Cohesion : Nesneye dayalı programlamada sıınf tasarımı yaparken tanımlanan değişkenlerin ve metotların birbiriyle ne kadar sıkı şekilde bağlantılı/ilgili olduğunu tanımlar. Yüksek kohezyona sahip sınıflar iyi tasarlanmış sınıflardır. Kohezyonu yüksek olan sınıfların fonksiyonel birliktelikleri üst seviyededir. Bu sınıflardaki değişkenler birbiriyle ilişkilidir : sınıf içindeki metotlar üye değişkenleri birlikte kullanır.

Coupling : Coupling nesneye dayalı programlamada bir sınıf diğer sınıflara ne kadar bağımlı olduğunu tanımlar. Bu bağımlılık çok fazlaysa : sınıfta olan değişiklikler daha çok sayıda sınıf etkiler ; bağımlılık azsa sınıftaki bir değişiklik daha az sınıfı etkiler yada hiçbir sınıfı etkilemez. Yazılımın doğası gereği sınıfların birbiriyle iletişimi olması gerektiği için couplingi yani bağımlılığı sıfıra indirmek çoğu zaman mümkün olmasa da bağımlılığı olabildiğince düşük seviyede tutacak şekilde tasarımlar yapmaya çalışırız.

Open/Closed Principle

Yazılımların genişlemeye açık(open), değişmeye(kapalı) olması gerektiğini söyler : bir sınıfa yeni bir özellik eklendiğinde sınıfın var olan diğer özelliklerinde bir değişiklik söz konusu olmamalıdır. Örneğin aldığı string türünden değer göre işlem yapan bir metota yeni bir özellik eklemez istediğimizde string'in yeni değeri için yeni bir kontrol ifadesi ve işi yapacağı kodları eklememiz gerecektir. Burada yeni bir özellik eklemek için varolan yapıyı bozmuş oluruz. Ayrıca bir metotta stringin değerine göre farklı işlemler yapıldığı için Single Responsbility Prensibine aykırı bir durum da vardır. Değişime kapalı, gelişime açık sınıflar tasarlayabilmek için soyutlamaları kullanmamız gerekir.

Liskov Substitution Principle

Sırf birbirine benziyor diye farklılık arzeden ve tasarımda uyuşmazlık oluşturabilecek nesneleri birbirinin yerine kullanılmaması gerektiğini söyler. Sınıflardaki üye değişkenler/özellikler bu uyuşmazlığa neden olabileceği gibi metotlar da olabilir.

Alt sınıflar üst sınıftan kalıtım aldığı metotları override ederken yeni kısıtlar eklememelidir, istemci base sınıfa ait referans üzerinden alt sınıflara erişirken hangi alt sınıfa ait referansı tuttuğunu(upcast ettiğini) bilme zorunluluğunda olmamalıdır : instance of gibi yapılarla çalışma anında tip kontrolü yapılıp buna göre işlem yapmamalıdır. Bu yüzden bir sınıf bir yerden kalıtım alarak ekstra dikkatli olunmalıdır. Çünkü istemci elinde sadece base sınıfa ait referansa sahip olacaktır neyi upcast ettiğini bilmez: Base sınıfa ait referansla metot çağırımları yapılacak ve burada yapılan işlemlere, buradan dönen değerlere/exceptionlara göre client tarafında aksiyomlar alınacaktır. Base sınıfın metotlarını override eden alt sınıflar bu metotlara yeni kısıtlar eklemişse client bundan habersiz olacaktır ve buna göre aksiyom veremeyecektir. Örneğin : üst sınıftaki metotta döndürülmeyen bir exceptionun alt sınıftaki metotta döndürülmesi; üst sınıfta normal bir şekilde hesaplanıp dönülen değerin alt sınıftaki metotda özel durumlara göre farklı değerler döndürmesi.

Bunlar yanında alt sınıf üst sınıftaki metotun döndürdüğü tipin daha alt tipini döndürebilir. Örneğin üst tip Çalışan tipinden bir nesne döndürüyorsa alt tip bundan kalıtım alan İşçi tipinden bir referans döndürebilir. Client sadece üst tipin döndürdüğü tipi bildiği ve buna göre işlem yapacağı için, alt tipin

burada üst tipin döndürdüğü tipten daha alt hiyerarşide bir tip döndürmesi sorun yaratmayacaktır. Client burada da upcasting yaparak bu dönen değeri yakalayarak kullanabilir.

Kısaca bu prensip istemcinin üst sınıftaki metotların vermesini beklediği aksiyomların alt sınıfta override edilen metotlarda da beklemesidir(metot kısmı için). Alt sınıflar üst sınıftaki metotun yaptığı işin daha spesifik/özel bir şeklini yapabilir fakat client bu farklılıktan etkilenmemelidir, üst sınıfın vereceğini bildiği aynı tepkilerin aynısı alt sınıflardaki metotlar için de geçerli olmalıdır.

Interface Segragation Principle

Bu prensip tasarlanan arayüzlerin olabildiğince daha küçük bir işi gruplaması gerektiğini söyler. İçinde bir çok alt role sahip ve bu alt role ait işlemleri gerçekleştiren bir sınıf olduğunda bu sınıftaki her alt rol dolayısıyla metot grupları için farklı arayüzler tasarlamalıyız. Tasarladığımız arayüzler birden fazla alt rolü birlikte kapsamamalıdır : böyle bir durumda istemci bu fazlaca alt rolü kapsayan(şişman arayüz) arayüzler ile işlemlerini gerçekleştirmeye çalıştığında arayüzü kullanmak istediği rol dışındaki diğer rollere ait metotlardan soyutlanamaz. Bunun yanında sınıf tasarımı yaparken oluşturulan arayüzler eğer şişman olarak tasarlanırsa tekrar kullanımı düşecektir : şişman bir arayüzü bir başka sınıf uygulamaya çalıştığında bu sınıf buradaki farklı rollere ait metotların hepsini implement etmek zorunda kalacaktır. Bu rollerden bazıları bu sınıfla alakalı olmayan roller olabilir : bu durumda sınıf bu roller ait metotları boş bir şekilde/hata fırlatacak şekilde implement etmek zorunda kalacaktır. Bunun sonucu olarak istemci hatasız bir program yazmak istiyorsa çalışma anında tip dönüşümü yapmalı ve tipe göre işlem yapmalıdır. Bu da Liskov Substitution Prensibine aykırı bir durumdur, kötü tasarımdır.

Dependecy İnversion Principle

Yazılımda yüksek seviyeli modüller daha alt seviyedeki modüllere direkt olarak bağımlı olmak yerine onları kapsayan arayüzlere bağımlı olmalıdır. Aksi bir tasarımda alt seviyeli bir modüldeki küçük bir değişiklik onun bir üst seviyesindeki değişikliği o da daha üst seviyedeki bağımlıları etkileyecek şekilde devam edecektir. Bağımlılıklar yazılımın doğası gereği sıfıra indirilemez; indirilirse yazılım denen şey olmaz. Fakat bu bağımlılıkları olabildiğince düşük yapmak değişime daha açık yazılımlar geliştirilmesini sağlar. Dependecy Inversion Prensibi de bunu yapar.