Creational Design Pattern'lar incelenmelidir. Örneklerle anlatınız.

Creational Design Patterns, esnek bir yapı kullanarak daha önce belirtilen durumlara uygun nesneler yaratır. Creational Design Patterns ile sistemin uygun nesneyi çağırması sağlanır. Hangi nesnelerin oluşacağı ve bu nesnelerin nasıl oluşturulacağına göre beş farklı pattern vardır.

Factory Method: Factory method, bir superclass'ta(üst sınıf) nesne oluşturabilmek için bir interface sağlar. Sisteme esneklik ve yapısal olarak sadelik ve daha az karmaşa vaat eden tarafı, subclassların(alt sınıf) yaratılan nesnelerin türünü değiştirmesine olanak sağlamasıdır.

Örneğin yalnızca ufak bir şehirde ve bu şehrin çevresinde, satılan çeşitli ürünlerin taşınması için tasarlanacak olan bir uygulama yazma işinde olalım ve bölgemiz ufak olduğundan, mallarımızı ilk başta yalnızca kamyonetlerle taşıyalım. İlk olarak işimizin hemen hemen hepsini sadece "kamyonet" adlı bir sınıfın içinde görürüz çünkü ihtiyacımız olan tek obje kamyonet objesidir.

Diyelim ki uygulamamız çok büyük ilgi gördü ve bağlı çalıştığımız firmaya uzak, hatta deniz ötesi veya sadece havayoluyla ulaşabileceğimiz yerlerden talepler gelmeye başladı. Firma da bize uygulamayı bu yönde geliştirecek gerekli kodları yazma görevi verdi. Eğer factory method bilmiyorsak yapacağımız şey, talep gelen ve uygulamaya eklememiz gereken, örneğin deniz yolu için, bütün kodumuzu değiştirmektir. Hatta ve hatta, daha sonra havayolu için de aynı işlemleri yapmaktır. Günün sonunda ise araç nesnelerinin ait olduğu sınıfa bağlı olarak uygulamanın yapacağı işi değiştiren oldukça karmaşık bir koda sahip oluruz.

Eğer factory method biliyorsak yapacağımı şey, "new" keyword'ü ile yaptığımız nesne yaratma işlemini bir factory method ile değiştirmektir. Factory method tarafından yaratılan objeler çoğunlukla "products" olarak anılır.

Bu yöntemi standart obje yaratımından ayıran durum ise bu factory method'u alt sınıflarda override edip factory method tarafından yaratılan objelerin ya da products'ın sınıfını değiştirebilmemizdir. Yani bu şekilde, factory method'u override ederek yarattığı objelerden hem başta elimizde olan kamyonet sınıfı objeleri, hem de daha sonra bize verilen görev neticesinde ihtiyacımız olan gemi ve uçak sınıfı objelerini elde edebiliriz. Yani factory method içinde yaratılan objeleri, ihtiyacımız olan yerde kamyonete, ihtiyacımız olan yerde ise gemi ya da uçağa çevirebiliriz. Bu da projemizi, taşıta kodun davranışının değişeceği bir dolu koşuldan ve karmaşıklıktan kurtarır.

Abstract Factory: Tüm metotları için implementasyonu bulunan sınıflara somut sınıf denir.

```
// This is an interface
interface X {
    int product(int a, int b);
    int sum(int a, int b);
}

// This is an abstract class
abstract class Product implements X {

    // this method calculates
    // product of two numbers
    public int product(int a, int b)
    {
        return a * b;
    }
}

// This is a concrete class that implements
class Main extends Product {

    // this method calculates
    // sum of two numbers
    public int sum(int a, int b)
    {
        return a + b;
    }

    // main method
    public static void main(String args[])
    {
        Main ob = new Main();
        int p = ob.product(5, 10);
        int s = ob.sum(5, 10);

        // print product
        System.out.println("Product: " + p);

        // print sum
        System.out.println("Sum: " + s);
    }
}
```

