9. HAFTA NOTU

(Design Patterns)

Bu not dersin 9. haftasında yapacağınız uygulamaya hazırlanmanızı sağlayacak olup temel kavramları anlatan bilgileri içermektedir.

DERSİN ÖĞRETİM ÜYESİ

Prof. Dr. Bahriye AKAY

DENEYİ YAPTIRAN ÖĞRETİM ELEMANI

Arş. Gör. Buğra Alperen ULUIRMAK

2026

KAYSERİ

1. DESİGN PATTERNS HAKKINDA TEMEL BİLGİLER

1.1. Design Pattern Nedir?

Yazılım geliştirirken karşılaşılan problemlerin büyük çoğunluğu aslında daha önce başkaları tarafından da yaşanmıştır. Design pattern (tasarım kalıbı), bu tekrarlayan problemlere karşı deneyimli yazılımcıların zaman içinde geliştirdiği, denenmiş ve kabul görmüş çözüm şablonlarıdır.

1994 yılında Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides tarafından yayımlanan *Design Patterns: Elements of Reusable Object-Oriented Software* kitabı bu alanın temel kaynağı olmuştur. Bu dört yazar yazılım dünyasında "Gang of Four (GoF)" olarak anılmaktadır. Kitapta 23 farklı pattern sistematik biçimde tanımlanmış ve sınıflandırılmıştır.

> **Önemli Not**
> Design pattern bir kütüphane ya da hazır kod değildir. Bir problemi çözmek için izlenebilecek genel bir yaklaşım biçimidir. Aynı pattern'ı C#, Java veya Python'da uygulayabilirsiniz; yalnızca söz dizimi değişir, mantık aynı kalır.

1.2. Pattern Kategorileri

GoF pattern'larını üç ana kategoride incelemiştir:

Creational (Yaratımsal) pattern'lar nesne oluşturma sürecini yönetir ve esnekleştirir. Nesnenin nasıl, ne zaman ve hangi koşullarda yaratılacağını belirler. *Singleton* ve *Factory Method* bu gruptadır.

Structural (Yapısal) pattern'lar sınıfları ve nesneleri birbirine bağlama biçimini tanımlar. Var olan yapıları daha büyük ve işlevsel yapılara dönüştürür. *Decorator* ve *Adapter* bu kategoriye girmektedir.

Behavioral (Davranışsal) pattern'lar nesneler arasındaki iletişimi ve sorumluluk dağılımını düzenler. *Observer* ve *Strategy* bu gruptadır.

Bu Deneyde	Kategori
Singleton	Creational
Factory	Creational
Observer	Behavioral
Strategy	Behavioral

1.3. Neden Kullanılır?

Ortak dil sağlar. Ekip içinde "burada Observer kullandım" demek, sayfalar dolusu açıklama yazmaktan çok daha hızlı anlaşılır. Pattern isimleri yazılımcılar arasında evrensel bir kelime dağarcığı oluşturur.

Değişime kapalı, genişlemeye açık kod üretir. Yeni bir ödeme yöntemi eklemek için Strategy kullandığınızda mevcut koda dokunmanız gerekmez; yalnızca yeni bir sınıf yazarsınız. Bu yaklaşım *Açık/Kapalı İlkesi (Open/Closed Principle)* olarak bilinir.

Denenmiş çözümler sunar. Her pattern yıllarca gerçek projelerde test edilmiştir. Sıfırdan çözüm üretmek yerine kanıtlanmış bir şablonu kullanmak hata riskini önemli ölçüde azaltır.

Bakım kolaylaşır. Pattern'larla yazılmış kodun sorumlulukları net biçimde ayrılmıştır. Bir modülü değiştirdiğinizde diğerlerini etkileme riskiniz düşer.

1.4. Ne Zaman Kullanılmamalı?

Her pattern her duruma uymaz. Küçük ve tek kullanımlık projelere pattern uygulamak gereksiz karmaşıklık yaratabilir. Pattern kullanmak amacı değil, araçtır. Problemi çözüyorsa kullanın; sırf "pattern kullandım" demek için değil.

> **Dikkat:** Pattern'ları ezberlemek değil, hangi problemin hangi pattern ile daha temiz çözüleceğini anlamak önemlidir. Bu yetkinlik yalnızca pratikle kazanılır.

