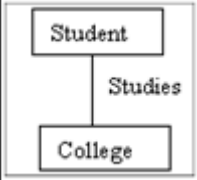
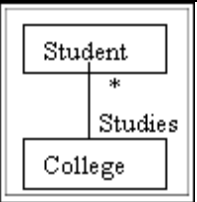
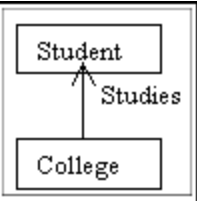
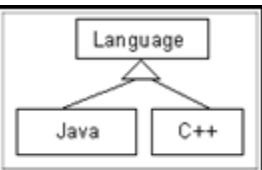
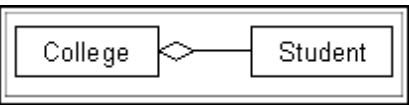
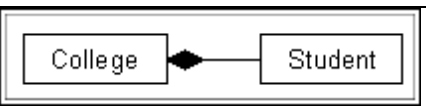
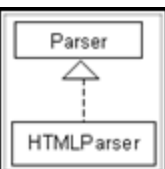


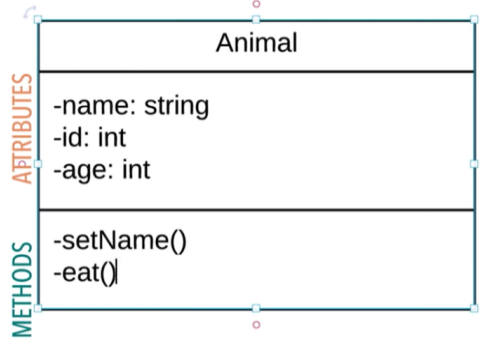
UML SINIF DIYAGRAMLARI

Sınıf İlişkileri Özet Tablosu

No	İlişki	Örnek Gösterim	Açıklama
1	Bağıntı(Association)		İki sınıfın herhangi bir şekilde birbirleriyle iletişim kurmasıdır. Örneğin : Öğrenci ve Okul arasındaki ilişki
1.a	Çokluk (Multiplicity)		İki sınıf arasındaki 1:n ilişkidir. Birden fazla öğrencinin okul ile olan ilişkisi olarak açıklanabilir. Şekildeki * işareti 1:n ilişkiyi göstermektedir.
1.b	Yönlü Bağntı (Directed Association)		Sınıflar arasındaki tek yönlü ilişkiyi gösterir. Ok işareti ile gösterilir.
1.c	Dönüslü Bağntı(Reflexive Association)	Belirgin bir çizimi yoktur.	Bir sınıfın kendisiyle kurduğu ilişkidir.Bu tür ilişkiler genellikle bir sınıfın sistemde birden fazla rolü varsa ortaya çıkar
2	Kalıtım/Genelleme(Inheritance/Generalization)		Bu ilişki, bir nesnenin bir diğerinin özel bir türü olduğu gerçeğini modellemek için kullanılır.
3	İçerme(Aggregation)		İki sınıf arasındaki “sahiptir” veya içerir türünden bağıntıları modellemekte kullanılır.Bütün parça içi boş elmas ile gösterilir.
3.a	Oluşum(Composition)		İçerme ilişkisinin farklı bir çeşitidir. Tüm bağıntılar içinde en güçlü olanıdır. Parça-bütün ilişkilerini modellemekte kullanılır.
4	Gerçekleştirim(Realization)		Kullanıcı arayüzlerinin modellemesinde kullanılır. Arayüz yalnızca method adlarını ve bunların parametrelerini içermektedir.

UML sınıf diyagramlarında genel olarak aşağıdaki yapılar ve ilişkiler bulunur.

1. **Classes/interfaces** : Somut sınıfları,abstract sınıfları ve interface'leri temsil eder.



En üstteki kısımda sınıfın ismi,onun altında üye değişkenler ve en altta da metotlar bulunur.

<<abstract>> veya *sınıf_adı* -> Abstract sınıf

<<interface>> -> arayüzleri belirtir.

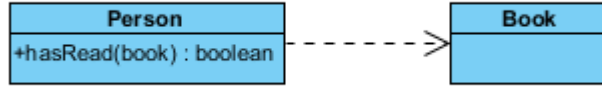
2. **Bağıntı(Association)** : Bir sınıfın bir başka sınıfın örneğini tutmasıdır. Yönü ve çokluk durumu belirtilmez. Örneğin bir İnsan sınıfı içinde onun arabasını gösteren Araba sınıfı türünden referans bulunması.
3. **Çokluk(Multiplicity)** : İki sınıf arasındaki ilişkiyi Association'dan farklı olarak çokluğunu belirterek kurar. Örneğin tablodaki No 1.a satırında bir College'in 0 veya çok sayıda Student'i vardır ilişkisi oluşturulmuştur. Tablo okuması seçilen sınıftan ilişkili sınıfın yakınındaki *,0,1,3.5 vb. çoklukla ilişkilendirilecek şekildedir.
4. **Yönlü Bağlantı(Directed Association)** : İki sınıf arasındaki bağlantının yönünü de belirler. A sınıfından B sınıfına doğru bir ok varsa A sınıfı B sınıfına ait bir referansı içinde bulunduruyordur(B sınıfıyla ilişki kurmak için). İlişkiler çok yönlü olabilir bu durumda çift yönlü ok kullanılır ya da oksuz düz çizgi kullanılır.
5. **Kalıtım/Genelleme(Inheritance/Generalization)** : Bir sınıfın somut bir sınıftan veya abstract bir sınıftan kalıtım alma durumudur. Kalıtım alan sınıftan base sınıfa doğru içi boş üçgen ile gösterilir(Tablodaki no 2).
6. **Gerçekleştirim(Realization)** : Bir sınıfın bir arayüzü gerçekleştirmesidir(Tablo No 4).
7. **İçerme(Aggregation)** : Bir sınıfın bir başka sınıfın parçası olması durumu söz konusudur. Bütün olan kısımda boş elmas(deltoid) bulunur. Composition'a göre daha zayıf bir ilişki söz konusudur.

8. **Oluşum(Composition)** : Aggregation'a gibi bir bütün parça ilişkisi vardır. Fakat buradaki ilişki daha güçlüdür. Bütün olan sınıfta içi dolu elmas/deltoid bulunur.

Aggregation ve Composition'u ayıran noktalar

- Aggregation bağlantısında parçalar tek başına da bir anlam ifade eder. Bütün olarak tanımladığımız sınıf oluşturulduğunda Composition sınıfların(parçaların) oluşma şartı yoktur.
- Oluşum bağlantısında her parça aynı anda sadece tek bir bütüne ait olabilir. Bu nedenle oluşum bağlantısında bütün tarafı çokluk değeri daima 1 olmak zorundadır. İçerim bağlantısında ise bir parçanın birden fazla bütün tarafından paylaşımı söz konusu olabilir.
- Oluşum ilişkisinde parçanın yaşam süresi doğrudan bütünün yaşam süresine bağlıdır. Bütün nesnesi yaratılmadan parça nesnesi yaratılamaz ve bütün tarafındaki nesne silindiğinde parça nesnelerde onunla birlikte silinmelidir.

9. **Bağımlılık(Dependency)** : Bir sınıf başka bir sınıfa ait örneği kullanıyor fakat bu örneği tutan referansı sınıfı içindeki bir değişkenle tutmuyorsa bu ilişki söz konusudur. Bu ilişkide bir sınıf başka bir sınıfa ait örneği metod parametresi olarak alır ve kullanır fakat depolamaz. En zayıf ilişki türüdür. Kesikli çizgi ve okla gösterilir.



YAZILIM TASARIM DESENLERİ

Bir yazılım deseni kapsamına göre 2 ayrılır:

- **Sınıf** : Sınıf kapsamında olan bir yazılım deseni sınıflar ve alt sınıfları arasındaki ilişkiyi belirler, statiktir : derleme anında belirlenir ve değişmez.
- **Nesne** : Nesneler arasındaki ilişkileri belirler, dinamiktir : çalışma anında değişir.

Genel ve yaygın kategorilendirmede ise yazılım tasarım desenleri 3 kategoriye ayrılır:

1. Yaratımsal(Creational) Tasarım Kalıpları : Bu tasarım desenlerinin hepsi sınıf örneği ile ilgilidir. Bu model daha sonra sınıf oluşturma modellerine ve nesne oluşturma modellerine ayrılabilir. Sınıf oluşturma kalıpları, örnekleme sürecinde mirasını etkin bir şekilde kullanırken, nesne oluşturma kalıpları işi yapmak için temsilciliği etkin bir şekilde kullanır. Yaratımsal kalıplar nesnelerin ilklendirilme işlemlerini soyutlaştırırlar. Sistem bu işle alakalı Ne, Kim, Nasıl, Ne Zaman sorularını yaratımsal kalıptaki önerilerle(abstract sınıf ve interfacerler) yardımıyla yapar.

