SELÇUK ÜNİVERSİTESİ
TEKNOLOJİ FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

TASARIM DESENLERİ

[DESIGN PATTERNS]

(ITERATOR– OBSERVER – MEDIATOR)

YRD.DOÇ.DR. TAHİR SAĞ

KONYA, 2017

Iterator Tasarım Deseni

- Davranışsal desenler nesnelerin belirli davranışlarını ele alan ve bu noktada bazı doğru çözümler sağlayan tasarımlara yöneliktir.
- Iterator deseni davranışsal tasarım desenlerindendir.
- Kompozit bir nesnenin dizi, koleksiyon, liste, vektör gibi bir taşıyıcıyı kullanarak barındırdığı nesnelere, bu taşıyıcıyı nasıl ve hangi biçimde yapılandırıldığını bilmeksizin sıralı olarak erişmeyi sağlar.
- Amacı; çoğu zaman yapısını bilmediği bir listenin elemanlarına sıralı olarak erişmektir.

Iterator Tasarım Deseni

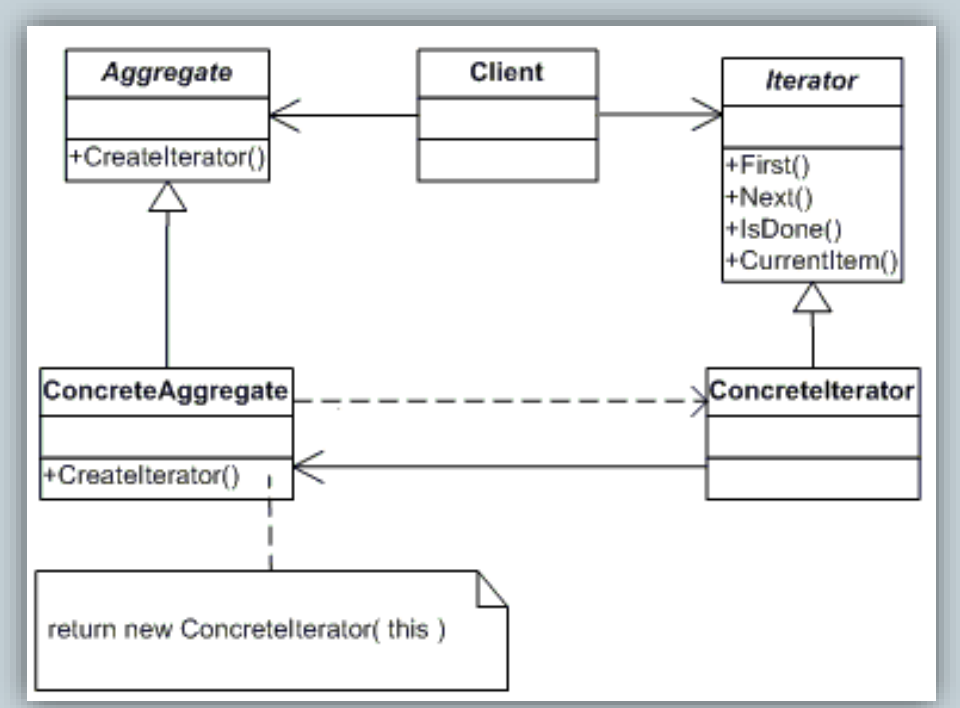
- Iterator deseni kullanılmadığında yaşanacak olumsuzluk değerlendirilecektir.
- Senaryo gereği bir firmada farklı departmanlarda çalışanlar olacaktır.
- Çalışan varlığı bir sınıfla temsil edilmiş olup, bu sınıf türündeki nesneleri, departmanlara ait sınıflar *aggregation* yoluyla içerecektir.
- Ancak bu noktada sorun; her departman sınıfında bu nesne koleksiyonunun farklı yapılandırılmış olmasıdır.

