**Berkan İrez**

**SOLID**

Yazılım, insanların gündelik aktivitelerinin bilgisayar sistemleri ile kolay hale getirilmesidir. Bu yeni gündelik ihtiyaçlar ve düzenlemeler sürekli olarak değişmektedir ve yazılım tarafında da aynı şekilde bu yeni şeyleri sürekli değiştirmek ve düzenlemek gerekir. Dolayısıyla bu yeni değişiklikleri yazılım tarafına kolay entegre etmek için sürdürülebilir bir yazılım daha doğru bir yaklaşımdır. Bunun sonucunda ise Solid (Tek Sorumluluk İlkesi (SRP), Açık/Kapalı İlkesi (OCP), Liskov Yerine Geçme İlkesi (LSP), Arayüz Ayırma İlkesi (ISP), Bağımlılıkların Tersine Çevrilmesi İlkesi (DIP)) prensipleri ortaya çıkar. Solid prensipleri, yazılımda nesnesel tasarımları daha anlaşılır, esnek kılan ve sürdürülebilirliği desteklemek adına kullanılan prensiplerdir. Bu ilkeler, Amerikalı yazılım mühendisi ve eğitmen Robert C. Martin tarafından, ilk olarak 2000 yılında yayımlanan Tasarım İlkeleri ve Tasarım Modelleri makalesinde tanıtılmıştır.

**Tek Sorumluluk İlkesi (Single Responsibility Principle)**

Bu beş ilkeden ilki olan, Tek Sorumluluk İlkesi (SRP) bütün yapıların ve sınıfların sadece bir iş yapmasını öne sürer. Her yapı ve sınıf sadece bir iş yapmalı ve bir sorumluluğa sahip olmalıdır. Aynı sınıfa birden fazla iş verilmesi sakıncalıdır çünkü gelecekte olası düzenlemeler ve değişimlerde bu durum kodun kalitesini bozar ve yazılımcı için karmaşık, zor ve zaman kaybettiren bir süreç oluşturur. Örneğin bir sınıf üzerinde   
“Görev 1, Görev 2,” gibi iki farklı görev olması yerine bu durum iki farklı yapıda “Görev 1” ve “Görev 2” diye ayrılmalıdır. Bu durum kodun karmaşıklığını azaltır ve bakımını kolaylaştırır.

**Açık/Kapalı İlkesi (Open/Closed Principle)**

İkinci ilke olan bu prensip, yazılımların değişimi ve gelişimi için temel bir uygulamada bulunur. Bu prensibin savunduğu düşünce, Yazılımların geliştirmeye açık fakat değişime kapalı olması gerektiğini vurgular. Yazılımlar yeni özellikler eklemeye açık olmalıdır fakat bu değişiklikler, mevcut kodlar değiştirilerek yapılmamalıdır. Örneğin bir ödeme yöntemi hazırlanırken tüm farklı ödemeleri alacak bir kod satırı oluşturmak ileride gelecek yeni ödemeyi eklerken kod satırını değiştirmemizi gerektirir. Bunun yerine ödeme sınıfı oluşturmak ve kodları bunun içinde düzenlemek, gelecek yeni ödemelerde güncel kodumuzu değiştirmemizi gerektirmez, ekleme yapmamız yeterli olacaktır.

**Liskov Yerine Geçme İlkesi (Liskov's Subsititution Principle)**

Üçüncü ilke ise, alt sınıfların, temel sınıfların yerine kullanılabilmesini ve programın doğruluğunu bozmaması gerektiğini savunur. Bu, türetilmiş sınıfların, temel sınıfların davranışını değiştirmeden genişletebileceği anlamına gelir. Bir fonksiyon, temel sınıfın bir nesnesiyle çalışabiliyorsa, herhangi bir alt sınıfın nesnesiyle de sorunsuz çalışabilmelidir. Örneğin bir “dikdörtgen” sınıfı üzerinden bir “kare” sınıfını oluşturup dikdörtgen sınıfının özelliklerini değiştirmektense daha üst bir “dörtgen” sınıfı oluşturup “dikdörtgen” ve “kare” sınıflarını bu sınıf üzerinden yazmak bu prensibe uygun hareket etmemizi sağlar.

**Arayüz Ayırma İlkesi (Interface Segregation Principle)**

Dördüncü prensip olan bu prensipte, ana fikir istemcilerin kullanmadıkları arayüzlere bağımlı olmamaları gerektiğini savunur. Arayüzler ayrı ayrı ve kendilerine has görevler için oluşturulmalıdır. Bir arayüzde kullanılmayacak koşullar içeren herhangi bir durumla karşılaşılmamalıdır. Bu prensip, daha küçük ve spesifik arayüzlerin oluşturulmasını amaçlar. Örneğin bir şirket çalışanlarının hepsinde aynı metotların bulunması yerine, her birisinin kendisine özel bir metodu olmalıdır.