- Singleton Pattern
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern

2. Yapısal(Structural) Tasarım Kalıpları : Bu tasarım kalıpları OOP sistemlerinde derin kalıtım hiyerarşileri yerine composition'ların kullanılmasının daha iyi bir yaklaşım olduğunu önerir. Yapısal tasarım kalıpları sınıfların ve nesnelerin nasıl bir araya gelerek(kalıtım ve composition) ile nasıl daha büyük yapıları oluşturacağını belirler.

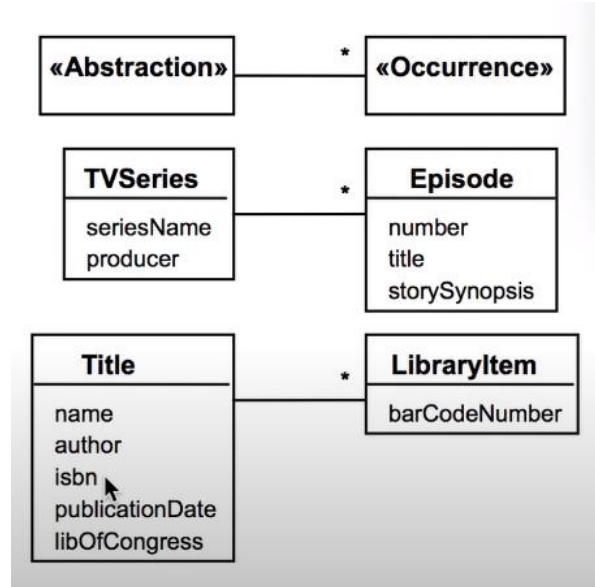
- Singleton Pattern
- Factory Pattern
- Abstract Factory Pattern
- Builder Pattern
- Prototype Pattern

3. Davranışsal(Behavioral) Tasarım Kalıpları : Bu tasarım desenleri, tamamen Class'ın nesne iletişimi ile ilgilidir. Davranışsal desenler, nesneler arasındaki iletişimle en özel olarak ilgilenen kalıplardır. Çalışma anında kontrol edilmesi zor olan karmaşık akış

kontrolünü düzenlerler. Bu sayede akış kontrolü ile ilgilenmek yerine nesnelerin birbiriyle nasıl bağlı olduğuyula ilgileniriz.

- Chain of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern
- Observer Pattern
- Null Object Pattern
- Strategy Pattern
- State Pattern
- Visitor Pattern

1. Abstraction-Occurence Kalıbı



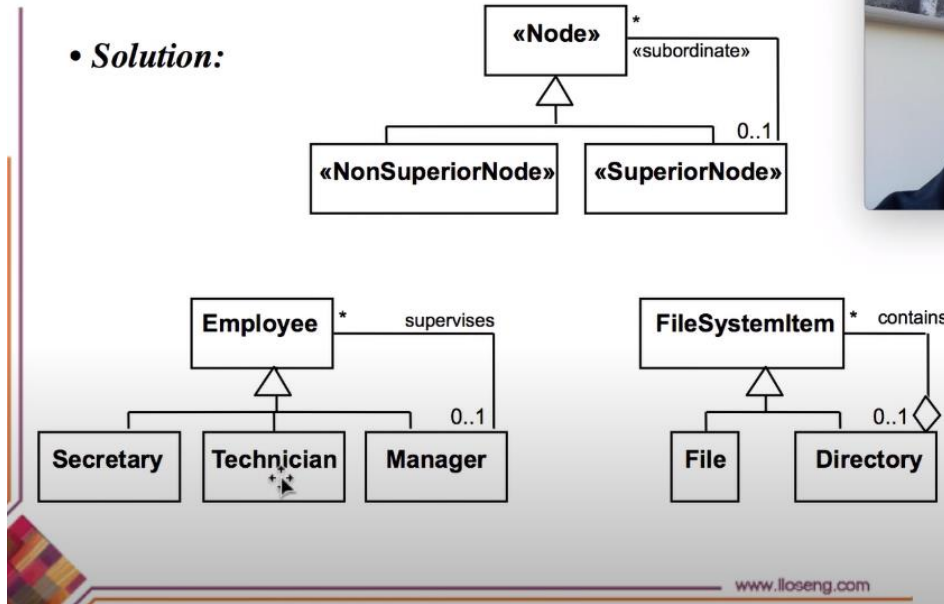
Bazen ilk bakışta kalıtım ilişkisi ile çözülebilecek fakat daha da dikkatli bakınca kalıtımla kurulduğunda hatalı, veri tekrarının bulunduğu bir sınıf ilişkisi olabilir. Bu ilişki : kalıtım alan çocuk sınıfa ait örneğin kalıtım aldıkları base sınıftaki sadece tek bir tekil örneğe bağlı olmaları gerektikleri durumdur. Bu yazılım deseni veritabanı tasarımı yaparken tek tabloda ifade etmekten kaçınıp 2. Bir tablo açarak foreign key ile ifade ettiğimiz bire çok ilişkilerine

benzetilebilir. Örneğin bir müşterinin birden fazla adresi olduğu durumda bunları müşteri tablosuna ifade etmek yerine adres adında yeni bir tablo açarız ve açtığımız bu tabloda müşteri ID adında bir bir değişken tanımlayarak bu adresler ile Müşteri arasında bir ilişki kurarız. Buradaki Müşteri tablosuna: Abstraction, Adres tablosuna : Occurence benzetmesi yapabiliriz. Abstraction-Occurence tasarım desenine sınıflar ile örnek verecek olursak:

Örnek 1 : Bir kütüphane yazılımında daha önce belli bir yazar tarafından yazılmış, ismi verilmiş kitaplar olsun. Bu kitaplar bir kavram olarak ele alındığında Abstraction, bu kitabın canlı bir örneği(elimize alıp okuyabildiğimiz, hissebildiğimiz) Occurence olarak ele alınır.

Her bir kitap örneği bir kişi tarafından belli bir tarihte bağışlanmış, unik bir kitap numarası olan somut örneklerdir. Fakat tüm bu kitap örnekleri ortak bazı değerler içerecektir. Bu ortak değerler kitap yazarı,cilt numarası,kitap adı gibi değerler olabilir. Bu değerleri içeren KatalogKitap adında bir sınıf tanımlarsak ve kitap örneklerini Kitap adında bir sınıf ile tanımlarsak aynı KatalogKitap'a sahip kitap örneklerinin tek bir KatalogKitap örneğini göstermesini sağlayabiliriz. Bunu yapmak için Kitap sınıfında KatalogKitap sınıfına ait bir referans tutmamız yeterlidir. KatalogKitap sınıfında, kitap örneklerini tutmak istiyorsak da bu sınıf içinde bir koleksiyon(LinkedList,ArrayList vb.) tanımlayarak bu koleksiyon içinde kitap örneklerini tutabiliriz.

2. General Hierarchy Kalıbı

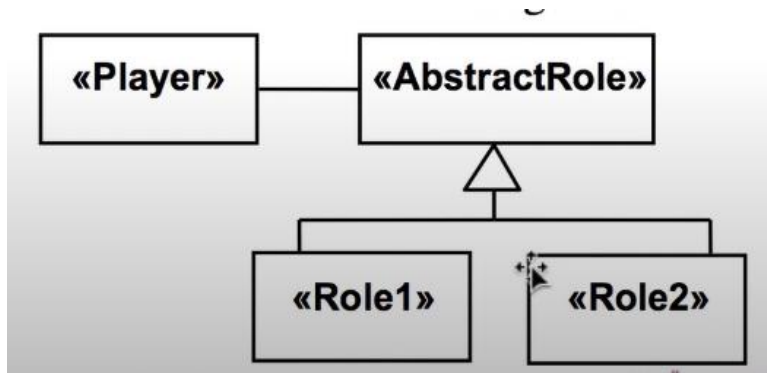


Aralarında kalıtım ilişkisi bulunan sınıflar(alt ve üst sınıflar arası) arasında ayrıca altlık-üstlük, birbirini yönetme/yönetilme gibi ilişkilerin de bulunması durumudur. Örneğin bir firmadaki çalışanları modelliyor olalım. Çalışanlar adında bir base sınıf, eleman ve yönetici adında da alt sınıflar olsun. Yönetici bir çalışandır. Elemanları yönetir ve başka bir yönetici tarafından yönetilebilir. Her bir çalışanın yöneticisinin kim olduğu bilinmek isteniyorsa base sınıf olan

Çalışan sınıfına Yönetici tipinden bir referans koymak yeterli olacaktır. Bu sayede bir çalışan yöneticisine erişebildiği gibi yöneticisinin de yöneticisine erişebilir. Bu durum en üst seviyedeki yöneticiyi bulana kadar devam edebilir.

