SOLID

SOLID, Bad Architecture Design diyebileceğimiz 3 problemi çözmeyi amaçlıyor.

Bad Design Nedir? Biz neye bad design diyebiliriz?

1. Rigidity - Katılık
   Değişiklik yapmak zordur çünkü her değişiklik sistemin diğer birçok parçasını da etkiler.
2. Fragility - Kırılganlık
   Değişiklik yapıldığında sistemin beklenmedik parçaları kırılır.
3. Immobility - Hareketsizlik
   Mevcut uygulamadan ayrılamadı için başka bir uygulamada tekrar kullanılması zordur.

1.Rigidity
Büyük ölçüde birbirine bağımlı yazılımda yapılan bir değişikliğin, bağımlı modüllerde bir dizi değişikliği başlatmasından kaynaklanmaktadır. Bu başlatılan değişikliğin kapsamı geliştiriciler tarafından tahmin edilemediğin, değişimin etkisi tahmin edilemez. Bu da değişikliğin maliyetini tahmin etmeyi engeller. Tüm bu engeller göz önüne alındığında değişiklik yapılması yöneticiler tarafından hoş karşılanmaz ve böylece katılık dediğimiz durum oluşur.

2.Fragility
Bir programın tek bir değişiklik yapıldığında birçok yerde kırılma eğilimidir. Bu kırılmalar, tasarım ve bakım ekiplerinin güvenilirliğini azaltır. Yöneticiler ürün kalitesini tahmin edemezler. Uygulamada yapılan basit değişiklikler, beklenmedik kısımlarda arızalara yol açabilmektedir. Bu arızaların giderilmesi daha büyük sorunlara yol açar ve bakım süreci hiç bitmeyecek şekilde devam eder.

3.Immobility
Tasarımda ihtiyaç duyulan kısım, büyük ölçüde diğer detaylara bağımlı olduğunda tasarım hareketsizdir. Tasarım son derece birbirine bağımlıysa, bu durumda istenilen kısmı diğer kısımlardan ayırmak için gereken çalışma miktarı da göz korkutacaktır. Çoğu zaman ayırma maliyetinin, yeniden geliştirme maliyetine göre daha fazla olmasından dolayı bu tür tasarımlar yeniden kullanılmaz.

