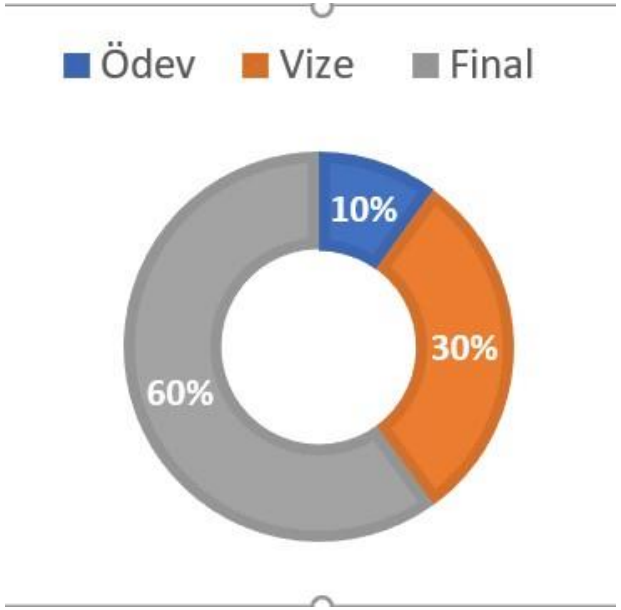
Yazılım Mühendisliği
1906003082015

Dr. Öğr. Üy. Önder EYECİOĞLU
Bilgisayar Mühendisliği

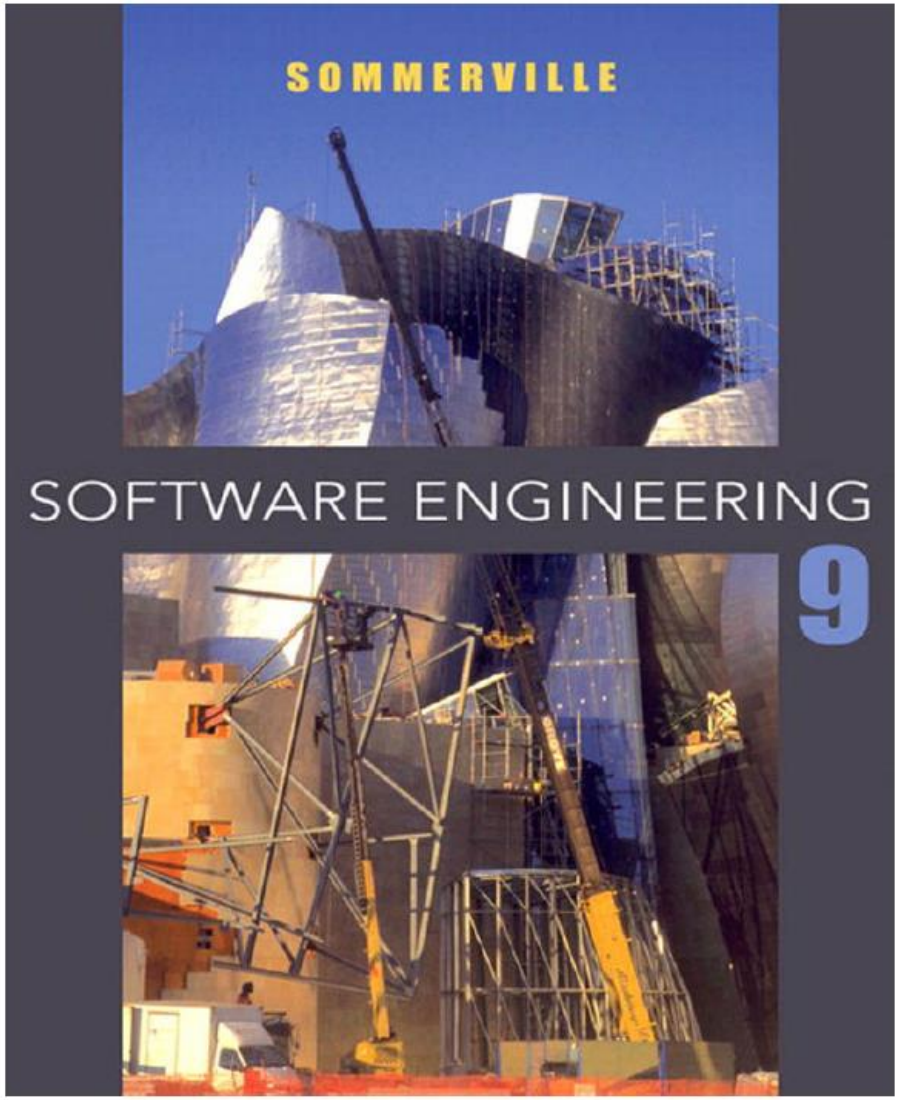
Giriş

Ders Günü ve Saati:
Salı: 09:15-13:00
Devam zorunluluğu %70



HAFTA	KONULAR
Hafta 1	Yazılım Mühendisliğine Giriş
Hafta 2	Yazılım Geliştirme Süreç Modelleri
Hafta 3	Yazılım Gereksinim Mühendisliği
Hafta 4	Yazılım Mimarisi
Hafta 5	Nesneye Yönelik Analiz ve Tasarım
Hafta 6	Laboratuar Çalışması: UML Modelleme Araçları
Hafta 7	Yazılım Test Teknikleri
Hafta 8	Ara Sınav
Hafta 9	Yazılım Kalite Yönetimi
Hafta 10	Yazılım Bakımı - Yeniden Kullanımı ve Konfigürasyon Yönetimi
Hafta 11	Yazılım Proje Yönetimi (Yazılım Ölçümü ve Yazılım Proje Maliyet Tahmin Yöntemleri)
Hafta 12	Yazılım Proje Yönetimi (Yazılım Risk Yönetimi)
Hafta 13	Çevik Yazılım Geliştirme Süreç Modelleri
Hafta 14	Yazılım Süreci İyileştirme, Yeterlilik Modeli (CMM)

Kaynaklar



13.
Tasarım Desenleri

Strategy (Policy) Patterns-Behavioral (Davranışsal)

GoF Tanımı: Strategy tasarım deseni, bir algoritma ailesi tanımlamamızı, her birini ayrı bir sınıfa koymamızı, her birinin kapsüllenmesini ve nesnelerinin birbiriyle değiştirilebilir hale getirmenizi sağlayan davranışsal bir tasarım modelidir.

Bir algoritmanın davranışını çalışma zamanında dinamik olarak seçebiliriz.

Birbirinin yerine geçen işlevleri biraraya getirir ve delagasyon kullanarak hangisinin kullanılacağına çalışma zamanında karar verilmesini sağlar. Bu örüntü kullanılarak bir işlevin farklı gerçekleştirimleri veya farklı algoritmaları isteğe bağlı olarak uygulanabilir.

Önemli olan nokta, bu uygulamaların birbirinin yerine geçebilmesidir - göreve bağlı olarak, uygulama iş akışını bozmadan bir uygulama seçilebilir..

Konsept

Strateji modeli, bir algoritmayı ana bilgisayar sınıfından kaldırmayı ve aynı programlama bağlamında, çalışma zamanında seçilebilecek farklı algoritmalar (yani stratejiler) olabilmesi için onu ayrı bir sınıfa koymayı içerir.

Strategy (Policy) Patterns-Behavioral (Davranışsal)

Konsept

Strateji modeli , bir istemci kodunun ilgili ancak farklı algoritmalar ailesinden seçim yapmasını sağlar ve istemci bağlamına bağlı olarak çalışma zamanında herhangi bir algoritmayı seçmesi için basit bir yol sağlar.

