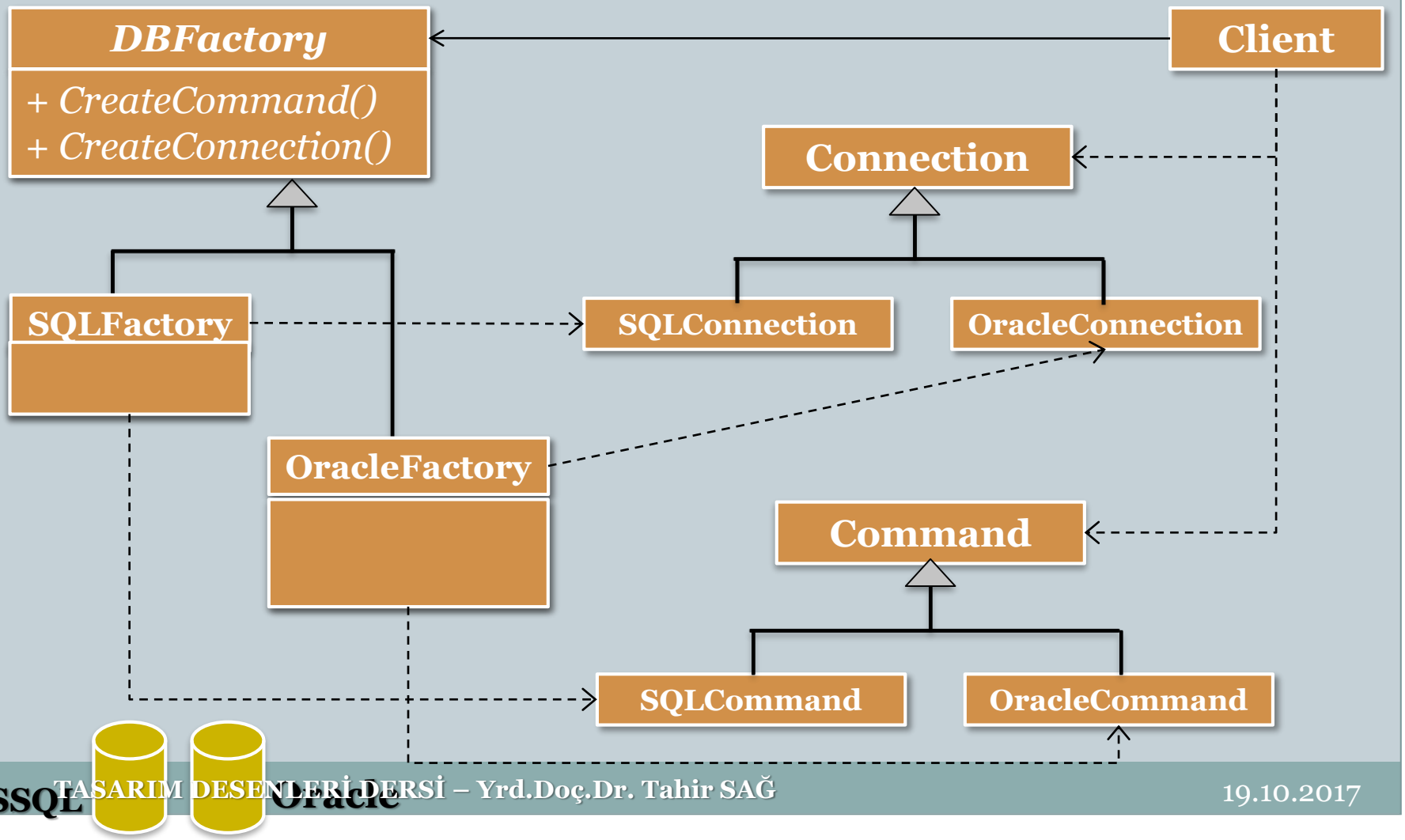
Abstract Factory: Özet

İstemcinin farklı nesne ailelerinden nesneler ile çalışması istenildiğinde bu tasarım deseni kullanılır.

