T.C.

ONDOKUZ MAYIS ÜNİVERSİTESİ

LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

BİLGİSAYAR MÜHENDİSLİĞİ

TASARIM DESENLERİ-BM634

Observer Pattern

FIRAT KAAN BİTMEZ

23281855

SAMSUN, 2024-2025 Eğitim Öğretim Yılı Güz Yarıyılı

1. Giriş

Yazılım geliştirme süreçleri, artan karmaşıklıkla başa çıkmak ve etkili çözümler üretmek amacıyla belirli prensipler ve desenler etrafında şekillenir. Tasarım desenleri, yazılım mimarilerinde sık karşılaşılan sorunlara yeniden kullanılabilir çözümler sunarak geliştiricilere esneklik, modülerlik ve kolay bakım sağlar. Observer Pattern bu desenler arasında nesneler arası dinamik etkileşimleri yönetmede öne çıkar.

Observer Pattern, temel olarak bir "yayın-abone" (publish-subscribe) modelini temsil eder ve bir nesnenin durumundaki değişiklikleri diğer ilgili nesnelere otomatik olarak bildirir. Bu yapı, sistem bileşenleri arasında gevşek bağlılık (loose coupling) oluşturarak, yazılım sistemlerinin daha esnek, şekillendirilebilir ve bakımı kolay olmasını sağlar.

Observer Pattern'in özellikleri, MVC (Model-View-Controller) mimarilerindeki rolü ve grafik kullanıcı arayüzlerinde etkin bilgi akışı sağlamasıyla dikkat çeker. Ayrıca, mikroservisler, sensör sistemleri ve oyun motorları gibi çoklu platform senaryolarında bu desenin kullanımı çarpıcı avantajlar sunar. Diğer tasarım desenlerinden farklı olarak, Observer Pattern bilginin dağıtılmasına ve zaman içinde değişken sistemlerde uyarlanabilirliğe odaklanır.

Özellikle dinamik veri akışının kritik olduğu senaryolarda Observer Pattern, sistem bileşenleri arasındaki bağımlılıkları azaltarak modern yazılım tasarımında önemli bir rol oynar.

2. Observer Pattern’in Tanımı ve Temel Prensipleri

Observer Pattern, bir nesnenin durumunda bir değişiklik olduğunda, bu değişikliği takip eden diğer nesnelere otomatik olarak bilgi ileten bir tasarım desenidir. Yazılım bileşenleri arasındaki bağımlılığı azaltarak esnek ve modüler bir yapı sağlar.

Bu desenin temel prensibi, bir veya birden fazla gözlemcinin, izlenen nesne üzerindeki değişikliklerden haberdar edilmesine dayanır. Gözlemciler, sürekli olarak izlenen nesneden veri almak zorunda kalmaz; bunun yerine izlenen nesne, durumu değiştiğinde gözlemcilere otomatik olarak bilgi gönderir. Bu mekanizma şu avantajları sunar:

- **Bağımsızlık:** İzlenen nesne, gözlemcilerin kim olduğunu veya kaç tane olduğunu bilmek zorunda değildir. Bu, bileşenler arasındaki bağımlılığı azaltır.
- **Esneklik:** Gözlemciler istendiğinde abonelikten çıkabilir veya yeni gözlemciler kolayca eklenebilir.
- **Otomatik Bildirim:** Sistemdeki değişiklikler otomatik olarak ilgili bileşenlere iletilir, böylece kodun tekrar kullanılabilirliği ve bakımı kolaylaşır.

Observer Pattern, özellikle dinamik veri akışlarının önemli olduğu sistemlerde ideal bir çözüm sunar.

3. Tarihsel Arka Plan ve Motivasyon

Observer Pattern, 1994 yılında Gamma, Helm, Johnson ve Vlissides (“Gang of Four”) tarafından yazılan "***Design Patterns: Elements of Reusable Object-Oriented Software***" kitabında popülerleşmiştir. Bu eser, yazılım geliştiricileri için yeniden kullanılabilir çözümler sunarak yazılım mimarisinde bir dönüm noktası olmuştur.

Observer Pattern'in tarihsel gelişiminde, grafik kullanıcı arayüzlerinin (GUI) yaygınlaşması önemli bir motivasyon kaynağı olmuştur. Model-View-Controller (MVC) gibi mimari yapılarda, modeldeki (Subject) değişikliklerin görünüme (Observer) otomatik olarak yansıtılması gereksinimi bu deseni öne çıkarmıştır.

