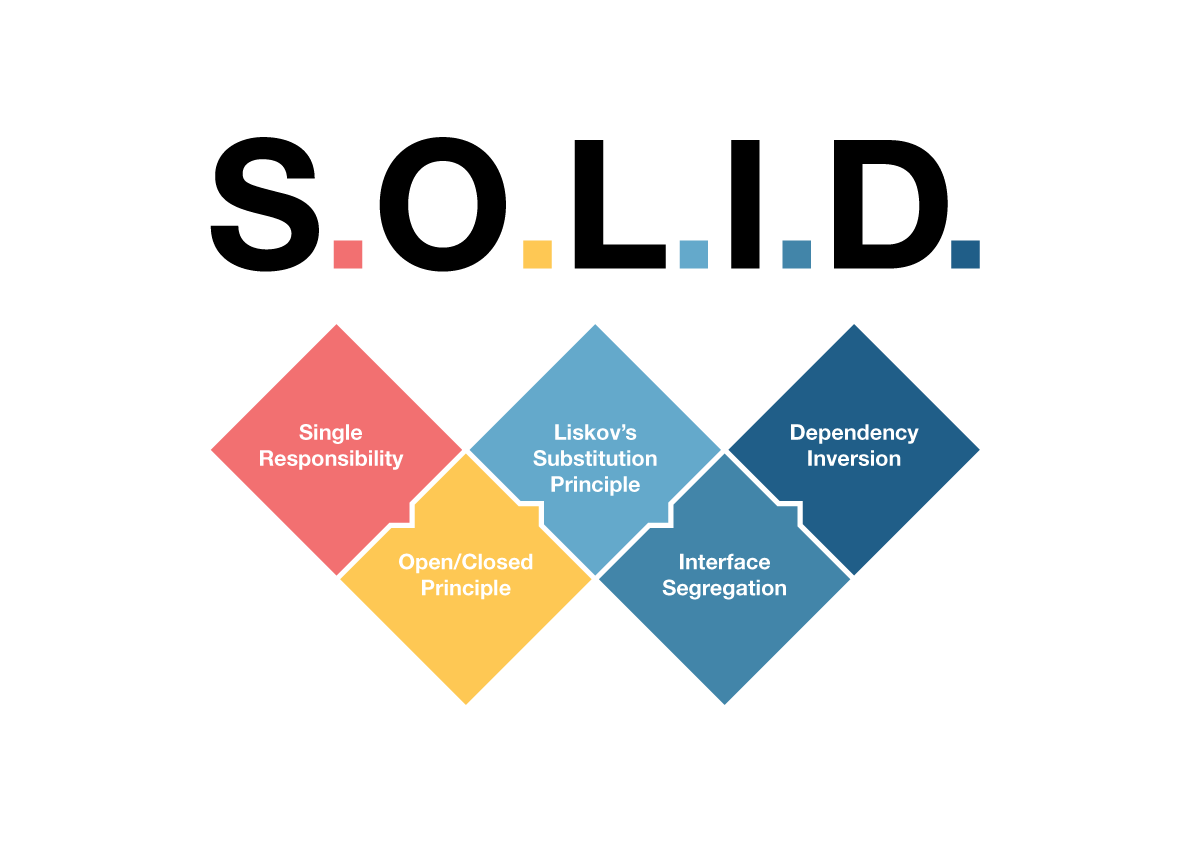
***SOLID PRENSİPLERİ***

Robert C. Martin (Uncle Bob)



* Single Responsibility ==> Tek Sorumluluk Prensibi / İlkesi
* Open Closed ==> Açık Kapalı Prensibi / İlkesi
* Liskov Substitution ==> Liskov’un Yerine Geçme Prensibi / İlkesi
* Interface Segregation ==> Arayüzlerin Ayrışması ve Ayrılması Prensibi
* Dependency Inversion ==> Bağımlılıkların Tersine Çevrilmesi Prensibi

*1. Single Responsibility Principle:*

* Bir sınıfın, bir metodun, her bir iş yapan yapının tek bir görevi olması gerektiğini savunur.
* Bir metod sadece kendi görevini yapmalı, bir sınıf sadece o sınıfla ilgili metodları, alanları, özellikleri barındırmalıdır.
* Birden fazla işle veya katmanla ilişkilendirilmemeli, her bir yapının tek bir sorumluluğu olmalıdır.