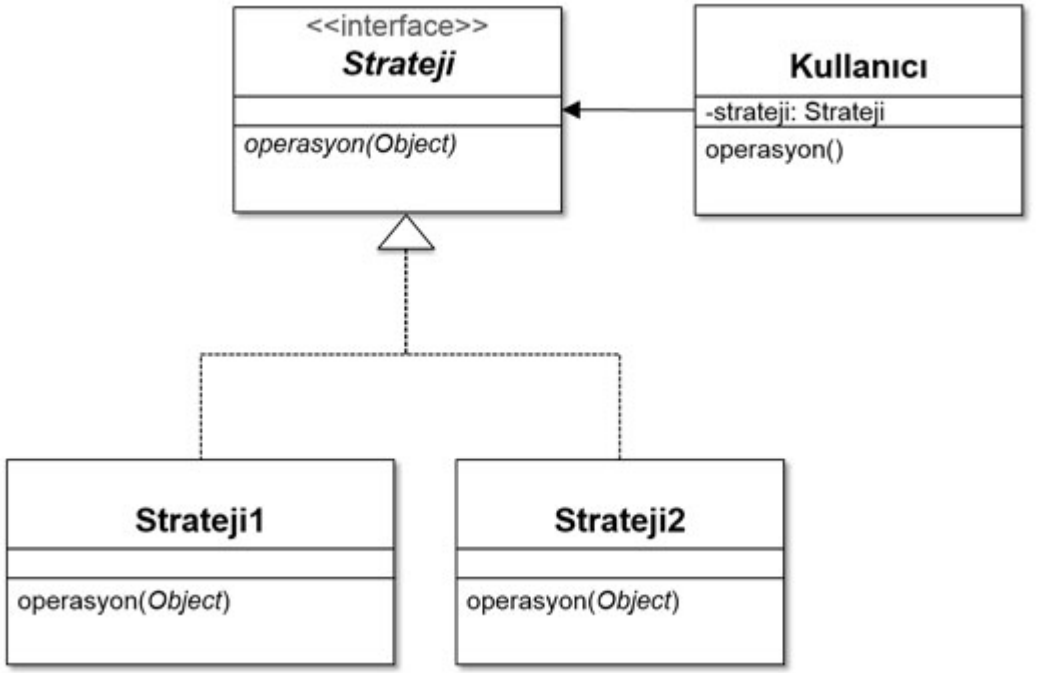
Açık/kapalı Prensibi ile Hareket Eder

Bu model Açık/kapalı ilkesine dayanmaktadır . [Değişiklik için kapalı] bağlamı değiştirmemize gerek yok, ancak herhangi bir uygulamayı [uzantıya açık] seçip ekleyebiliriz.

Örneğin, Collections.sort()– farklı sıralama sonuçları elde etmek için sıralama yöntemini değiştirmemize gerek yoktur. Çalışma zamanında sadece farklı karşılaştırıcılar sağlayabiliriz.

Strategy Patterns-Behavioral (Davranışsal)

Strateji örüntüsünü gösteren sınıf diyagramı : Burada Kullanıcı sınıfı, aynı işlevi (operasyon) farklı stratejileri uygulayarak yerine getirir. Farklı stratejileri gerçekleştirmek için Strateji türünde bir alan tutar (strateji) ve bu sınıftan oluşturulan nesnelere, atanan farklı stratejileri (aynı anda yalnızca bir strateji) uygulayabilir.



Strategy Patterns-Behavioral (Davranışsal)

Bilgisayar Dünyası Örneği

•Facebook, Google Plus, Twitter ve Orkut gibi dört sosyal platformda (örneğin sake) arkadaşlarımla bağlantı kurmamı sağlayan bir sosyal medya uygulaması tasarlamak istiyorum. Şimdi, müşterinin arkadaşının adını ve istenen platformu söyleyebilmesini istiyorum - o zaman uygulamam ona şeffaf bir şekilde bağlanmalıdır.

•Daha da önemlisi, uygulamaya daha fazla sosyal platform eklemek istersem, uygulama kodu tasarımı bozmadan buna uyum sağlamalıdır..

Strategy Patterns-Behavioral (Davranışsal)

İllüstrasyon

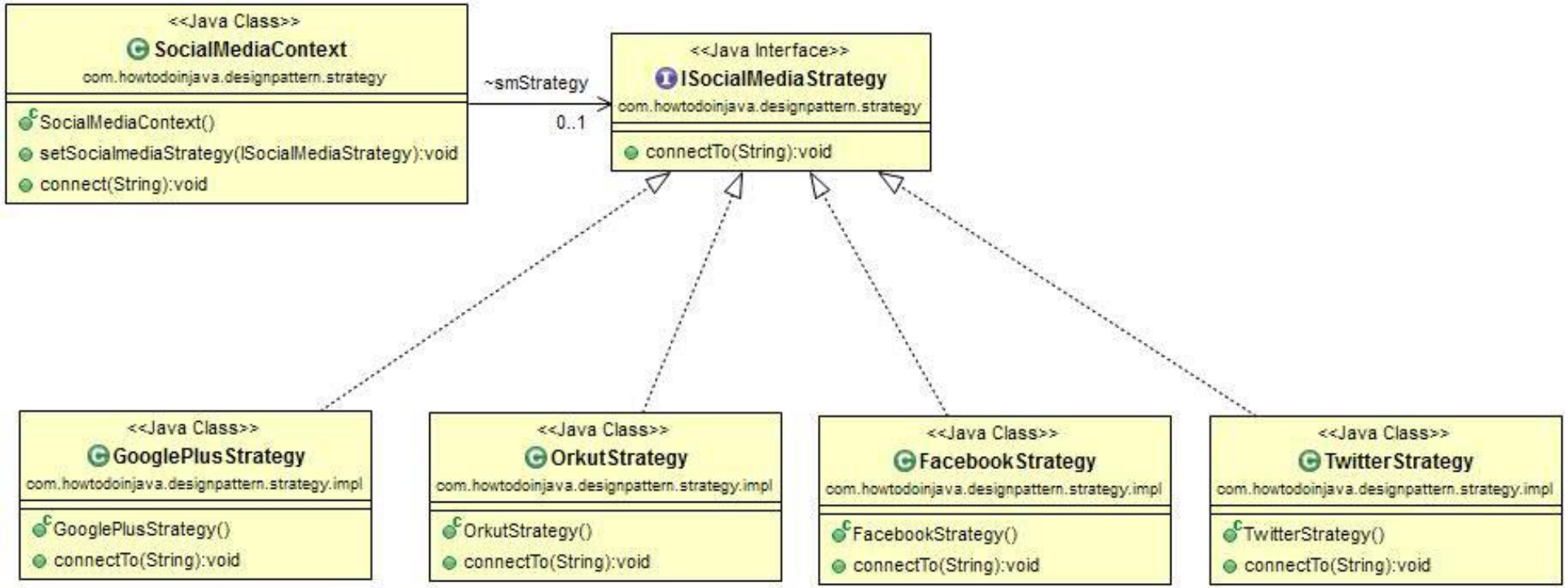
Yukarıdaki problemde, birden çok yolla (arkadaşla bağlantı kur) yapılabilecek bir işlemimiz var ve kullanıcı çalışma zamanında istediği yolu seçebiliyor. Bu yüzden strateji tasarım modeli için iyi bir adaydır.