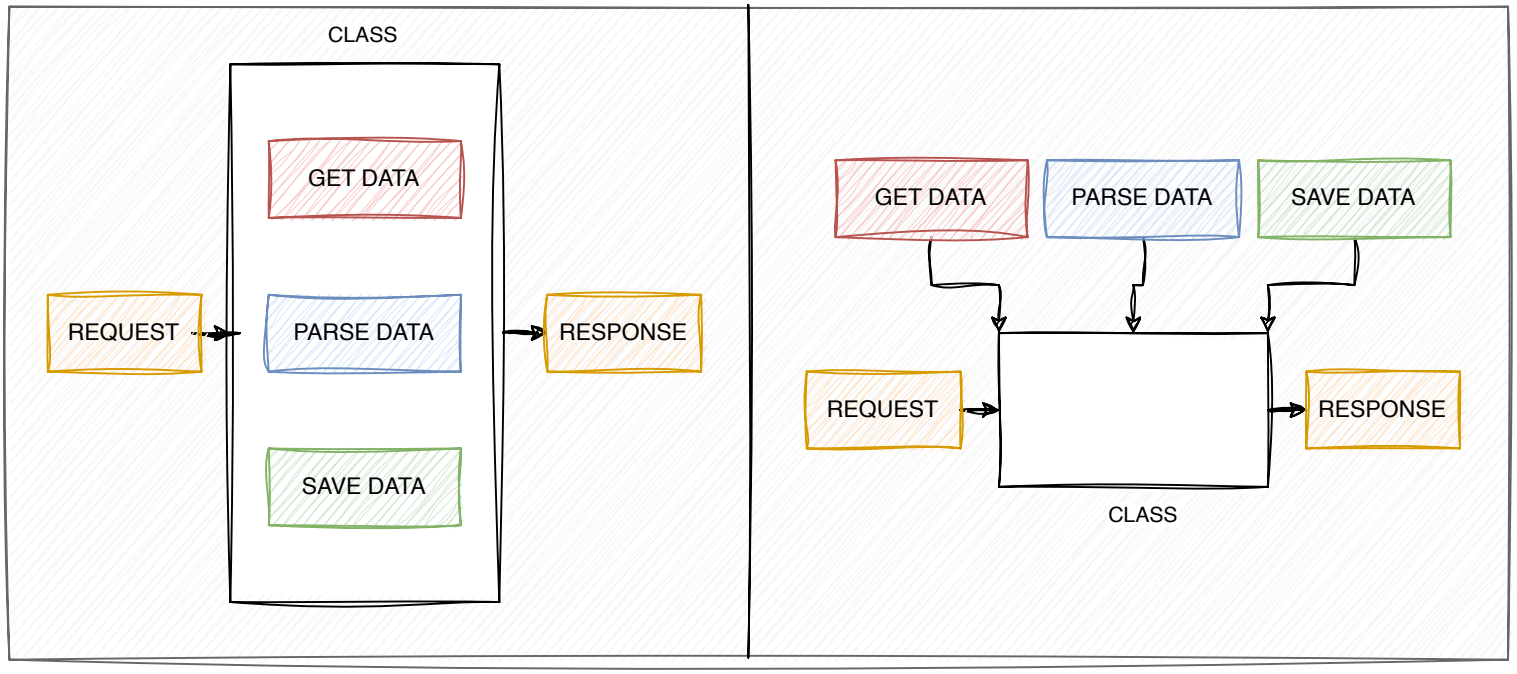
Single Responsibility Principle

Sınıf sadece bir işten sorumlu olmalıdır.

Bu prensip sınıfları çok daha basit ve temiz tutmamızı sağlar. Böylece test yazmamız kolaylaşır. Anlaması çok basit olan bu ilkenin uygulanması oldukça karmaşık ve zor olabilmektedir.

Kısacası

- Sınıfınız sadece belirli bir işten sorumlu olmalıdır.
- Kodumuz daha test edilebilir bir yapıya dönüşmüştür.
- Uygulamanın farklı bir yerinde tekrar benzer kodları kullanabiliriz.
- Eğer bir değişiklik yapmak istersek, çok daha büyük kod parçalarını değil sadece kodda sorumlu olan kısmı değiştirmemiz yeterli olacaktır.



Open-Closed Principle

Sınıflar, modüller, fonksiyonlar vs. uzantılara (extension) açık, değişikliklere kapalı olmalıdır.

Hiç değişmeyen modüller tasarlanmamız gerektiğini söylüyor. Gereksinimler değiştinde, hali hazırda çalışan eski kodu değiştirerek değil, yeni kod ekleyerek bu tür modüllerin davranışını genişletmeliyiz.

Open-Closed ilkesinin iki temel özelliği vardır.

1. Open for Extension
   Modülün davranışının genişletilebileceği anlamına gelir. Uygulamanın gereksinimleri değiştikçe veya ihtiyaçlarını karşılayacak şekilde
2. modülün yeni ve farklı davranmasını sağlayabilmeliyiz.
3. Closed for Modification
   Modülün kaynak kodu ihlal edilemez. Hiç kimse kaynak kodunda değişiklik yapmamalı

Bir kodun değişikliklere %100 kapalı hale getirilmesi neredeyse imkansızdır. Her zaman düşünülemez veya ele alınmamış senaryolar olacaktır.

Bazı buluşsal yöntemler ve gelenekler tam da bu prensip nedeniyle ortaya çıkmıştır. Bunlardan iki tanesi; tüm özellikleri özel yapmak ve Global değişkenleri kullanmamaktır.