Iterator Tasarım Deseni: Olumsuz Örnek

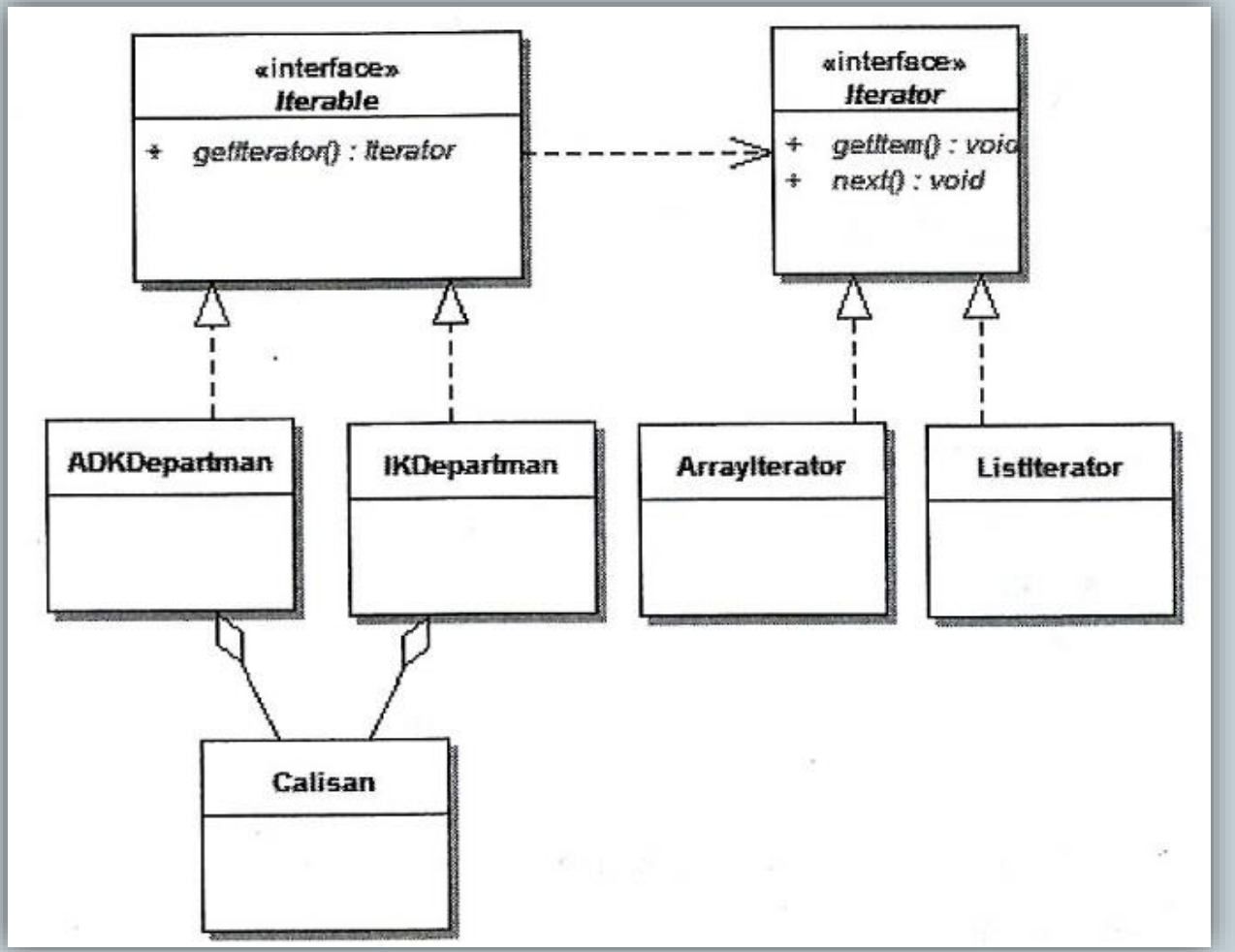

```
namespace IteratorExample
{
    class Program
    {
        static void Main(string[] args)
        {
            IKDepartmani ik = new IKDepartmani();
            Calisan[] elemanlar = ik.GetCalisanlar();
            for (int i = 0; i < elemanlar.Length; i++)
            {
                Console.WriteLine(elemanlar[i].ToString());
            }

            // farklı yöntem kullanılmak zorunda kalınmış !

            ADKDepartmani adk = new ADKDepartmani();
            List<Calisan> elemanlar2 = adk.GetCalisanlar();
            for (int i = 0; i < elemanlar2.Count; i++)
            {
                Console.WriteLine(elemanlar2[i].ToString());
            }

            Console.ReadKey();
        }
    }
}
```