Çözümü uygulamak için her seferinde bir katılımcı tasarlayalım.

- **ISocialMediaStrategy** – İşlemi özetleyen arayüz.
- **SocialMediaContext** – Uygulamayı belirleyen bağlam.
- **Uygulamalar** – Çeşitli uygulamalar: ISocialMediaStrategy. Örneğin FacebookStrategy, GooglePlusStrategy TwitterStrategy, OrkutStrategy.

Strategy Patterns-Behavioral (Davranışsal)

İllüstrasyon

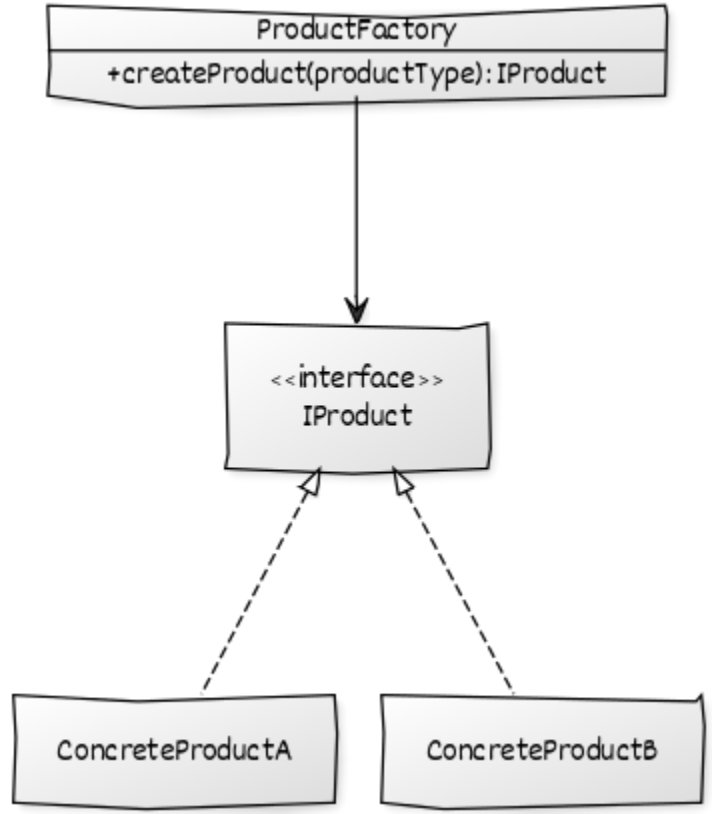


Factory Patterns-Creational (Oluşturucu)

GoF Tanımı: Factory tasarım deseni birbirleri ile ilişkili nesneleri oluşturmak için bir arayüz sağlar ve alt sınıfların hangi sınıfın örneğini oluşturacağına olanak sağlar..

Konsept

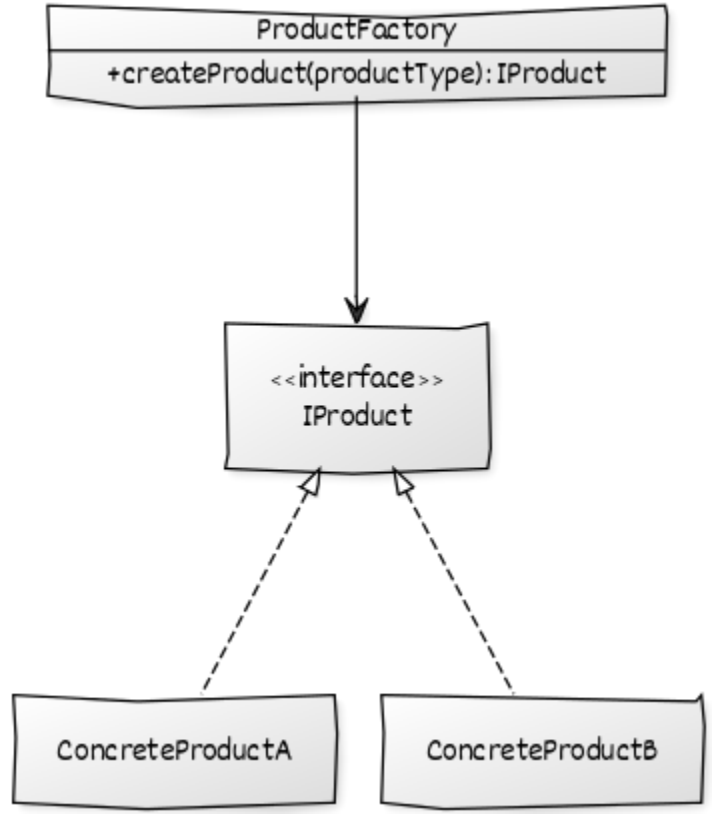
Buradaki amaç istemci tarafından birbirleri ile ilişkili nesnelerin oluşturulma anını soyutlamak, istemci hangi sınıf örneğini alabileceğini bilebilir ama oluşturulma detayları bilmez. Detaylar yani nesnenin nasıl oluşturulacağı soyutlanır.



Factory Patterns-Creational (Oluşturucu)

Konsept

Fabrika modeli , uygulama mimarisi tasarlanırken göz önünde bulundurulması ve uygulanması gereken en önemli ilke olan **sınıflar arasında gevşek bağlantı sağlar.** Gevşek bağlantı, somut uygulamalar yerine soyut varlıklara karşı programlama yaparak uygulama mimarisinde tanıtılabilir. Bu, mimarimizi yalnızca daha esnek kılmakla kalmaz, aynı zamanda daha az kırılgan hale getirir.



Factory Patterns-Creational (Oluşturucu)

