3.1 Design Patterns

Tasarım desenleri(Design Patterns), yazılım tasarımında sürekli karşılaşılan genel sorunlara esnek, yeniden kullanılabilir, başarılı çözümler getiren bir takım hazır kalıplardır. Hazır olarak kodun içine konulup çalışabilen, bitmiş tasarımlar değildir. Çeşitli durumlarda sorunların nasıl giderileceğini açıklayan, bunlara yol gösteren açıklamalardır. Nesneye dayalı programlamada, tasarım desenleri sınıf ve nesneler arasındaki ilişkilerin en iyi şekilde nasıl olmaları gerektiğini açıklayan yöntemlerdir. Algoritmalar, tasarım deseni değildir. Çünkü bunlar hesaplama sorunlarına çözüm getirirler, oysaki tasarım desenleri yazılım tasarımı sorunlarıyla ilgilenir. Günümüzde sıkça yapılan bir diğer hata ise, çok spesifik bir soruna karşılık kullanılan ve esnek olmayan yapılara tasarım deseni adını vermemizdir [1].

Nesneye dayalı programlamada sınıfların kendi içinde tutarlı, fakat diğer sınıflara en az bağımlı olmaları beklenir. Yazılım parçalarının tekrar kolayca kullanılabilir olmaları, kolayca genişleyebilir veya sistemden kolayca çıkarılabilir olmaları yani kısaca esnek olmaları beklenir. Yeni ihtiyaçların, yazılımın diğer kısımlarını en az biçimde etkileyerek yazılıma kolayca dahil olmaları beklenir.(Open-Closed Principle) İşte tasarım desenleri, nesneye dayalı programlamanın bu prensiplerini doğru bir şekilde uygulamamızı sağlar. Sonuç olarak tasarım desenlerini uygulayarak, etkin kodlar üretiriz, yazılımın kalitesini arttırırız. Zamandan ve yazılım için harcanan iş gücünden de tasarruf etmiş oluruz.

Yazılımcılar en çok kullanılan ve en etkin yöntemlere isimler verdiler. Bu yüzden aynı yöntemi kaynaklarda pek çok farklı isimle görebilmek mümkündü. 1994 yılında ise, Gang of Four (dörtlü çete) olarak tanınan "Erich Gamma", "Richard Helm", "Ralph Johnson" ve "John Vlissides" isimli yazarlar "Design Patterns: Elements of Reusable Object-Oriented Software" adlı kitabı yazdılar ve bu en yaygın olarak kullanılan 23 deseni en çok kullanılan isimleriyle bu kitapta topladılar [2].

1- **Creator Design Patterns (Kurucu Tasarım Desenleri)**
   A. Abstract Factory (Soyut Fabrika)
   B. Builder (Kurulum Nesnesi)

C. Factory Method (Fabrika Yordamı)
D. Prototype (Kopya Nesne)
E. Singleton (Tek Nesne)

2- Structural Design Patterns (Yapısal Tasarım Desenler)

A. Adapter (Adaptör)
B. Bridge (Köprü)
C. Composite (Ağaç Yapısı)
D. Decorator (Dekorasyon)
E. Facade (Ön Yüz)
F. Flyweight (Hafif Ağırlık)
G. Proxy (Özdeş Nesne)

3- Behavioral Design Patterns (Davranışsal Tasarım Desenler)

A. Chain of Responsibility (Sorumluluk Zinciri)
B. Command (Komut)
C. Interpreter (Yorumlayıcı)
D. Iterator (Tekrarlayıcı)
E. Mediator (Arabulucu)
F. Memento (Hatırlayıcı)
G. Observer (Gözlemci)
H. State (Durum)
I. Strategy (Strateji)
J. Template Method (Kalıp Yordam)
K. Visitor (Ziyaretçi)

3.2 Nesne Yönelimli Programlama

Nesne Yönelimli programlama (OOP) yazılımsal süreçlerde kullanılan bir yaklaşım tarzıdır. Eskiden sadece fonksiyonlar, inputlar ve outputlar ile ilgilenen yazılımcılar, sonraları bu yaklaşımdan vazgeçtiler. Kısacası bir problem ve bu problemin çözümüyle ilgilenirken, bu problemin çözüm şeklinin ve çözümün nasıl tasarlanması gerektiği de önem kazanmaya başladı. Bu proje tasarlanırken, sınıflar ve bu sınıflardan türetilen nesneler üzerinden problemi çözmeye çalıştılar. İşte bu sebeple nesne yönelimli programlama doğmuştur [3].

