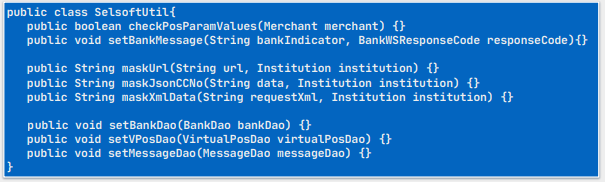
Cohesion

Birliktelik tek bir işin tek bir nesne tarafından yapılması olayıdır(Single Responsiblty). Karmaşık bir yazılım varsa bunlar ayrılmalıdır.

* Karmaşıklık düşüktür
* Bakım Maliyeti düşüktür
* Tekrar kullanıma (reusability) daha yatkındır.
* Netbeans > LowCohesionExample



Coupling

Bir işin veya nesnenin diğerleriyle ne kadar bağımlı olduğu durumudur. Bağımlılık arttığı sürece karmaşıklık artar.

* Bağımlılık arttıkça karmaşıklık artar bakım maliyeti yükselir.
* Geliştirilebilirlik ve Genişletilebilirlik azalır.
* Yazılımın tekrar kullanımı zor hale gelir.
* Netbeans örneğini incele > Engine

Solid Prensipleri

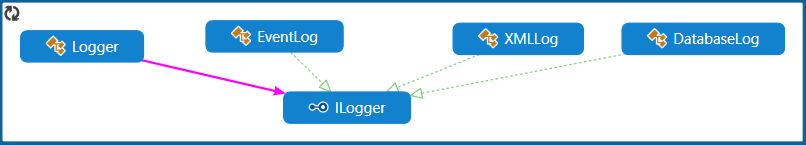
**Single Resposibility(Tek Sorumluluk):** Bir nesneye yalnızca bir görevin verilmesi durumuna denir.

****

Netbeans örneğine bak > FileOperation

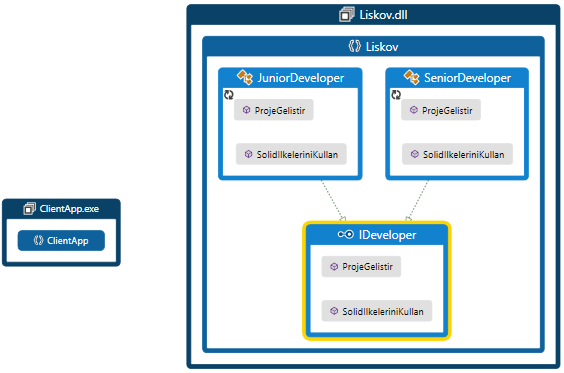
**Open/Closed : Açık kapalı prensibi, yazılım geliştirirken kullandığımız varlıkların (class, method vs.) gelişime açık, kodların ise değişime kapalı olması ilkesidir.**

**Örneğin; bir loglama altyapısı oluşturduğunuzu düşünün, Veritabanına ve XML’e kayıt tutuyorsunuz. Daha sonradan Eventloglara da log tutma ihtiyacı hissettiğinizde, sadece Eventloglara kayıt tutan kodları yazmanız yetecek, kodunuzda hiçbir değişiklik yapmadan bu yapı sisteme entegre olacak. Bunun için uygulayacağımız çözüm şu şekilde olabilir:**

****

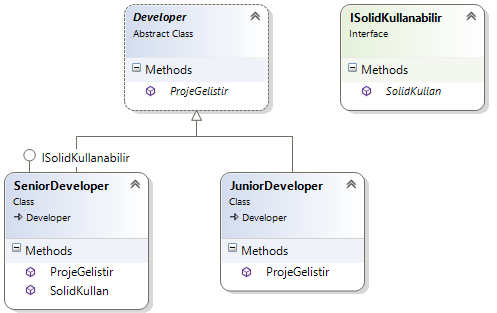
**Netbeans örneği> Rectangle**

**Liskovs Substitution:** Liskov’un yerine geçme prensibi alt sınıflardan oluşturulan nesnelerin üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermek zorunda olduklarını söyler. Yani; türetilen sınıflar, türeyen sınıfların tüm özelliklerini kullanmak zorundadır. Eğer kullanmaz ise ortaya işlevsiz, dummy kodlar çıkacaktır. Bu durumda üst sınıfta if else blokları kullanarak tip kontrolü yapmamız gerekebilir ve böylelikle Açık Kapalı prensibine de ters düşmüş oluruz.