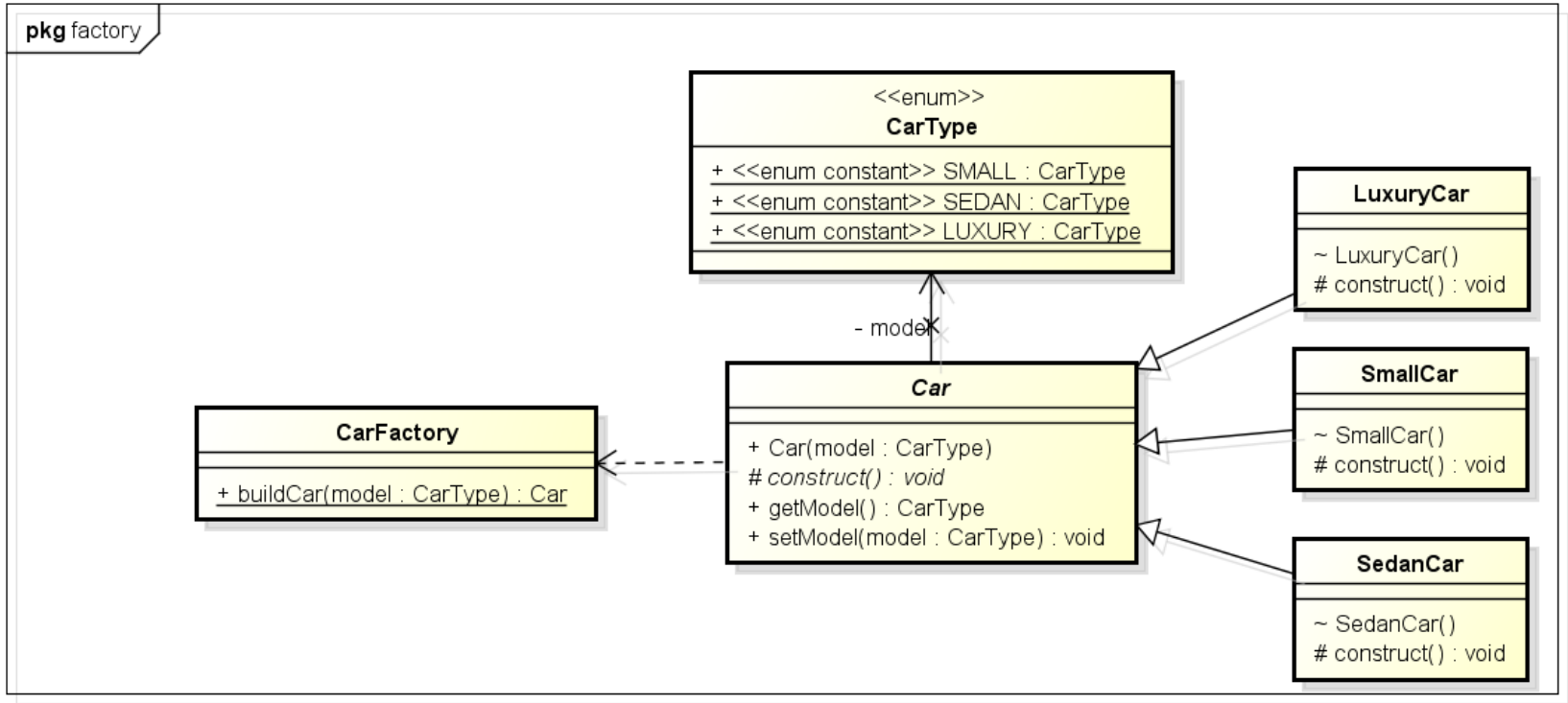
Bilgisayar Dünyası Örneği

•küçük, sedan ve lüks olmak üzere 3 tip araba üretebilen bir araba fabrikası örneğini kullanan ortak bir senaryoyu göstermektedir. Bir araba inşa etmek, aksesuarların tahsis edilmesinden son makyaja kadar birçok adım gerektirir. Bu adımlar, programlamada yöntemler olarak yazılabilir ve belirli bir araba tipinin bir örneğini oluştururken çağrılmalıdır.

•Eğer şanssızsak, SmallCaruygulama sınıflarımızda araba tiplerinin (örn.) örneklerini oluşturacağız ve böylece araba inşa mantığını dış dünyaya açacağız ve bu kesinlikle iyi değil. Ayrıca kod merkezi olmadığı için araba yapım sürecinde değişiklik yapmamızı da engelliyor ve tüm beste sınıflarında değişiklik yapmak mümkün görünmüyor.

Factory Patterns-Creational (Oluşturucu)

İllüstrasyon



Abstract Factory Patterns-Creational (Oluşturucu)

GoF Tanımı: Abstract Factory tasarım deseni birbirleri ile ilişkili ürün ailesini oluşturmak için bir arayüz sağlar.

Konsept

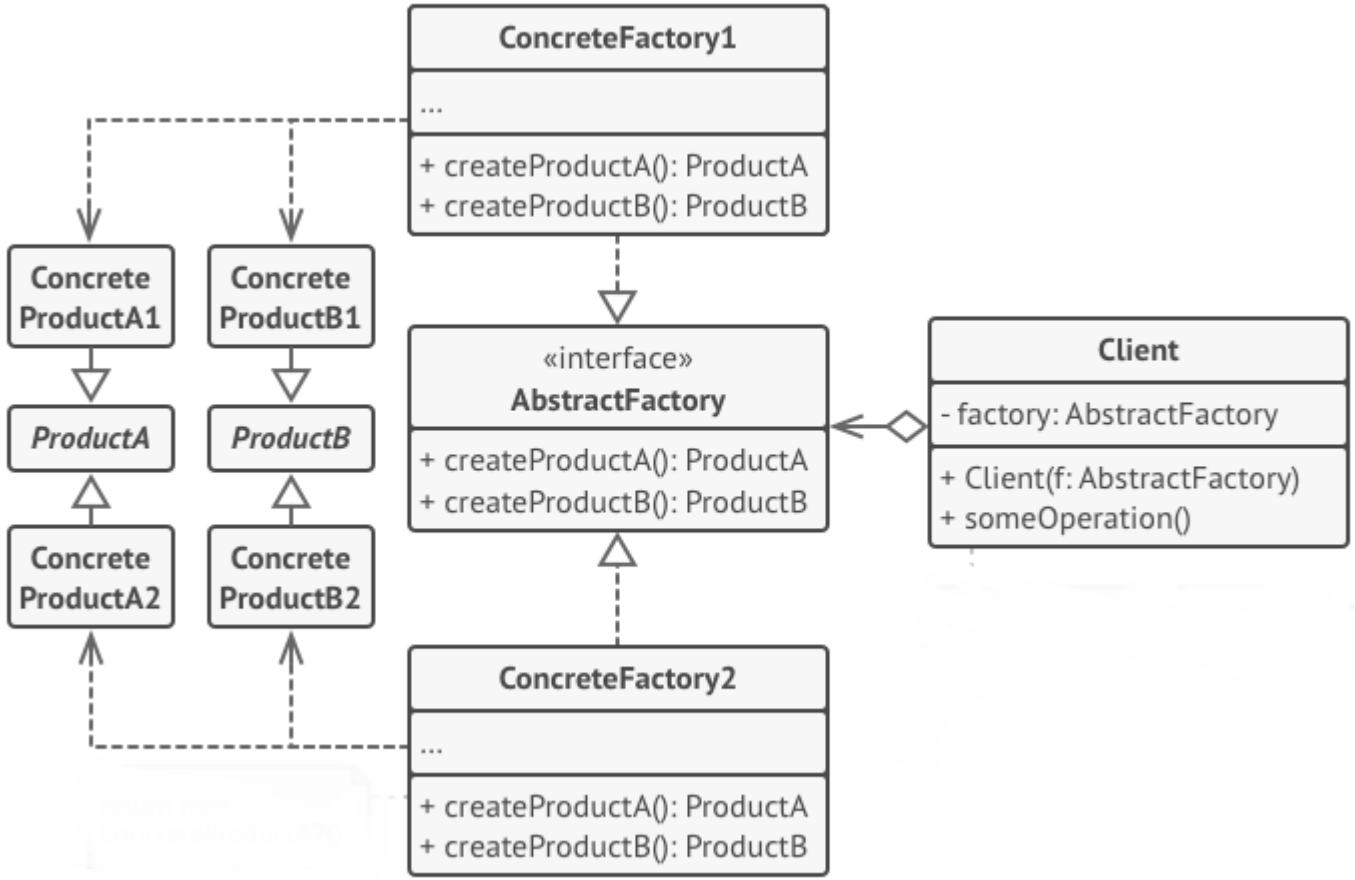
Soyut fabrika deseni , başka bir yaratıcı tasarım desenidir ve **fabrika deseni** üzerinde başka bir soyutlama katmanı olarak kabul edilir . Soyut fabrika örüntüsü, bir istemcinin, somut sınıfları belirtmeden nesneleri oluşturmasını sağlar. Bunun için bir fabrika nesnesi kullanır. Fabrika nesneleri istenen türde nesne oluşturmakla görevlidir. Aşağıda soyut fabrika örüntüsünün sınıf diyagramı görülmektedir. Burada istemci SoyutFabrika arayüzünü gerçekleştiren SomutFabrika nesnelerinden birini üretir. Bu somut fabrika da SoyutÜrün tiplerinden birinden istenen türde nesneleri üretmekle görevlidir. Sonrasında istemci hangi somut fabrika ve hangi somut ürün olduğu ile ilgilenmeden soyut fabrika arayüzünü kullanarak istediği tipte ürünü üretmek için metodu çağırarak nesne ürettirir. Böylece istemci fabrika ve ürün detaylarından tamamen soyutlanmış olur.