Iterator Tasarım Deseni

- Doğru çözümde koleksiyonun yapısından bağımsız şekilde iterasyonu gerçekleştirmek üzere farklı koleksiyonlar için farklı iterator sınıflar yazılmalıdır.
- Bu sınıflar ortak bir interface-den türetilmelidir.
  - Parça nesneyi barındıran sınıflar böylesi bir iterasyona uygun olma anlamında bir interfaceden türemeli ve bu interface bize doğru iterator sınıfını kullandırmalıdır.



Iterator Tasarım Deseni



Iterator Tasarım Deseni: Örnek


```
namespace IteratorExample
{
    class Program
    {
        static void Main(string[] args)
        {
            IKDepartmani ik = new IKDepartmani();
            Calisan[] elemanlar = ik.GetCalisanlar();
            for (int i = 0; i < elemanlar.Length; i++)
            {
                Console.WriteLine(elemanlar[i].ToString());
            }

            // farklı yöntem kullanılmak zorunda kalınmış !

            ADKDepartmani adk = new ADKDepartmani();
            List<Calisan> elemanlar2 = adk.GetCalisanlar();
            for (int i = 0; i < elemanlar2.Count; i++)
            {
                Console.WriteLine(elemanlar2[i].ToString());
            }

            Console.ReadKey();
        }
    }
}
```

Iterator Tasarım Deseni – idiom-ları

- C# ve Java dillerinde Iterator desenine ilişkin idiomatik tasarımlar söz konusudur.
- .NET kütüphanesinde **`IEnumrable`** ve **`IEnumerator`**
- Java'da ise **`iterable`** ve **`iterator`** arayüzleri ve bu arayüzlerin sağladığı altyapı üzerinde çalışabilen for döngüsü aslında Iterator desenine ait idiomlardır.

