

Tasarım Kılflıkları

- Singleton Design Pattern** → Bir sınıfın yalnızca bir tane nesne oluşturmasını sağlar ve bu nesneye global erişim sunar.
- Veritabanı bağlantısı gibi tek bir koynak gereklidir.
 - Ayarlar (config), loglama (logger), önbellek (cache) gibi ortak ve tekil yapılar varsa.
 - Çok fazla nesne oluşturmalarının sistemi yavaşlatacağı durumlar varsa.
- “Bir uygulamanın içindeki tek bir yönetici paneli gibi düşünün. Her kullanıcı aynı panele erişir ama yeni bir panel yapılmaz.”
- ⊕ Koynak israfını engeller
 - ⊕ Küresel erişim sağlar
 - ⊕ Merkezi yönetir
 - ⊖ Asırı kullanımca bağımlılık yaratır (test yapmak zorlaşabilir)
 - ⊖ Çoklu iş forgacığına dikkat edilmezse hata olabilir (Java'da özellikle)
- Factory Method Design Pattern** → Bir üst sınıf, hangi sınıfın örneği oluşturulacağına alt sınıfların karar verdiği bir yapıdır. Amaç nesne oluşturmamıza alt sınıflara bırakmak.
- Hangi sınıfın nesnesi oluşturulacağı önceden bilinmeyecektir.
 - Nesne oluşturma işini korosiksa ve bunu koddan soyutlamak istiyorsak.
 - Kodda bağımlılığı azaltmak ve genişletilebilirliği artırmak istiyorsak.
- “Bir pizzacı düşün. Mükteri sadexe 'Pizza istiyorum' der, ama mutfak hangi tür pizza yapacağını karar verir.”
- ⊕ Kodun genişletilmesini kolaylaştırır.
 - ⊕ Nesne oluşturmamıza mecbur değiliz.
 - ⊕ Genişek bağıllılık (low coupling) sağlar.
 - ⊖ Her yeni ürün türü için kodda yeni sınıf ve kontrol gereklidir.
 - ⊖ Yapıtı boşlukta basit proseler için fazla detaylı gelebilir.
- Abstract Factory ~~Method~~ Design Pattern** → Birbirisyle ilişkili nesne gruplarını (masa + sandalye + kanepe) türlerine göre oluşturmak için kullanılabilecek fabrika yapısıdır.
- Factory method'un bir üst modelidir. — fabrika üretir, ama bu fabrika başka nesneler üretir.
- Birbirisyle ilişkili ürünlerin ailelerini oluşturmak istiyorsan.
 - Aynı ortamda farklı varyasyonlar üretmek istiyorsan.
 - Uygulamanın temasını, platformunu, mimarisi kolayca değiştirmek istiyorsan.
- “IKEA düşün. Bir masa üreticisi (factory) var ama bu üretici aynı zamanda masa, sandalye ve sehpası gibi bir tane setini birlikte üretiyor:
- örneğin: Modern set → modern masa, modern sandalye, modern sehpası
Klasik Set → klasik masa, klasik sandalye, klasik sehpası”
- ▷ Ürün ailelerini birlikte değiştirmek kolaydır.
 - ▷ Kullanıcıya net ve anlaşılırce uygundur.

④ Bağımlılığı azaltır (ürün oluşturma sayılır)

⑤ Yapısı daha karmaşık

⑥ Her gün ihan çizgi ve sınıf yazmak gereklidir (fazla kod)

Builder Design Pattern → Karmaşık nesnelerin adım adım oluşturmalarını sağlar. Aynı anda silecekle farklı nesne türleri oluşturabilmeni olanak tanır. Yani ne yapacağın değil nasıl yapacağının kontrol edersin.

• Bir nesne oluşturken birçok adım varsa (özellik, varyasyon, seçenek)

• Aynı nesnenin farklı versiyonunu üretmek istersen

• Göz fazla constructor parametresi varsa kodun okunabilirliğini artırmak için,

"bir hamburger sınıfı düşün, her seferinde aynı mutfağı (yapım süreci) kullanılır. ama bizer ekstra peynir, bizer büyük boy elmek istersem. Yani üretim süreci aynı gün farklı olabilir."