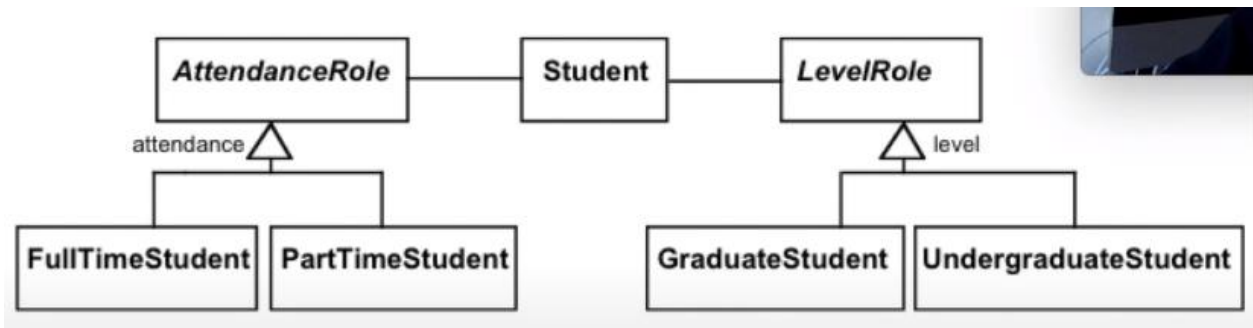
Yöneticiler altında çalışan kişilere erişmek istiyorsa : bir yöneticinin altında çalışan çok sayıda kişi olabileceği için yönetici sınıfında içinde Çalışan tipinden referans tutan bir koleksiyon tanımlamak gerecektir.

3. Player-Role Tasarım Kalıbı



Bir sınıf aynı anda birden fazla tanımlayıcı özellik(rol) içerebilir. Tek bir role sahip bir sınıf için kalıtım(çoklu olmayan) kullanmak yeterli olacaktır. Örneğin bir sekreterin tek rolü çalışan olmasıysa Sekreter sınıfının Çalışan sınıfından kalıtım alması yeterli olacaktır. Tek bir rol olma durumuna bir örnek daha verecek olursak bir şirkette outsource ve yerel çalışanlar olmak üzere iki tip çalışan olsun : Bir iki tip çalışan sınıfı da Çalışan sınıfından kalıtım alacaktır ve bir çoklu kalıtım durumu söz konusu değildir.

Player-Role tasarım kalıbı çoklu kalıtım olmayan durumlarda da uygulanabiliyor olsa da esas olarak çoklu kalıtım gerektiren durumlarda kullanılması daha faydalıdır. Örneğin aşağıdaki şekilde de gözüktüğü üzere bir Öğrenci, seviyesine(Lisans,LisansÜstü) ve devam tipine göre(full time,part time) iki ayrı sınıfta incelenebilir. Bu ilişki sadece çoklu kalıtım ve sadece hiyerarşik kalıtım kullanılarak tasarlanabiliyor olsa da Player-Role kalıbı ile tasarlamak daha iyi bir yol olacaktır.



Player-Role tasarım kalıbında bizim yaşayan sınıfımız(Player), farklı rolleri içinde referans olarak tutar. Tuttuğu referanslar her farklı rol grubu için bu rol grubuna ait base sınıfı tutacak şekildedir. Bu sayede her rol grubunda sonsuz tane alt rol tanımlamak mümkündür. Player-Role tasarım kalıbının bu özelliği sayesinde genel tasarım bozulmadan tasarımı genişletmek mümkündür.

4. Singleton Tasarım Deseni

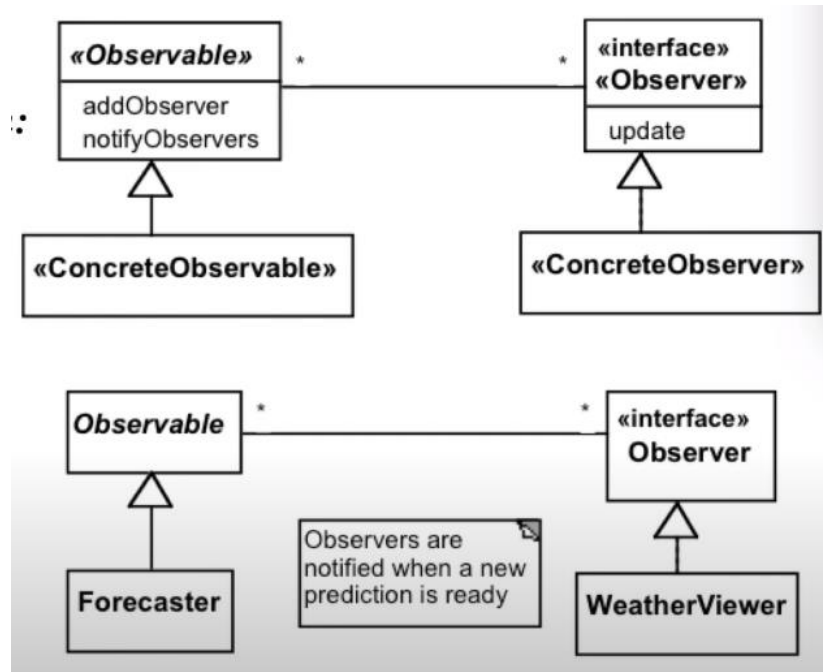
Bir sınıftan sadece bir tane örnek oluşturulmasını istediğimiz durumlarda Singleton tasarım desenini kullanırız. Örneğin örnekleri SQL Connection yapan bir sınıftan fazlaca örnek oluşturulup belleğin şişirilmesini istemeyeceğimiz için bu sınıfı Singleton olarak tasarlayabiliriz.

Bir sınıfı Singleton yapmak için :

- Yapıcı metodu private yapılır.
- Sınıf içinde kendi türünden referans tutan statik bir değişken tanımlanır.
- Kendi sınıf türünden bir referans döndüren statik bir metot tanımlanır. Bu metot eğer daha önce bir örnek oluşturulmamışsa oluşturur, oluşturulmuşsa sürekli bu örneği döndürür.

Singleton tasarım deseni temel alınarak örnek oluşturma sayısını 1 ile kısıtlamak yerine 1'den fazla bir sayı ile de kısıtlayabiliriz.

5. Observer Tasarım Deseni

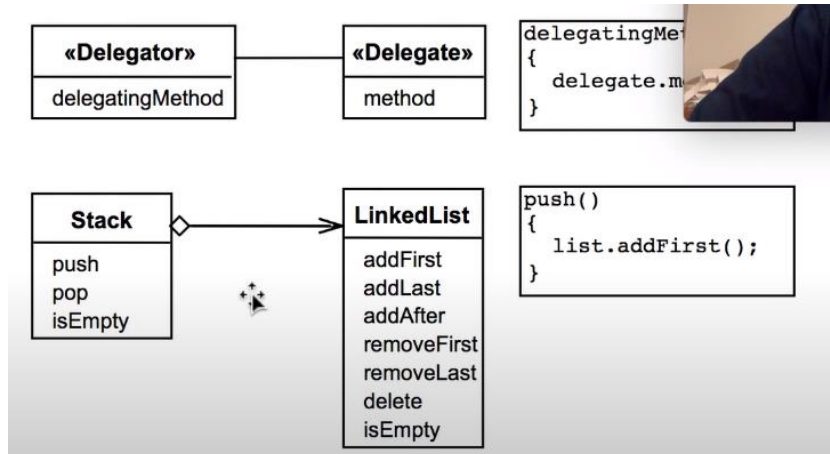


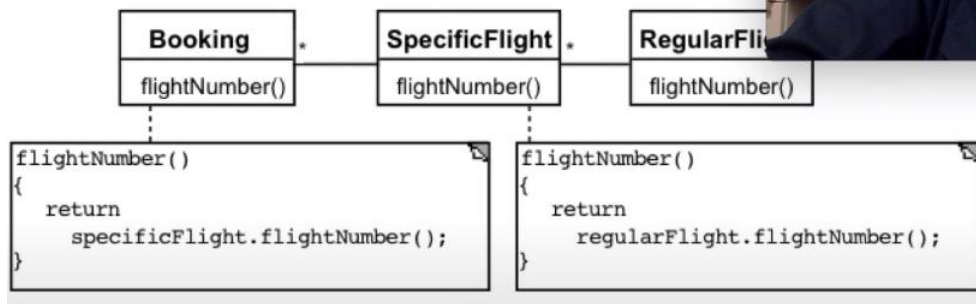
Elimizde sürekli yeni veriler eklenebilen/silinebilen/değiřtiribilen veri kaynakları(Observable/Subject) ve bu veri kaynaklarındaki her deęiřimde kendini g¼ncellemesi gereken(Observer) yapılar bulunsun. Bunu saęlamak için baęımlı yapıların her an veri kaynaklarına bir deęiřim var mı, varsa ne diye istek g¼ndermesi verimli bir yol olmayacaktır. Bunun yerine veri kaynakları(observerlar), verilerinde bir deęiřim olduęunda/observerları bir olay hakkında bilgilendirmesi gerektięinde bunu yaparlar. T¼m observerların aynı interface'i implement etmesi saęlanarak, Observable sınıfı ierisinden t¼r¼ ne olursa olsun tek bir metod aęırımıyla tetiklenmesi saęlanmıřtır.