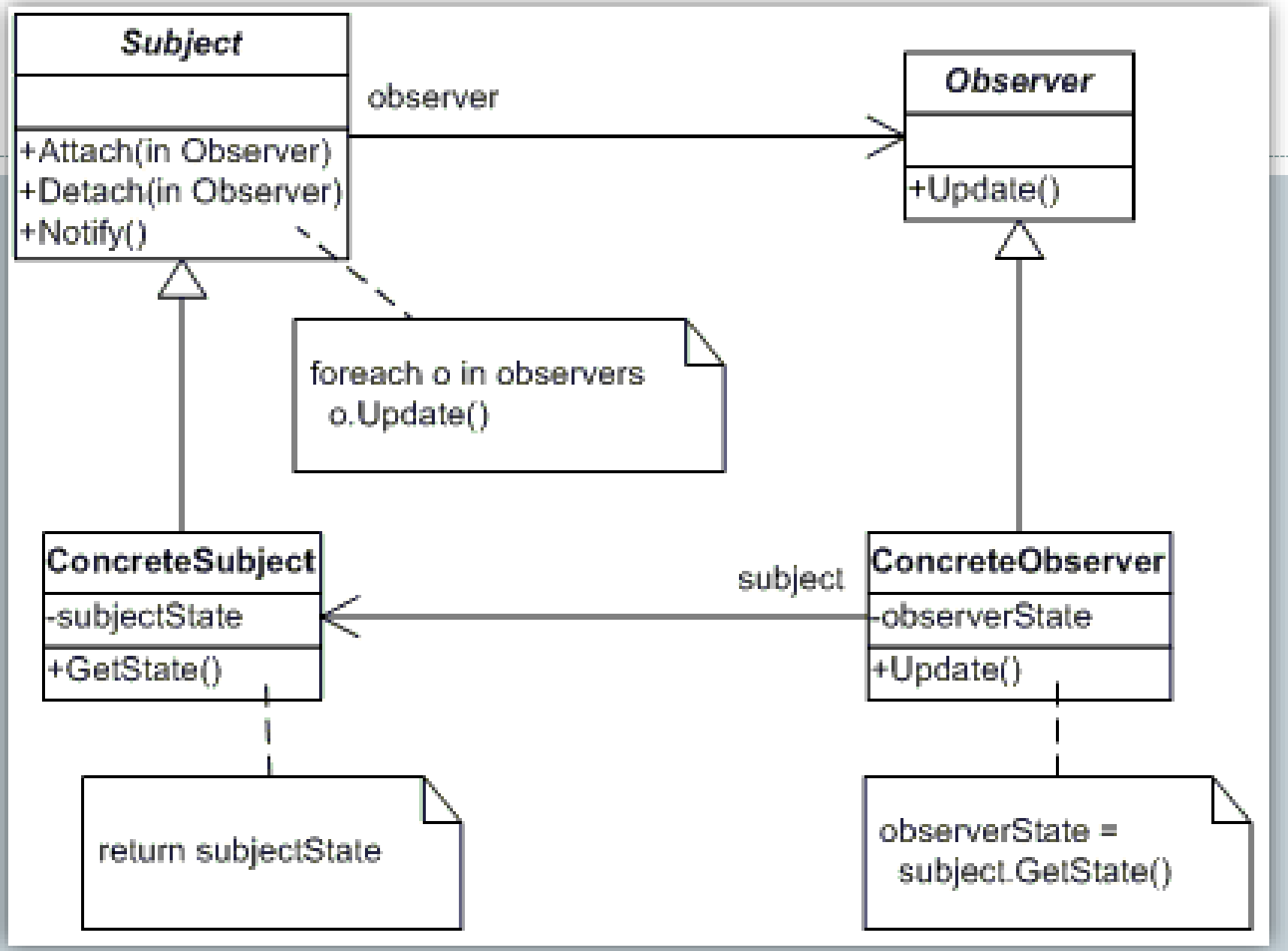
Iterator Tasarım Deseni – idiom-ları


```
class Program
{
    static void Main(string[] args)
    {
        SampleContainer sc = new SampleContainer();

        foreach (SampleObject item in sc)
        {
            Console.WriteLine(item.Value);
        }

        SampleIterator iter = (SampleIterator)sc.GetEnumerator();
        while(iter.MoveNext())
        {
            Console.WriteLine(((SampleObject)iter.Current).Value);
        }
    }
}
```

Observer (Publish-Subscribe) Tasarım Deseni

- Sistemimizdeki bir sınıfta herhangi bir değişiklik olduğunda, bu sınıftaki değişiklikten haberdar olması gereken sınıflar olabilir.
- Bu sınıflara gözlemci (observer) sınıflar denir.