④ Kodun okunabilirliği artır (özellikle uzun constructorlarda)

⑤ Aynı nesneye ait varyasyonları kolayca üretirsin

⑥ "Immutable" nesneler üretmek kolaylaşır.

⑦ Küçük projelerde osuri yapılama gibi olumsuzluklar

⑧ Her karmaşık nesne için tek sınıf (builder) yazmak gereklidir

Prototype Design Pattern → Ver olsan bir nesnenin kopyasını oluşturarak yani bir nesne üretmesini sağlar. Ancak nesnesi sıfırdan oluşturmak yerine, mevcut bir nesneyi kopyalayarak yine de kullanmak.

• Nesne oluşturmak maliyetliyse (gözfazla veri, yeri, yükleme vs.)

• Bir nesnenin farklı varyasyonları gerekiyorsa ana aynı temel yapı kullanırsa

• Nesneleri runtime'da değiştirilebilir

"bir word belgesi şablonu var. Onu her seferinde sıfırdan yazmak yerine kopyalayıp içeriği değiştirirsin."

④ Karmaşık nesnelerin hızlı ve kolay klonlanmasını sağlar.

⑤ Sıfırdan oluşturma maliyetini azaltır.

⑥ Runtime'da nesne üretimi daha esnek hale gelir

⑦ Derin kopya(deep copy) ile yüzeysel kopya(shallow copy) karışabilir.

⑧ Klonlanabilir nesneler iyi tanımlanması gereklidir.

Özetimsel Tasarım Kalıfları Özette →

Singleton → Tek nesne

Factory method → Alt sınıfa nesne üretimini bırakır

Abstract factory → ilişkili nesne grupları üretmek

Builder → karmaşık nesneleri adım adım oluşturmak

Prototype → Ver olsan nesneyi kopyalamak

Adapter Design Pattern → farklı arayüzlerle sahip iki sınıfın birlikte çalışmasını sağlar. Yeni uyumlu bir sınıfı uyumlu hale getirir. Tipki priz adaptör gibi.

- Verilen bir sınıfı, başka bir sisteme entegre etmek istiyorsa
- Yeni bir sistem, eski kodla uyumlu hale gelirmek gerekiyorsa
- Kodun arayüzüni değiştirmek istiyorsa ama başka arayüze kullanmak istiyorsa

“Telefonun USB-C, ama bilgisayarın USB-A, Araya bir dönüştürücü (Adapter) takarsın. Yani istek var uyum yok. Gidon → Adapter”

④ Eski kodu değiştirmeden yeni sisteme entegre eder.

⑤ Mevcut sistemleri birbirine uydırır.

⑥ Kod yeniden kullanılabilir hale gelir.

⑦ Çok fazla adapter, kodda karmaşıklik yaratır.

⑧ Hatalı ve eksik çeviriler olabilir.

Bridge Design Pattern → Sınıflama (abstraction) ile uygulama (implementation) arasında bir köprü kurar. Baylere ikisini bağımsız olarak geliştirebilir ve geliştirebilirsin.

• Hem sınıflama hemde gerçekleme sıklıkla değişebilse

• Farklı özelliklerin karıştırılmak gerekiyorsa (örneğin: şekiller + renkler)

• Interface yerine composition ile esneklik istiyorsa

“Bir uzaktan kumanda (sınıflama) farklı televizyonlara (uygulama) çalışabilir, kumandanın aradığı değişmeden yeni markalar eklenebilir.”

④ Sınıflama ve gerçekleme bağımsız gelişir.

⑤ Koda sınıf patomiasını (her kombinasyon için alt sınıf) önlüyor.

⑥ Çok daha esnek ve yapılabılır bir yapı sunar.

⑦ Yer, biraz daha karmaşık hale gelebilir.