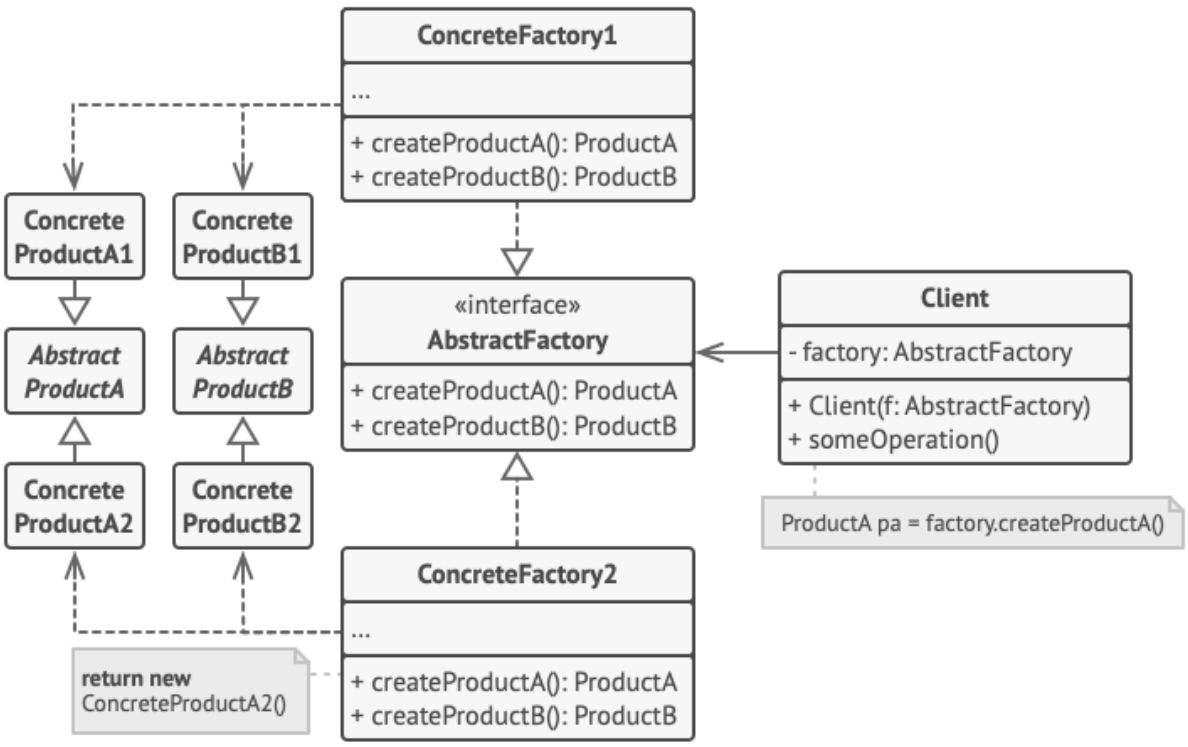
Örneğin yukarıdaki kodda, X interface'ini implement eden Product sınıfında "product" metodu için bir implementasyon varken "sum" metodu için yoktur. Dolayısıyla bu, bir abstract(soyut) sınıftır. Öte yandan Product sınıfını extend eden Main sınıfında "sum" metodu için bir implementasyon vardır ve sonuçta bu sınıf bir concrete(somut) sınıftır.

Şimdi konumuza dönersek, abstract factory, birbiriyle ilişkili objelerin ailelerini bunların somut sınıflarını belirtmeden yaratmamızı sağlar. Örneğin, bir mobilya mağazası uygulaması yapıyoruz ve elimizde birbiriyle ilişkili olan sandalye ve masa ürünlerini içeren bir ürün ailesi var. Ayrıca bu ailenin de, örneğin modern ve klasik olmak üzere çeşitli varyasyonları var. Mobilya objelerinin ailenin diğer elemanları ile eşleşmesi lazım. Yoksa modern sandalyelerin yanına gelen klasik masa gibi uyumsuzluklarla karşılaşabiliriz.