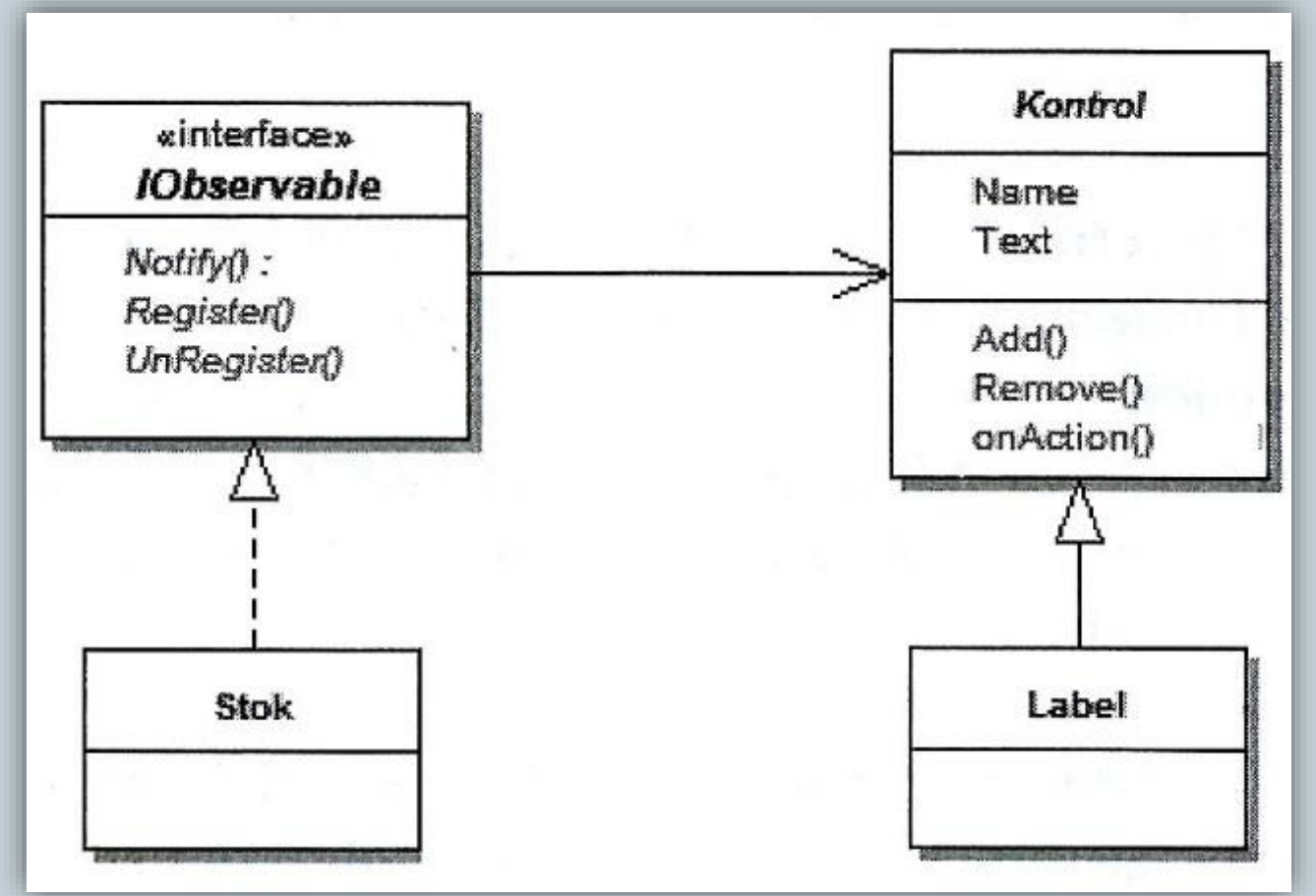
- Bu desende; durumu değiştiğinde kendisini izleyen (observe eden) nesnelere otomatik olarak uyarı gönderen reaktif nesnelerin davranışları ele alınır.
- Uyarıyı alan observer (gözlemci) nesneler ise tipik olarak durumlarını buna göre günceller.


Subject
+Attach(in Observer)
+Detach(in Observer)
+Notify()
observer
Observer
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

Observer Tasarım Deseni

- Observer genelde veri bağlama (databinding) işlemlerinde kullanılmaktadır.
- Pratikte çoğu zaman veri taşımakta olan bir entity nesnesi görsel arabirimdeki kontrollere bağlanmakta ve
- Nesnenin verileri, kontrol tarafından kullanıcıya sunulmaktadır.
- Hatta iki yönlü bir binding'te kontroldeki veri değişirse otomatik olarak entity de değişmektedir.