Abstract Factory Patterns-Creational (Oluşturucu)

Konsept

Factory tasarım deseninde bir ürünün oluşturulması soyutlanmış iken Abstract Factory deseninde birbirleri ile ilişkili ürün ailelerininin oluşturulması soyutlanmıştır. **Factory üreten Factory deseni olarak da düşünülebilir.**

Anlayacağımız; birden fazla ürün ailesi ile çalışmak zorunda kaldığımız durumlarda, istemciyi bu yapılardan soyutlamak için Abstract Factory doğru bir yaklaşım olacaktır.



Abstract Factory Patterns-Creational (Oluşturucu)

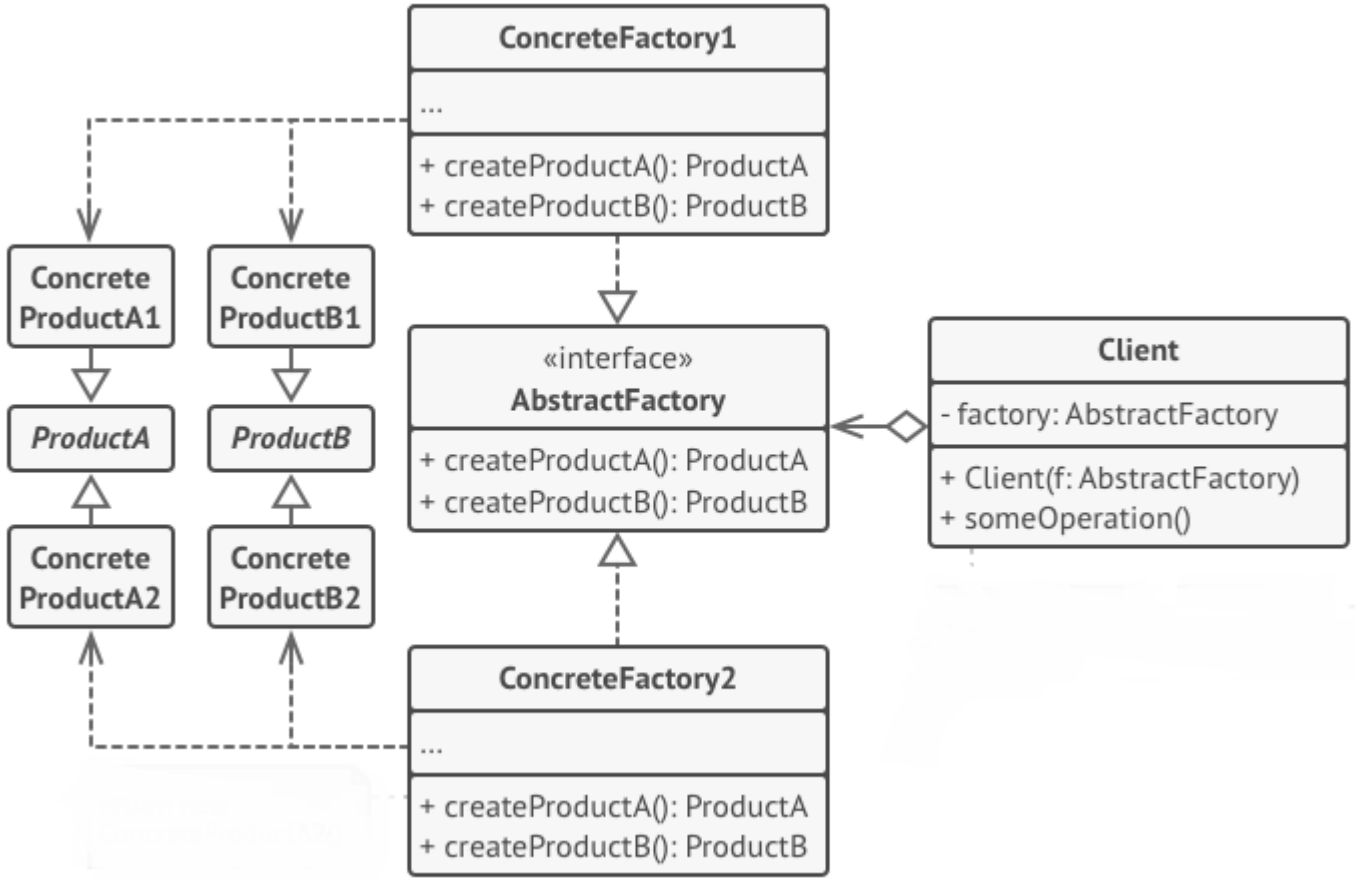
Konsept

ProductA, ProductB: Temel sınıflarımız, soyuttur ve oluşturulmasını istediğimiz sınıflar bunlardan türer.

ConcreteProduct: Üretmek istediğimiz sınıflardır.

AbstractFactory: Herbir sınıfın oluşturulması için metotların tanımlandığı arayüzdür.

ConcreteFactory: AbstractFactory arayüzünü uygulayarak gerekli sınıfların oluşturulmasını sağlar.



Abstract Factory Patterns-Creational (Oluşturucu)

Bilgisayar Dünyası Örneği

•Soyut fabrika modelini kullanarak küresel otomobil fabrikasını tasarlayalım.

•Küresel operasyonları desteklemek için , farklı ülkeler için farklı araba yapım tarzlarını desteklemek için sistemi geliştirmemiz gerekecek. Örneğin bazı ülkelerde direksiyonu sol tarafta, bazı ülkelerde ise sağ tarafta görüyoruz. Arabaların farklı kısımlarında ve yapım süreçlerinde bunun gibi daha birçok farklılık olabilir.

•Soyut fabrika modelini tarif etmek için 3 tür markayı ele alacağız – ABD , Asya ve diğer tüm ülkeler için varsayılan . Birden fazla konumun desteklenmesi, kritik tasarım değişikliklerine ihtiyaç duyacaktır.