rnek olarak farklı kaynaklardan s¼rekli veri akıřının olduęu haber kaynaklarımız olsun. Bu haber kaynaklarının herbirini farklı bir yapı olarak d¼ř¼n¼p bunları farklı sınıflar altında inceledięimiz d¼ř¼nelim. Observable sınıfını abstract veya normal sınıf olarak tanımlarız. Bu ana sınıf iinde Observerları tutar, bunları ekleyen,silen ve tetikleyen metotları tutar. T¼m alt observerlable sınıfları da bu sınıfı implement eder. Tek bir tane tipte haber kaynaęı(observable) olsa bile tasarımı bu řekilde yapmak deęiřime daha kolay uyum saęlayan bir tasarım saęlayacaktır. Observer tarafında ise t¼m baęımlıları/clientleri tutan bir interface tanımlarız. Bu interface iinde update metodunu bildirir ve bu update metodu gerektięinde Observable sınıfı iinden aęırılacaktır. Bu rnekte observerlara, observable'den aldıęı haberleri bir grafiksel aray¼ze kullanıcıya g¼steren mobil uygulama, web sitesi vs. verilebilir. Haber kaynaęına(observable) yeni bir haber geldięinde iindeki koleksiyonda tuttuęu t¼m Observer(interface) t¼r¼nden nesnelerin update metodunu aęıracaktır.

- Observer sınıfı iinde ayrıca Observable sınıfı tipinden referans tutulması iyi bir tasarım deęildir.
- Observer(interface ile) ve Observable(birden fazla tipteyse abstract veya normal sınıf ile) kendi aralarında gruplanmalıdır.
- Observer deseni C#'da bulunan delegate-event iliřkisine benzemektedir.

6. The Delegation Tasarım Kalıbı





Bu kalıpta Delegator bir işi kendi becerileriyle, kaynaklarıyla yapmak yerine işi bir başkasına(Delegate) yaptırır. İş yaptırmak istediği kişi de bu işi bir başkasına devredebilir(Şekil 2). Bu devretme işlemi komşu sınıflar arasında olmalıdır. Örneğin şekil 2’de Booking sınıfından specificFlight.regularFlight.flightNumber() şeklinde bir çağrım yapmak doğru değildir.

Delegation desenini bebek ve anne ilişkisine benzetebiliriz. Bebeğin yemek yemesi bir zorunluluktur ve yemekYe isminde bir metodu bulunmalıdır. Fakat bebek nasıl yemek yenileceğini bilmez bu yüzden yemek yeme işini ona annesi yaptırmalıdır. Bebek kendi Anne sınıfına ait bir referans ile Anne sınıfının yemekYe metodunu çağırarak yemek yeme işlemini ona yaptırır. Bu işlem delegation, bebek delegator, anne de delegate’tir.

- Delegation kalıbında bir işi başka sınıfa devrederken sadece devredilen sınıfa ait karşılık gelen metod çağrılır. Delegator’ın bu metodun içeriği hakkında bir fikri yoktur. Yani bu tasarım kalıbında soyutlama söz konusudur.

7. Immutable Tasarım Kalıbı

Bir sınıfın Immutable olması :

- Sınıfa ait bir örneğin bilgilerinin değiştirilmek istendiğinde o anki örnek üzerinde bir değişiklik yapmak yerine o anki örneğin bir kopyası alınarak onun üzerinde değişiklik yapılmasıdır.

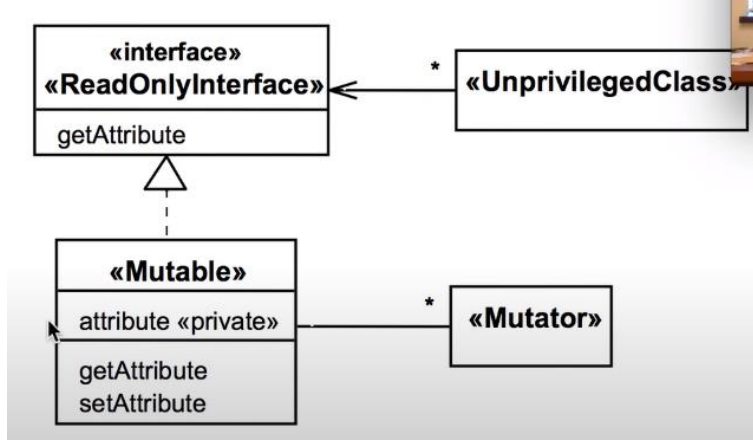
Örneğin Java’da String sınıfı Immutable bir sınıftır. Java’da bir String’in belli bir indeksinde karakterine erişip bu karakteri değiştiremeyiz. Replace gibi metotlarla bunu yaptığımızda ise Java arka planda mevcut String değerinin bir kopyasını alır, bunun üzerinde istenilen değişiklikleri yapar ve bu yeni String değerini döner.

Bir sınıfı Immutable tanımlamak için sınıfın üye değişkenlerinin değiştirilmesi engelleriz. Bunu yapmanın bir yolu üye değişkenleri private yapmak ve bu değişkenlere ait sadece get metotları yazmaktır. Üye değişkenleri ilklendiren bir yapıcı fonksiyon da tanımlayarak sınıfın değerleri değiştirilmek istendiğinde yeni bir örneğin oluşturulmasını garanti altına almış oluruz.

Bir sınıfı ne gibi durumlarda Immutable yapmak gerekli olur aşağıdaki linkte incelenmiş :

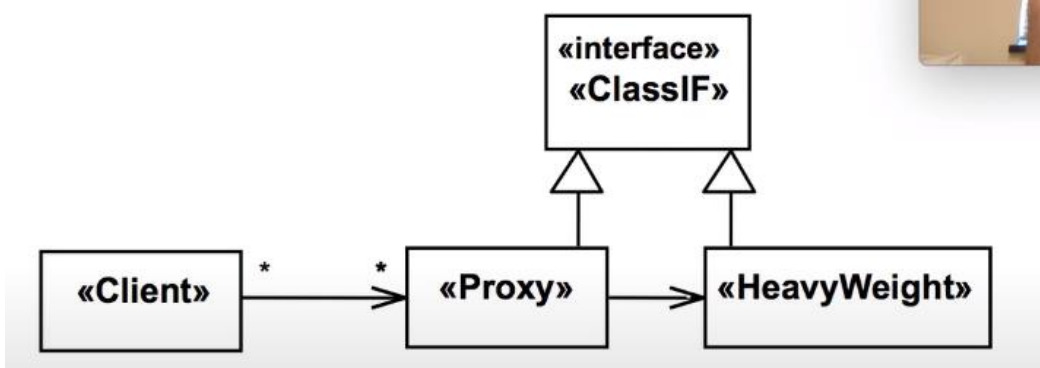
[https://www.buraksenyurt.com/post/Sabit-Deger-Tipleri-\(Immutable-Value-Types\)-Analizi-bsenyurt-com-dan](https://www.buraksenyurt.com/post/Sabit-Deger-Tipleri-(Immutable-Value-Types)-Analizi-bsenyurt-com-dan)

8. Read-only Tasarım Kalıbı



Read-only tasarım kalıbıyla bir sınıftaki üye değişkenlerin değerlerini belli referans tiplerinin değiştirmesini engelleriz. Bu referans tipi erişimi engellediğimiz üye değişkenin kendi sınıfı ise bunu interface kullanmadan yapabiliriz fakat bunu interface ile yaparsak Sınıfın referansı üzerinden sorunsuz bir set işlemi sunmakla birlikte, interface referansı üzerinden sadece get işlemi yapılmasını(read-only) sağlayabiliriz. Böylece direkt olarak Mutable sınıfın referansına sahipken set işlemi yapabiliyorken, bu interface tipinden referansları kullanan diğer sınıflar read-only yapmak istediğimiz sınıfın sadece belirlediğimiz değişkenlerinin get metotlarına erişebilirler ve daha güvenli ve esnek bir tasarım söz konusu olur.

9. Proxy Design Pattern



Bu yazılım kalıbında client tarafından bir işi yapacak nesneye(HeavyWeight/Real Subject) direkt olarak erişmek yerine bir Proxy(Vekil) sınıf üzerinden erişim yapılmasını sağlarız. Client Real Subject'in yapması gereken bir işi yaptırmak için Proxy nesnesini kullanılır, Proxy nesnesi kendi içinde bu işlemi (isterse)belirli şartlar, düzenlemeler, kısıtlamalar altında yapar. Proxy ve Real Subject sınıfların ikisi de aynı interface'den(Subject olarak adlandırılır) türemiştir.