Abstract Factory Paattern ile ilk olarak, doğal olarak, ailedeki her ürün çeşidi için bir interface kurmalıyız ve yine tabi ki doğal olarak, ailenin her ürünün varyasyonları kendi ilgili interfacelerini esas almalıdır. Örneğin, modern ve klasik sandalyeler sandalye ürünü için yazılan interface'i takip etmeli. Şimdi ise ürünleri oluşturacağımız Abstract Factory interface'ini oluşturabiliriz. Ailenin tüm ürün çeşitlerinin yaratılacağı metotlar burada bulunmalıdır(createChair, createTable vb). Bu metotlar, şimdilik abstract olan ve daha somutlaştırılacak objeler döner.

Elimizde şimdilik olduğunu varsaydığımız ama aynı zamanda bunların sayısını da artırabileceğimiz varyasyonlar için de Abstract Factory interface'inden gelecek abstract özellikteki ürünleri alacak Factory Classları yaratalım. Bu Factory Classlar sadece ve sadece ilişkili olduğu ürünü döndürür. Örneğin modern ürünler için yazılan Factory Class yalnızca modern sandalye ve modern masa verir.

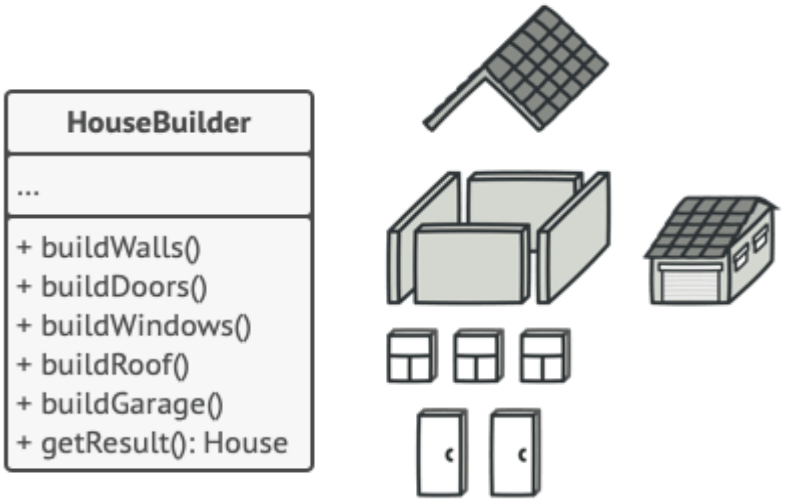
İstemci yazılım, factory ve ürünlerle ilgili istekte bulunurken bu abstract interface'i kullanırsa yukarıdaki kodu değiştirmeden gelen talebe göre factory'yi ve ürün varyasyonunu değiştirebiliriz. Diyelim ki istemciden sandalye objesi üretecek bir factory için talep geldi. İstemcinin factory'nin çeşidi ya da ürünün varyasyonu hakkında bilgi sahibi olması gerekmez. O sadece, her varyasyonu aynı görerek sandalye için olan abstract interface'ten bir şekilde implemente edilen sandalye metotlarını bilir. Böylece bu kodu değiştirmeye gerek kalmadan, uygun Factory Class ile işleme devam edebiliriz. Ayrıca, bu şekilde, ortak Factory Class'tan dönen sandayenin varyasyonu neyse masanınki de o olacaktır.



Builder: Builder, aynı construction'ı kullanan objelerin farklı ve özelleştirilmiş türlerini oluşturmamıza imkan tanır. Elimizde bir objenin en ham ve özelliksiz halinin olduğunu varsayalım. Builder Pattern ile ihtiyaca göre bu objeye geliştirmeler verebiliriz.