Ancak önemli olan nokta;

istemcinin somut fabrikaları ve bunların ilgili nesneleri nasıl ürettiğinin bilmesini istenmiyorsa bu desenin kullanılmasıdır.

Abstract Factory: Örnek



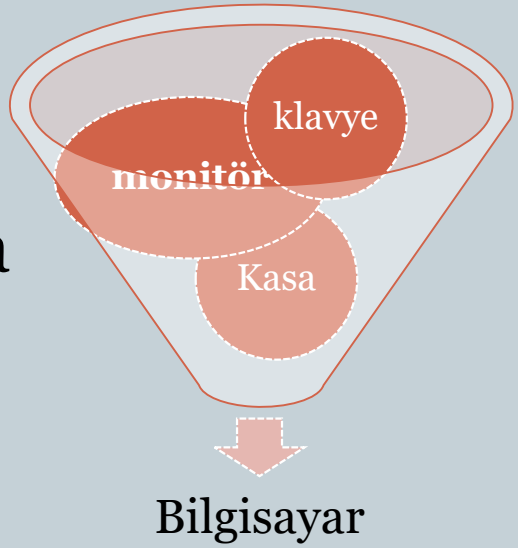
Abstract Factory: Örnek


```
public class FactoryUtil
{
    public static DBFactory GetFactory(string db_name)
    {
        if (db_name == "SQL")
        {
            return new SQLFactory();
        }

        if (db_name == "Oracle")
        {
            return new OracleFactory();
        }

        throw new Exception("error");
    }
}

// ABSTRACT FACTORY
public abstract class DBFactory
{
    public abstract Connection CreateConnection();
    public abstract Command CreateCommand();
}
```

```
// Concreate Factory'ler
public class SQLFactory : DBFactory
{
    public override Command CreateCommand()
    {
        return new SQLCommand();
    }

    public override Connection CreateConnection()
    {
        return new SQLConnection();
    }
}

public class OracleFactory : DBFactory
{
    public override Command CreateCommand()
    {
        return new OracleCommand();
    }

    public override Connection CreateConnection()
    {
        return new OracleConnection();
    }
}
```

Builder Tasarım Deseni

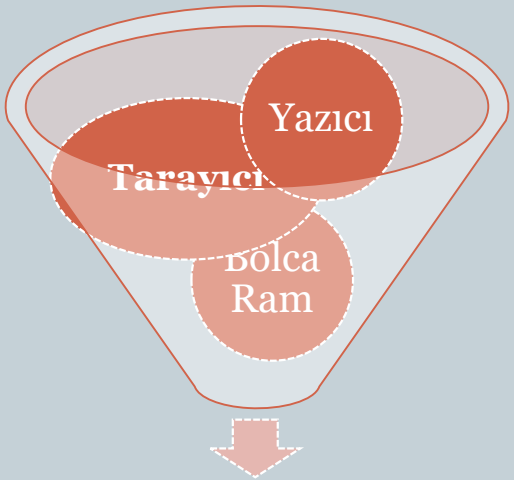
- Günlük hayatta bazı varlıklar pek çok alt parçanın birleşmesiyle oluşur. Örneğin;



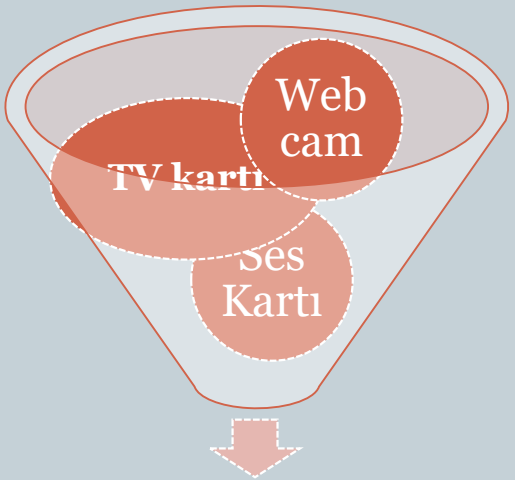
- Bu desen adını komposit nesneleri, alt parçalarını bir araya getirerek oluşturmaya bir çözüm getirmesinden almaktadır.