- **Virtual Proxy:** Ağır sistem kaynağı kullanan sınıflarda kullanılır. Uygulama başlar başlamaz; sınıfı oluşturmak yerine, sadece ihtiyaç olduğunda hedef sınıfı oluşturur.

- **Protection Proxy:** Client'ın, sadece belirli durumlarda sınıfı çalıştırması gerekiyorsa uygulanır.
- **Logging Proxy:** İstenilen belirli işlemlerden önce veya sonra, log göndermek için kullanır.
- **Caching Proxy:** Sürekli olarak gönderilen aynı tipte isteklerin aynı sonucu geri dönderdiği durumlar kullanılır.

Proxy yazılım kalıbında :

- Vekil sunucu belli yetkilendirmeleri kontrol ederek duruma göre Real Subject'in işi yapmasını sağlayabilir,
- Çok kaynak harcayan objelerin her defasında tekrar oluşturulması yerine bu nesneleri vekil eden Proxy'lerin oluşturulmasını sağlar. Bu sayede gereksiz kaynak kullanımı engellenmiş olur. Örneğin Web serviste tutulan ve aşırı kaynak harcayan bir nesneyi(Real subject), client tarafında bir başka sınıf(Proxy) temsil edebilir.

[https://bidb.itu.edu.tr/sevir-defteri/blog/2013/09/08/vekil-kal%C4%B1p-\(proxy-pattern\)](https://bidb.itu.edu.tr/sevir-defteri/blog/2013/09/08/vekil-kal%C4%B1p-(proxy-pattern))

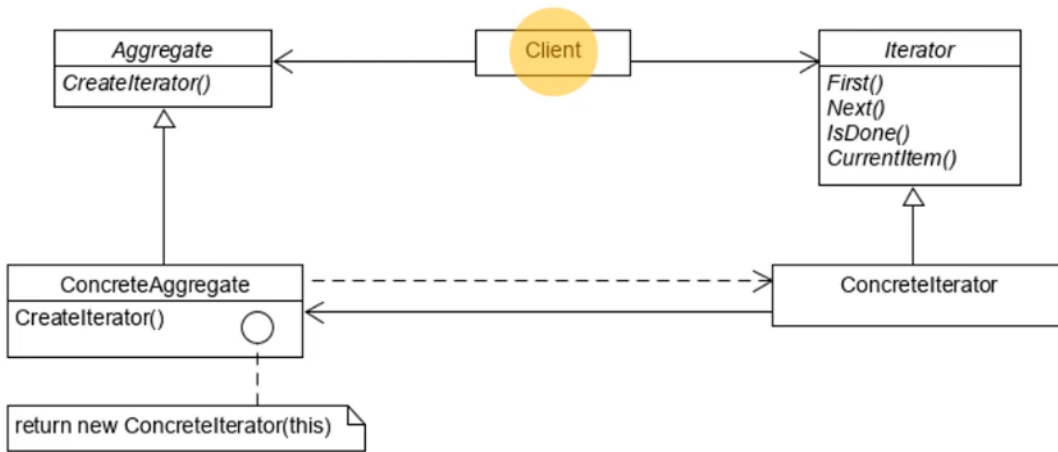
<https://www.turkayurkmez.com/proxy-design-pattern/>

<https://medium.com/@ibrahimozdogan/proxy-tasar%C4%B1m-deseni-251b4f69a8ab>

<https://www.gencayyildiz.com/blog/c-proxy-design-patternproxy-tasarim-deseni/>

10. Iterator Tasarım Kalıbı

Iterator tasarım kalıbı bir grup(koleksiyon,ağaç gibi veri yapılarında tutulan) veri içinde belli şartlara göre gezinmek veya belli şartlar içinde itere edilebilir veriler(mesela bir aralık içindeki tarihlere göre iteratif bir işlem yapılıyorsa) içinde gezinmek için kullanılır.



Aggregate : İtere edilmek istenilen Concrete Aggregate tüm sınıfların implement etmesi gereken arayüzdür. İçinde genelde sadece bir iterator oluşturup döndüren CreateIterator metodunun bildirimi bulunur.

ConcreteAggregate : Üstünde iteratif bir yapı kurulabilecek değerleri içeren sınıftır. Daha önce bahsedildiği üzere bu değerler koleksiyon türünde veya başka yollarla itere edilebilecek değerler olabilir. Dikkat edilmesi gereken bu sınıfların itere edilen asıl sınıflar olmadığıdır. Örneğin Urunler sınıf Concrete Aggregate sınıf olamaz. Fakat ürünler listesini koleksiyon olarak içeren bir sınıf Concrete Aggregate sınıfı olabilir. UrunlerAggregate isminde adlandırabiliriz.

Concrete Aggrate sınıfı implement ettiği CreateIterator metodu ile ona karşılık gelen ConcreteIterator sınıfından bir nesne döner. Örneğin UrunlerAggregate için UrunlerIterator adında bir nesne döner.

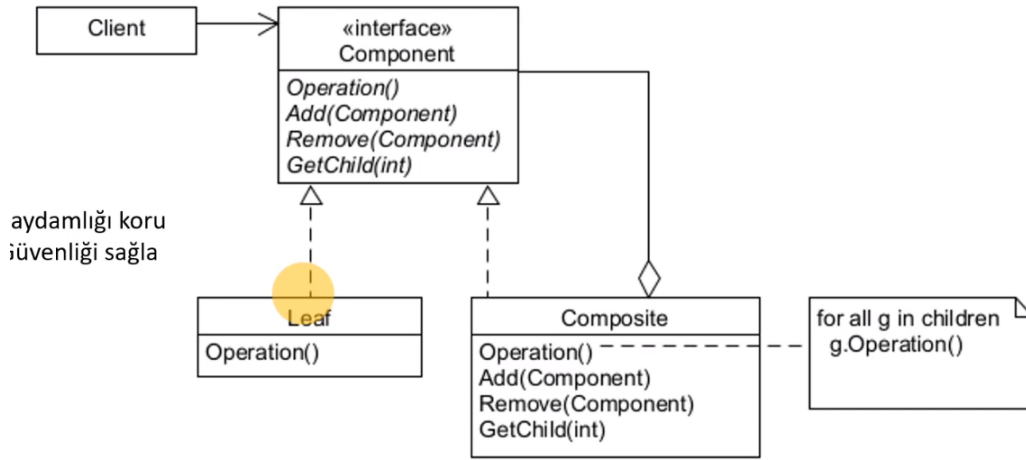
Concrete Aggrete içinde veri kümesi tutması ve bir iterator oluşturup bunu döndüren metodu olması yanında bu veri kümesi üstünde ekleme,silme, veriye erişme gibi işlemleri yapan metotlar da bulundurulur. Bu metotlarla vasıtasıyla ConcreteIterator iterasyon işlemlerini gerçekleştirir.

Iterator : Tüm iterator sınıflarının implement etmesi gereken arayüzdür. Biz iterasyon yaparken kullanılan veri yapılarına göre farklı işlemler yaparız. Buna rağmen bu arayüzdeki metotları veri yapısından bağımsız olarak tasarlamalıyız. Örneğin First metodu bir ağaç için ilk elemanı getirirken bir arraylist için de ilk elemanı getirir. Değişen tek şey arayüzdeki metotları implemente eden ConcreteIteratorler sınıfındaki metotların içeriğidir.

ConcreteIterator : İtere edeceğimiz veritipi için(kullanılan dilde hazır olarak tanımlanmış veya bizim kendi tanımladığımız veritipi olabilir) özel olarak yazılmış iterator sınıfıdır. Iterator arayüzünü uygulamak zorundadır.

Bu desen Composite ve Factory deseni ile bağlantılıdır.

11. Composite Tasarım Deseni

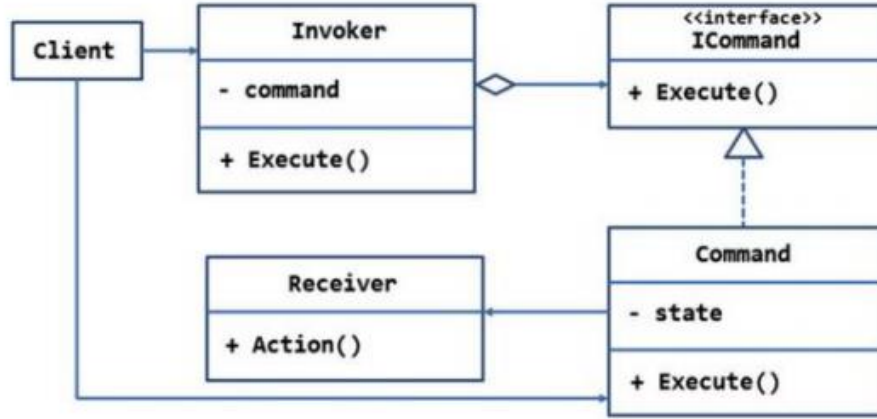


Composite yazılım deseni hiyerarşik/aęaç yapılarının bulunduęu durumlarda kullanılır. Örneęin; Çalışan(Component) sınıfından türeyen İşçi(Leaf) ve Yönetici(Composite) sınıfları; Asker(Component) sınıfından türeyen general/onbaşı(Composite), er(Leaf) gibi sınıflar.