1.5. Deneyde Kullanılacak C# Yapıları

interface

Bir sınıfın hangi metotları içermesi *zorunlu* olduğunu belirleyen sözleşmedir. Gövde içermez; yalnızca imzaları tanımlar. Birden fazla sınıf aynı interface'i gerçekleyebilir ve birbirinin yerine kullanılabilir hâle gelir. Bu yapı *gevşek bağlılık (loose coupling)* sağlar.

```
public interface IOdemeYontemi {
    void Ode(double tutar);   // gövde yok, sadece imza
}
public class Nakit : IOdemeYontemi {
    public void Ode(double t) { /* gerçekleme buraya */ }
}
```

sealed class

`sealed` ile işaretlenmiş sınıftan başka sınıf türetilemez. Singleton'da bu kritik önem taşır: biri sınıfı miras alıp kendi kurucusunu ekleseydi tek nesne garantisi bozulurdu. `sealed` bu riski tamamen ortadan kaldırır.

```
public sealed class Logger { }
// public class GelismisLogger : Logger { } → DERLEME HATASI!
```

static alan ve metot

Normal alanlar her nesneye aittir; `static` alanlar ise sınıfın kendisine ait olup bellekte yalnızca tek bir kopyası bulunur. Singleton'da tek nesne örneği `static` alanda tutulur. `static` metotlar nesne olmadan, doğrudan sınıf adıyla çağrılır.

```
Logger.GetInstance().Kaydet("mesaj"); // nesne olmadan çağrı
```

List<T> ve foreach

Generic koleksiyon yapısıdır. `T` yerine interface tipi yazılırsa yalnızca o türden nesneler kabul edilir. Observer'da gözlemci listesi bu yapıyla yönetilir; `foreach` ile her gözlemci sırayla bilgilendirilir.

```
List<IHavaGozlemci> liste = new List<IHavaGozlemci>();
liste.Add(new TelefonApp());
foreach (var g in liste) g.Guncellendi(...);
```

switch ve throw

Factory'de hangi nesnenin üretileceğine `switch` ile karar verilir. Tanınmayan bir tür geldiğinde `throw` ile exception fırlatılır; bu sayede hatalı kullanım sessizce geçilmez, anında fark edilir.

```
switch (tur) {
    case "email": return new EmailBildirim();
    case "sms":   return new SmsBildirim();
    default: throw new ArgumentException("Bilinmeyen: " + tur);
}
```

2. SİNGLETON PATTERN

2.1. Problem

Bazı nesnelerin programın tüm yaşam süresi boyunca yalnızca bir tane olması gerekir. Örneğin bir uygulamanın log dosyasına yazan sınıf, veritabanı bağlantı havuzu ya da uygulama ayarlarını tutan nesne birden fazla oluşturulursa tutarsızlık ve kaynak israfı kaçınılmaz olur. Farklı kod noktalarında `new Logger()` yazmak bu sorunu doğurur.

2.2. Çözüm

Singleton, nesne oluşturma işlemini sınıfın kendi içine çeker. `private` yapıcı metot dışarıdan `new` yapılmasını engeller. Sınıf, tek örneği `static` bir alanda saklar. `GetInstance()` adlı statik metot her çağrıldığında bu alanı kontrol eder; nesne yoksa oluşturur, varsa mevcut olanı döndürür.

> **Gerçek Dünya Analojisi**
> Bir ülkedeki Merkez Bankası gibi düşünün. Merkez Bankası her ilde ayrı ayrı kurulmaz; tüm ülke tek bir kurumu paylaşır. Singleton da tam olarak budur: tek kurum, her yerden erişilebilir.

2.3. Ne Zaman Kullanılır?

- Log kayıt sistemi
- Veritabanı bağlantı havuzu
- Uygulama konfigürasyon yöneticisi
- Önbellek (cache) yöneticisi

> **Dikkat:** Singleton global durum taşıdığından aşırı kullanımı kodun test edilmesini zorlaştırır. Gerçekten tek olması gereken durumlarda tercih edin.