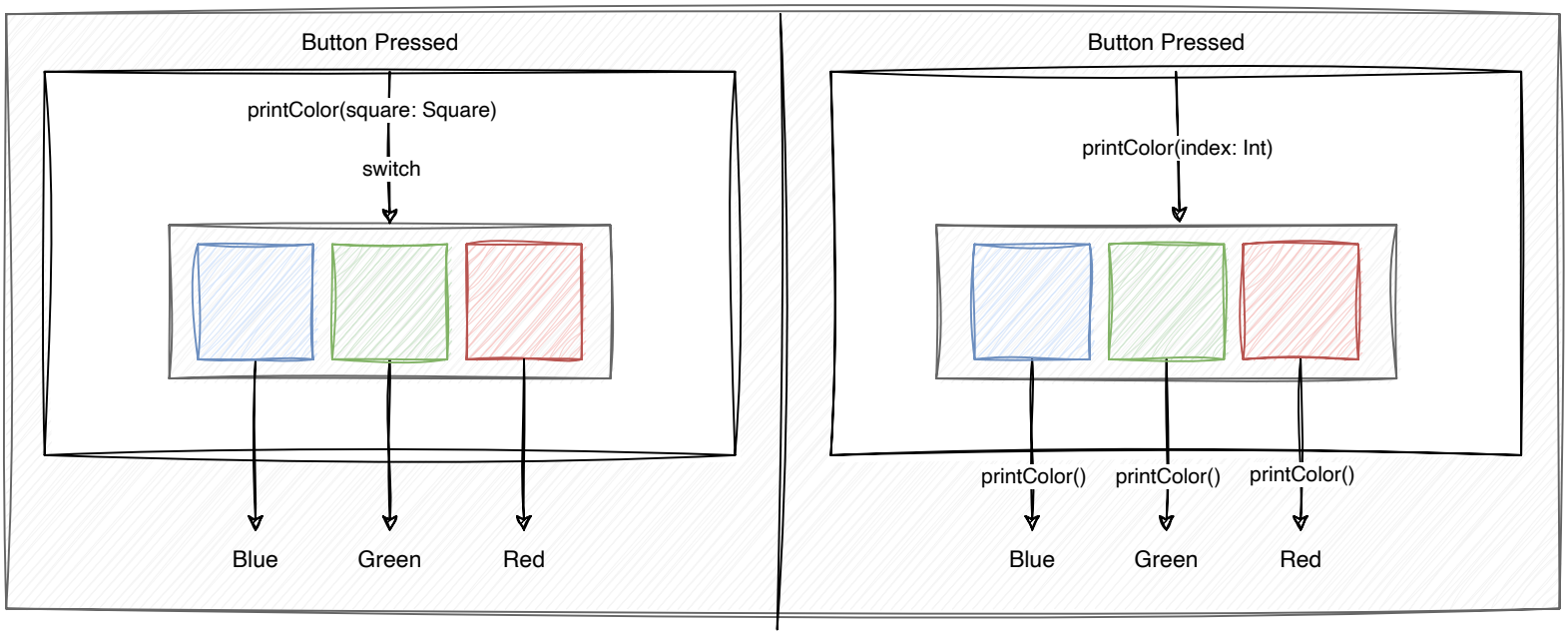
Bir sınıfın tüm özelliklerini özel yapmak, diğer sınıfların bu özelliklere gereksiz yere erişmemesini sağlayacağından OCP'ye saygı gösterilmesine yardımcı olacaktır. Buna OOP'ta kapsülleme(encapsulation) diyoruz. Ayrıca burada belirtilmesi gereken diğer nokta, özelliklerin özel olmasının, söz konusu sınıfın tüm durumları üzerinde daha iyi kontrol edilmesine olanak sağlamasıdır. Örneğin, bir özellik herkese açıksa, bu, diğer sınıfların herhangi bir anda onun durumunu kullanmasına ve değiştirmesine izin verecektir; bu, eşzamanlılıkla başa çıkmanın ve bunun için atomikliği sağlamanın gerekli olduğu gerçeğinin yanı sıra, tüm uygulama boyunca bazı beklenmeyen davranışlara neden olabilir. bazı durumlarda değişkendir.

Global değişkenlerin kullanılmama sebebi de özellikleri özel yapma sebebiyle çok benzer. Global bir değişkene bağlı olan hiçbir modül kapalı olarak değerlendirilemez, çünkü diğer herhangi bir modül bu değişkeni kullanabilir ve durumunu değiştirebilir. Bunun olumsuz etkisi de daha önce de bahsettiğimiz gibi beklenmedik davranışlardır.

