SOLID Prensipleri

- Solid prensipleri ile ilgili bilmemiz gereken ilk nokta gelecekte karşımıza çıkacak işlemler için kodumuzun kalitesinin artmasında fayda sağlayacak prensiplerdir. Başarılı bir şekilde uygulayabilirsek gelecekte karşımıza çıkacak değişikliklere en iyi şekilde entegre olmak SOLID ile daha kolay olacaktır.

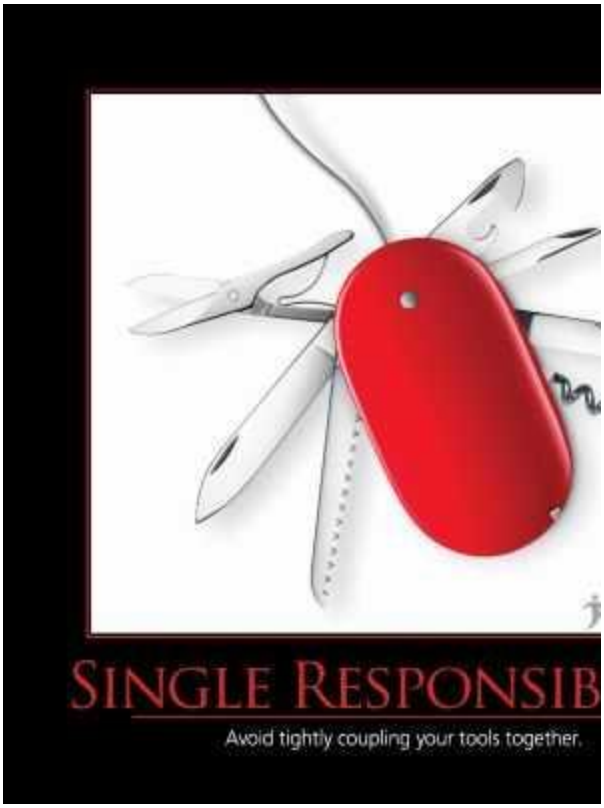
Single Responsibility Prensibi

- Bir şeyi yapabiliyor olmamız her şeyi yapmamız gerektiği anlamına gelmemektedir. Metot yazarken de bu yaklaşımla ilerlemeliyiz. Bir metotun bir sorumluluğu olsun sadece bir iş yapsın mantığında ilerlemek gün sonunda daha doğru olabilir.
- "Single Responsibility Principle" (SRP), yazılım geliştirme alanında SOLID prensiplerinden biridir. SOLID prensipleri, yazılım tasarımındaki temel prensipleri düzenler ve sürdürülebilir, esnek ve bakımı kolay sistemler oluşturmayı amaçlar. Single Responsibility Principle (Tek Sorumluluk Prensibi), Robert C. Martin'in belirttiği beş SOLID prensibinden biridir. Bu prensip, bir sınıfın yalnızca bir nedenle değişebileceğini ve bir sınıfın sorumluluklarının tek bir konsepte odaklanması gerektiğini savunur. Bu, bir sınıfın sadece bir tür görevi veya sorumluluğu olması gerektiği anlamına gelir. İdeali, bir sınıfın değişiklik nedeni sadece bir tane olmalıdır. Eğer bir sınıf birden fazla sorumluluğu üstlenirse, bu sınıfın herhangi bir sorumluluğundaki bir değişiklik diğer sorumlulukları etkileyebilir ve bu da kodun karmaşıklığını ve bakım zorluğunu artırabilir.

SRP'nin avantajları şunlar olabilir:

1. Daha İyi Bakım ve Anlaşılabilirlik: Sınıfların tek bir sorumluluğa odaklanması, kodun daha anlaşılabilir ve bakımı daha kolay olmasını sağlar. Bu sayede, bir sınıfın içindeki değişikliklerin diğer kısımları etkileme olasılığı azalır. **2. Daha Az Bağımlılık:** Sınıflar arasındaki bağımlılıkların azaltılması, kodun daha esnek ve değiştirilebilir olmasını sağlar. Bir sınıfın içinde yapılan değişiklikler, diğer sınıfları minimum düzeyde etkiler. **3. Kodun Tekrar Kullanılabilirliği:** Sınıfların tek bir sorumluluğa odaklanması, bu sınıfların daha genel amaçlı ve yeniden kullanılabilir olmasını sağlar. Bir işlevselliği değiştirmek istediğinizde, sadece ilgili sınıfı değiştirmeniz yeterli olacaktır.