- Bu desen ile aynı kompleks nesnenin farklı parçalarla oluşturulup farklı durumlarda elde edilebilmesi sağlanır.

Builder Tasarım Deseni



1.Bilgisayar

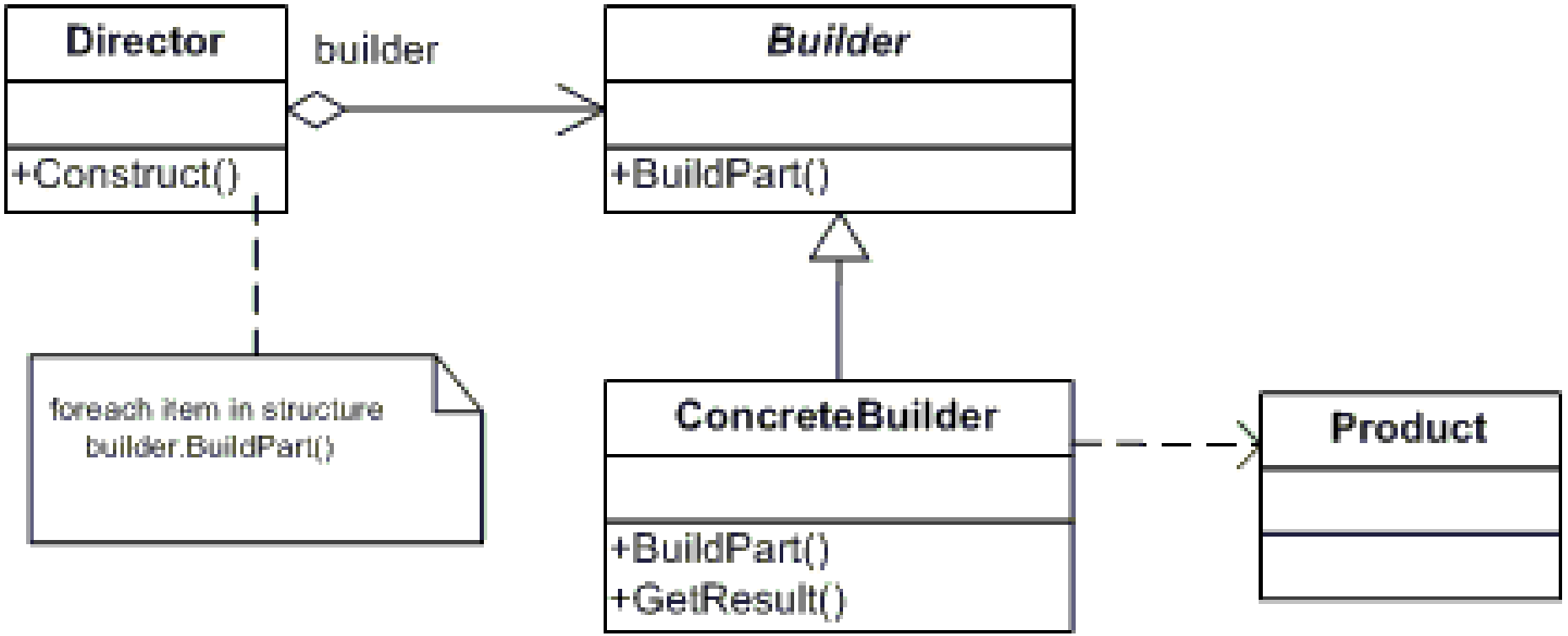


2.Bilgisayar

Eğer bir sınıfı oluşturduğumuzda kurucu sayımız fazla ise bu deseni kullanmak için sinyali aldık demektir.