Yine de bazen özellikleri özel yapmak veya Global değişkenleri kullanmanın mantıklı olabileceği senaryolar olabilir. Bu sebeple her zaman ciddi bir şekilde bunlar engellenmeye çalışılmamalıdır.

Kısacası

- Modüller uzantılara (extension) açık, değişikliklere kapalı olmalıdır.
- Bu ilkenin uygulanmaması, bir modülde yapılan değişiklik diğer modüllerde de değişik yapılmasına yol açılmasına, geliştiricinin fazla çalışmasına ve bu nedenle bazı modüllerin yeniden test edilmesine sebep olacaktır.
- Bu ilkenin uygulanmaması, diğer modüllerde beklenmedik yeni hataların çıkmasına sebep olacak ve iş kurallarının değiştirilmesini zorlaştıracaktır.
- Protocol kullanımı (Abstraction) genel olarak OCP ihlallerini azaltacaktır.
- Enumlar genel olarak OCP'yi ihlal etmeye yatkındır. Bu nedenle kullanımına dikkat edilmelidir.
- Bir kodun %100 kapalı hale getirilmesi mümkün değildir. Ana değişenlerin ölçümlenerek kapalı hale getirilmeye çalışılması
- önceliklendirilmelidir.



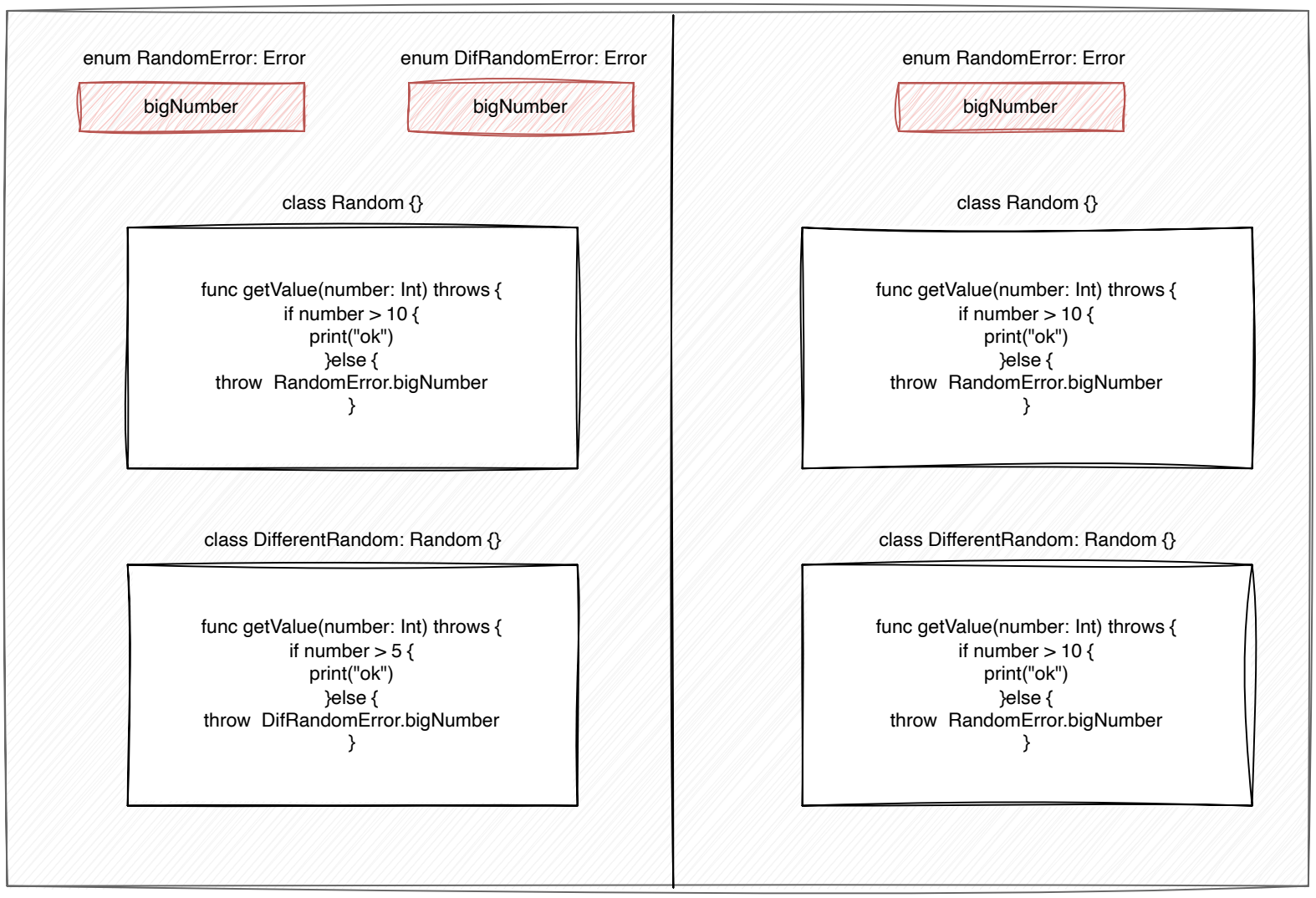
Liskov Substitution Principle

Alt seviye sınıflardan oluşan nesnelerin/sınıfların, ana(üst) sınıfın nesneleri ile yer değiştirdikleri zaman, aynı davranışı sergilemesi gerekmektedir. Türetilen sınıflar, türeyen sınıfların tüm özelliklerini kullanabilmelidir

Bir sınıfta bulunan herhangi bir işlev, kendisinden kalıtım alan sınıflarda kullanılmayacaksa eğer işte bu durum LSP'ye aykırı bir durumdur.

Kısacası

- Çoğu durumda LSP'nin ihlali sonuç olarak OCP'yi de ihlal eder
- Alt sınıflar, üst sınıfın bir yöntemini boş bir yöntemle geçersiz kılmamalı
- Üst sınıftan miras alınan belirli yöntemler istemciler tarafından çağrıldığında sorunlar çıkarıyorsa LSP ihlal edilmiş demektir.
- Alt sınıflar exception throw yapıyorsa