2.4. Yapısı

Singleton'ın üç temel bileşeni vardır:

- **private static alan:** Tek nesneyi saklar
- **private yapıcı:** Dışarıdan nesne üretimini engeller
- **public static GetInstance():** Tek erişim noktası

2.5. Pattern Olmadan vs. Pattern İle

❌ **Pattern olmadan**

```
// Her yerde yeni nesne
Logger log1 = new Logger();
Logger log2 = new Logger();
Logger log3 = new Logger();
// 3 farklı nesne — tutarsızlık!
```

✓ **Singleton ile**

```
// Her zaman aynı nesne
Logger log1 = Logger.GetInstance();
Logger log2 = Logger.GetInstance();
// log1 == log2 → True
```

2.6. Kod Örneği — Log Kayıt Sistemi

```
using System;

public sealed class Logger          // sealed: kalıtım engellenir
{
    private static Logger _instance = null;
    private int _kayitNo = 0;

    private Logger()                // private: dışarıdan new Logger() yapılamaz
    {
        Console.WriteLine("[Logger] Sistem başlatıldı.");
    }

    public static Logger GetInstance()
    {
        if (_instance == null)      // ilk çağrıda nesne oluşturulur
            _instance = new Logger();
        return _instance;           // sonraki çağrılarda aynı nesne döner
    }

    public void Kaydet(string seviye, string mesaj)
    {
        _kayitNo++;
        Console.WriteLine($"  [{_kayitNo:D3}] [{seviye.ToUpper()}] {mesaj}");
    }
}

class Program
{
    static void Main()
    {
        Logger log1 = Logger.GetInstance();
        Logger log2 = Logger.GetInstance();   // yeni nesne OLUŞTURULMAZ

        log1.Kaydet("INFO",  "Uygulama başlatıldı.");
        log2.Kaydet("WARN",  "Bağlantı yavaş.");
        log1.Kaydet("ERROR", "Veritabanı hatası.");

        // Singleton doğrulaması
        Console.WriteLine($"\nAynı nesne mi? {object.ReferenceEquals(log1, log2)}");
        // Çıktı → Aynı nesne mi? True
    }
}
```

3. OBSERVER PATTERN

3.1. Problem

Bir nesnenin durumu değiştiğinde, buna bağlı başka nesnelerin de güncellenmesi gerekebilir. Ancak bu nesnelerin birbirini doğrudan tanıması istenilmez; aksi hâlde aralarında sıkı bir bağ (tight coupling) oluşur ve biri değiştiğinde diğerini de değiştirmek zorunlu hâle gelir.

Örneğin bir hava durumu uygulaması düşünün: telefon uygulaması, web sitesi ve akıllı saat aynı sıcaklık verisini göstermektedir. Sıcaklık değiştiğinde üçü de güncellenmelidir. Ama hava durumu servisi bu üç uygulamanın varlığından haberdar olmak zorunda mıdır?

3.2. Çözüm

Observer, **subject** (yayıncı) ve **observer** (abone) rollerini tanımlar. Subject, abonelerini bir listede tutar. Durum değiştiğinde listedeki herkesi bilgilendirir. Aboneler dinamik olarak eklenip çıkarılabilir. Subject, abonelerin kim olduğunu veya ne yaptığını bilmez; yalnızca ortak arayüz üzerinden onları çağırır.

> **Gerçek Dünya Analojisi**
> Gazete aboneliğini düşünün. Gazete (subject) her sabah yeni baskısını çıkarır ve abonelerine (observer) gönderir. Yeni bir abone eklendiğinde ya da ayrıldığında gazete bunu kaydeder. Gazete her abonesiyle tek tek ilgilenmez; gönderir, biter.

3.3. Ne Zaman Kullanılır?

- Bir nesne değiştiğinde diğerlerinin de değişmesi gerektiğinde
- Kaç nesnenin bilgilendirileceğinin önceden bilinmediği durumlarda
- Event (olay) sistemleri tasarlanırken
- Model-View ayrımı gerektiren mimarilerde (MVC)

> **C# İle Bağlantısı**
> Windows Forms'da kullandığınız `button.Click += IslemYap` satırı tam olarak Observer pattern'ıdır. Button (subject) tıklandığında kayıtlı tüm metotları (observer) çağırır.

3.4. Katılımcılar

Rol	Açıklama
IObserver	Tüm gözlemcilerin uyması gereken arayüz. Güncelleme metotunu tanımlar.
Subject	Durumu değişen nesne. Gözlemci listesini yönetir ve haber verir.
ConcreteObserver	Arayüzü gerçekleyen sınıf. Haberi aldığında kendi işlemini yapar.