Örneğin, bir sınıfın hem veritabanı işlemlerini yönetmesi hem de kullanıcı arayüzü işlemlerini gerçekleştirmesi durumunda, bu sınıf SRP'ye uymuyor demektir. İdeal olarak, bu sorumluluklar iki ayrı sınıfa ayrılmalıdır.



Single Responsibility

Robert C. Martin

Open Closed Prensibi

- Yazdığımız kodların geliştirmeye açık olmalıdır ancak değişime kapalı olmalıdır.
- "Open/Closed Principle" (OCP), SOLID prensiplerinden biridir ve yazılım tasarımındaki iyi bir uygulama pratiğini temsil eder. Bu prensip, **Bertrand Meyer** tarafından ortaya atılmıştır ve yazılım tasarımında esneklik ve genişletilebilirlik sağlamayı amaçlar.
- Open/Closed Principle (Açık/Kapalı Prensibi), şu şekilde ifade edilir: **"Bir yazılım birimi (sınıf, modül, fonksiyon, vb.) genişletmeye açık, değişikliğe kapalı olmalıdır."** Bu ilkeye göre, bir yazılım birimi, yeni özellikler eklemek veya davranışını değiştirmek için açık olmalı, ancak mevcut davranışlarını değiştirmek için kapalı olmalıdır.
- Bu prensip, yazılım birimlerinin (genellikle sınıfların) uzantılarına açık, ancak mevcut kodlarında yapılan değişikliklere kapalı olması gerektiğini savunur. Yani, yeni bir özellik eklemek veya davranışı değiştirmek, mevcut kodu değiştirmek yerine, var olan kodu uzatarak yapılmalıdır.

1. Genişletilebilirlik: Yazılım birimleri, yeni gereksinimlere veya özelliklere uyum sağlamak için kolayca genişletilebilir. Yeni bir özellik eklemek istendiğinde, var olan kod değişmeden yeni bir kod eklenerek genişletme yapılabilir. **2. Daha Az Riskli Değişiklikler:** Mevcut kodun değişmeden kalması, hali hazırda çalışan sistemi etkileme riskini azaltır. Bu, birim testlerin daha güvenilir olmasına ve kod değişikliklerinin daha güvenli bir şekilde yapılmasına olanak tanır. **3. Daha Yüksek Uyum Sağlama:** OCP, birimlerin daha iyi bir şekilde uyum sağlamasını sağlar. Yeni özellikler eklemek, mevcut kodu değiştirmeksizin gerçekleştirilebileceği için, yazılım birimleri daha esnek ve yeniden kullanılabilir olabilir.

OCP, diğer SOLID prensipleriyle birlikte kullanıldığında, daha modüler, sürdürülebilir ve esnek yazılım tasarımları elde etmeye yardımcı olabilir.

Open Closed

Bertrand Meyer

Liskov Substitution Prensibi

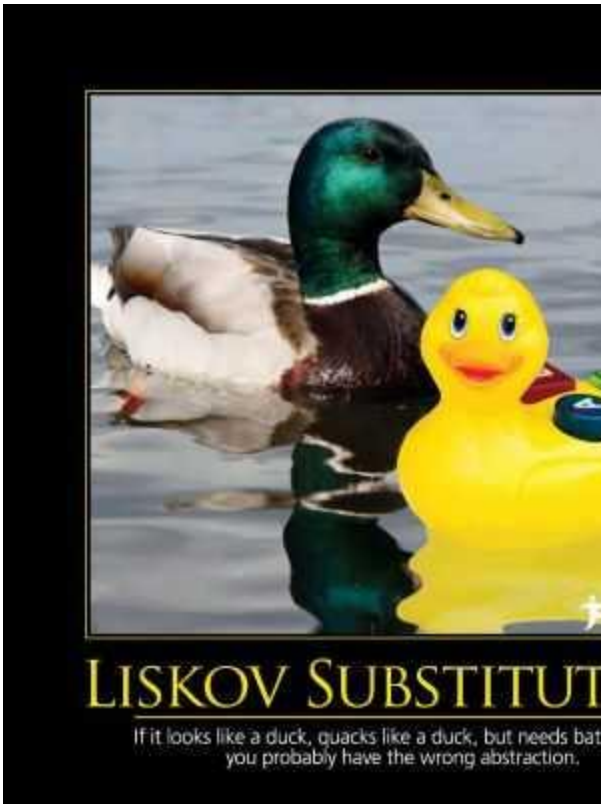
- Bu prensip bize der ki bir class inherit edildiği class gibi davranamıyorsa burada bir ihmal vardır. Abstraction ile ilgili yaptığımız bir sorun var demektir.