Bu yazılım tasarım desende aęaç hiyerarşisinde bulunan tüm sınıflar Component interface veya abstract sınıfındaki Operation metodunu uygularlar. Farklı sınıflar bu metodun içerięi farklı olacak şekilde uygulayabilirler. Component interface(veya abstract class)'ında ayrıca Add, Remove ve GetChild gibi metotlar bulunur. Add metodu Composite sınıftaki koleksiyon tipindeki bir veri yapısına eleman eklemek için, remove buradan eleman silmek için, getChild ise bu Composite'in belli bir indeksteki çocuęuna erişmek için kullanılır.

Leaf sınıflar ise hiyerarşinin en altında kalan sınıflardır. Örneęin asker örneęinde bir erin hiyerarşik olarak altında başka bir sınıf yoktur. Dolayısıyla bu sınıfta hiyerarşik olarak alt sınıflara ait nesneleri tutan bir koleksiyon olmaması gerektięi gibi bu koleksiyonla çalışan Add,Remove,getChild metotları da bulunmamalıdır. Fakat Leaf'de Composite gibi bu metotların bulunduęu Component interface/abstract sınıfından kalıtım aldığı için mecburen bu metotları içinde bulundurur. Bu metotların içinde sadece bu metotların tanımlanmadıęına/desteklenmedięine dair hata fırlatan bir throw ifadesi olması bu tasarımı gerçekleştirmek için iyi bir yoldur.

12. Command Tasarım Deseni



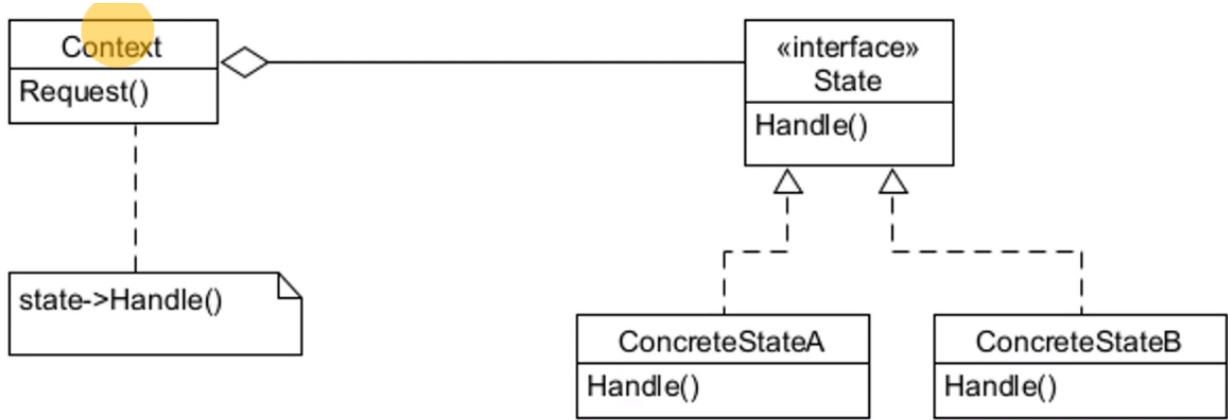
Command Tasarım Deseninde bir nesneye komutu gerçekleştirmek için bir sınıf tasarlarız. Bu komutu nesnenin kendi sınıfı içerisinde yapacak metodu yazmayız. Bu sayede SOLID prensiplerinin Open/Closed ilkesini uygulamış oluruz. Bu yazılım desenin faydaları:

- Uygulama daha modüler ve esnek olur.
- Composite deseni ile makro komutlar oluşturularak bu komutların da birlikte çalıştırılması sağlanabilir.
- Yeni işlevler istendiğinde, yeni komut nesneleri kolayca eklenebilir.
- İşlemi tetikleyen nesnelerle ile işlemi yapan nesneler birbirinden ayrılmış olur.

Command tasarım desenine gerçek hayattan örnek verecek olursak televizyon ve kumanda arasındaki ilişkiyi örnek verebiliriz. Televizyon bir nesnedir, biz kanal değiştirme işlemi direkt olarak televizyondan değil ona kumandayla komut göndererek yaparız. Burada client biz, ICommand kanal değiştirme dahil tüm(seç aç/kapa,menü aç vs.) komutların implement ettiği bir arayüz, Command somut tekil komut sınıfımız yani kanal değiştirme işlemi yapan sınıf, Receiver(Alıcı) televizyon, Invoker(çağırıcı) ise kumandadır.

Invoker bir grafiksel arayüz/menü olarak düşünülebilir. İçinde ICommand arayüzünü implement etmiş farklı komutları barındırır. TV-Kumanda örneğindeki kumandanın farklı komutları içeren tuşları olması gibi invokerında farklı komutları tutan referansları bulunabilir. Client bu düğmelerden birine bastığında yani invoker tipinden nesne ile bu metotları çağırdığında invoker ICommand arayüzünü implemente etmiş Command'leri çağırır. Bu Command'ler Receiver nesnesini kendi içerisinde barındırır ve istenilen işlemi gerçekleştirir.

13. State Tasarım Deseni



Bir sınıftan oluşturduğumuz nesneler farklı durumlarda farklı davranışlar sergiliyorsa bu tasarım desenini kullanmak faydalı olacaktır. Bu tasarım deseninde nesnenin farklı durumlarının her biri bir sınıf olarak tasarlanır. Bu durumların hepsi bir interface'i implement ederler. Bu interface'de nesnelerin farklı durumlarda yapabileceği işlemlerin tamamı bildirilmelidir. State tasarım deseni fine state makinelerinin yazılımsal uygulamasıdır. Bu desen uygulanmadan önce sistemin fine state diyagramının çizilmesi ardından bunun uml diyagramı çizilmesi ve bu 2 diyagramdan faydalanarak sistemin gerçekleştirilmesi yerinde olacaktır. Fine state makinesindeki her bir state UML sınıf diyagramında ConcreteState'leri ifade eder. Fine state makinesindeki oklar üstünde geçiş şartlarını sağlayan tüm koşullar ise ana interfaceimiz olan State interfaceindeki metot bildirimleri olarak karşımıza çıkmaktadır. Context ise tasarımını yaptığımız sistemdir. Örneğin kahve makinesi örneğindeki kahve Context'dir. Su yok, para girildi, bozuk para yok durumu gibi durumlar state; para ekle, su ekle gibi sistemin durumunu değiştirebilecek işlemler/koşullar ise State arayüzündeki bildirilen metotlardır.

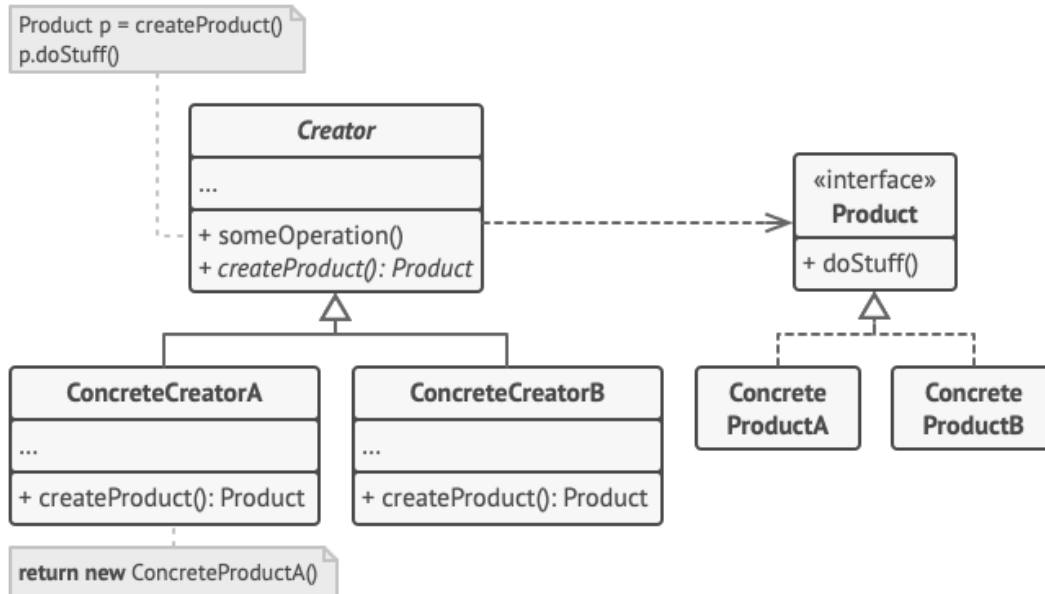
Her bir concrete state içinde Context'e ait referans barındırır ve bu referansa yapıcı fonksiyonuyla değer atar. Bu yapıcı metot Context sınıfının yapıcısı içinde çağrılır. Yani Context sınıfı kendi yapıcısında her bir durumu kendini referans vererek(this) oluşturur, bu oluşturduğu durum değerlerini kendi sınıfındaki üye değişkenlerle tutacak şekilde atar. Ayrıca Context sınıfı kendi yapıcısı içinde kendine bir başlangıç durumu belirler(fine state makinesindeki başlangıç durumu). Context bu başlangıç durumunu o anki durumunu tuttuğu State türünden referansa atar.