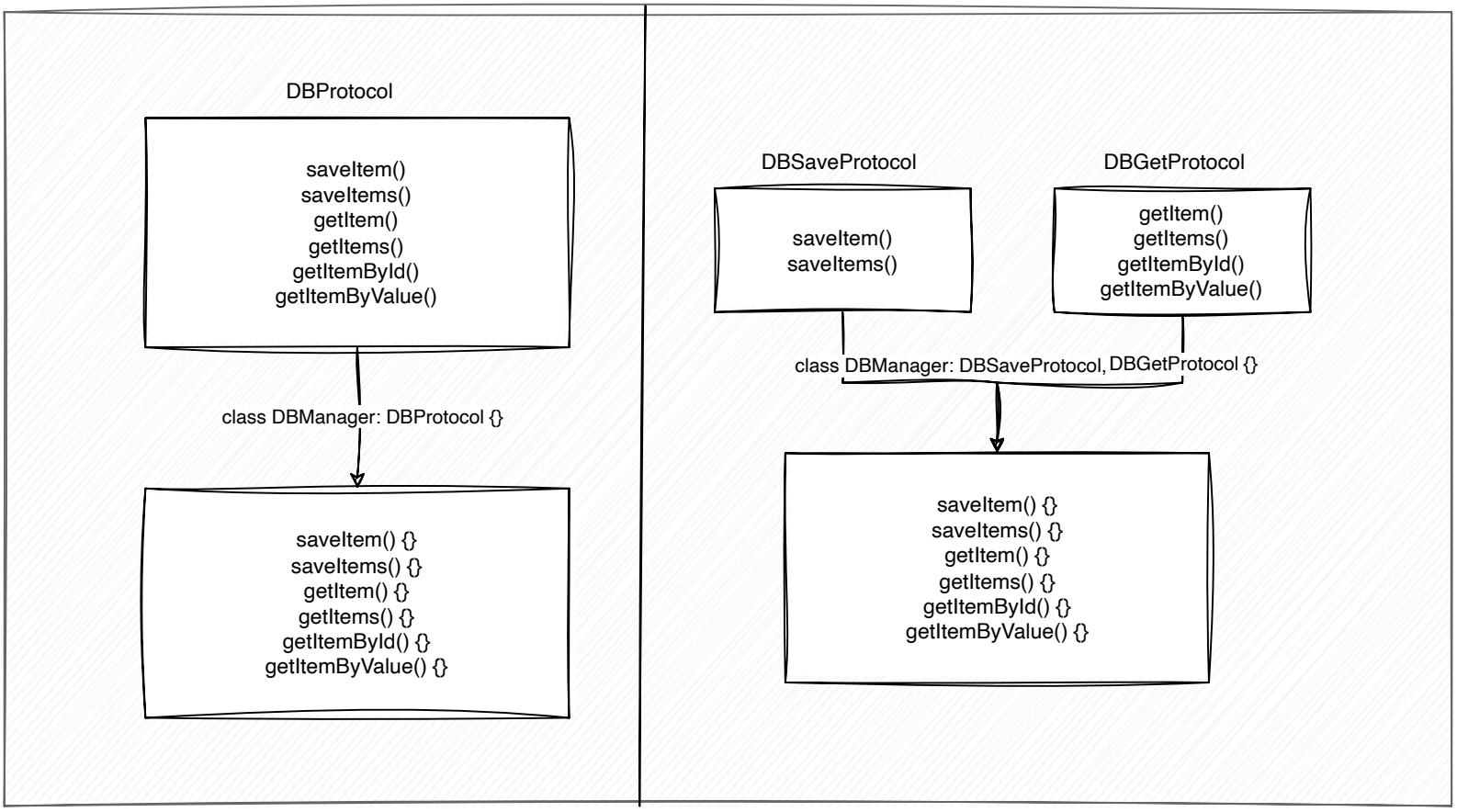
Interface Segragation Principle

İstemciler, kullanmadıkları arayüzlere bağımlı olmaya zorlanmamalıdır.

Bu prensibin amacı, uygulama arayüzlerini daha küçük arayüzlere bölerek daha büyük arayüzler kullanmanın yan etkilerini azaltmaktır. “Fat interface” bize gereksiz yeniden düzenleme, değişiklik yapıldığında yeniden test etme zorunluluğu gibi bazı sorunlar oluşturur. Bunun nedeni, bu tür arayüzlerin fazlaca bağlantı kurması ve bunu uygulayan tüm sınıfların da etkilenmesidir. Bu prensip “fat interfaces” sorununu çözmektedir.

Büyük arayüzleri ayırırken iki adet soru sorabiliriz.

- Bu arayüz ayrımı şu anda işime yarayacak mı?
- Bu arayüz ayrımı ileride bize fayda sağlayacak mı?