Örnek: Bir restoranda tüm işi tek bir kişi yapsaydı nasıl olurdu? Garsonluk, temizlik, yemekler ve bir de hesaba bakmak var tabii. Görseldeki örnek gibi her şey çok karışırdı değil mi? Peki o restoranda çalışan birçok kişi olsa ve herkesin farklı görevleri olup herkes kendi işini yapsa? İşte Single Responsibility Principle (Tek Sorumluluk İlkesi) de bu şekildedir.



Yazılım tarafında örnek verecek olursak her class’ın tek bir tane sorumluluğu olmalıdır diyebiliriz. Product sınıfında product dışında hiçbir class özelliği olmamalıdır. Özetle, SOLID Prensiplerinin S’si her class tek bir sorumluluktan ibaret olmalıdır diyor.

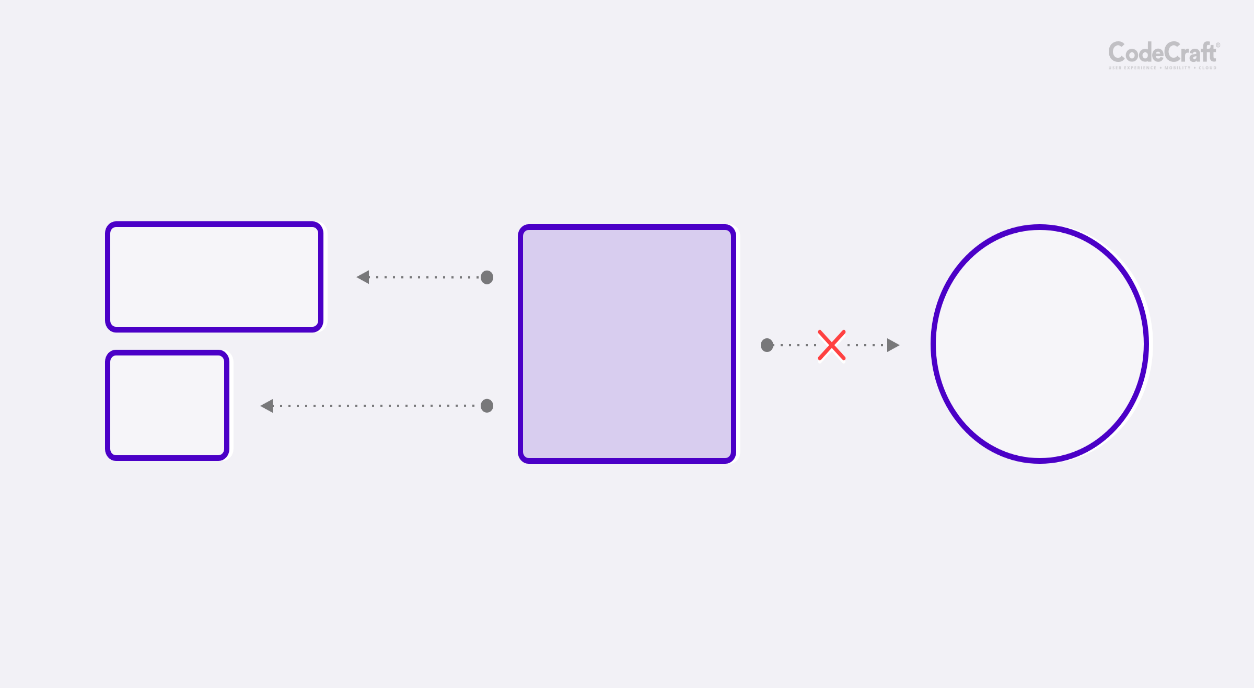
*2. Open Closed Principle:*

* Bir yazılımda yeni bir özellik talep edildiğinde mevcut kodların değiştirilmeden yeni özelliğin geliştirilebiliyor olması gerekmektedir.
* Yeni özellik, geliştirme istediğinde yazılımımız bize direnç göstermemelidir.
* Yazılım ***değiştirilmeye kapalı, geliştirilmeye açık olmalıdır.***

Örnek: Bilgisayarımızda ram yetersiz olduğunda bilgisayarın hiçbir yerini bozmadan işlemci ya da klavye gibi farklı yerlerine dokunmadan sadece ram takılan kısma gelip ekleyebiliyoruz.