Observer Tasarım Deseni



Observer Tasarım Deseni


```
public interface IObservable
{
    void Register(Kontrol k);
    void Unregister(Kontrol k);
    void Notify();
}

public class Stok : IObservable
{
    private int m_Adet;
    private List<Kontrol> kontroller;
    public Stok()
    {
        kontroller = new List<Kontrol>();
        m_Adet = 10;
    }

    public void ElemanCek()
    {
        --m_Adet;
    }

    public void Register(Kontrol k)
    {
        kontroller.Add(k);
    }

    public void Unregister(Kontrol k)
    {
        kontroller.Remove(k);
    }

    public void Notify()
    {
        for (int i = 0; i <= kontroller.Count-1; ++i)
        {
            kontroller[i].onAction(m_Adet.ToString());
        }
    }
}
```

Mediator Tasarım Deseni

- NYP; modellenen sisteme ilişkin varlıkların nesnelerle ifade edilmesi ve bu nesnelerin kendi aralarında kurdukları ilişkiler ve gerçekleştirdikleri etkileşimleri göz önüne alan bir yaklaşımla uygulama geliştirilmesidir.

- Nesneler arasındaki bu iletişim, bir nesnenin başka bir nesneye ait fonksiyonu çağırması biçiminde somutlanabilir.

Mediator Tasarım Deseni

- Sözgelimi x nesnesi, y nesnesine ait bir fonksiyonu çağırıyor olsun. Şüphesiz bu durumda en azından x nesnesinin y nesnesini tanıyor olması gerekir ki bu durum iki nesne arasında bir bağımlılık ortaya çıkarır.
- Bu durumda *sadece* iki nesnenin iletişim kurabilmek için birbirini tanıması; nesne sayısı az olduğu için karmaşık değildir.
- Oysa uygulama pratiğinde ikiden çok fazla nesnenin kendi aralarında karmaşık bir iletişim gerçekleştirmesi söz konusu olabilir.