- Elimizdeki class inherit ettiğimiz class gibi davranıyorsa ve bu class içerisindeki bütün metotlara sahipse ve kullanabiliyor ise herhangi bir sorun yoktur. Çünkü ola ki bir durumda ben inherit edilen class yerine inherit eden classı kullandığımda sıkntı çıkmaması gerekiyor.
- LSP'nin ana ilkesi, bir türetilmiş sınıfın, temel sınıfın yerine geçebilmesi ve aynı arayüzü sağlaması gerektiğidir. Bu prensip, **Barbara Liskov** tarafından 1987 yılında **"A Behavioral Notion of Subtyping"** adlı makalede ortaya konulmuştur. LSP'nin temel ifadesi şu şekildedir: **"S, T'nin bir alt türü ise, T türünden nesnelerin yerine S türünden nesneler kullanılabilir olmalıdır."**

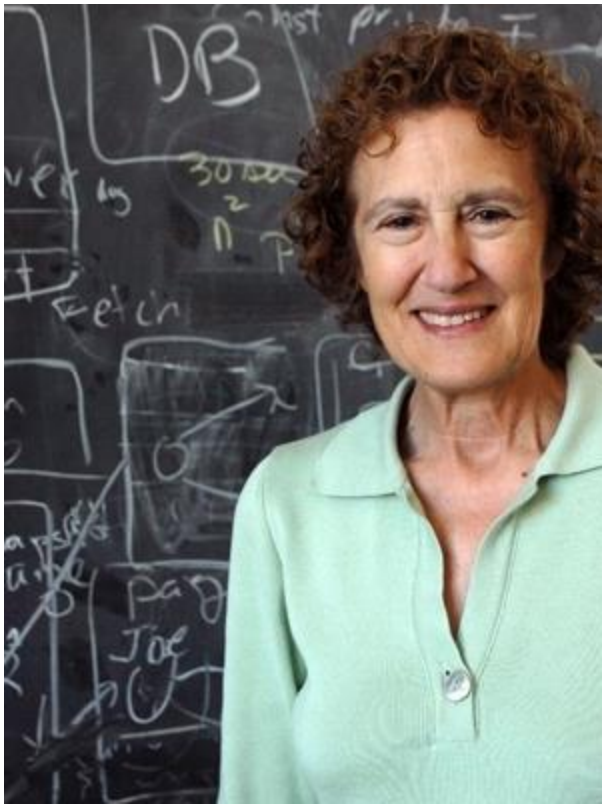
1. Alt Sınıfların Özel Durumlar Oluşturmaması: Alt sınıflar, temel sınıfların davranışlarını değiştirmemeli ve temel sınıfların yerine geçebilmelidir. Alt sınıflar, temel sınıfların kullandığı arayüzleri aynı şekilde desteklemelidir.

2. Miras Alan Sınıfların Davranışlarını Genişletme: Alt sınıflar, temel sınıfların davranışlarını genişletebilir ancak değiştiremez. Yani, temel sınıflardan gelen özellikleri kullanabilir ve ihtiyaç duydukları özellikleri ekleyebilirler.

LSP'nin bu kuralları, türetilmiş sınıfların kullanımının güvenli olmasını sağlar ve kodun daha öngörülebilir olmasına katkıda bulunur. Bu prensip, polymorphism (çok biçimlilik) konseptiyle de ilişkilidir. Eğer bir sınıf, bir üst sınıfın yerine geçebiliyorsa, bu durumda polymorphism kullanılabilir ve bu nesneleri bir arayüz üzerinden kullanmak daha kolay hale gelir. LSP'nin amacı, türetilmiş sınıfların kullanımını güvenli ve sorunsuz hale getirerek, yazılımın esnek ve sürdürülebilir olmasına katkıda bulunmaktır.



Liskov Substitution



Barbara Liskov

Interface Segregation Prensibi

Interface Segregation Principle" (ISP), SOLID prensiplerinden biridir ve arayüzlerin (interfaces) mümkünse mümkün olduğunca özel olması gerektiğini savunur. Bu prensip, bir sınıfın kullanmadığı metotlara sahip bir arayüzü uygulamamasını önerir ve müşterilerin yalnızca ihtiyaçları olan metotları içeren küçük arayüzleri kullanmalarını teşvik eder. ISP'nin temel prensibi şu şekildedir: **"Bir sınıf, ihtiyaç duymadığı metotları içeren bir arayüzü uygulamamalıdır."** Bu ilkeye göre, bir sınıf yalnızca kendi ihtiyaçlarına uygun olan metotları içeren arayüzleri uygulamalıdır. Böylece, bir sınıfın gereksinim duymadığı metotlarla bağlantılı olması ve bu metotları boş bir şekilde implemente etmesi önlenmiş olur.