3.2.1 Nesne Yönelimli Programlama Prensipleri

1. Abstraction – Soyutlama: Karmaşıklığı yönetmek için kullanılır. Nesneyi diğer tüm nesne türlerinden ayıran ve dolayısıyla izleyicinin bakış açısına göre net tanımlanmış kavramsal sınırlar sağlayan temel özelliklerini belirtir.

2. Encapsulation – Kapsülleme: Hangii bilgilerin diğer sınıflar ile paylaşılacağını belirlemek için kullanılır.

3. Inheritance-Miras Verme: Birbiriyle ortaklıklar içeren sınıfların birbirinden türetilmesini sağlamak için kullanılır.

4. Polymorphism – Çok Biçimlilik: Bir Fonksiyonun birden farklı şekilde davranış göstermesi demektir.

3.2.2 Solid Principles

Nesne Yönelimli Programlama'nın bahsedilmişken **Solid Principles'**dan bahsetmemek olmaz. Solid Principles' ın ana amacı programa istenen değişikliklere karşı en az değişiklik ile değiştirilebilme yeteneği kazandırmak bu sayede zaman ve enerji kazancı sağlamaktır [4].

- S — Single-Responsibility Principle(Tek Sorumluluk Prensibi)
- O — Open-Closed Principle(Açık Kapalı Prensibi)
- L — Liskov Substitution Principle (Liskov'un Yerine geçme Prensibi)
- I — Interface Segregation Principle (Arayüz Ayrımı Prensibi)
- D — Dependency Inversion Principle (Bağımlılıkların Terslenmesi Prensibi)

1- Single-Responsibility Principle(Tek Sorumluluk Prensibi)

Bir Sınıfın ya da fonksiyonun,metodun tek bir görevi, sorumluluğu olmalıdır. Başka sınıfların görevlerini gerçekleştirmemelidir.

Örneğin: Aşçı sınıfında **YemekYap metodu** olur ama **malzemeAl metodu** olmaz ama Aşçı istese malzemeyi gidip alabilir. Lakin bu aşçının ana görevi değildir dolayısıyla gerek yoktur [4].

2- Open-Closed Principle(Açık Kapalı Prensibi)

Open-Closed prensibi kısacası bir programın, applicationun veya objelerin ya da entitylerin geliştirilmeye açık ancak değiştirmeye kapalı olduğunu belirtir. Interface ve abstract sınıflar kullanılarak istenen eklemeler yapılabilir.

Örneğin : Bir şirkette çalışanların emeklilik günleri hesaplanacak.

Her sınıf ayrı ayrı tanımlı işçi, memur vs ve her birinin bilgileri, işe giriş tarihleri sınıflarında tanımlı. Ne zaman emekli olduklarını hesaplamak için sgkHesap isimli sınıfımızda **hesap metodu** var. Patronumuz dedi ki buraya yöneticileri de ekle

yöneticilerin de emeklilik günleri hesaplansın. Bunun için sadece sgkHesap kısmına if-else koşullarıyla kontrol edilip hesaplatılmak istenen gün sayısının sahibi olan kişinin ne olduğu tespit edilerek yaptırılabilir. Hesap aynı hesap sadece değişen şey obje [4].

3- Liskov Substitution Principle (Liskov'un Yerine geçme Prensibi)

Bir ana sınıftan ya da sınıflardan türetilen sınıfların bir üst hiyerarşideki sınıfların yerine geçmesini esas alan bir prensiptir.

Örneğin: Dörtgenler Üst sınıf ve kare alt sınıf olsun. dörtgen sınıfında tanımlı **boyut değiştir** metodu uzun kenarı 300 ve kısa kenarı 100 yapsın. Kare sınıfını bunun yerine geçirdiğiniz zaman(Kare de bir dörtgendir) karenin tüm kenarları eşit olduğu için bu metodu karşılayamaz ve bu yüzünden LSPye uymaz [4].

4- Interface Segregation Principle (Arayüz Ayrımı Prensibi)

Bir arayüze gerekli olmayan eklentilerin eklenmemesini belirten bir prensiptir. Arayüzde o an sadece kullanılacak olan eklentilerin ekli olması gerektiğini savunur.

Bir müşteri asla kullanmadığı bir arabirimi uygulamaya zorlanmamalı veya istemciler kullanmadıkları yöntemlere bağlı olmaya zorlanmamalıdır.