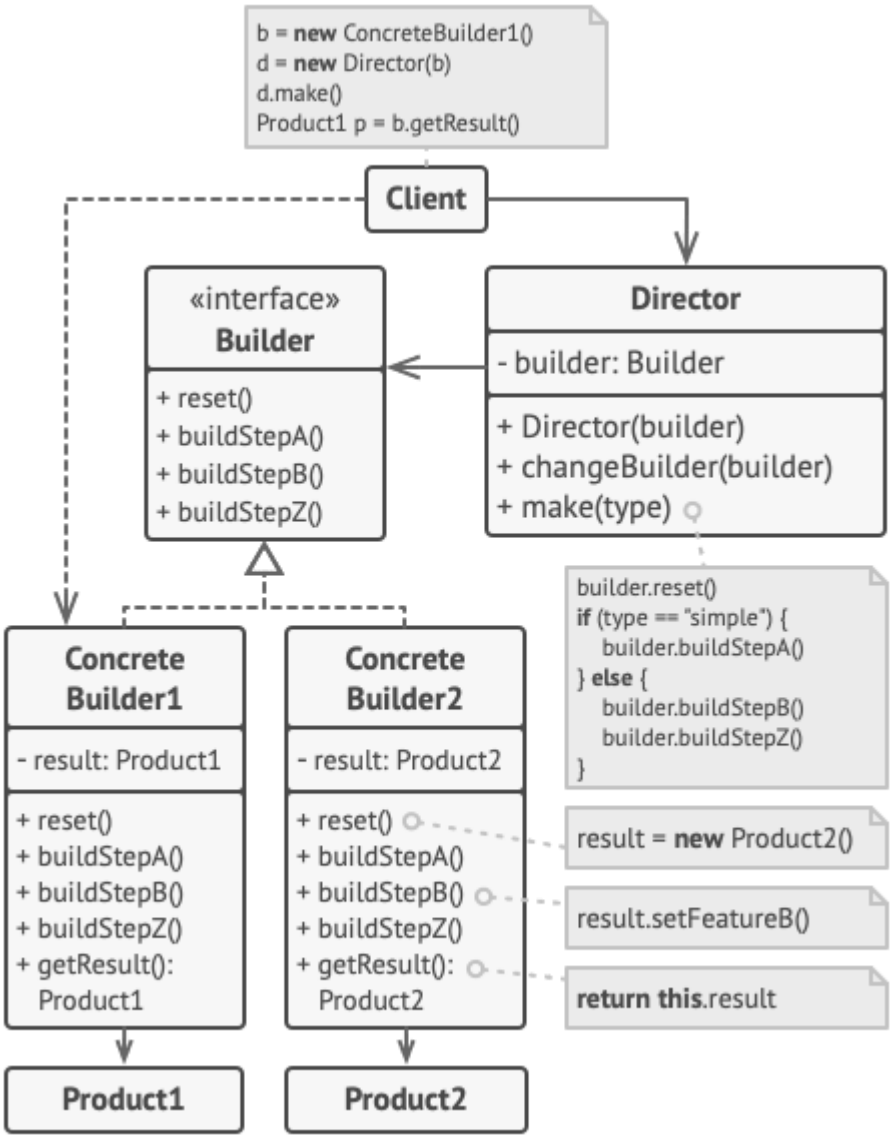
Örneğin elimizde bir ev objesi olsun. Bu ev de en ham durumda olsun. Peki, bu ham evden özelleşmiş, farklı özellikleri olan evleri nasıl oluşturabiliriz? Mesela bahçesi olan bir ev ya da garajı olan bir ev ya da spor alanı olan bir ev. Builder Pattern kullanmadan, katmak istediğimiz çeşitli özellikler için alt sınıflar oluşturup bunları kombine edebiliriz, ancak alt sınıf sayısını ve sınıfların birbirleriyle ilişkilerini nasıl kontrol edeceğiz? Öte yandan, yine Builder Pattern kullanmadan, istediğimiz tüm özellikleri dev bir cosntructor'a koyabiliriz ama bu sefer de obje oluştururken constructor'ın birçok parametresi boş kalacak ve kodumuz oldukça kötü gözükecek.