Günümüzde Observer Pattern, mobil uygulamalar, bulut tabanlı sistemler ve IoT platformlarında önemli bir yer tutar. Örneğin:

- **Mobil Uygulamalar:** Haber bildirimleri ve anılık veri akışı ihtiyacı.
- **Bulut Sistemleri:** Mikroservisler arası bildirimlerin organize edilmesi.
- **IoT Sistemleri:** Sensör verilerinin dağıtımı ve cihazların senkronizasyonu.

Bu desenin, modern yazılım geliştirme süreçlerine adaptasyonu, dinamik sistemlerdeki gereksinimlere yanıt verme kapasitesini göstermektedir.

4. Yapısal Elemanlar

Observer Pattern, iki temel bileşen üzerine kuruludur:

1. **Subject (Observable):**
    - Takip edilen nesnedir. Durumundaki değişiklikleri izlemek isteyen Observer'ları kaydeder ve aboneliklerini yönetir.
    - Görevleri:
        - **Kayıt (Attach):** Observer eklemek.
        - **Kaldırma (Detach):** Observer çıkartmak.
        - **Bildirim (Notify):** Değişiklikleri tüm kayıtlı Observer'lara iletmek.
2. **Observer (Gözlemci):**
    - Subject'in durumundaki değişikliklerden haberdar olmak isteyen nesnedir.
    - Görevleri:
        - Subject tarafından gönderilen bildirimleri almak.
        - Bildirimlere yanıt olarak kendi iç mantığını güncellemek veya eylemleri tetiklemek.

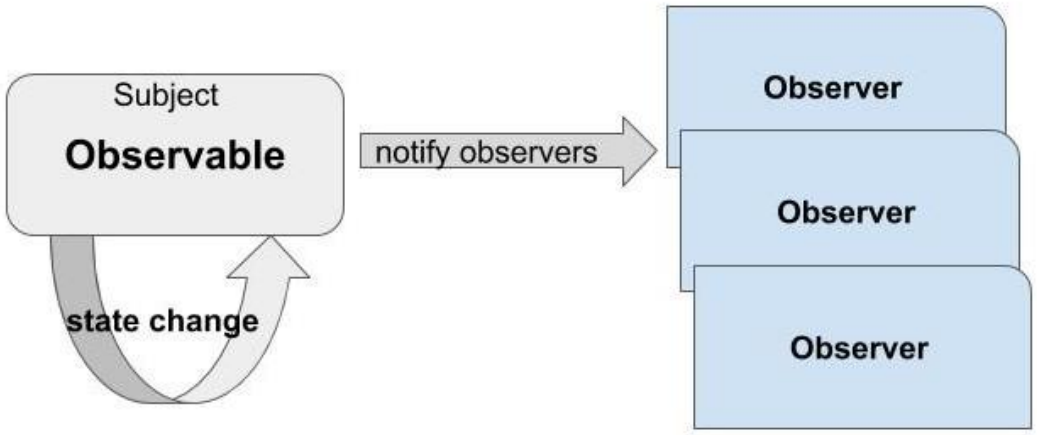
Bu iki bileşen arasındaki etkileşim gevşek bağlılık ilkesiyle yönetilir. Subject, Observer'ların kimliğini veya iç mantığını bilmez, sadece durumu yayınlar.

Pratikte Karşılaşılan Sorunlar:

- **Zincirleme Bildirimler:** Bir Observer'ın, diğer bir Subject'i tetiklemesi karmaşıklığı artırabilir.
- **Geri Bildirim Döngüleri:** Bir Observer, Subject'in durumunu güncellerse, bu durum sonsuz bir bildirim döngüsüne yol açabilir.
- **Performans Sorunları:** Fazla sayıda Observer olduğunda, bildirimlerin verimli şekilde yönetilmesi gereklidir.

Çözümler:

- Bildirim filtreleme ve gruplama mekanizmaları kullanılarak özelleştirilmiş güncellemeler yapılabilir.
- Eşzamanlılık kontrolü ve hiyerarşik bildirim yapısı uygulanabilir.



5. Teorik Temeller ve Matematiksel Modelleme

Observer Pattern, yazılım mühendisliğinde bilgi akışını ve nesneler arasındaki ilişkileri modellemek için matematiksel bir yaklaşımla da ele alınabilir. Aşağıda bu yapının teorik bir modellemesi ve kod örneğiyle desteklenmiş açıklaması bulunmaktadır.

5.1 Matematiksel Modelleme

Observer Pattern, bir Subject ve bir veya birden fazla Observer arasında bilgi akışını matematiksel olarak soyutlamaya olanak tanır. Bu yapı, şu şekilde formüle edilebilir:

- **Subject Durumu**: Subject, bir durum değişikliği meydana geldiğinde bunu bir dizi Observer'a bildirir. Subject'in durumu S(t), zaman ttt ile ifade edilir.