****

Burada IDeveloper interface’ini implemente eden Junior Developer ve Senior Developer class’ları olduğunu görüyoruz. Ancak proje geliştirilirken bir sorun olduğunu farkettik. Junior Developer’ımız solid ilkelerini kullanamıyor  Yapıyı bu şekilde kurmaya devam edersek ya bu methodun içini boş bırakıcaz ya da  NotImplementedException throw edicez ve ClientApp uygulamamız IDeveloper interface’i üzerinden, IDeveloper’dan implemente olan tiplerini kullanıp Açık Kapalı prensibine bağlı kalabilecekken eğer JuniorDeveloper değil ise SolidIlkeleriniKullan() diyen bir if else bloğuna sahip olucak.

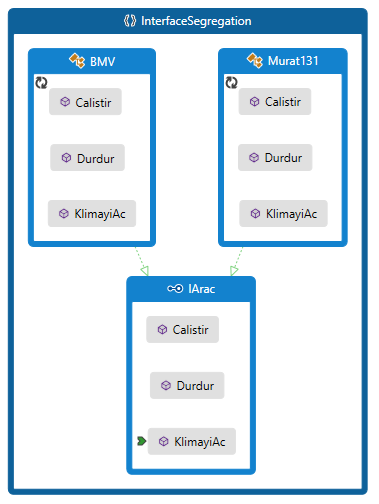
Peki böyle bir durumda çözüm nasıl olabilirdi? Bu yanlış tasarımın önüne nasıl geçeriz?

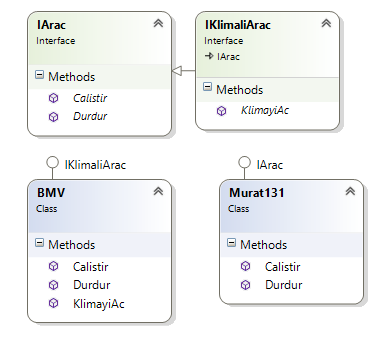


Class Diagramından da anlaşıldığı gibi Developer sınıfını soyut yaptık ve ISolidKullanabilir isimli bir Interface oluşturduk. Böylelikle Liskov prensibine bağlı kaldığımız gibi, oluştuduğum dll’i kullanan Client Uygulaması da if else bloğuna gerek kalmadan Açık Kapalı prensibine uygun olabilecek.

**Netbeans örneği > Shape**

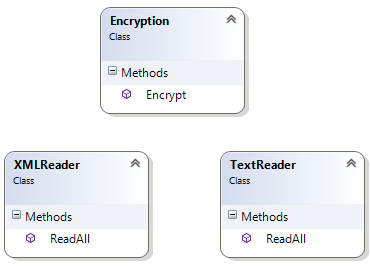
**İntarface Segregation:** Arayüz ayırım prensibi, bir arayüze gerektiğinden fazla yetenek eklemememiz gerektiği söyler.





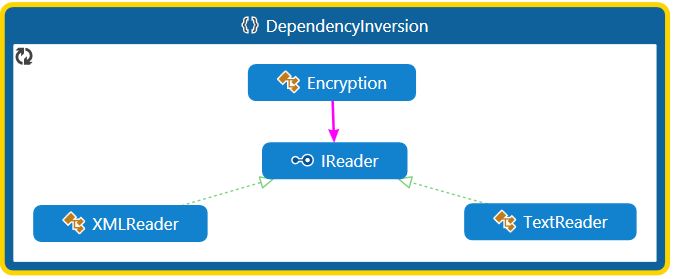
Netbeans örneği > WorkerImpl

**Dependency Inversion Principle:** Bağımlılığın ters çevirilmesi ilkesine göre üst seviye sınıflar, modüller, methodlar vs. alt seviyeli sınıflara bağımlı olmamalıdır. Alt sınıflarda yapılan değişiklikler üst sınıfları etkilememelidir.

****

Oluşturduğum örnekte üst sınıf olan Encryption, alt sınıfları olan XMLReader ve TextReader’dan belirli bir formatta dönen veriyi şifrelemektedir. Reader sınıfları birer nesne olarak Encryption sınıfının içinde kullanılmakta/bağımlı olarak çalışmaktadır. Alt sınıf olan Reader sınıflarından birini uygulama dışında tuttuğumuz anda Encryption sınıfı hata verecektir. Reader sınıflarının ReadAll methodlarında yapacağımız bir kod değişikliği Encryption sınıfının işlevselliğini/kurgusunu rahatlıkla bozabilecektir.