Context sınıfı ayrıca State arayüzündeki tüm metotları kendi bildirir ve tanımlar. Context sınıfı bu metotlar içinde o anki durumunu belirten referans üstünden metota karşılık gelen durum metodunu çağırarak şekilde düzenler.

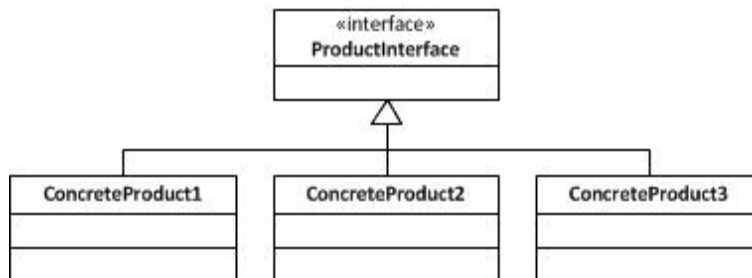
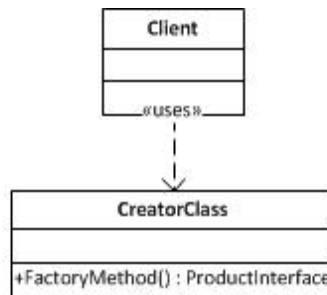
ConcreteState sınıfları ise arayüzdeki metotları uygularken kendi durumunu ve şartları göz önünde bulundurarak işlemler gerçekleştirir ve eğer fine state makinesinde karşılık gelen bir

durum geçişi varsa içinde tuttuđu Context referansını kullanarak Context nesnesinin durumunu deđiřtirir.

14. Factory Method Pattern



Kullanım 1



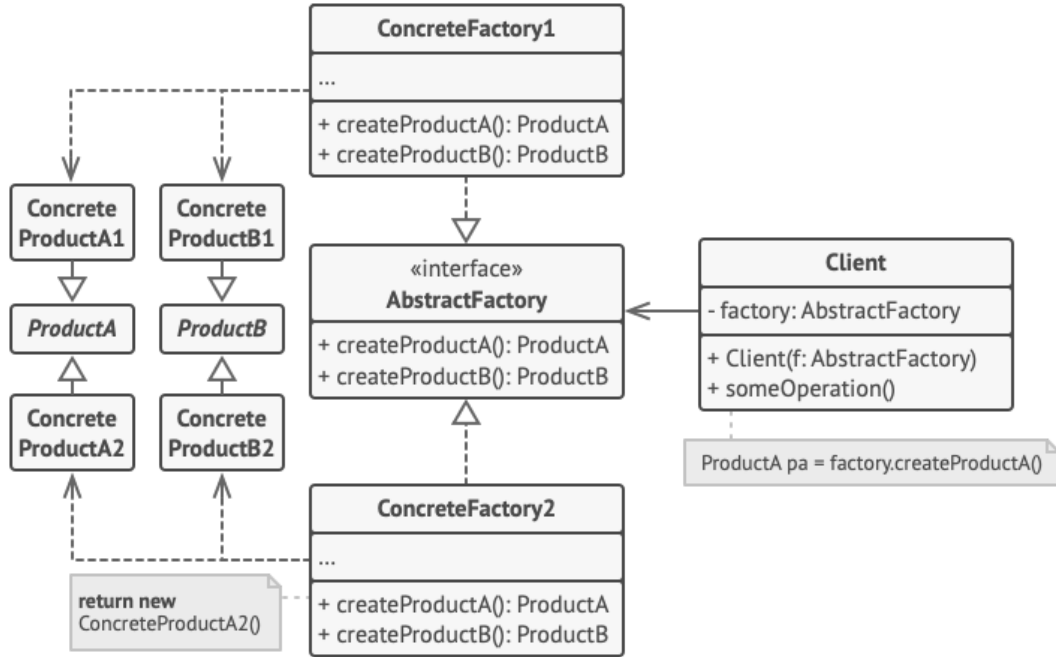
Kullanım 2

Factory method kalıbında birbiriyle kalıtım ilişkisi olan(üst sınıfları, arayüzleri aynı olan) sınıflarda nesne oluşturulma işlemi Factory isminde bir sınıfa devredilir : Kullanım 1. Bir diğer kullanım da ise bu kalıtım ilişkisi bulunan her bir sınıf ayrı bir Factory sınıfı tanımlamaktır. Bu ayrık Factorylerin hepsi aynı arayüzden veya abstract classtan kalıtım almaktadır : Kullanım 2.

Nesne ve creational(oluşumsal) bir tasarım kalıbıdır.

Birden fazla platform için yazılım geliştiriliyorsa kullanılabilir. Örneğin Windows için başka, Ubuntu için başka bir nesne oluşturulması otomatize edilebilir.

15. Abstract Factory Method Pattern



Nesne yaratımsal bir tasarım kalıbıdır. Abstract Factory Method’unda olduğu gibi yine burada da aynı interface’i(veya class/abstract class) implement ederek gruplanmış ürenler ve bu ürünler karşılık gelen yine bir interface/abstractClass’tan kalıtım alarak gruplanmış factoryler bulunmaktadır. Abstract Factory’i, Factory Method’tan ayıran şeyleri yukarıdaki UML sınıf diyagramı üzerinden inceleyecek olursak:

- ProductA ve ProductB adında gruplayıcı bir sınıf/abstract sınıf veya arayüzümüz bulunmaktadır.
- ProductA kapsayıcı yapısından türeyen bir sınıflar vardır : ProductA1, ProductA2... Aynı türeme işlemi ProductB için de geçerlidir. Buradaki önemli nokta ProductA ve ProductB’den türeyen bu sınıfların birbiriyle karşılıklı olarak ilişkili olmasıdır. UML diyagramına bakınca ProductA1 ve ProductB1’i ilişkilendiren bir yapı olmasa da(Factoryleri göz ardı edelim, onların amacı zaten bu ilişkiyi kurmak) biz tasarımı yaparken bunlar arasında bir ilişki olduğunu görebiliriz. Hatta isimlendirme yaparken de bu ilişkiye göre isimlendirme yapmamız iyi olacaktır.

- Factory method'ta olduğu gibi her bir ürünü oluşturmak için concrete factorylerimiz bulunmaktadır. Fakat bu concrete factoryler kapsayıcı sınıf/arayüz olan ProductA, ProductB'yi oluşturmak için değil de bunların çocukları olan ve aralarında bir ilişki olduğunu bildiğimiz ProductX1, ProductX2 ürünleridir. İşte abstract factory'inin, factory'den farkı budur. Concrete factorylerimiz bu ürünleri oluşturmakla sorumludur. Burada bir parça-bütün ilişkisinden bahsedebiliriz. ProductA1-ProductB1 parçaları birleşerek Product1'i, ProductA2-ProductB2 birleşerek Product2'yi oluşturur. Dolayısıyla Concrete Factorylerimiz de bu Product1 ve Product2'yi oluşturmak için tasarlanmıştır. Bundan ötürü de base AbstractFactory arayüzümüzde ProductA ve ProductB'yi oluşturmak için metotlar bulunur.
- Concrete Factory'deki parça sayısı kadar üretme fonksiyonunu kullanarak üretme işlemini gerçekleştirmeyi bir Application sınıfı tasarlayarak ona devredebiliriz yada direkt olarak Client'te yaparız.