Örneğin : İki boyutlu şekiller sınıfında **hacim** isimli bir metod tanımlamak. İki boyutlu şekillerin hacmi olmadığı için bu metod gereksizdir [4].

5- Dependency Inversion Principle (Bağımlılıkların Terslenmesi Prensibi)

Varlıklar(Alt sınıflar ve Üst sınıflar) somut olmayan soyutlamalara bağlı olmalıdır. Üst seviye modülün düşük seviye modülüne bağlı olmamasını, ancak soyutlamalara bağlı olması gerektiğini belirtir. Alt sınıflarda yapılan değişiklikler üst sınıfları etkilememelidir.

Örneğin : Yönetici ve işçi diye bir sınıfımız olsun. Yönetici sınıfımızın yeterince karmaşık olduğunu ve hem yöneticiye ait bilgilerin hem de işçiler üzerinde etki edebilecek metotları olduğunu düşünelim. Süper işçi diye yeni bir sınıf tanımladık. Ancak şu an yönetici sınıfındaki metotlar sadece işçilere hizmet veriyor ve bu durumda yönetici sınıfında çok büyük değişikliğe gitmemiz gerekecek çünkü oradaki metotların hepsi sadece işçiler için tasarlandı. Eğer dependency inversion prensibine uyulsaydı karmaşıklık minimum düzeye indirilebilirdi ve üst sınıf (yönetici) alt sınıf olan süper işçilere bağlı olmazdı [4].

3.3 Java

Java, Sun Microsystems mühendislerinden James Gosling tarafından geliştirilmeye başlanmış açık kaynak kodlu, nesneye yönelik, zeminden bağımsız, yüksek verimli, çok işlevli, yüksek seviye, adım adım işletilen (yorumlanan - interpreted) bir dildir. Java C ve C++'dan birçok sözdizim türetmesine rağmen bu türevler daha basit nesne modeli ve daha az düşük seviye olanaklar içerir. Java uygulamaları bilgisayar mimarisine bağlı olmadan herhangi bir Java Sanal Makinesi (Java Virtual Machine - JVM) üzerinde çalışabilen tipik bytecode'dur (sınıf dosyası) [5].

Şekil 3.1 Java Programlama Dili Simgesi

Java'nın sık kullanılan sloganlarından biri olan, çevirisi "bir defa yaz, her yerde çalıştır" olan "write once, run anywhere - WORA", Java'nın; derlenmiş Java kodunun, Java'yı destekleyen bütün platformlarda tekrar derlenmeye ihtiyacı olmadan çalışabileceğini ima eder. 2016 yılında bildirilen 9 milyon geliştiricisi ile, özellikle istemci sunucu web uygulamaları için olmak üzere, kullanımda olan en popüler programlama dillerinden birisidir [5].

Java ilk çıktığında daha çok küçük cihazlarda kullanılmak için tasarlanmış ortak bir düzlem dili olarak düşünülmüştü. Ancak düzlem bağımsızlığı özelliği ve tekbiçim kütüphane desteği C ve C++'tan çok daha üstün ve güvenli bir yazılım geliştirme ve işletme ortamı sunduğundan, hemen her yerde kullanılmaya başlanmıştır. Özellikle kurumsal alanda ve mobil cihazlarda son derece popüler olan Java özellikle J2SE 1.4 ve 5 sürümü ile masaüstü uygulamalarda da yaygınlaşmaya başlamıştır.

Java'nın ilk sürümü olan Java 1.0 (1995) Java Platform 1 olarak adlandırıldı ve tasarlama amacına uygun olarak küçük boyutlu ve kısıtlı özelliklere sahipti. Daha sonra düzlemin gücü gözlendi ve tasarımında büyük değişiklikler ve eklemeler yapıldı. Bu büyük değişikliklerden dolayı geliştirilen yeni düzleme Java Platform 2 adı verildi ama sürüm numarası 2 yapılmadı, 1.2 olarak devam etti. 2004 sonbaharında çıkan Java 5, geçmiş 1.2, 1.3 ve 1.4 sürümlerinin ardından en çok gelişme ve değişikliği barındıran sürüm oldu. Java SE 8 ise Java teknolojisinin günümüz sürümüdür. 13 Kasım 2006'da Java düzlemi GPL ruhsatıyla açık kodlu hale gelmiştir.