Görselde de gördüğünüz gibi kodlarımızın tamamı dikdörtgen ve kareden oluşuyor olsa ona yeni bir özellik eklemek istesek alttaki karenin yanına 1 kare daha ekleyip ortadaki büyük kareyi elde etmiş üstelik bunu hiçbir koda dokunmadan yapmış oluruz.

Fakat diğer yandan kodlarımız ortadaki kareden oluşuyor olsa ve onu Open Closed İlkesine göre yapmamış olsak bir özellik çıkartmak istediğimizde karenin 4 köşesinden de kesmemiz gerekir ki daire şeklini ancak alabilsin..



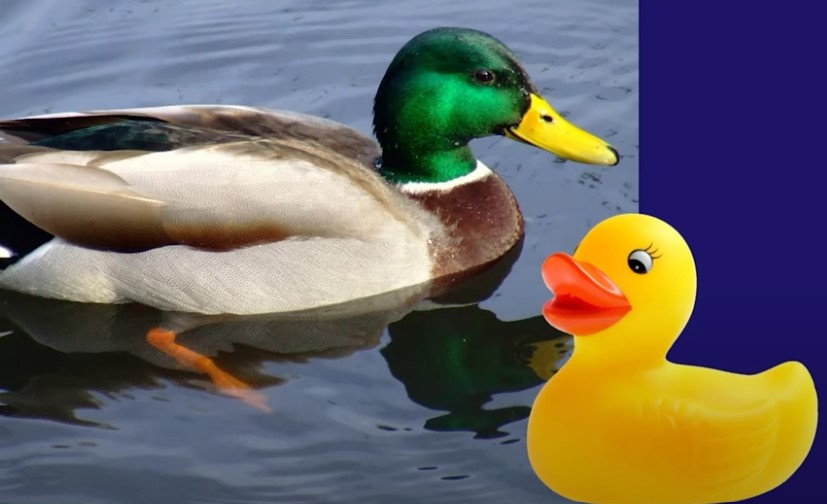
Yazılım tarafında da olay aynen bu şekilde olmalıdır. Yaptığımız programa yeni bir özellik ekleyeceğimiz zaman kodlarımızın hiçbir yerine dokunmadan o özelliği pıt pıt entegre edebiliyor olmalıyız.

*3. Liskov Substitution Principle:*

* Hiçbir değişikliğe ihtiyaç duymadan alt sınıfları, miras aldıkları – türedikleri üst sınıfların yerine kullanabilmeliyiz.
* Alt sınıflar miras aldıkları üst sınıfların tüm özelliklerini barındırıyor olabilirler.
* ***Sadece birbirine benziyor diye sınıflar birbirinden miras (kalıtım) almamalıdır.*** Alt sınıf üst sınıfın özelliklerini, metodlarını mutlaka kullanmalıdır ve yer değiştirdiğinde doğru sonuçları verecek şekilde bozulmadan çalışmalıdır.

Örnek: Yukarıda son paragrafta kalın ve italik yazdığım cümleyi tekrar okumanızı rica edip görselden örnek verecek olursak şöyle açıklayabiliriz: Canlı ve oyuncak ördeğin tabii ki yapabildikleri bazı şeyler kısıtlıdır. Bir metodumuz olduğunu düşünün bu metotta yüzmek, ses çıkartmak ve uçmak olsun. Her 2 ördek ses çıkartabilir oyuncak ördeğe bastığımızda ses çıkartacaktır. 2 ördek de yüzebilir oyuncak ördeği su üzerine koyduğumuzda suyun dibine çökmeyecektir. Fakat uçma konusunda oyuncak olan doğal olarak uçamayacaktır.

Dolayısıyla yazılım kısmında da sırf 2 ördek birbirine benziyor diye miras aldığımız zaman oyuncak ördeğin uçma fonksiyonu bize hata fırlatacaktır ya da çalışmayacak, boş kalacaktır.