ISP'nin avantajları şunlar olabilir:

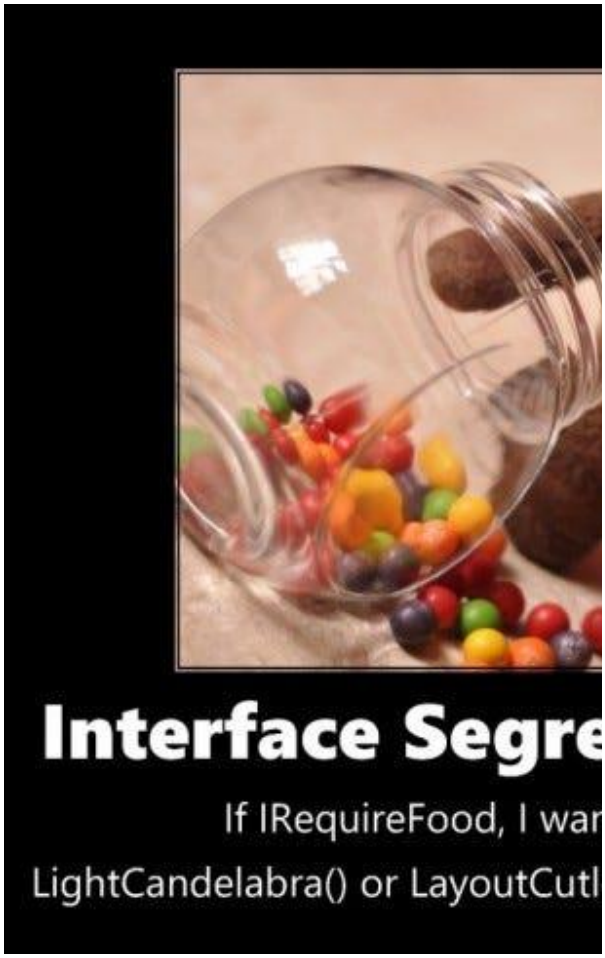
1- Daha Az Bağımlılık: Bir sınıf, yalnızca ihtiyaç duyduğu metotları içeren küçük arayüzleri uygulayarak, diğer sınıflara olan bağımlılığını azaltabilir.

2- Esneklik ve Bakım Kolaylığı: İhtiyaç duyulan metotları içeren küçük arayüzler, kodun daha esnek ve bakımı daha kolay olmasını sağlar. Bu sayede, değişiklikler yapıldığında sadece ilgili sınıflar etkilenir.

3- Kodu Anlama ve Kullanma Kolaylığı: Müşteriler, yalnızca ihtiyaç duydukları metotlara sahip arayüzleri kullanarak kodu daha iyi anlayabilir ve kullanabilirler.

Örneğin, bir arabayla ilgili bir arayüzde "Uçak Modu" gibi uygulanması gereksiz veya kullanılmayacak metotlar varsa, bu arayüzü implemente eden araba sınıfları için bu metotlar anlamsız olacaktır. Bu durum, ISP'nin ihlali olarak kabul edilebilir ve bu tür durumlar kaçınılmalıdır. Sınıflar, ihtiyaçlarına uygun ve anlamlı metotları içeren arayüzleri uygulayarak bu prensibi takip etmelidir.

Interface Segregation



Interface Segregation

Dependency Inversion Prensibi

- Daha üst seviyede bir class alt seviyedeki bir classa bağımlı olmamalı. Çünkü bu bağımlılıkların sayısı arttıkça bunlardan kurtulma olasığımız azabilir ve karmaşıklığa yol açılabilir. Dependency Inversion prensibi, SOLID prensiplerinden biridir ve yazılım tasarımında kodun esnekliğini ve sürdürülebilirliğini artırmayı amaçlar.

Dependency Inversion prensibi, yüksek seviyeli modüllerin düşük seviyeli modüllere bağımlı olmamalıdır. İki modül de soyutlamalara bağlı olmalıdır.