Mediator Tasarım Deseni

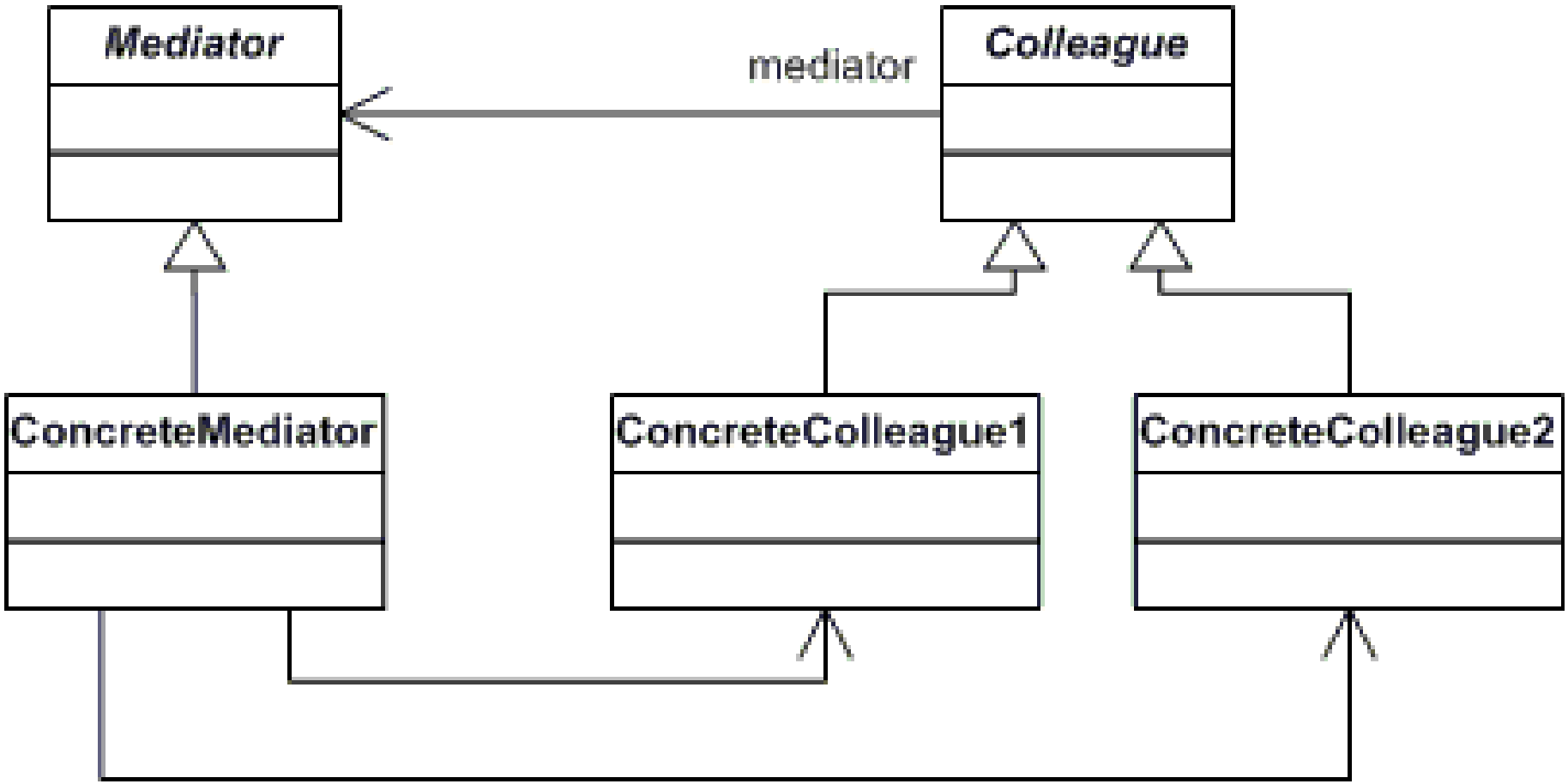
- Mediator deseni;

birbirini tanımak zorunda kalmadan birbirleriyle iletişim kurması istenilen nesneler için uygulanır.

Mediator Tasarım Deseni

- Bu desenin açıklanmasında kaynaklarda sıklıkla havaalanı modellemesi örneği yer almaktadır:
- Bilindiği gibi aynı havaalanını kullanan pek çok uçak olmaktadır. Bu uçakların pist kullanımı gibi işler için birbirleri ile haberleşmesi halinde kaos oluşacaktır. Onun yerine uçaklar birbirleri ile değil de bir kontrol kulesi üzerinden haberleşirler ve pist kullanımı gibi konuları merkezi bir unsur olan kule kontrol eder.
- Bu örnekte uçakları temsil eden nesneler arasındaki iletişim kule isimli mediator nesne üzerinden yürütülmektedir.