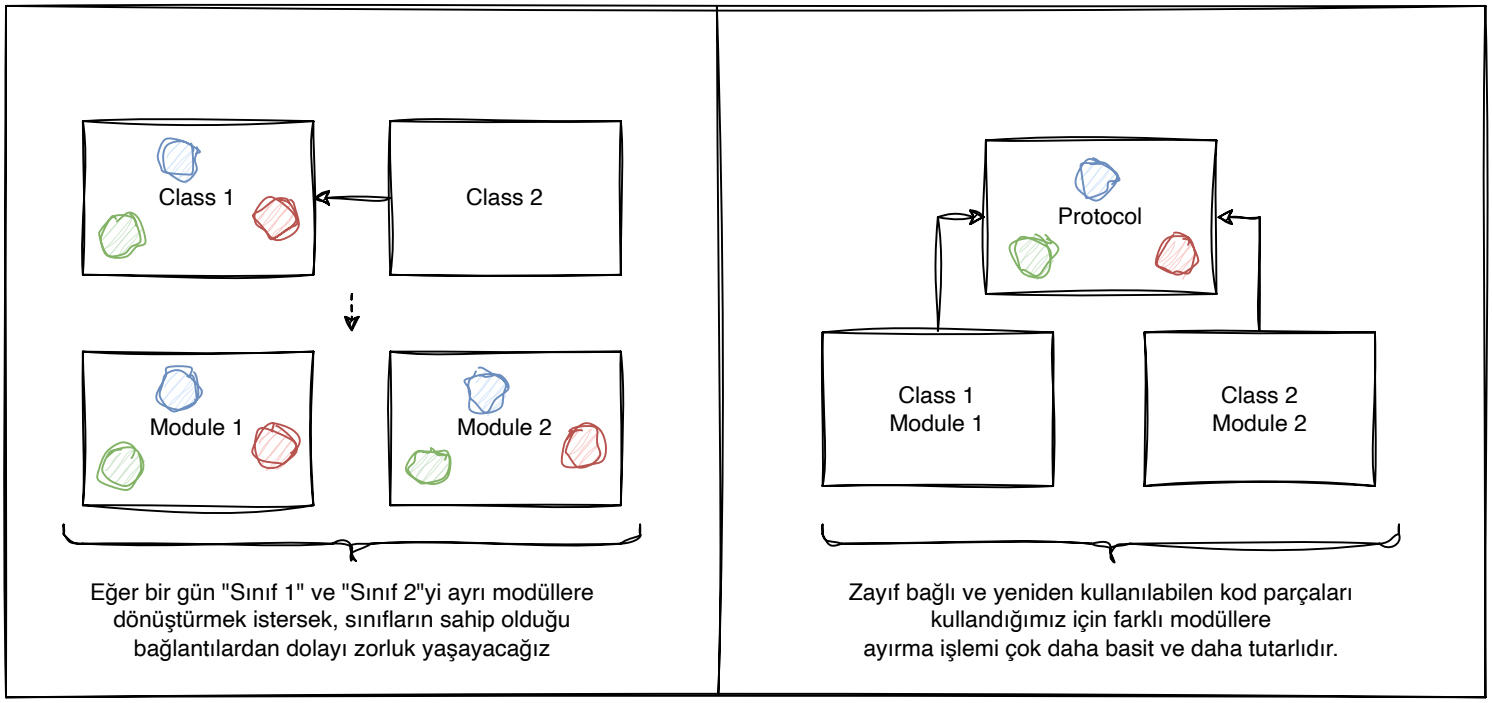
Dependency Inversion Principle

Dependency Inversion prensibi iki ana açıklamaya sahiptir.
A. Yüksek seviyeli modüller, düşük seviyeli modüllere bağlı olmamalıdır. Her ikisi de soyutlamalara dayanmalıdır.
B. Soyutlamalar ayrıntılara bağlı olmamalıdır. Ayrıntılar soyutlamalar üzerine olmalıdır.

Uygulamalar için kod yazarken projeyi birden fazla modüle bölebiliriz. Bununla birlikte, bu bağımlılıkları olan bir kodla sonuçlanacaktır. Dependency Inversion'ın amaçlarından biri bizi sıklıkla değişen modüllere bağlı kalmaktan alıkoymaktadır. Somut sınıflar, arayüzlerin aksine sıklıkla değişirler. Örneğin bug fix, refactor vs.

Dependency Inversion'ın asıl amacı modüllerimizin gerçekten bağımsız olmasına yardımcı olmaktır. Bir uygulamayı bağımsız olarak geliştirmeyi ve dağıtmayı kolaylaştırmaya yardımcı olur.

Eğer doğru şekilde Open - Closed Principle ve Liskov Substitution Principle'ı uygularsanız, modülünüz ayrıca Dependency Inversion Principle'ı da uygulamış olacaktır.



Eğer bir gün "Sınıf 1" ve "Sınıf 2"yi ayrı modüllere dönüştürmek istersek, sınıfların sahip olduğu bağlantılardan dolayı zorluk yaşayacağız

Zayıf bağlı ve yeniden kullanılabilen kod parçaları kullandığımız için farklı modüllere ayırma işlemi çok daha basit ve daha tutarlıdır.

Makalenin sonuna geldiniz. Eğer yazdığım konu hakkında daha fazla bilgi almak isterseniz veya makale ile ilgili muhabbet etmek isterseniz, benimle her zaman iletişime geçebilirsiniz. Düşüncelerinizi paylaşmak, sorular sormak veya daha derinlemesine tartışmak için buradayım. Bana ulaşmak için çekinmeyin! İletişim bilgilerime aşağıdaki kısmından erişebilirsiniz. Teşekkürler.

Zafer Çalışkan
zafer.caliskan@appcent.mobi - caliskan.zafer@outlook.com