Abstract Factory Patterns-Creational (Oluşturucu)

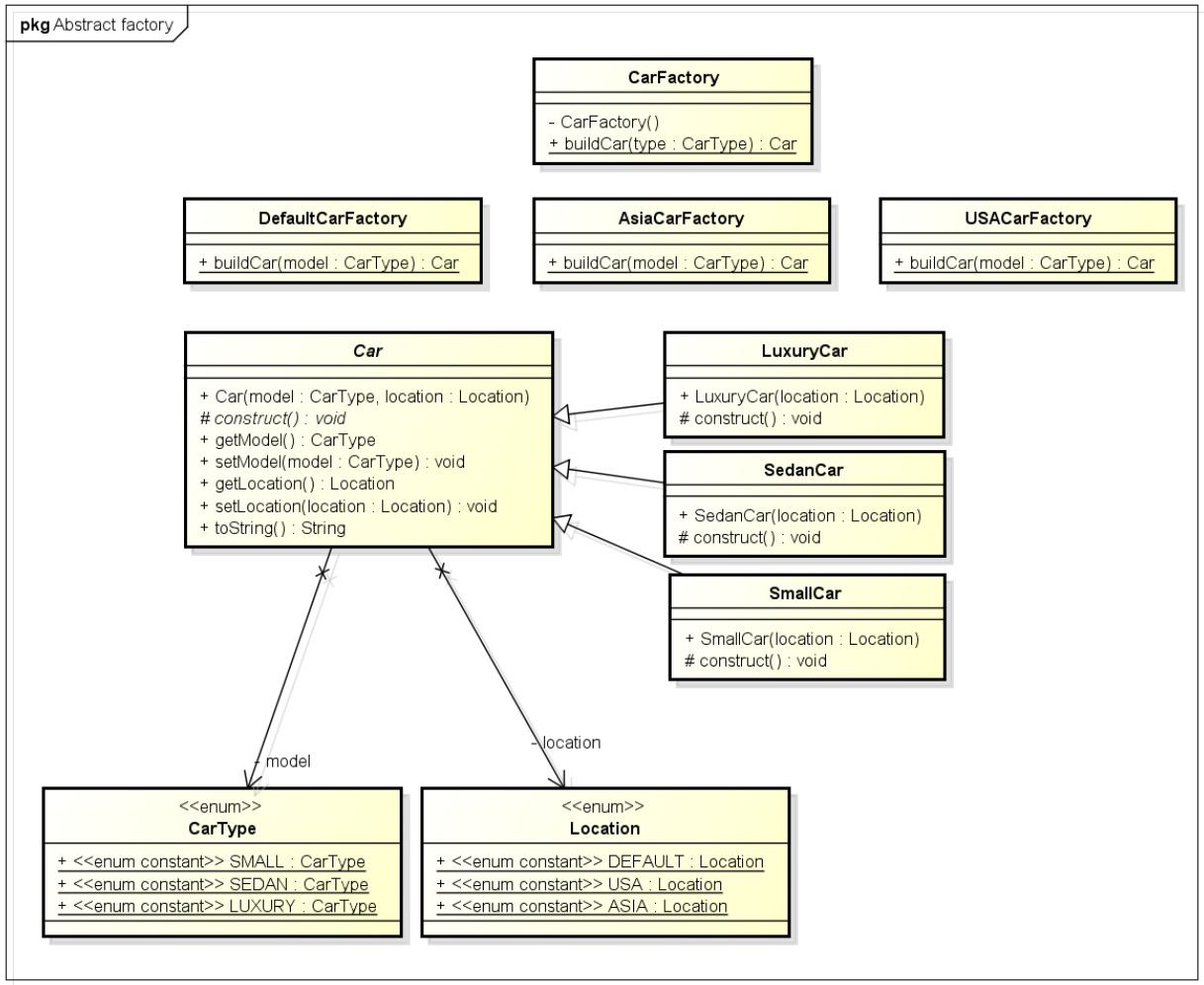
Bilgisayar Dünyası Örneği

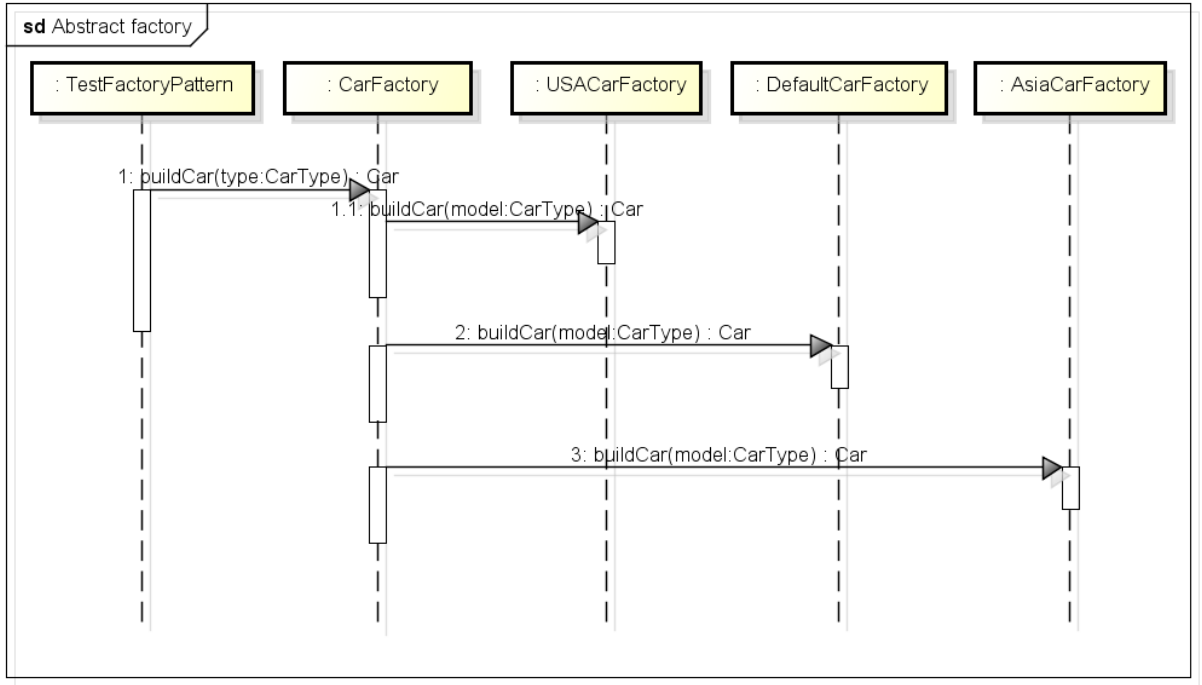
•Her şeyden önce, sorun bildiriminde belirtilen her lokasyonda araba fabrikalarına ihtiyacımız var. yani USACarFactory , AsiaCarFactory ve DefaultCarFactory . Şimdi, uygulamamız kullanıldığı yeri tespit edecek kadar akıllı olmalı, bu yüzden dahili olarak hangi araba fabrikası uygulamasının kullanılacağını bile bilmeden uygun araba fabrikasını kullanabilmeliyiz. Bu aynı zamanda bizi belirli bir konum için yanlış fabrikayı arayan birinden de kurtarır.

•Temel olarak, konumu belirleyecek ve kullanıcıya tek bir ipucu bile vermeden dahili olarak doğru araba fabrikası uygulamasını kullanacak başka bir soyutlama katmanına ihtiyacımız var. Soyut fabrika deseninin çözmek için kullanıldığı sorun tam olarak budur.

Abstract Factory Patterns-Creational (Oluşturucu)

İllüstrasyon





Singleton Patterns-Creational (Oluşturucu)

GoF Tanımı: Tek sorumluluk ilkesi (SRP), bir yazılım bileşeninin (genel olarak bir sınıf) yalnızca bir sorumluluğa sahip olması gerektiğini belirtir. Sınıfın tek sorumluluğa sahip olması, onun tek bir somut şey yapmaktan sorumlu olduğu anlamına gelir ve bunun sonucunda, değişmesi için tek bir nedeni olması gerektiği sonucuna varabiliriz..