Builder Tasarım Deseni: UML Diyagramı

Abstract Factory; belli bir ürün ailesini oluşturmayı hedefler. Builder ise tek bir ürünü oluşturmaya yöneliktir.



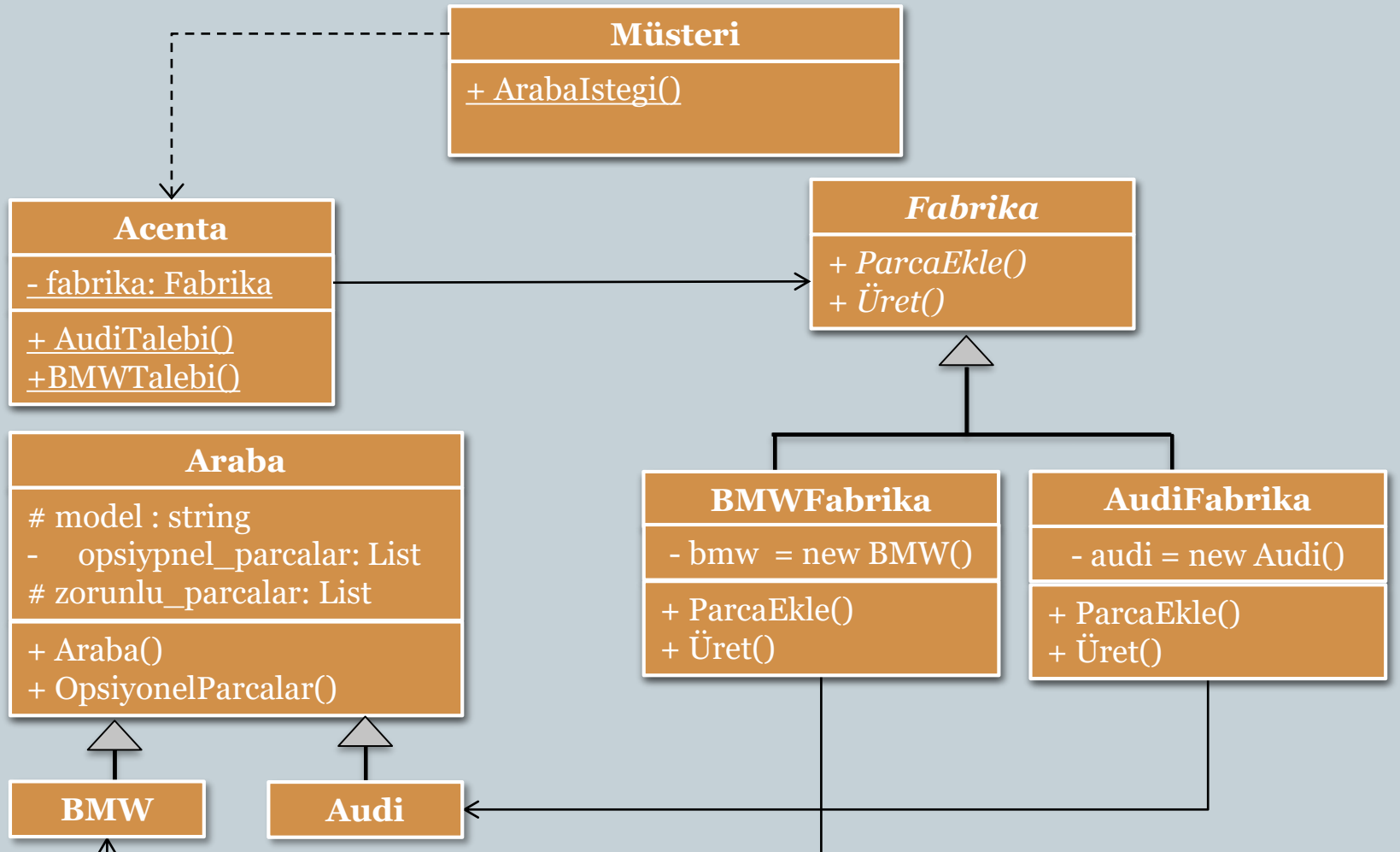
Builder Tasarım Deseni: Örnek


```
class Director
{
    public static void Construct(Builder builder,
                                 string[] parcalar)
    {
        foreach (string p in parcalar)
            builder.BuildPart(p);
    }
}

abstract class Builder
{
    public abstract void BuildPart(string parca_ismi);
    public abstract Product GetProduct();
}

class ConcreteBuilder1 : Builder
{
    private Product m_Product = new Product();

    public override void BuildPart(string parca_ismi)
    {
        m_Product.Add(parca_ismi);
    }

    public override Product GetProduct()
    {
       return m_Product;
    }
}

class ConcreteBuilder2 : Builder
{
    private Product m_Product = new Product();

    public override void BuildPart(string parca_ismi)
    {
        m_Product.Add(parca_ismi);
    }

    public override Product GetProduct()
    {
        return m_Product;
    }
}
```

```
class Product
{
    private List<string> m_Parts = new List<string>();

    public void Add(string part)
    {
        m_Parts.Add(part);
    }

    public override string ToString()
    {
        string s = "Ürüne ait parçalar : \n\r";

        foreach (string part in m_Parts)
            s += part + "\n\r";

        return s;
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] parca_listesi1 = { "a", "b", "c" };
        string[] parca_listesi2 = { "d", "e", "f" };

        Builder b = new ConcreteBuilder1();
        Director.Construct(b, parca_listesi1);
        Product p1 = b.GetProduct();
        Console.WriteLine(p1.ToString());
        ////////////////////////////////////
        b = new ConcreteBuilder2();
        Director.Construct(b, parca_listesi2);
        Product p2 = b.GetProduct();
        Console.WriteLine(p2.ToString());
    }
}
```