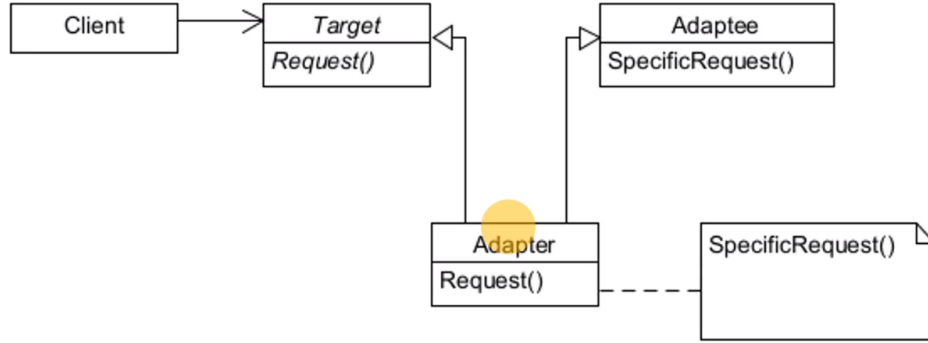
AbstractFactory' buradaki UML Sınıf diyagramını da göze alarak bir örneklendirme yapacak olursak :

- ProductA : SQL Connection sınıfı
- ProductA1 : MS-SQL için connection sınıfı, ProductA2 : PostgreSQL için connection sınıfı
- ProductB : SQL Command(Komut) sınıfı
- ProductB1 : MS-SQL için command sınıfı, ProductB2 : PostgreSQL için command sınıfı
- ConcreteFactory1 : MS-SQL veritabanı için gerekli parçaları oluşturan factory
- ConcreteFactory2 : PostgreSQL veritabanı için gerekli parçaları oluşturan factory

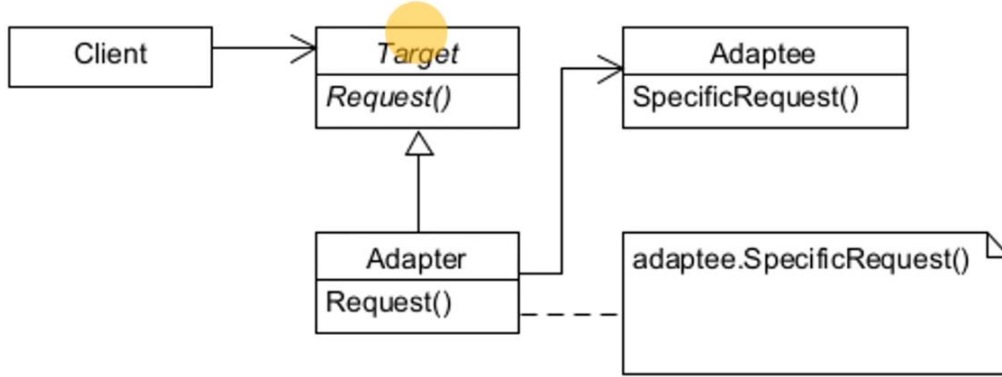
Görüldüğü üzere ProductA1 ve ProductB1 arasında hiçbir ilişki UML sınıf diyagramında yok. Fakat biz bunların birbiriyle alakalı, aynı bütün parçası olduğunu biliyoruz ve bu parçaları efektif bir şekilde oluşturmak için Abstract Factory tasarım desenini kullanıyoruz. Bu sayede istemci üretim aşamaları hakkında hiçbir fikri olmadan bu ürünleri oluşturup kullanılıyor yani soyutlama sağlanmış oluyor.

Bu tasarım deseninde yeni var olan yeni bir ürün gruba eleman ekleyerek genişletme yapmak nispeten kolay olsa da yeni bir ürün grubu ekleyince kodda fazlaca değişiklik yapılması gerekecektir. Bu tasarım dezavantajı olarak söylenebilir.

16. Adapter Design Pattern



Çoklu kalıtım ile

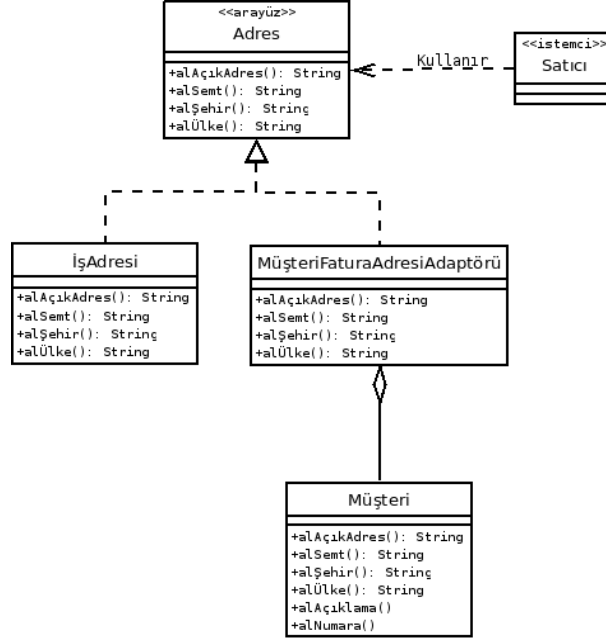


Composition ile

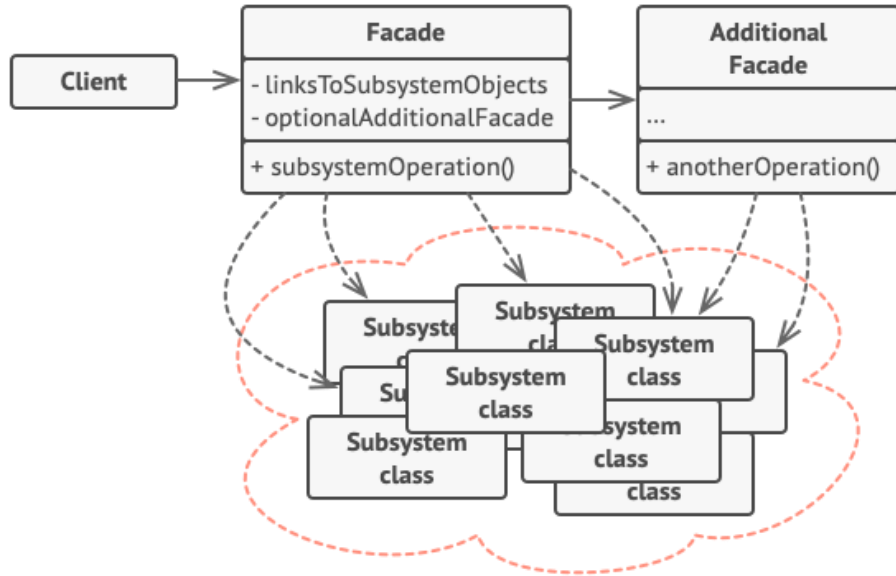
Elimizde bir arayüzü uygulayarak veya sınıftan kalıtım alarak gruplanmış bir veya birden fazla sınıf olsun. Daha sonrasında elimize üzerinde değişiklik yapmak istemediğimiz/yapamadığımız bir başka sınıf geçsin. Bu yeni sınıfı daha önceki gruba dahil etmek yani onun base sınıfı/interfaceinden kalıtım aldırarak istiyoruz. Bunu direkt olarak yapmamız çoğu zaman mümkün olmayacaktır : yeni gelen sınıfta değişiklik yapmadığımız gibi direkt olarak bu sınıfın aynı arayüzden/sınıftan kalıtım alması elimizdeki sınıfın üstüne eklemeler yapmamız gerektirecektir ve bu sınıfın ana yapısını bozmamızı gerektirecektir. Onun yerine bu yeni sınıfımızı diğer sınıflarla aynı gruba koyabilmek için bir başka sınıf daha oluşturuz. Bu sınıf **Adapter** sınıfı olarak da adlandırılır. **Adaptee** ise bizim adapte etmek istediğimiz sınıftır. Adapter sınıfı gruplamayı sağlayan interface/sınıftan kalıtım alarak onlarla aynı grup içine girer. Fakat bu haliyle Adaptee yani adapte edilmesi gereken asıl sınıfımız henüz gruplamaya hiçbir şekilde dahil edilmemiştir. Bu dahil etme işlemini gerçekleştirmek için 2 seçenek vardır : Çoklu kalıtım destekleniyorsa Adapter sınıfının Adaptee sınıfından da kalıtım alması; Adapter sınıfı içinde Adaptee sınıfına ait bir referansı içinde barındırması(composition). Adapter sınıfı çoklu kalıtım ile oluşturulmuşsa Adaptee sınıfındaki metotları/değişkenleri kullanarak ana gruplandırmayı yapan interface/sınıftaki metotları implemente

eder. Composition kullanılmışsa Adapter sınıfı Adaptee sınıfına ait referansı üzerinden gruplandırma yapan sınıf/interface'e ait metotları implemente eder yada override eder.

Bu tasarım kalıbında yeni bir sınıfın tüm özellikleri varolan gruba göre düzenleneceği gibi yeni sınıftaki özelliklerin sadece küçük bir kısmı kullanılarak varolan gruba adaptasyon yapılabilir :



17. Facade Design Pattern



Bu yazılım kalıbında istemcinin/istemcilerin sistemdeki alt sınıflara direkt olarak erişip karmaşık işlemleri gerçekleştirmesi yerine bu karmaşık işlemleri gerçekleştiren bir sınıf : **Facade** sınıfı tasarlanır. Bu sınıf kendi içindeki metotlarla sistemdeki alt sınıflara erişerek bu karmaşık işlemleri gerçekleştirir. Client'in ise bu alt sınıflara erişmesi engellenerek daha soyut bir tasarım yapılması sağlanabilir. Face sınıfları zincir şeklinde de olabilir.