İşte Builder Pattern burada devreye giriyor. Builder Pattern'de yapılan işlem basitçe objenin constructor'ını "builders" denen objeye taşımaktır. Burada evin ham özelliklerinin yanı sıra "buildGarden", "buildeGarage" ve benzeri özellikler bulunur.



Tek yapmamız gereken, bu özelliklerden gerekli olanları çalıştırıp evimizde istediğimiz kombinasyonu sağlamaktır. Tüm özellikleri tek tek çağırmamıza gerek yoktur.

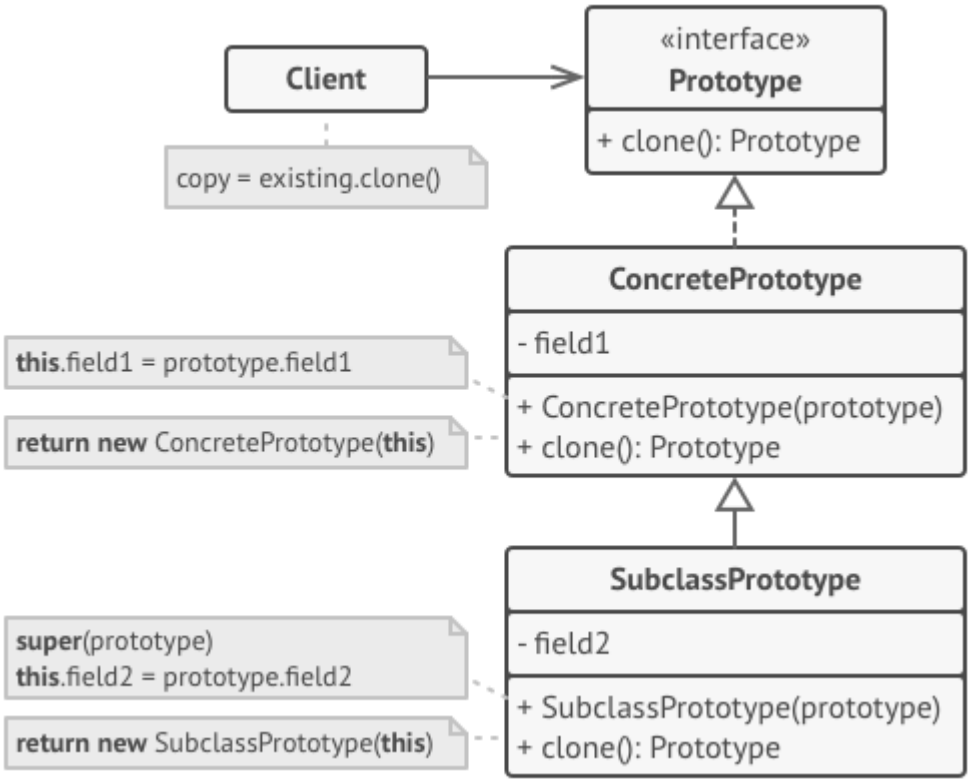
Ek olarak, yukarıdaki özelliklerden bazıları da kendi içlerinde özelliklere ayrılabilir. Örneğin gövdenin ham maddesinin ahşap mı taş mı olacağı gibi. Bu durumda, yukarıdaki özellikleri farklı şekillerde, burada kullanılan hammaddeyi değiştirerek, kapsayana farklı builderlar yazmamız gerekir.



Prototype: Prototype, sınıflara bağlı olmadan objenin kopyasını kodlamamıza imkan verir. Bu işlemi, aynı sınıftan yeni bir obje oluşturup ilk objemizin bütün parametrelerini yeni objeye kopyalayarak yapabiliriz. Ancak ilk objenin alanları "private" olabilir ya da obje dışında bir yerden erişilemiyor olabilir. Ayrıca bu durumda ilk objenin sınıfını bilmemiz gerektiğinden sınıflara bağlı hale geliriz.