- http://dofactory.com/net/builder-design-pattern

Builder Tasarım Deseni: Örnek2



Builder Tasarım Deseni: Örnek2


```
enum Markalar
{
    BMW, AUDI
}

class Musteri
{
    public static void ArabaIstegi(Markalar marka, string model,
        params string[] istek_parcalar)
    {
        Acenta acenta = new Acenta();
        switch (marka)
        {
            case Markalar.AUDI:
                Audi a = acenta.AudiTalebi(model, istek_parcalar);
                Console.WriteLine(a.ToString());
                break;

            case Markalar.BMW:
                Bmw b = acenta.BmwTalebi(model, istek_parcalar);
                Console.WriteLine(b.ToString());
                break;
        }
    }
}
```

Object Pool Tasarım Deseni

- Performans çoğu zaman kaliteli bir yazılımın temel niteliklerinden birisidir.
- Belirli sayıda nesneyi önceden yaratıp havuzlama ihtiyacına cevap verir.

Tipik uygulamaları;

- Çok kullanıcıya hizmet veren ağır iş yükü altındaki uygulamalarda genelde orta katmandaki iş nesnelerinin applicaion server'da havuzlanması…
- Veritabanı bağlantılarının havuzlanması…

Object Pool Tasarım Deseni

- İstemcilerin ihtiyaç duyduğu bir anda böyle bir nesneyi yaratmaktansa, daha önceden yaratılmış hazır bir nesnenin kullanılmak üzere havuzdan çıkarılarak istemciye tahsis edilmesi;
- İstemcinin işini bitirdikten sonra da nesneyi yok etmeksizin tekrar havuza geri atması;

Sıklıkla uygulanan ve performansı da ciddi şekilde arttıran bir yöntemdir.