**Bağımlılıkların Tersine Çevrilmesi İlkesi (Dependency Inversion Principle)**

Beşinci ve SOLID ilkesi olan bu ilke, yüksek seviyeli sınıfların, düşük seviyeli sınıfları somut hallerinden ziyade soyut halleriyle kullanmasıdır. Seviyesi düşük olan sınıflar, seviyesi yüksek olanlara bağımlı olmamalıdır. İki sınıf da soyutlamalara dayanmalıdır. Bu ilke, yazılım içindeki bağımsızlığı korur ve bu durum daha esnek bir yazılım ortaya çıkarır.

Sonuç olarak yazılım dünyasındaki bu SOLID prensipleri yazılımın sürdürülebilir ve daha kaliteli olmasında önemli bir yere sahiptir.

**TASARIM KALIPLARI (DESIGN PATTERNS)**

SOLID prensibinin yanı sıra değinilmesi gereken diğer bir konu (Design Patterns) Tasarım Kalıplarıdır. 1994 yılında “Dörtlü çete (Gang of Four)” olarak da bilinen Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides tarafından yazılan "Tasarım Desenleri: Yeniden Kullanılabilir Nesne Tabanlı YazılımınÖğeleri (Design Patterns: Elements of Reusable Object-Oriented Software)" adlı kitapla beraber bu düşüncenin kavramsallaşmış ve popülerleşmiştir.

23 temel tasarım kalıbı içeren bu kitap, yazılım dünyasında zaman içerisinde tekrarlanmış sorunlara ve problemlere yine zaman içerisinde üretilmiş çözümler sunmuştur. Tasarım kalıpları, yazılım dünyasında sıkça karşılaşılan sorunlar ve problemler için yeniden kullanılabilmeyi amaçlayan yazılımlardır. Yazılımcıların benimsemesi gereken Tasarım Kalıpları, hemen uygulanabilirlikten ziyade, yazılımcının düşünce yapısına ve kodlamasına yerleşmesi gereken bir düşünme biçimidir.

Tasarım kalıpları, Yaratıcı (Creational), Yapısal (Structural) ve Davranışsal (Behavioral) kalıplar olarak üç ana kategoride sınıflandırılır.

**Yaratıcı (Creational) Tasarım Kalıbı**

Bu tasarım kalıplarından biri olan Yaratıcı tasarım kalıbı nesneleri oluşturmakla ilgilidir ve nesnelerin gerekli ve durumuna uygun şekilde yaratılmasını amaçlar. Daha esnek ve yeniden kullanılabilir bir yazılım üretmeyi amaçlayarak, nesne yaratma işlemlerini soyutlar. Kısaca yazılım içerisinde nesne oluşturma sürecini optimize etmek ve sistem içindeki karmaşıklığı azaltmak bu tasarım kalıbının birincil amacıdır. Yaratıcı Tasarım Kalıpları, Tek Nesne Kalıbı (Singleton), Fabrika Metodu Kalıbı (Factory Method), Soyut Fabrika Kalıbı (Abstract Factory), Kurucu Kalıp (Builder) ve Prototip Kalıbı (Prototype) olarak 5 alt kavramdan oluşmaktadır. Bunlardan yaygın olarak kullanılanlar ise şunlardır.

**Tek Nesne Kalıbı (Singleton)**

Tek Nesne Kalıbı, bir sınıftan sadece bir nesne oluşturmak ortaya konulmuş bir kavramdır.

**Fabrika Metodu Kalıbı (Factory Method)**

Fabrika Metodu Kalıbı, nesne oluşturma sürecini soyutlayarak, nesne oluşturma işlemini merkezi bir yerden yönetmeyi sağlayan bir kavramdır.

**Yapısal (Structural) Tasarım Kalıbı**

Bir diğer tasarım kalıbı olan bu tasarım kalıbı, nesneler ve sınıflar arasındaki ilişkileri tanımlar. Bu tasarım kalıbında nesneler ve sınıfların daha büyük yapılar oluşturması için bir araya gelmesi amaçlanmaktadır. Yazılımdaki yeniden kullanılabilirlik ve esnekliği, sistem bileşenlerinin birbirleri ile ilişkilerini ve nasıl beraber çalışacaklarını belirleyerek arttırır. Yapısal Tasarım Kalıbı, Adaptör Kalıbı (Adapter), Köprü Kalıbı (Bridge), Bileşik Kalıp (Composite), Süsleyici Kalıp (Decorator), Cephe Kalıbı (Facade) ve Hafif Sıklet Kalıbı (Flyweight) olmak üzere 6 alt kavramdan oluşmaktadır ve bu kavramlardan yaygın olarak kullanılanlar şunlardır.