Dependency Inversion prensibi üç ana ilkeye dayanır:

1- Yüksek Seviyeli Modüller (High-Level Modules) ve Düşük Seviyeli Modüller (Low-Level Modules)

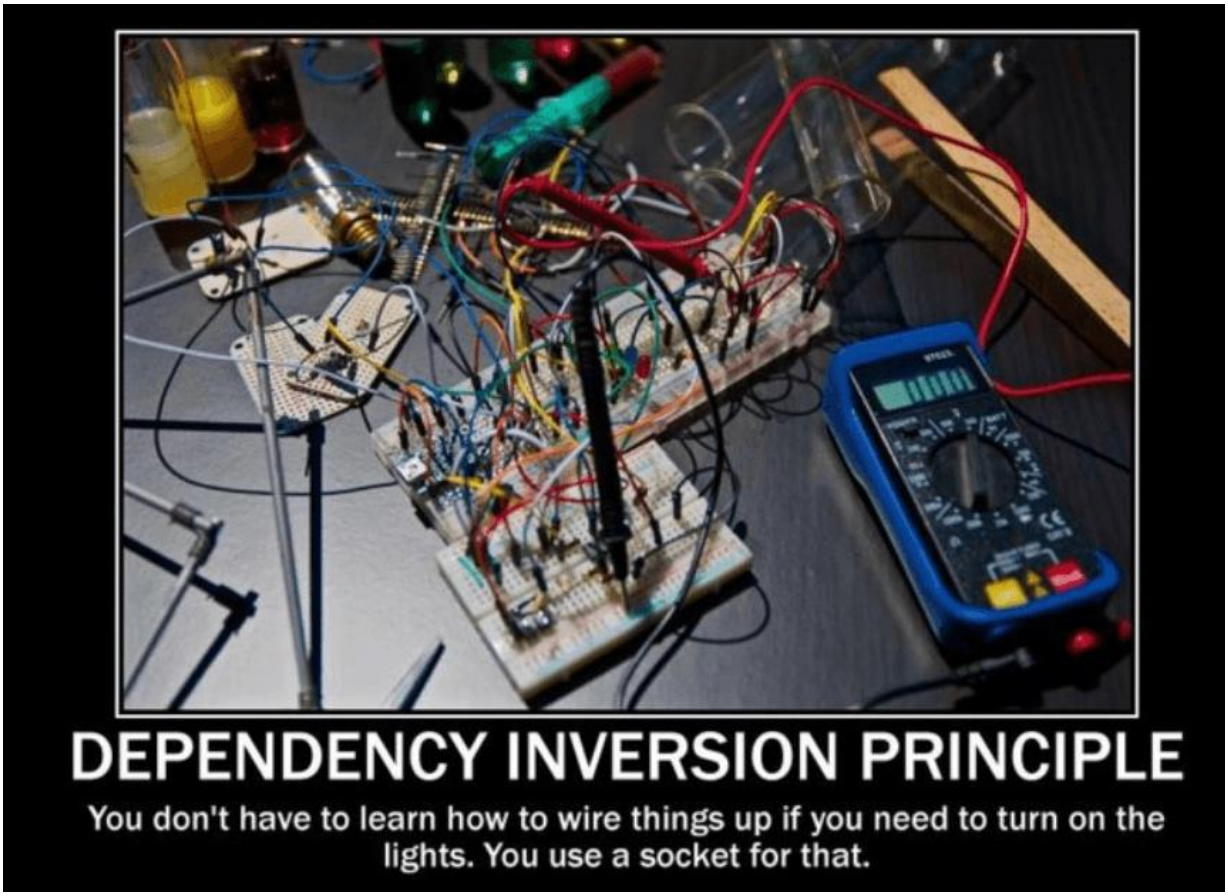
- Yüksek seviyeli modüller, uygulamanızın temel iş mantığını oluşturan ve genellikle daha soyut düzeydeki modüllerdir.
- Düşük seviyeli modüller, daha spesifik ve uygulama detaylarına odaklanan modüllerdir.

2- Soyutlamalara Bağımlılık (Dependency on Abstractions)

- Yüksek seviyeli modüller, düşük seviyeli modüllere doğrudan bağımlı olmamalıdır. Bunun yerine, her iki seviyedeki modüller de soyutlamalara (interface veya abstract class gibi) bağımlı olmalıdır.
- Bu, modüller arasında gevşek bağlantılar sağlar, çünkü yüksek seviyeli modül düşük seviyeli modülü somut bir uygulama yerine soyut bir kavram üzerinden kullanır.

3- Çalışma Prensibi

- Yüksek seviyeli modüller ve düşük seviyeli modüller, her ikisi de soyutlamalara bağımlı olduklarından, değişiklikler bir modülde yapıldığında diğerini etkilemez.
- Bu prensip, kodun değişikliklere daha dayanıklı, esnek ve sürdürülebilir olmasını sağlar.



Dependency Inversion

Kaynakça

- https://www.youtube.com/watch?v=oLVETlMyJZM
- https://www.youtube.com/watch?v=kF7rQmSRlq0&t=3s
- https://gokhana.medium.com/solid-nedir-solid-yaz%C4%B1l%C4%B1m-prensipleri-nelerdir-40fb9450408e
- https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/