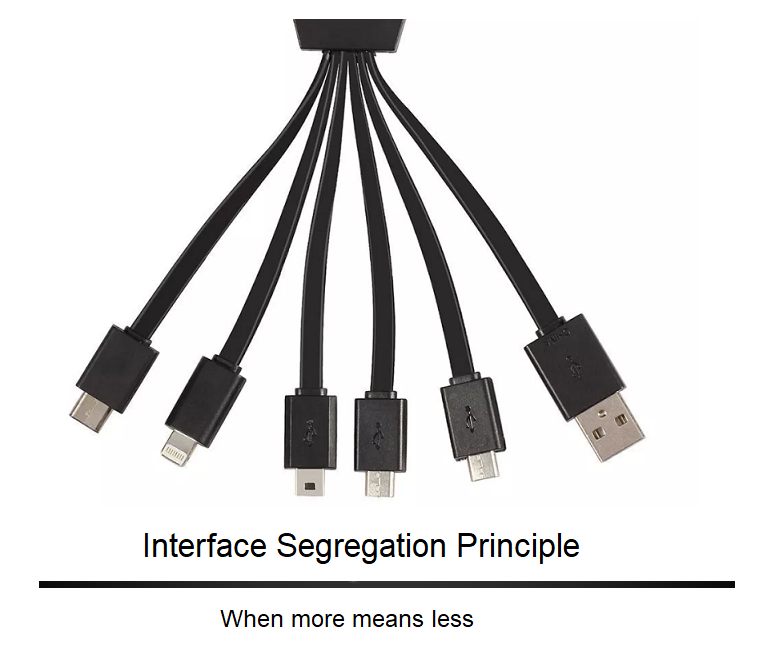
*4. Interface Segregation Principle:*

* Arayüzü implement eden sınıfların gereksiz metod bulundurmaması gerekir.
* Sorumlulukları tek bir arayüze tanımlamak yerine daha özelleştirilmiş birbirinden ayrı arayüzler oluşturmak gerekir.
* Arayüzler birbirinden ayrılmadığında bu arayüzü implement eden sınıflarda gereksiz metodlar veya özellikler olacaktır. Ortak olmayan her bir sorumluluk için ayrı bir arayüz oluşturulması gerekmektedir.

Örnek: Düşünün ki bir araba üreticisisiniz. Arabanıza default (varsayılan) olarak gaza bas, frene bas ve müzik çalar gibi özellikler vereceksiniz. Bu özellikleri bir arayüzde topladınız ve Audi marka bir arabaya implement ettiniz.

Fakat daha düşük model bir araba olan Murat 131’ e de bu arayüzü implement etseniz o arabada müzik çalar özelliği olmadığından müzik çalar kısmına gereksiz, boş düğmeler eklemek zorunda kalırsınız. Yani interfaceler içerisine yazılan şeyleri zorunlu hale getirdiği için bu özelliği desteklemeyen bir arabaya müzik çaları default olarak koymak zorunda olursunuz. Bu durumda Audi’yi ayrı Murat131’i ayrı implement etmeniz gerekli. Ortak olan özellikleri aynı arayüzde ortak olmayanları ise ayrı arayüzlere böl, parçala…

Kısacası, SOLID Prensiplerinin I’sı senin bir özelliğin ortak değilse sen de bu ortak olmayan özelliği ayrıştır diyor.