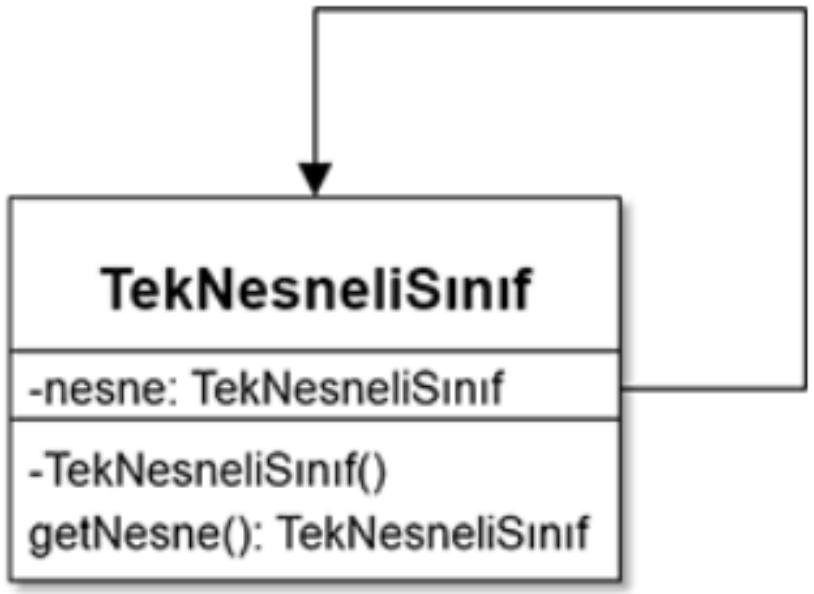
Konsept

Bazı sınıflar için tam olarak bir örneğe sahip olmak önemlidir. Bir sistemde çok sayıda yazıcı olmasına rağmen, yalnızca bir yazıcı biriktiricisi olmalıdır. Yalnızca bir dosya sistemi ve bir pencere yöneticisi olmalıdır. Dijital filtrede bir A/D dönüştürücü bulunur. Bir şirkete hizmet vermek için bir muhasebe sistemi tahsis edilecektir. Bir sınıfın yalnızca bir örneği olduğundan ve örneğin kolayca erişilebilir olduğundan nasıl emin olabiliriz? Genel bir değişken, bir nesneyi erişilebilir kılar, ancak birden çok nesneyi başlatmanızı engellemez. Daha iyi bir çözüm, tek örneğini takip etmekten sınıfın kendisini sorumlu kılmaktır. Sınıf, başka bir örneğin oluşturulmamasını sağlayabilir (yeni nesneler oluşturma isteklerini engelleyerek) ve örneğe erişmenin bir yolunu sağlayabilir. Bu Singleton modelidir..

Singleton Patterns-Creational (Oluşturucu)

Konsept

Java ve diğer nesnesel dillerde herhangi bir sınıftan istenen sayıda nesne oluşturulabilir. Fakat bazı nesnelerin yalnızca bir kere oluşturulması gerekebilir. Yani o sınıftan birden fazla nesne oluşturulması engellenmek istenebilir. Örneğin, bilgisayarda kullanılan donanıma hizmet veren donanım sürücüleri, log kayıtları alan bir servis, belirli bir servis sağlayıcının hizmet kuyruğu gibi. Bu nesnelerden birden fazla oluşturulması işlevlerinde probleme yol açacaktır. Örneğin servis sağlayıcının iki kuyruğu olduğunda hangisinden gelen istemcilere hizmet verilecektir, öncelikler ne olacaktır gibi problemler ortaya çıkabilir. Bu şekilde bir kere oluşturulması istenen nesneler için Tekli Örüntü geliştirilmiştir.

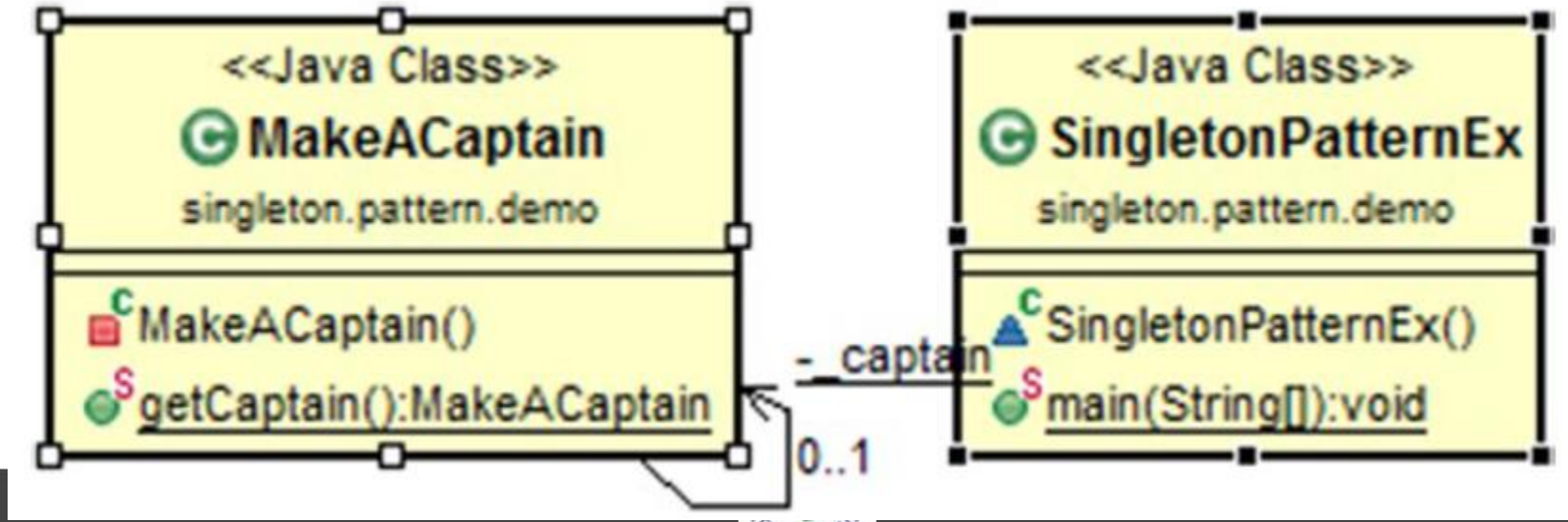


Singleton Patterns-Creational (Oluşturucu)

Bilgisayar Dünyası Örneği

•Bir yazılım sisteminde bazen sadece bir dosya sistemi kullanmaya karar verebiliriz. Genellikle kaynakların merkezi yönetimi için kullanabiliriz.

•Bu örnekte, normal şekilde somutlaştıramamak için ilk önce yapıcıyı özel yaptık. Sınıfın bir örneğini oluşturmaya çalıştığımızda, elimizde mevcut bir kopya olup olmadığını kontrol ediyoruz. Böyle bir kopyamız yoksa, onu oluşturacağız; aksi takdirde, mevcut kopyayı yeniden kullanacağız.



Command Pattern - Behavioral (Davranışsal)

GoF Tanımı: Command tasarım deseni, kullanıcı isteklerini gerçekleştiren kod yapısını sarmallayarak nesneler halinde saklanmasını daha sonra da bu isteklerin gerçekleştirilmesini veya geri alınmasını sağlayan tasarım desenidir..