Mediator Tasarım Deseni

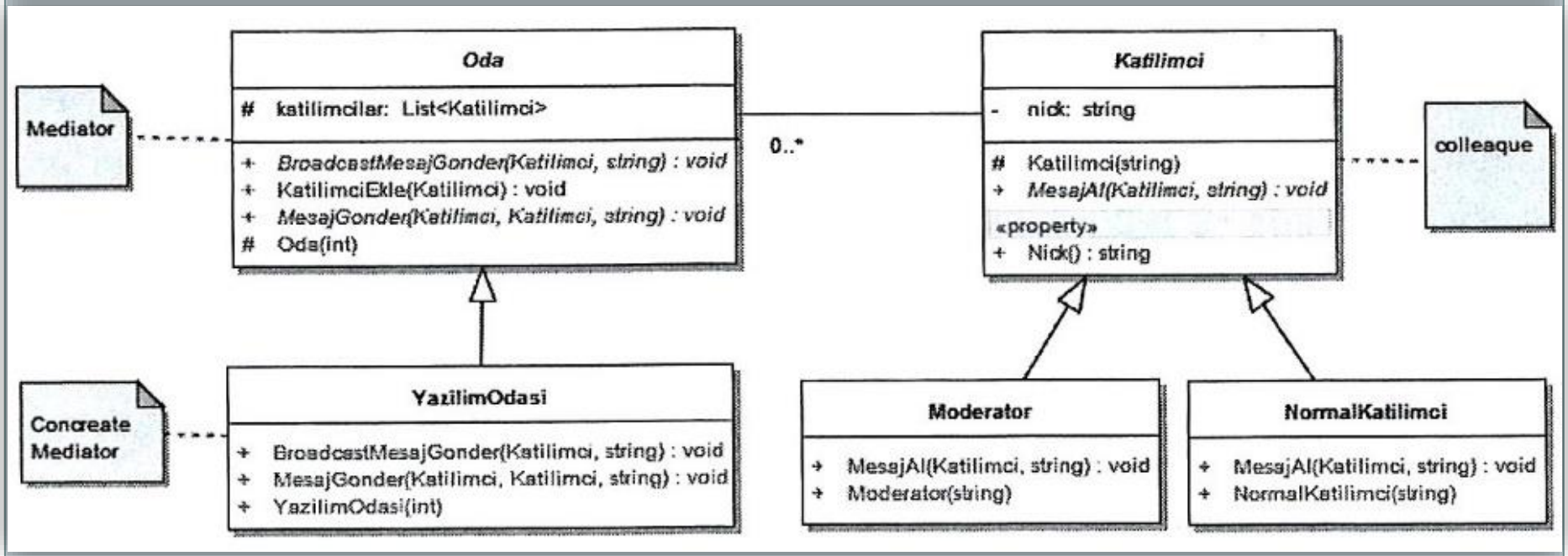


Mediator Tasarım Deseni

- Mediator deseni, aralarında iletişim kuracak nesnelerin (colleague nesneler) birbirlerini bilmeksizin merkezi bir (mediator) nesne üzerinden haberleşmesini öngörür.
- Mediator taban sınıfından türemiş bir sınıfa ilişkin nesne (ConcreteMediator) tüm colleague nesneleri içsel bir dizi yada kolleksiyonda saklar.
- Ayrıca her bir colleague nesne ise ConcreteMediator olan nesneyi referanse eder (association ilişkisi biçiminde).

Mediator Tasarım Deseni: Örnek

- Bu desene verilmiş en güzel ve basit örneklerden birisi de; sohbet (chat) odası örneğidir.



Mediator Tasarım Deseni: Örnek

- Örnek uygulamada Mediator sınıf olan Oda, Katilimci sınıfı türündeki nesneleri bir liste içinde barındırmaktadır.
- Böylece mediator nesne tüm katılımcı nesneleri tanır durumdadır.
- Ancak bir katılımcı nesne asla diğerini tanımaz (doğrudan ilişki kuramaz).
- Oda nesnesi tüm katılımcıları bilir ve haberleşmeyi yönetir.
- Oda ve Katilimci sınıflarından türeyen sınıflar ConcreteMediator ve ConcreteColleague durumundadır.

Mediator Tasarım Deseni: Örnek


```
// Soyut Mediator
abstract class Oda
{
    protected List<Katilimci> katilimcilar;

    protected Oda()
    {
        katilimcilar = new List<Katilimci>();
    }

    public void KatilimciEkle(Katilimci k)
    {
        if (! katilimcilar.Contains(k))
        {
            katilimcilar.Add(k);
        }
    }

    public abstract void MesajGonder(Katilimci gonderen,
        Katilimci alan, string mesaj);
    public abstract void BroadcastMesajGonder(
        Katilimci gonderen, string mesaj);
}
```