3.5. Pattern Olmadan vs. Pattern İle

❌ **Pattern olmadan**

```
void SicaklikGuncelle(double d) {
  telefon.Guncelle(d);
  webSitesi.Guncelle(d);
  saat.Guncelle(d);
  // yeni ekran → kodu değiştir!
}
```

✓ **Observer ile**

```
void SicaklikGuncelle(double d) {
  foreach (var g in _gozlemciler)
    g.Guncellendi(d);
  // yeni ekran → sadece ekle!
}
```

3.6. Kod Örneği — Hava Durumu Bildirimi

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

// Observer arayüzü — tüm gözlemciler bunu gerçekler
public interface IHavaGozlemci
{
    void Guncellendi(string sehir, double sicaklik, double nem);
}

// Subject — durumu değişen, gözlemcileri yöneten sınıf
public class HavaMerkezi
{
    private List<IHavaGozlemci> _gozlemciler = new List<IHavaGozlemci>();

    public void GozlemciEkle(IHavaGozlemci g)   { _gozlemciler.Add(g);    Console.WriteLine($"  + Abone: {g.GetType().Name}"); }
    public void GozlemciCikar(IHavaGozlemci g)  { _gozlemciler.Remove(g); Console.WriteLine($"  - Abone çıkarıldı: {g.GetType().Name}"); }

    public void Guncelle(string sehir, double sicaklik, double nem)
    {
        Console.WriteLine($"\n[Merkez] {sehir} güncellendi → {sicaklik}°C, %{nem} nem");
        foreach (var g in _gozlemciler)     // tüm aboneleri bilgilendir
            g.Guncellendi(sehir, sicaklik, nem);
    }
}

// Concrete Observer 1
public class TelefonApp : IHavaGozlemci
{
    public void Guncellendi(string sehir, double s, double n)
        => Console.WriteLine($"  [Telefon] 📱 {sehir}: {s}°C");
}

// Concrete Observer 2
public class HavaDurumEkrani : IHavaGozlemci
{
    public void Guncellendi(string sehir, double s, double n)
    {
        string durum = s > 30 ? "Çok Sıcak" : s > 15 ? "Ilık" : "Soğuk";
        Console.WriteLine($"  [Ekran]   🖥  {sehir}: {s}°C / %{n} nem — {durum}");
    }
}

// Concrete Observer 3
public class UyariSistemi : IHavaGozlemci
{
    public void Guncellendi(string sehir, double s, double n)
    {
        if (s > 38)
            Console.WriteLine($"  [UYARI]   ⚠  {sehir} tehlikeli sıcaklık: {s}°C!");
    }
}

class Program
{
    static void Main()
    {
        HavaMerkezi merkez = new HavaMerkezi();
        merkez.GozlemciEkle(new TelefonApp());
        merkez.GozlemciEkle(new HavaDurumEkrani());
        merkez.GozlemciEkle(new UyariSistemi());

        merkez.Guncelle("Kayseri",  8.5, 65);
        merkez.Guncelle("Ankara",  34.0, 30);
        merkez.Guncelle("Adana",   40.5, 55);   // uyarı tetiklenir
    }
}
```

4. STRATEGY PATTERN

4.1. Problem

Bir işlemi farklı algoritmalarla gerçekleştirmeniz gerektiğinde, bunları tek bir sınıfın içinde `if-else` veya `switch` bloklarıyla yönetmek başlangıçta kolay görünür. Ancak her yeni algoritma eklendikçe bu blok büyür, test edilmesi zorlaşır ve bir kısmı değiştirildiğinde diğerlerinin de etkilenme riski artar.

4.2. Çözüm

Strategy, her algoritmayı ayrı bir sınıfa taşır. Bu sınıflar ortak bir interface'i gerçekler. Algoritmayı kullanan bağlam sınıfı (context) yalnızca interface'e bağlıdır; hangi algoritmanın arkasında çalıştığını bilmez. Algoritma çalışma zamanında değiştirilebilir.

> **Gerçek Dünya Analojisi**
> Bir navigasyon uygulaması düşünün. "En hızlı yol", "en kısa yol" veya "otoyolsuz yol" seçenekleri vardır. Uygulama her seferinde aynı hedefe gider fakat arkada farklı bir rota algoritması çalışır. Kullanıcı seçimi değiştirir — uygulama kodu değişmez.

4.3. Ne Zaman Kullanılır?

- Bir işlemin birden fazla yapılış biçimi olduğunda
- Algoritmanın çalışma zamanında değiştirileceği durumlarda
- Büyük `if-else` bloklarını temizlemek için
- Farklı müşteri/kullanıcı türlerine göre davranış değiştiğinde

4.4. Katılımcılar

Rol	Açıklama
IStrategy	Tüm stratejilerin uyması gereken arayüz
ConcreteStrategy	Her bir algoritmayı ayrı sınıfta gerçekler
Context	Stratejiyi tutan ve kullanan bağlam sınıfı

4.5. Pattern Olmadan vs. Pattern İle

❌ **Pattern olmadan**