- **Observer Fonksiyonu**: Her bir Observer OiO_iOi, Subject'in durumuna bağlıdır ve bu bağımlılık Oi=fi(S(t)) ile ifade edilebilir. Burada fi, Observer'ın Subject'ten aldığı durumu nasıl işlediğini tanımlar.
- **Senkron ve Asenkron Bildirimler**: Subject, Observer'lara senkron veya asenkron olarak bilgi gönderebilir. Senkron bir modelde tüm Observer'lar aynı anda bilgilendirilir, asenkron bir modelde ise bu bildirim bir sıra dahilinde gerçekleşir.

Matematiksel modelleme, Observer Pattern'in bilgi teorisi açısından bir sinyal yayma mekanizması olarak ele alınmasını sağlar. Özellikle büyük ölçekli sistemlerde, bu modelleme veri akışının optimize edilmesi için kullanılabilir.

5.2 Kodla Desteklenen Teorik Çıkarımlar

Observer Pattern, C# gibi dillerde delegeler ve olaylar ("events") kullanılarak uygulanabilir. Aşağıda, Observer Pattern'in soyut matematiksel modelini somut bir kod örneğiyle açıklıyoruz.

Kod Örneği: Sayı Üreticisi ve Gözlemciler

```
using System;
using System.Collections.Generic;

// Subject (Observable)
public class NumberGenerator
{
    private List<INumberObserver> observers = new List<INumberObserver>();
    private Random random = new Random();

    public void RegisterObserver(INumberObserver observer)
    {
        observers.Add(observer);
    }

    public void RemoveObserver(INumberObserver observer)
    {
        observers.Remove(observer);
    }

    public void GenerateNumber()
    {
        int number = random.Next(1, 101); // Rastgele 1-100 arası bir sayı
        NotifyObservers(number);
    }

    private void NotifyObservers(int number)
    {
```

```
        foreach (var observer in observers)
        {
            observer.Update(number);
        }
    }
}

// Observer Interface
public interface INumberObserver
{
    void Update(int number);
}

// Concrete Observer
public class ConsoleObserver : INumberObserver
{
    public void Update(int number)
    {
        Console.WriteLine($"Güncellenen Sayı: {number}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var generator = new NumberGenerator();
        var observer1 = new ConsoleObserver();
        var observer2 = new ConsoleObserver();

        generator.RegisterObserver(observer1);
        generator.RegisterObserver(observer2);

        generator.GenerateNumber();
        generator.GenerateNumber();
    }
}
```

- **NumberGenerator** bir Subject olarak davranır ve rastgele bir sayı üreterek tüm kayıtlı Observer'lara bu bilgiyi iletir.
- **ConsoleObserver**, Subject'ten gelen güncellemeleri dinleyen bir Observer'dır.
- Kodda matematiksel model, **Update** metodunda temsil edilir: Her bir Observer, Subject'ten aldığı yeni durumu kullanarak çıktısını günceller.

5.3 Observer Pattern'in Sınırları ve Alternatifleri

Observer Pattern, yazılım mimarilerinde sıkça kullanılsa da, belirli durumlarda yetersiz kalabilir veya daha modern yaklaşımlarla değiştirilebilir. Bu başlık altında bu sınırlamaları ve alternatifleri inceliyoruz:

5.3.1 Memory Leak Sorunları

Observer Pattern, yanlış uygulandığında bellek sızıntılarına yol açabilir. Bu, genellikle bir Observer'ın, Subject üzerinde abonelikten çıkarılmadan sistemden kaldırılması durumunda meydana gelir. Modern diller, bu tür sorunları azaltmak için çeşitli çözümler sunar:

- **Weak References (Zayıf Referanslar)**: Observer nesnelerinin, Subject tarafından zayıf referanslarla tutulması, bu nesnelerin çöp toplayıcı (garbage collector) tarafından serbest bırakılmasına olanak tanır. Bu sayede, bellek sızıntıları önemli ölçüde önlenir.
- **Unsubscribe Mekanizmaları**: Observer'ların, artık kullanılmadıklarında abonelikten ayrılmalarını sağlamak için düzenli olarak Unsubscribe yöntemlerinin çağrılması önerilir.
- **Otomatik Abonelik Yönetimi**: Modern framework'lerde (örneğin C#'da) IDisposable arayüzü veya benzeri yöntemlerle Observer'ların yaşam döngüsü yönetilebilir

5.3.2 Performans Kısıtları

- Çok sayıda Observer'a bildirim gönderilmesi, yüksek sistem kaynakları tüketimine yol açabilir. Bildirimlerin filtrelenmesi veya gruplanması gibi yöntemler bu sorunları hafifletmek için kullanılabilir.