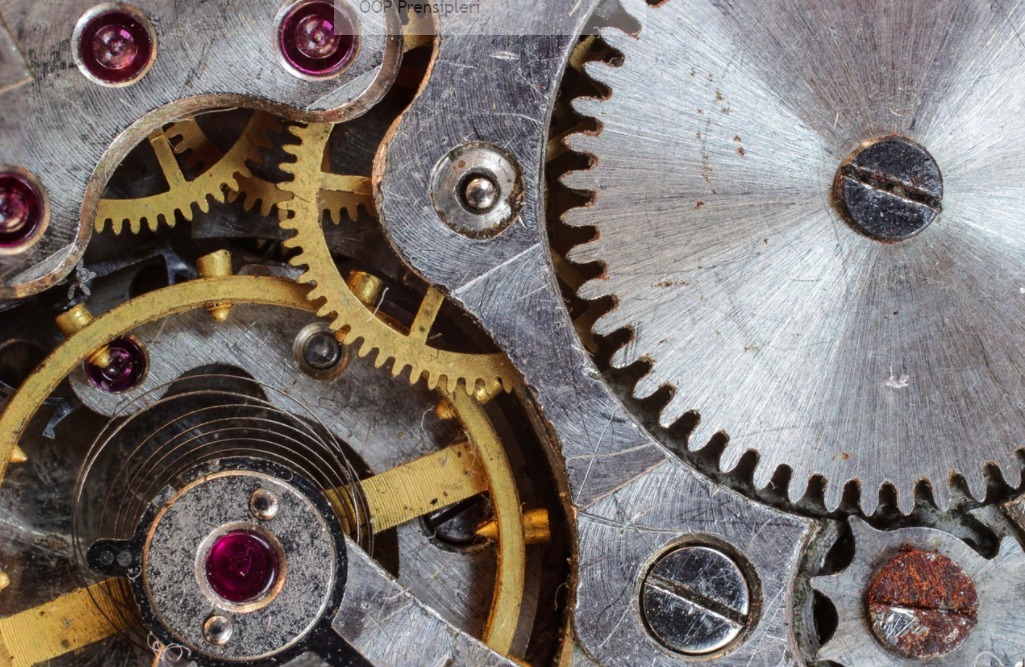


5. Dependency Inversion Principle:

* Sınıflar arasındaki bağımlılık en aza indirgenmelidir.
* Üst sınıflar, alt sınıflara bağımlı olmamalıdır. Tüm somut sınıflar soyutlanma yöntemi ile bağımlılıkları azaltılmalıdır.
* Alt sınıflardaki yapılacak değişiklikler üst sınıfları etkilememelidir.
* Peki, bütün bu sorunlardan kurtulmanın yolu nedir ?  
  Cevap: Yüksek seviye ve düşük seviye sınıflar arasında bir soyutlama katmanı oluşturabiliriz.

**Üst Seviye Sınıflar -> Soyutlama Katmanı -> Düşük Seviye Sınıfları**

Örnek: Mobilyaları zemine sabitlenmiş bir ev olduğunu düşünün hiçbir eşyasını yerinden dahi oynatamadığınız bir ev ne kadar mantıklı olabilir ki? Burada tasarımsal bir problemin olduğu aşikâr. Bir örnek daha vermek gerekirse; evinizdeki ampul patladığında koskoca elektrik tesisatını değiştirdiğinizi düşünün. Yani büyük modül (elektrik tesisatı) küçük modüle (ampul) bağlı olmamalı her ikisi de soyut kavrama (lamba –duy ve ampul-) bağlı olmalı. Üstelik burada, ampulün kaç Watt olduğu (detay), lambanın duy kısmını (soyut) ilgilendirir mi?



Yazılım açısından ve daha teknik bakacak olursak ünümüzde aynı altyapıyı kullanan fakat birden fazla arayüze (web, mobile gibi) sahip olan birçok uygulama var. Burada da altyapı üzerinde çalışan ve iş süreçlerini yöneten katman bizim “yüksek seviye” modülümüzdür. Bu açıdan bakıldığında kullanıcı arayüzünün düşük seviye olduğunu söylememiz mümkün.

***DİĞER YAZILIM PRENSİPLERİ***

DRY (Don’t Repeat Yourself)

* “Kendini tekrar etme” anlamına gelir.
* Kod tekrarlarından sakınılması gerektiğini öneren bir prensiptir.

KISS (Keep it Simple Stupid)

* Basit ve aptalca tutun anlamına gelir.
* Basitlik için çabalamayı öneren bir prensiptir.

YAGNI (You Aren’t Gonna Need It)

* Buna ihtiyacın olmayacak anlamına gelir.
* Gelecekte lazım olacak düşüncesiyle ihtiyacımız olmayacak şeyleri sisteme dahil etmemeyi söyleyen bir prensiptir.