⑧ Küçük projelerde fazla gereksiz sınıflama hissi yaratır.

Composite Design Pattern → Nesneleri ağaç yapısı içinde hiyerarşik şekilde düzeltmeni sağlar. Hem tekil hem de bilesik nesnelere aynı şekilde durumabilmesi izin verir.

• Nesneler hiyerarşik yapıdaysa (menüler, klasörler...)

• Tek bir nesne ile bir grup nesneye aynı işlemi yapmak istiyorsa

• Uyandırılabilir yapı kurmak gerekiyorsa (örneğin bir ağaç)

“Bir klasör dosyası, içinde dosyalar ve boşta klasörler var. Hem tek bir dosyaya, hem klasöre (ve içindeki klasörler) aynı şekilde işlem yapılabilirsin.”

④ Tekil ve bilesik yapılda tek tip işlem yapılır

⑤ Ağaç yapları temiz ve düzenli bir şekilde dastururlar

⑥ Yenilemebilirlik ve genişletebilirlik sağlar

⑦ Uygunlukta debug ve yönetim zorlaşır

⑧ Fazla sınıflama karmaşıklik yaratır.

Decorator Design Pattern → Bir nesneye, davranışını değiştirmeden dinamik olarak yeni özellikler eklemeye yarar. Kullanım yerine kompozisyon kullanarak genişletme sağlar.

- Bir nesneye özellik eklemek istiyorsan ama alt sınıf türetmek istemiyorsan
- Özellikleri runtime'da değiştirebilmek istiyorsan

• Farklı kombinasyonlarda özellik eklemek gerekiyorsa (sarma mantığıyla)

"Kohve alırsın → Üstüne sütlü eklersin → Şeker eklersin → Çikolata eklesin. Her biri bir sistemlidir (decorator) ve kahvenin tadını değiştirir. Anna kahve hala kahvedir."

④ Nesneleri alt sınıf oluşturmadan genişletir.

④ Durumları değişme zamanında değiştirebilirsin

⑤ Farklı özellik kombinasyonları kolaydır

⑥ Göz fazla decorator sınıfı karmaşıklik oluşturabilir

⑦ Kod tabibi zorlaşır ("normal etkisi")

Facade Design Pattern → karmaşık bir sistemi basitleştir; bir çok işlev sunar. Kullanıcı, alt sistemin detaylarını bilmeden işini halledebilir.

• Birde çok sınıfın karmaşık etkisini tek bir işlevle gizlemek istiyorsan

• Sistemin dışından kolayca erişilemesini istiyorsan

• Kodun dış batırılmasını iş detaylarından yaktırmak istiyorsan

"Bilgisayar开机电源按钮按压. Altta on/off, RAM, BIOS, işletim sistemi deneye girer - ama son sadece tek tuşa basırsın."

④ Karmaşıklığı sınırlar, sistemin kullanımını kolaylaştırır

⑤ Kodun dışına basit API sunar.

⑥ Alt sistemler bağımsız kalır, daha iyi bir şarapnılıklık sağlar.

⑦ Fazla soyutlama bazı işlevleri engellemez.

⑧ Facade çok fazla sorumluk alırsa God Object olabilir.

Flyweight Design Pattern → Çok sayıda benzer nesne yaratmak yerine, bu nesneleri paylaşarak bellek kullanımını azaltır. Paylaşım ve paylaşılmayan nesnelere çalışır.

• Nihayetinde nesne yaratma gerekiyorsa

• Bu nesnelerin büyük kısmı aynı veriyi taşıyorsa

• Performans ve bellek optimizasyonu önemlidir

④ Ron kullanımını büyük oranda azaltır

⑤ Nesneleri paylaşarak performans artışı sağlar

⑥ Milyonluk nesne yerine birkaç tanesini tekrar kullanır.

⑦ Kod karmaşıklasabilir

⑧ Dış veri(extrinsic state) iyi yönetilmemezse hata olur.