5.3.3 Alternatif Modeller

- **Reactive Programming ve Event Streams:** Observer Pattern'in modern bir alternatifi olarak veri akışlarını daha esnek şekilde işlemek için Reactive Extensions (Rx) kullanılabilir.
- **Mediator Pattern:** Karmaşık sistemlerde, Observer Pattern yerine merkezi bir aracı kullanılarak iletişim yönetilebilir.

6. Kullanım Alanları

Observer Pattern, dinamik veri akışlarının yönetimi ve bileşenler arasındaki bağımlılıkların azaltılması gereken pek çok farklı senaryoda kullanılmaktadır. Bu desen, genellikle değişikliklerin otomatik olarak ilgili tüm bileşenlere iletilmesini gerektiren sistemlerde önemli bir rol oynar. Özellikle kullanıcı arayüzleri, finansal uygulamalar, oyun motorları, sensör izleme sistemleri ve dağıtık mimariler gibi alanlarda yaygın olarak uygulanmaktadır.

Kullanıcı arayüzü tasarımlarında, modeldeki bir değişikliğin görsel bileşenlere anlık olarak yansıtılması gereksinimi sıklıkla karşılaşılan bir durumdur. Örneğin, bir e-ticaret uygulamasında

sepetteki ürünlerin değişmesiyle, arayüzdeki toplam fiyat ve ürün listesi bilgilerinin otomatik olarak güncellenmesi Observer Pattern ile kolayca yönetilebilir.

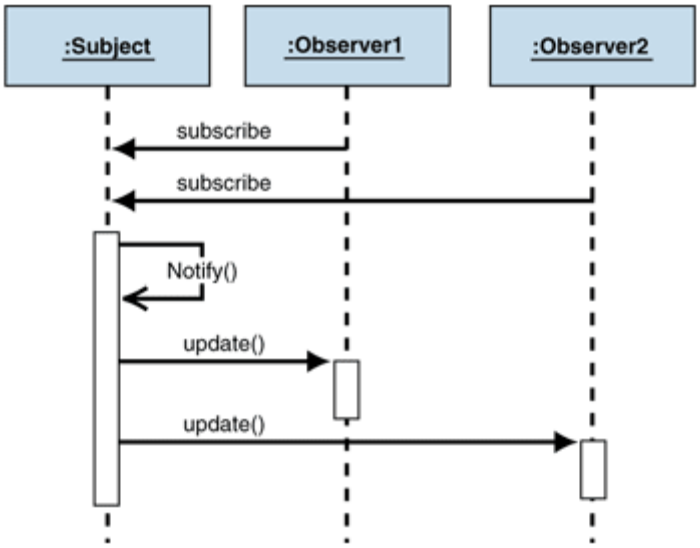
Finansal uygulamalarda, hisse senedi fiyatları veya döviz kurları gibi değişkenlerin gerçek zamanlı olarak izlenmesi kritik bir ihtiyaçtır. Bu desen, kullanıcıların portföylerinde meydana gelen değişikliklerin hızlı ve güvenilir bir şekilde iletilmesini sağlar. Benzer şekilde, oyun motorlarında bir nesnenin durumundaki değişikliklerin diğer bileşenlere aktarılması gereklidir. Örneğin, bir oyuncunun sağlık durumundaki bir değişikliğin, arayüzdeki sağlık göstergesine veya oyunun skor sistemine yansıtılması Observer Pattern ile etkin bir şekilde gerçekleştirilebilir.

IoT sistemlerinde, sensörlerden gelen anlık verilerin ilgili cihazlara veya kontrol panellerine aktarılması Observer Pattern'in uygulandığı diğer bir alandır. Örneğin, bir sıcaklık sensöründen gelen verilerin hem bir kontrol merkezine hem de ilgili alarm sistemlerine iletilmesi bu desenin sağladığı kolaylıklar arasında yer alır.

Mikroservis mimarilerinde ise bir servis üzerinde meydana gelen değişikliklerin diğer servis veya bileşenlere iletilmesi gereklidir. Bu, Observer Pattern'in dağıtık sistemlerde yaygın olarak kullanılan bir çözüm olduğunu göstermektedir. Örneğin, bir stok yönetim servisinde meydana gelen güncellemelerin, sipariş yönetimi servisine bildirilmesi gibi senaryolar bu desenle kolayca uygulanabilir.

7. C# .NET Ortamında Uygulama

C# ve .NET ortamı, Observer Pattern'i uygulamak için çok çeşitli araçlar sunar. Bu bölümde hem temel hem de ileri seviye teknikleri açıklıyoruz.