Object Pool Tasarım Deseni

- .NET kütüphanesinde hızlı thread yaratmayı sağlayan **ThreadPooling** mekanizması vardır.
- İşletim sistemi düzeyinde bir thread'in yaratılması karmaşık ve zor bir işlemdir ve zaman alır.
- Windows bunu hızlandırmak için belli sayıda thread'i önceden yaratıp havuzda tutar ve yeni bir thread açılmak istendiğinde havuzdan daha hızlı bir şekilde bu ihtiyacı karşılar.

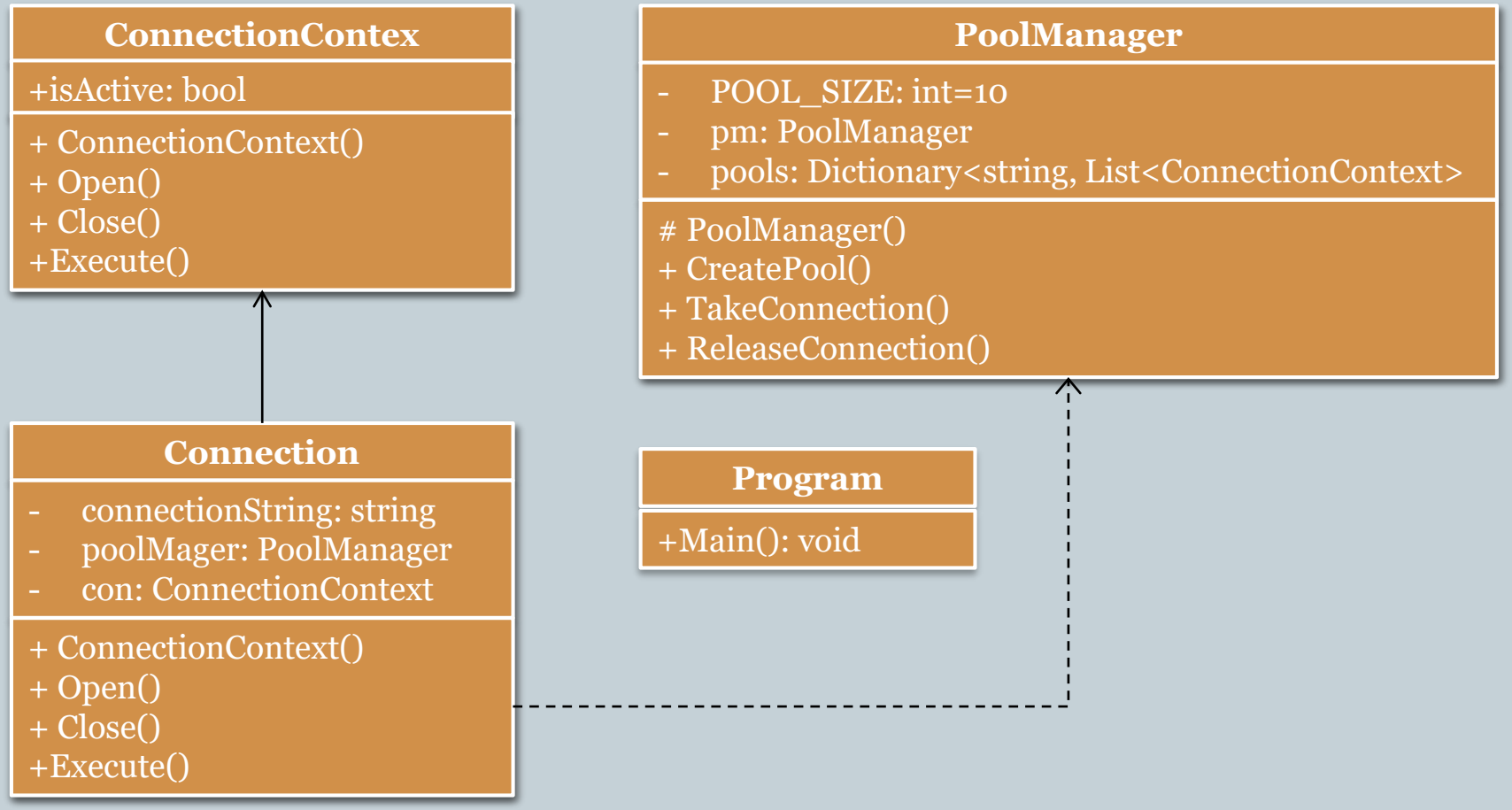
Object Pool Tasarım Deseni

- Bu işlemler OOP'ın doğasına uygun olarak yönetici bir sınıf tarafından gerçekleştirilir. **ConnectionPool** gibi.
- Bu sınıf önceden yarattığı n tane Connection nesnesini içsel olarak bir **container** nesnede tutar.
- **acquireXXX()** gibi bir fonksiyon ile talep üzerine istemciye tahsis eder.
- **releaseXXX()** gibi bir fonksiyon ile işi biten istemci nesneyi havuza geri gönderir.

Object Pool Tasarım Deseni

- Önceden çok fazla sayıda nesneyi havuzlamak da bellek kullanımını zararlı şekilde arttırabilir.
- Bu nedenle nesne sayısı makul (optimal) bir sevide tutulmalıdır.
- Şayet istemci nesne talep ettiği anda uygun bir nesne havuzda bulunmuyorsa, istemci beklemeye geçer.
- Ayrıca **ConnectionPool** gibi yönetici nesneler sistemi tek başına yönetmeleri gerektiği için Singleton deseni ile gerçekleştirilmelidir.

Object Pool Tasarım Deseni: Örnek



Object Pool Tasarım Deseni: Örnek


```
public class ConnectionContext
{
    private bool is_Active;
    // Nesnenin kullanılıp kullanılmadığı bilgisini tutar

    public bool isActive
    {
        get
        {
            return is_Active;
        }
        set
        {
            is_Active = value;
        }
    }

    public ConnectionContext()
    {
        is_Active = false;