Bu durumda Protoype kullanmalıyız. Prototype, kopyalamaya uygun her obje için ortak bir interface gösterir. Bu, genellikle yalnızca "clone" metodunu içeren, interface sayesinde objeleri, kodu tekrarlamadan kopyalayabiliriz.

"clone" metdou oldukça basit ve çoğu durumda aynıdır. Metot mevcut sınıftan bir tane obje üretir ve ilk objenin tüm alan değerlerini yenisine taşır. Buna "private" olanlar da dahildir. Çünkü birçok dilde aynı sınıftan yaratılan iki objenin birbirlerinin "private" alanlarına erişim imkanı vardır.

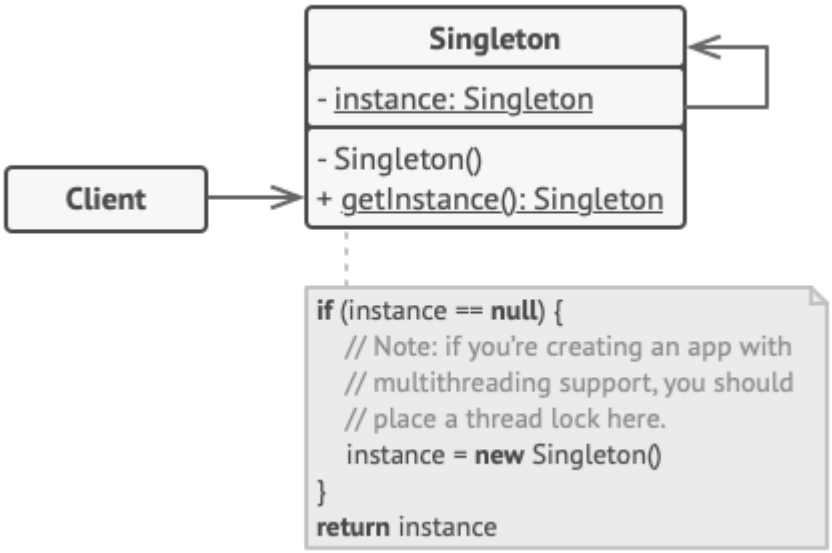


Singleton: Singleton, bir sınıfın yalnızca bir "instance"a sahip olmasını sağlar ve bunu yaparken bu instance'a global bir erişim verir.

Singleton ile sınıfın yalnızca bir instance'ı olmasını sağlayarak veritabanı gibi yapılara erişimi kontrol edebiliriz. Constructor ile obje oluşturduğumuzda, her zaman yeni bir obje elde ederiz ancak yeni bir obje istediğimizde ve Singleton kullandığımızda daha önce oluşturduğumuz objeyi alırız.

Global erişim durumunda ise global değerler kullanışlı olsa da güvenli değildir. Global erişimin mevcut olduğu durumlarda, başkaları gelip bu değerleri ya da alanları değiştirebilir ve kodu bozabilir. Singleton da aslında bazı objelere global, yani projenin her yerinden erişim olanağı sağlar ama aynı zamanda bunların değiştirilmesini de engeller.

Singleton uygulaması genelde bütün durumlarda benzerdir. Öncelikle standart constructor "private" yapılır ve objelerin Singleton ile birlikte bir de "new" keyword'ünü kullanması engellenir. Daha sonra "static" bir yaratma metodu yazılır. Constructor gibi davranan bu metot aslında "private" yapılan constructor'u çağırır ve objeyi yaratarak bunu static alanda tutar. Bu metodu çağırdığımızda artık tutulan bu obje döner.


Singleton
- instance: Singleton
- Singleton()
+ getInstance(): Singleton
Client
if (instance == null) {
// Note: if you're creating an app with
// multithreading support, you should
// place a thread lock here.
instance = new Singleton()
}
return instance