7.1 Event ve Delegate Tabanlı Uygulama

C# dilinin doğal bir parçası olan "events" ve "delegates" yapıları Observer Pattern'i etkili bir şekilde gerçekleştirir.

Kod Örneği: Olay Bildirim Sistemi

```
using System;

public class WeatherStation
{
    public event Action<int> TemperatureChanged;

    public void ChangeTemperature(int newTemperature)
    {
        Console.WriteLine($"Yeni Sıcaklık: {newTemperature}");
        TemperatureChanged?.Invoke(newTemperature);
    }
}

public class Display
{
    public void OnTemperatureChanged(int temperature)
    {
        Console.WriteLine($"Ekran Güncellendi: Sıcaklık {temperature} °C");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var station = new WeatherStation();
        var display = new Display();

        station.TemperatureChanged += display.OnTemperatureChanged;

        station.ChangeTemperature(25);
        station.ChangeTemperature(30);
    }
}
```

- **WeatherStation**, sıcaklık değişimini bir event aracılığıyla yayınlar.
- **Display**, bu event'e abone olarak güncellemeleri alır.

7.2 Reactive Extensions (Rx) ile Uygulama

Reactive Extensions (Rx), Observer Pattern'in modern bir yaklaşımını sunar. Rx, veri akışlarını asenkron ve fonksiyonel olarak işlemek için çok çeşitli operatörler sağlar.

Kod Örneği: Rx ile Sıcaklık Takibi

```
using System;
using System.Reactive.Subjects;

public class WeatherStation
{
    private readonly Subject<int> _temperatureSubject = new Subject<int>();

    public IObservable<int> TemperatureStream => _temperatureSubject;

    public void ChangeTemperature(int newTemperature)
    {
        Console.WriteLine($"Yeni Sıcaklık: {newTemperature}");
        _temperatureSubject.OnNext(newTemperature);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var station = new WeatherStation();

        station.TemperatureStream.Subscribe(temp =>
        {
            Console.WriteLine($"Observer: Sıcaklık {temp} °C olarak
güncellendi.");
        });

        station.ChangeTemperature(22);
        station.ChangeTemperature(28);
    }
}
```

- **Subject**, bir veri yayını oluşturur.
- **Subscribe** metodu ile Observer'lar bu yayına abone olabilir ve gelen değerleri alabilir.

Rx kullanımı, özellikle karmaşık veri akışı senaryolarında daha esnek bir yapı sunar.

8. Avantajlar ve Dezavantajlar

8.1 Avantajlar

- **Gevşek Bağlılık**: Subject Observer'ların kim olduğunu bilmez, sadece değişikliği yayınlar. Bu sayede bileşenler arasında sıkı bir bağ yoktur.
- **Kolay Genişletilebilirlik**: Yeni Observer eklemek veya mevcut bir Observer'ı çıkarmak, Subject tarafında değişiklik gerektirmez.
- **Yeniden Kullanılabilirlik**: Aynı Subject birden fazla farklı Observer tarafından izlenebilir, bu da kodun tekil bileşenlerinin birden çok senaryoda tekrar kullanımını kolaylaştırır.
- **Bakım Kolaylığı**: Değişiklik yaparken yaygın etkiyi azaltır, sistemdeki güncellemeler kolayca izlenip yönetilebilir.

8.2 Dezavantajlar

- **Çoklu Observer Yönetimi**: Çok sayıda Observer olduğunda performans ve yönetim açısından karmaşıklık artabilir.
- **Bildirim Sıkıntıları**: Hatalı veya gereksiz bildirimler sistemde gereksiz yüke yol açabilir.
- **Senkronizasyon Sorunları**: Çok iş parçacıklı (multi-threaded) ortamlarda güncellemelerin senkronizasyonu zor olabilir. Eşzamanlılık konularında dikkatli tasarım ve bazen ek senkronizasyon mekanizmaları şarttır.

8.3 Kritik Eleştiriler ve Çözümler

Observer Pattern'in eleştirilmesi gereken bazı yönleri bulunmaktadır:

1. **Zincirleme Bildirim Sorunları:** Bir Observer, başka bir Subject'e bildirim gönderebilir. Bu, zincirleme bir reaksiyona yol açarak sistemde karmaşıklık yaratabilir. Çözüm olarak, Observer'lar arasında bağımsızlık sağlanmalı veya bildirimlerin hiyerarşik yapısı iyi tasarlanmalıdır.
2. **Geri Bildirim Döngüleri:** Observer'ların aynı anda Subject'i güncellemesi, sonsuz bir geri bildirim döngüsüne yol açabilir. Bu durumun önüne geçmek için eşzamanlılık kontrolü gereklidir.