        Console.WriteLine("ConnectionContext yaratıldı");
    }

    public void Open()
    {
        Console.WriteLine("Connection açıldı");
    }

    public void Close()
    {
        Console.WriteLine("Connection kapandı");
    }

    public void Execute(string query)
    {
        Console.WriteLine(query + " çalıştı");
    }
}
```

```
public class Connection
{
    private string connectionString;
    private PoolManager poolManager;
    private ConnectionContext con;

    public string ConnectionString
    {
        get
        {
            return connectionString;
        }
        set
        {
            connectionString = value;
        }
    }

    public Connection(string ConnectionString)
    {
        this.connectionString = ConnectionString;
    }

    public void Open()
    {
        poolManager = PoolManager.createPool();
        con = poolManager.TakeConnection(this.connectionString);

        if (con != null)
        {
            con.Open();
        }
        else
        {
            Console.WriteLine("Havuzda hiç uygun nesne yok");
        }
    }

    public void Close()
    {
        poolManager.ReleaseConnection(con);
        con.Close();
    }

    public void Execute(string query)
    {
        con.Execute(query);
    }
}
```

ÖDEV-2

- Abstract Factory, Builder, Object Pool tasarım desenleri için ayrı ayrı bir senaryo tasarlayınız.
- Projeleri C# ile kodlayınız. (tek solution içinde 3 proje)
- Her bir desen için tasarladığınız senaryo, araştırma ve kodlamalarınızın açıklamalarını içeren raporu elyazınızla hazırlayınız.
- Proje dosyalarınızı rar-layarak moodle.selcuk.edu.tr üzerinden 26.10.2017 saat 05.00'e kadar göndermeniz gerekmektedir. (ogrenci_no.rar)
- Raporları aynı gün ders saatinde getirmelisiniz.