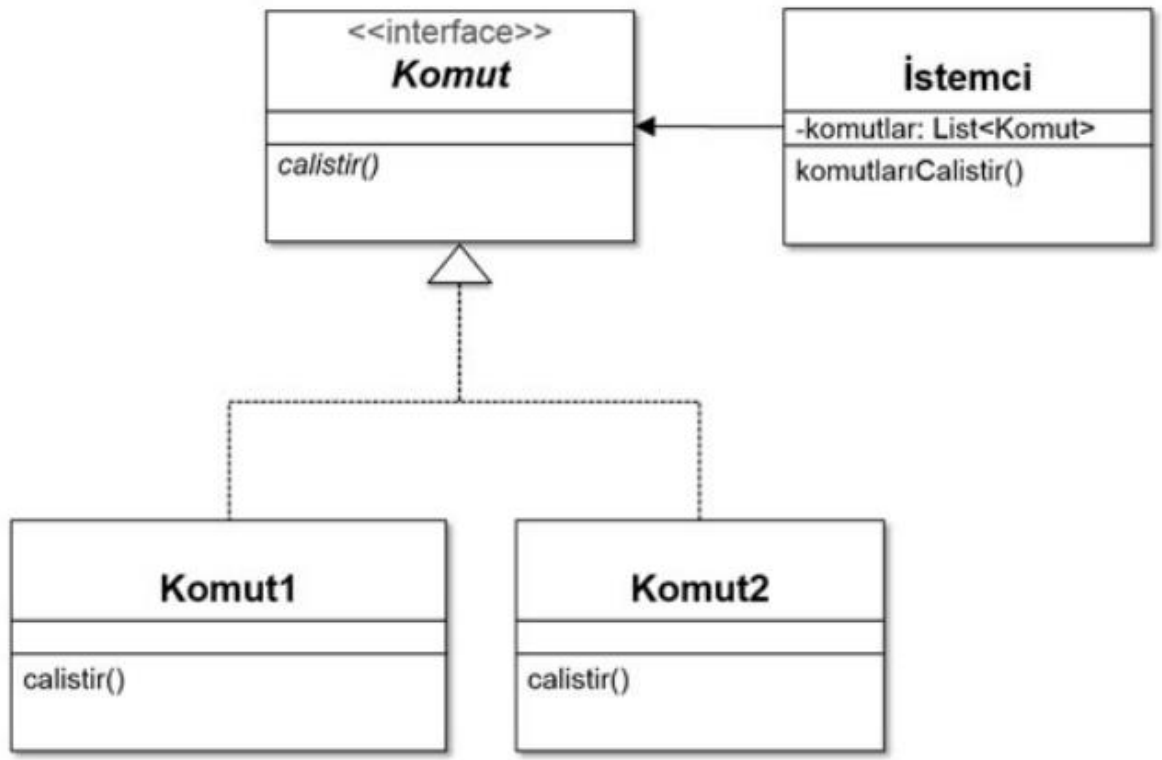
Konsept

Burada istekler nesneler olarak kapsüllenir. Genel olarak, dört terim ilişkilidir: çağıran, istemci, komut ve alıcı. Bir komut nesnesi, alıcıda belirli bir yöntemi çağırabilir. Metotların parametrelerini alıcıda saklar. Çağıran yalnızca komut arabirimini bilir, ancak somut komutlardan tamamen habersizdir. İstemci nesnesi, çağıran nesneyi ve komut nesnesini/nesnelerini tutar. İstemci, bu komutlardan hangisinin belirli bir zamanda yürütülmesi gerektiğine karar verir. Bunu yapmak için, o belirli komutu yürütmek için komut nesnesini çağırıcıya iletir.

Command Pattern - Behavioral (Davranışsal)

Konsept

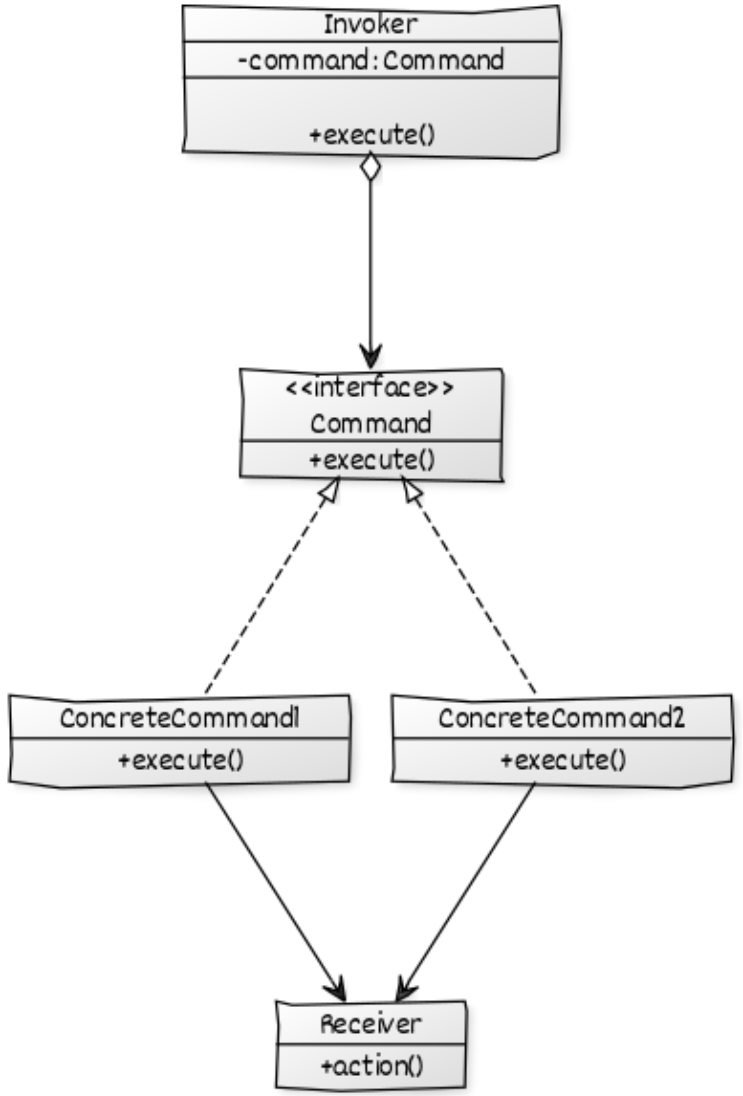
Bir isteğin nesne olarak oluşturulmasını sağlar. Davranışsal bir örüntüdür. Bir istek bir komut nesnesi olarak çerçevelenir ve işleyecek nesneye gönderilir. İşlemci nesne bu komutu uygun nesneyi kullanarak çalıştırır. Aşağıda komut örüntüsünün genel sınıf diyagramı yer almaktadır. Burada Komut soyut sınıfı calistir soyut metodunu tanımlar. Somut Komut sınıfları ise Komut soyut sınıfını gerçekleştirir, yani calistir metotunu özelleştirirler ve kendilerine özel işlemleri yerine getirirler. İstemci ise bir dizi Komut'u istenen sırada ve seçici olarak işleyebilir (komutlarıCalistir metotu)..



Command Pattern - Behavioral (Davranışsal)

Konsept

Bu tasarım desenine ait UML diyagramı yandaki gibidir.



Command: Temel arayüzdür, komutların çalıştırılması için temel metotu içermektedir.

ConcreteCommand: Nesnelere dönüştürdüğümüz her bir isteğe denk gelmektedir, Command arayüzünü uygular.

Invoker: Command referansını tutan, metotun ne zaman çağrılacağını belirtir.

Receiver: Client tarafının asıl iletişime geçeceği sınıftır.

Command Pattern - Behavioral (Davranışsal)

Bilgisayar Dünyası Örneği

•Senaryo olarak bir ürüne ait stok takibi verilen basit bir sistem olduğunu var sayalım. Burada stok ile ilgili işlemler bu tasarım desenine göre yapılsın.