9. Test, Bakım ve Performans Değerlendirmesi

Observer Pattern tabanlı sistemlerin test edilmesi görece kolaydır. Subject için test senaryolarında, sahte (mock) Observer'lar kullanılarak yayınlanan bildirimlerin doğru zamanda ve doğru sayıda yapıldığı doğrulanabilir. Observer için ise sahte Subject aracılığıyla tetiklenen bildirimlerin doğru tepkiyi verdiği test edilebilir.

Performans tarafında, çok sayıda Observer söz konusu olduğunda, Subject her değişimde tüm Observer'lara bildirim gönderir. Bu, bazı durumlarda bant genişliği ve işlemci tüketimini artırabilir. Bu tür sorunları aşmak için filtreleme, durum değişimlerinin toplu bildirimleri, önbellekleme veya Rx üzerinde operatörlerle veri akışını optimize etme yöntemleri kullanılabilir.

Bakım aşamasında, Observer Pattern sayesinde sistemin bileşenlerini bağımsız olarak değiştirmek daha kolaydır. Bir Observer'ın kaldırılması ya da eklenmesi, Subject'te bir değişiklik gerektirmez. Bu modüler yapı uzun vadede yazılımın evrimini ve güncellenmesini destekler.

Performans Optimizasyonları

Performans sorunlarını çözmek için kullanılabilecek stratejiler:

- **Gruplanmış Bildirimler:** Subject, birden fazla durumu toplu olarak Observer'lara bildirebilir.
- **Eşikleme:** Küçük değişiklikleri toplu güncellemelerle birleştirmek, bildirim sıklığını azaltır.
- **Filtreleme:** Observer'lara yalnızca ilgilendikleri güncellemelerin gönderilmesi sağlanabilir.

10. Diğer Tasarım Desenleriyle İlişki

Observer Pattern çoğu zaman diğer tasarım desenleriyle birlikte kullanılır:

- **MVC (Model-View-Controller):** Model (Subject) değişikliklerini View (Observer) bileşenlerine bildirir.
- **Event Aggregator:** Birden fazla Observer'a yapılan bildirimleri merkezileştirir.
- **Strategy Pattern:** Observer Pattern, Strategy Pattern ile birlikte kullanılarak dinamik davranışlar sunar.

Observer Pattern çoğu zaman MVC, MVP, MVVM gibi yaygın mimari kalıplar içerisinde kullanılır. Model verilerini izleyen View veya ViewModel bileşenleri, Observer Pattern mantığıyla güncellenir. Ayrıca Decorator, Mediator, Publisher-Subscriber, Strategy, Command gibi desenlerle birlikte kullanılarak daha zengin mimariler oluşturulabilir.

11. Observer Pattern ile Sepet Yönetimi Örnek Uygulaması

Yazılım geliştirme süreçlerinde sıklıkla kullanılan **Observer Design Pattern**'in bir alışveriş sepeti senaryosu üzerinden nasıl uygulanabileceği detaylı bir şekilde ele alınmaktadır. Observer Pattern, bir nesnenin durumundaki değişikliklerin, bu nesneyi gözlemleyen diğer nesnelere bildirilmesini sağlar. Bu mekanizma, nesneler arasındaki sıkı bağımlılığı ortadan kaldırarak daha esnek ve ölçeklenebilir bir sistem geliştirilmesine olanak tanır.

Ele alınan örnekte, bir alışveriş sepeti uygulaması üzerinde Observer Pattern'in temel bileşenleri, işlevleri ve sınıflar arasındaki ilişkiler incelenmektedir. **Cart** sınıfı, sepetin durumu değiştiğinde bağlı olan gözlemcilere bilgi göndermekten sorumlu bir "Subject" olarak görev yapar. **StockManager**, **PromotionEngine** ve **UIUpdater** gibi gözlemciler ise bu duruma farklı şekillerde tepki verir.

Bu senaryo, Observer Pattern'in gerçek dünyadaki alışveriş uygulamaları, stok yönetimi, kampanya entegrasyonu ve kullanıcı arayüzü güncellemeleri gibi ihtiyaçlarda nasıl etkili bir şekilde kullanılabileceğini gözler önüne sermektedir. Örnek kod parçaları ve açıklamalarla birlikte, bu güçlü tasarım deseninin işleyişi ve avantajları detaylandırılmaktadır.

Aşağıda, Observer Pattern'in bu senaryodaki temel bileşenleri ve örnek bir çalışma akışı sunulmaktadır.