**Adaptör Kalıbı (Adapter)**

Uyumsuz arayüzlerin beraber çalışmasını sağlar. Bunu bir arayüzü, başka bir arayüze dönüştürerek yapar.

**Köprü Kalıbı (Bridge)**

Birbirinden bağımsız olarak değiştirilebilen ve geliştirilebilen iki sınıf hiyerarşisini ayırır.

**Davranışsal (Behavioral) Tasarım Kalıbı**

Son tasarım kalıbı olan bu tasarım kalıbında, nesneler ile sınıflar arasındaki görev, sorumluluk dağılımını ve bu yapılar arasındaki iletişim üzerine yoğunlaşılmıştır. Davranışsal Tasarım Kalıbı, nesneler ve sınıflar arasındaki bağlantıyı ayarlayarak, algoritma ve iş birliklerinin yönetimini geliştirir. Diğer kalıplar gibi bu tasarım kalıbı da yazılımların daha esnek ve yeniden kullanılabilir olmasını amaçlar. Davranışsal Tasarım Kalıbı, Sorumluluk Zinciri Kalıbı (Chain of Responsibility), Komut Kalıbı (Command), Yorumlayıcı Kalıbı (Interpreter), Yineleyici Kalıbı (Iterator), Arabulucu Kalıbı (Mediator), Hatıra Kalıbı (Memento), Gözlemci Kalıbı (Observer), Durum Kalıbı (State), Strateji Kalıbı (Strategy), Şablon Yöntem Kalıbı (Template Method), ve Ziyaretçi Kalıbı (Visitor) olmak üzere 11 alt kavramdan oluşmaktadır.

Sonuç olarak günümüzde Tasarım Kalıpları, yazılım dünyasında esneklik, sürdürebilirlik ve yinelenen sorunlara kanıtlanmış çözümler ile gelen bir kavramdır.

**EN İYİ UYGULAMALAR (BEST PRACTICES)**

Yazılım geliştirme dünyasında En İyi Uygulamalar (Best Practices) denilen bu kavram kod kalitesini, kod verimliliğini ve kod bakımında ve düzenlemesinde kolaylığı sağlar. Bu uygulamalar arasında Kod İncelemeleri, Otomatik Testler, Versiyon Kontrolü, Dokümantasyon ve Sürekli Entegrasyon/Sürekli Dağıtım (CI/CD) gibi bazı kavramlar bulunur. Bu kavramlar şu şekilde açıklanabilir:

**Kod İncelemeleri**

Kodları düzenli bir şekilde incelemek, oluşacak sorunları önceden belirlemede ve kodlama standartlarına uyum sağlamada önemli bir yere sahiptir. Bireysel ve grup çalışmalarındaki kodları kontrol etmek, hataları erkenden yakalamayı sağlar ve kodun kalitesini arttırmada yardımcı olur.

**Otomatik Testler**

Entegrasyon, birim ve uçtan uca testler uygulamak, kodlardaki değişikliklerin yeni hatalara yol açmadığından emin olmak için kullanılır.

**Versiyon Kontrolü**

Versiyon kontrol sistemleri kullanmak, diğer yazılımcılarla işbirlikleri yapmayı sağlar. Bunun dışında değişikliklerden çabuk faydalanmak ve projelerin geçmişini korumakta da etkilidirler. Yazılımcıların özellikleri ve sürümleri verimli bir şekilde yönetmesine de olanak tanır.

**Dokümantasyon**

Diğer geliştiriciler ve yeni başlayanlar için bilgilendirici ve öğretici dokümantasyonlar yazmak, bu kişilerin yazılımı daha iyi kavraması ve anlamasında çok yardımcı olur.

**Sürekli Entegrasyon/Sürekli Dağıtım (CI/CD)**

Yazılım geliştirme süreçlerinde verimliliği ve otomasyonu arttırmayı amaçlar. Yazılım değişikliklerinin entegrelerinin otomatikleşmesi, test edilip dağıtılmasıdır.

Sonuç olarak SOLID ilkeleri, tasarım kalıpları ve en iyi uygulamalar yazılım dünyasında kaliteyi ve sürdürülebilirliği arttırmak için çok önemli kavramlardır. Bu kavramlar, koddaki esnekliği arttırarak yüksek kalitede yazılım geliştirmek için faydalıdırlar. Yazılım geliştiricilerin bu kavramları benimsemesi ve yaptıkları işlerde bu kavramları kullanması yazılım dünyası ve kişisel gelişimleri için büyük bir öneme sahiptir.

**Berkan İrez**