"Bir metin editörü düşün. Her 'a' harfi için ayrı nesne oluşturmayı. Sadece bir 'ö' harfi nesnesi varmış. Geri kalanlar sadece pozisyonla ayrılır. Yani harf bir nesnedir. Pozisyon dış verisi"

Proxy Design Pattern → Gerçek nesneye erişimi kontrol etmek için bir vekil(proxy) nesne oluşturular. Gerçek nesneye doğrudan erişim yerine, araya aracı koyarak işlem yapılır.

- Gerçek nesne doğrudan erişimi istemek
- Güvenlik, güvenlikli yüklenme(log), loglama, önbelleklere gibi kontrol mekanizmaları gerekiyor
- Gerçek nesneye doğrudan erişimi sınırlamak istiyorsan
- ① Ofis çalışanı bir dokümana ulaşmak ister ama önce sekretere gider. Sekreter uygun koşullarda dokümana erişir. Sekreter = proxy, Dokuman = Gerçek nesne
- ② Gerçek nesne güvenlikli yüklenir (log)
- ③ Güvenlik, önbelleklere, loglama gibi ek işlevler kolay eklenir.
- ④ Gerçek nesneye erişim kontrol altında tutulur.
- ⑤ kod karmaşıklığı
- ⑥ Vektör fazla sorumluluk alırsa bakım zorlaşabilir.

~~Buglece 7 genel tasarım kalıpları~~ ~~sıradı sıradı Dövenissal tasarım kalıpları~~
Chain of Responsibility → Bir isteğin işlemesi için birde fazla nesne sırasıyla denecektir. Her nesne kendisi işini yapar ya da sıradaki nesneye devreder. Zincir kırılmaz, ama hangi nesnenin işleyeceği önceden bilinmez.

- Bir isteğin kimin tarafından işlenerası önceden bilinmiyorsa
- İşleyiciler (handlers) zincir şeklinde bağlıysa
- Genel bağlılık istiyorsan (nesneler birbirini hakkında fazla bilgi sahibi olmasın)
- ① Bir masteri şikayet ettiğinde sırasıyla ① Masteri temsilcisine gider ② Gereklense minden işlevi iletilir
- ③ O da yetmezse genel masteri gider, her ascıma kendi düzeyinde sorumluluğu alabilir ya da devredebilir.
- ④ İstek işleyicileri genelde bağlı hale gelir.
- ⑤ Yeni işleyici eklemek çok kolaydır
- ⑥ Zincir sırasını değiştirmek dinamik yapılabilir
- ⑦ Zincir çok uzun durussa performans düşebilir
- ⑧ Hatalı zincir yapısı bazı isteklerin hiç işlememesine neden olabilir.

Command Design Pattern → Bir işlemi (komut, Galistiracak Nesneden ayrı olarak nesne haline getir). Bu komut nesnesi daha sonra soklanabilir, sıralanabilir, geri alınabilir veya tekrar kullanılabilir. Bir komut nesnesi daha sonra soklanabilir, sıralanabilir, geri alınabilir veya tekrar kullanılabilir.

- Geri al, tekrar et, kaydet ve sonra galistir gibi ihtiyaçları varsa
- İşlemleri nesne şeklinde temsil etmek istiyorsanız
- Kullanıcı eylemleri veya uzaktan (remote) kontrol gereklisi

① Gerekçe sipariş al desin. Gerekçe bu sipariş (komut) mutfağı yollar. Mutfak siparişi bilmem; sadece verilen komutu şaire getirir. Sipariş bir nesne halindedir, ki ne yapacığı içinde yazılıdır

- ② Komutlar nesne olarak taşınabilir, soklanabilir, geri alınabilir

③ İşlem ile nesne birbirinden ayrıldığı için genelde bağlılık sunar

- ④ İşlem sıralama ve kuyulama (queue) kolaylaşır

⑤ Her işlem için bir sınıf oluşturmak karmaşıklaştırır

⑥ Küçük projelerde ağır mühendislik gibi görülebilir.