1. Observer Interface

```
public interface ICartObserver
{
    void Update(Cart cart);
}
```

Observer interface'i, Observer Pattern'in temel yapı taşlarından biridir. Bu arayüz, tüm gözlemcilerin uygulaması gereken Update metodunu tanımlar. Bu metodun amacı, Subject tarafından sağlanan güncellemeleri almak ve işlemek için bir şablon oluşturmaktır.

2. Subject Interface

```
public interface ISubject
{
    void Attach(ICartObserver observer);
    void Detach(ICartObserver observer);
    void Notify();
}
```

Subject interface’i, gözlemcilerin yönetiminden sorumlu metodları tanımlar. Attach ve Detach metodları, Observer'ların eklenmesi ve kaldırılması işlemlerini gerçekleştirir. Notify metodu ise durumu güncellenen Subject’in, Observer'lara bilgi göndermesini sağlar.

3. Concrete Subject: Cart

```
public class Cart : ISubject
{
    private readonly List<ICartObserver> _observers = new List<ICartObserver>();
    public List<CartItem> CartItems { get; private set; } = new List<CartItem>();
    public decimal TotalPrice => CalculateTotalPrice();

    public void Attach(ICartObserver observer)
    {
        _observers.Add(observer);
    }

    public void Detach(ICartObserver observer)
    {
        _observers.Remove(observer);
    }

    public void Notify()
    {
        foreach (var observer in _observers)
        {
            observer.Update(this);
        }
    }

    public void AddItem(CartItem item)
    {
        CartItems.Add(item);
        Notify();
    }

    public void RemoveItem(CartItem item)
    {
        CartItems.Remove(item);
        Notify();
    }

    private decimal CalculateTotalPrice()
```

```
    {
        decimal total = 0;
        foreach (var item in CartItems)
        {
            total += item.UnitPrice * item.Quantity;
        }
        return total;
    }
}
```

Cart sınıfı, Concrete Subject rolünü üstlenir ve gözlemcileri yönetir. Sepete ürün ekleme veya çıkarma işlemi gerçekleştirildiğinde, Notify metodu çağrılır ve tüm Observer'lar bilgilendirilir.

4. Concrete Observers

StockManager

```
public class StockManager : ICartObserver
{
    public void Update(Cart cart)
    {
        foreach (var item in cart.CartItems)
        {
            Console.WriteLine($"Stock updated for Product ID: {item.ProductId},
Quantity: {item.Quantity}");
        }
    }
}
```

StockManager, Cart üzerinde bir güncelleme olduğunda stoğu kontrol eder ve değişiklikleri yansıtır. Bu Observer, sepet durumunu stok sistemine entegre etmek için kullanılır.

PromotionEngine

```
public class PromotionEngine : ICartObserver
{
    public void Update(Cart cart)
    {
        if (cart.TotalPrice > 100)
        {
            Console.WriteLine("Promotion applied: 10% discount for orders over
$100.");
```

```
        }
    }
}
```

PromotionEngine, toplam fiyat üzerinden kampanyaları değerlendirir ve gerekli durumlarda indirimleri tetikler.

UIUpdater

```
public class UIUpdater : ICartObserver
{
    public void Update(Cart cart)
    {
        Console.WriteLine("Cart updated. Total items: " + cart.CartItems.Count +
", Total Price: " + cart.TotalPrice);
    }
}
```

UIUpdater, kullanıcı arayüzünü güncel tutmak için kullanılır. Bu Observer, kullanıcı deneyimini geliştirmek için gerçek zamanlı bilgiler sağlar.

5. CartItem Sınıfı

```
public class CartItem
{
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public decimal UnitPrice { get; set; }
    public int Quantity { get; set; }
}
```

CartItem sınıfı, sepet öğelerini temsil eder ve Observer Pattern'in temel işlemleri için veri sağlar.

Run Exampla Script

```
var cart = new Cart();

        var stockManager = new StockManager();
        var promotionEngine = new PromotionEngine();
        var uiUpdater = new UIUpdater();
```

```
        cart.Attach(stockManager);
        cart.Attach(promotionEngine);
        cart.Attach(uiUpdater);

        cart.AddItem(new CartItem { ProductId = 1, ProductName = "Laptop",
UnitPrice = 1000, Quantity = 1 });
        cart.AddItem(new CartItem { ProductId = 2, ProductName = "Mouse",
UnitPrice = 50, Quantity = 2 });

        cart.RemoveItem(new CartItem { ProductId = 2, ProductName = "Mouse",
UnitPrice = 50, Quantity = 2 });
```

Observer Pattern'in çalışma mekanizmasını uygulamalı olarak gösterir. Cart ve Observer'ların etkileşimleri, Observer Pattern'in gerçek dünyadaki bir alışveriş uygulamasında nasıl kullanılabileceğini somutlaştırır.