Burada, üst sınıfın alt sınıflara olan bağımlılığı nasıl ters çevirebiliriz? Örneğini verdiğimde ters çevirmekten neyi kast ettiğimizi daha rahat anlayabileceğiz.



Yeni oluşturmuş olduğumuz yapıda, Encryption sınıfının XMLReader ve TextReader sınıflarına olan bağımlılığı, araya IReader interface’i koyularak ters çevrilmiştir. Alt seviyeli olan Reader sınıfları IReader interface’ine bağımlı durumdadır ve içlerinde yapılan değişiklikler Encryption sınıfını kesinlikle etkilemeyecektir. IReader interface’ini implemente edecek FileReader gibi bir sınıf, Encryption üzerinde hiçbir değişiklik yapılmadan kullanılabileceği bir tasarıma da uygun hale gelmiştir.

Netbeans Örnek > PostgreSQLDatabaseService

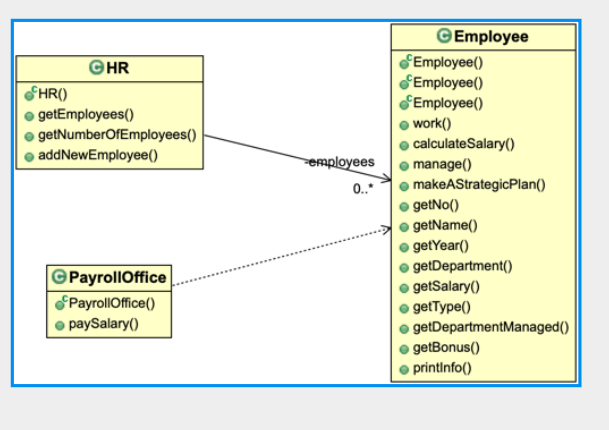
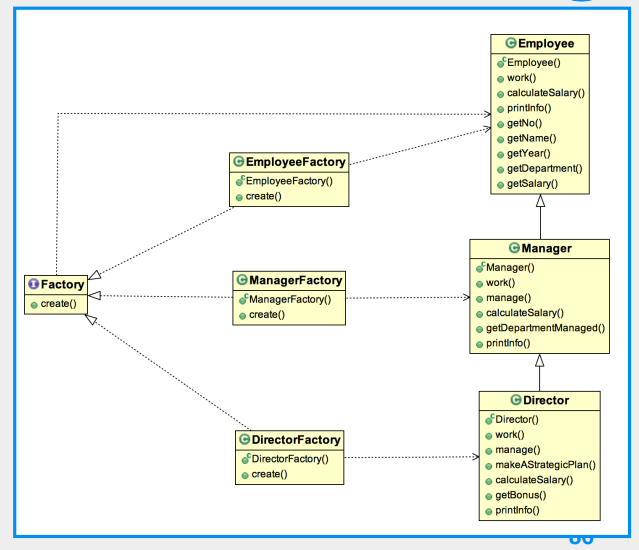
Kaynal : <http://cagataykiziltan.net/solid-prensipleri/>

**Yaratımsal Kalıplar**

**Singleton:** Bir sınıftan yalnızca bir tane nesne yaratmak. Bir sınıfın sadece bir tane nesnesinin olduğundan emin ol ve ona global bir erişim noktası sağla. Bir sınıftan bir tane nesne olmasından kasıt ise, herkesin istediği zaman bu sınıfın bir nesnesini oluşturmaya çalışmaması, oluşturamaması, var olan tek nesneyi kullanmasıdır.

Netbeans örneği > AppConfig

**Factory Method:** Bir nesne yaratmak için bir arayüz tanımla, fakat hangi sınıfın nesnesinin oluşturulacağına alt sınıflar karar versin. Factory Method bir sınıfın nesne oluşturmasını alt sınıflarına ötelenmesine izin verir.

****

Burada bir diğer önemli nokta da, nesne oluşturmak için gerekli durum bilgisinin olabildiğince create() metodunda oluşturması ya da bu metot tarafından elde edilmesidir. Yani karmaşık nesneleri oluşturmak için gerekli parametrelerinin olabildiğince create() metodu tarafından temin edilmesi tercih edilmelidir. Bu durumda create() metoduna parametre geçmeye gerek kalmayabilir ya da daha az geçilir.

Netbeans Örneği > Client