12. Sonuç

Bu çalışma kapsamında Observer Pattern'in teorik temelleri, tarihsel gelişimi, avantaj ve dezavantajları, kullanım alanları ve bir yazılım dili (C#) üzerinde örnek uygulamaları detaylı bir şekilde incelenmiştir. Observer Pattern, yazılım mühendisliğinde özellikle kullanıcı arayüzleri, dağıtık sistemler, veri akışı yönetimi ve gerçek zamanlı bildirim gereksinimlerinin bulunduğu alanlarda etkin bir çözüm sunmaktadır.

Observer Pattern, sistem bileşenleri arasındaki gevşek bağlılık sayesinde yüksek esneklik ve modülerlik sağlarken, sistemin genişletilebilirliğini de artırır. Bu özellikleriyle yazılım projelerinde değişikliklere hızlı uyum sağlama ve bakım kolaylığı gibi avantajlar sunar. Ancak, Observer Pattern'in özellikle çoklu gözlemci yönetimi, bildirim performansı ve potansiyel geri bildirim döngüleri gibi zorluklar taşıdığı da unutulmamalıdır. Bu zorluklar, uygun tasarım prensipleri, filtreleme yöntemleri ve modern yaklaşımlar (ör. Reactive Programming) ile giderilebilir.

Pratik uygulama örneği, Observer Pattern'in gerçek dünya senaryolarındaki kullanımını göstermiştir. Özellikle e-ticaret senaryosunda, bir sepet üzerindeki değişikliklerin stok yönetimi, indirim motoru ve kullanıcı arayüzü gibi bağımsız bileşenlere bildirimi, bu desenin gücünü ortaya koymaktadır. Bu gibi uygulamalarda, Observer Pattern, kodun bakımını kolaylaştırırken, yeniden kullanılabilirliği ve sistemin toplam performansını artırmaktadır.

Sonuç olarak, Observer Pattern, yazılım mühendisliğinin temel tasarım desenlerinden biri olarak modern uygulamalarda vazgeçilmez bir rol oynamaktadır. Ancak, her tasarım deseninde olduğu gibi, Observer Pattern'in de doğru bağlamda ve ihtiyaçlara uygun bir şekilde uygulanması gerekmektedir. Yazılım projelerinde bu desenin kullanımı, uzun vadede maliyet etkin çözümler ve sürdürülebilir sistemler geliştirilmesine olanak tanır.

Kaynaklar

1. Maier, I., Rompf, T., & Odersky, M. (2010). *Deprecating the Observer Pattern*. EPFL Technical Report. Retrieved from https://infoscience.epfl.ch/record/148043/files/DeprecatingObserversTR2010.pdf
2. Pattern, O. (2021). *Observer Pattern*. Computer Engineering Department, Dankook University. Retrieved from https://dis.dankook.ac.kr
3. André, É. (2013). *Observer Patterns for Real-Time Systems*. 18th International Conference on Engineering of Complex Computer Systems. Retrieved from IEEE Xplore.
4. Piveta, E. K., & Zancanella, L. C. (2003). *Observer Pattern Using Aspect-Oriented Programming*. Scientific Literature Digital Library. Retrieved from https://cin.ufpe.br
5. Firat Kaan Bitmez. *Shopping Master App Repository*. Retrieved from https://github.com/firatkaanbitmez/ShoppingMasterApp
6. Firat Kaan Bitmez. *Design Patterns Repository*. Retrieved from https://github.com/firatkaanbitmez/DesignPatterns
7. Kodcular. (2021). *Observer Design Pattern Nedir?*. Retrieved from https://medium.com/kodcular/observer-design-pattern-nedir-671f61969c91
8. Yıldız, G. A. (2021). *C# Observer Design Pattern/Observer Tasarım Deseni*. Retrieved from https://www.gencayyildiz.com/blog/c-observer-design-patternobserver-tasarim-deseni/
9. Gökalp, G. (2021). *C# Observer Pattern Kullanımı*. Retrieved from https://www.gokhan-gokalp.com/tr/c-observer-pattern-kullanimi/
10. Küçükoğlu, A. (2021). *Observer Design Pattern (.NET)*. Retrieved from https://ahmetkucukoglu.com/observer-design-pattern-dotnet
11. Atılkan, Y. (2021). *Observer Design Pattern/Tasarım Deseni Nedir?*. Retrieved from https://www.yasinatilkan.com/observer-design-pattern-tasarim-deseni-nedir/
12. Alkan, F. (2021). *Observer Design Pattern/Tasarım Deseni (C#)*. Retrieved from https://alkanfatih.com/observer-design-pattern-tasarim-deseni-c/