```
void OdemeYap(string tur, double t) {
  if (tur == "kredi")
    // kredi kodu...
  else if (tur == "nakit")
    // nakit kodu...
  else if (tur == "kripto")
    // kripto kodu...
  // yeni ödeme → bloğu değiştir!
}
```

✓ **Strategy ile**

```
class Sepet {
  IOdemeYontemi _yontem;
  void YontemSec(IOdemeYontemi y)
    { _yontem = y; }
  void OdemeYap(double t)
    { _yontem.Ode(t); }
}
// yeni ödeme → yeni sınıf!
```

> **Dikkat:** Strategy ile Observer benzer görünebilir. Fark şudur: Strategy *nasıl yapılacağını* değiştirir (algoritma seçimi), Observer ise *ne zaman haber verileceğini* yönetir (olay bildirimi).

4.6. Kod Örneği — Ödeme Sistemi

```
using System;

// Strateji arayüzü
public interface IOdemeYontemi
{
    void Ode(double tutar);
    string Ad { get; }
}

// Strateji 1
public class KrediKartiOdeme : IOdemeYontemi
{
    public string Ad => "Kredi Kartı";
    public void Ode(double tutar)
        => Console.WriteLine($"  [Kredi Kartı] {tutar:F2} TL tahsil edildi.");
}

// Strateji 2
public class NakitOdeme : IOdemeYontemi
{
    public string Ad => "Nakit";
    public void Ode(double tutar)
        => Console.WriteLine($"  [Nakit] {tutar:F2} TL alındı, para üstü verildi.");
}

// Strateji 3
public class TaksitliOdeme : IOdemeYontemi
{
    private int _taksit;
    public string Ad => $"{_taksit} Taksit";
    public TaksitliOdeme(int taksit) { _taksit = taksit; }
    public void Ode(double tutar)
        => Console.WriteLine($"  [Taksit] {_taksit} × {tutar / _taksit:F2} TL = toplam {tutar:F2} TL");
}

// Bağlam sınıfı (Context)
public class AlisverisSepeti
{
    private IOdemeYontemi _yontem;
    private double _toplam;

    public AlisverisSepeti(double toplam) { _toplam = toplam; }

    public void YontemSec(IOdemeYontemi yontem)
    {
        _yontem = yontem;
        Console.WriteLine($"\nÖdeme yöntemi: {_yontem.Ad}");
    }

    public void OdemeYap() { _yontem.Ode(_toplam); }
}

class Program
{
    static void Main()
    {
        AlisverisSepeti sepet = new AlisverisSepeti(3750.00);

        sepet.YontemSec(new KrediKartiOdeme());   sepet.OdemeYap();
        sepet.YontemSec(new TaksitliOdeme(12));   sepet.OdemeYap();
        sepet.YontemSec(new NakitOdeme());         sepet.OdemeYap();
        // Yeni ödeme yöntemi eklemek için bu koda dokunmaya gerek yok!
    }
}
```

5. FACTORY PATTERN

5.1. Problem

Nesne oluşturmak için `new` anahtar kelimesini her yerde kullanmak başlangıçta sorunsuz görünür. Ancak hangi sınıfın oluşturulacağına karar veren kod birçok yere yayıldığında, yeni bir sınıf eklediğinizde bu kararların bulunduğu her yeri güncellemeniz gerekir. Bu hem zahmetlidir hem de hata riski taşır.

5.2. Çözüm

Factory pattern, nesne oluşturma sorumluluğunu merkezi bir noktaya toplar. Bu merkez bir statik metot veya ayrı bir sınıf olabilir. Çağıran kod sadece "bu türde

5.5. Pattern Olmadan vs. Pattern İle

✗ **Pattern olmadan**

```
// Her yerde new + if blokları
if (tur == "email")
  bildirim = new EmailBildirim();
else if (tur == "sms")
  bildirim = new SmsBildirim();
// aynı blok 5 ayrı yerde!
```

✓ **Factory ile**

```
// Tek satır, her yerde
IBildirim b =
  BildirimFabrika.Olustur("email");

// Fabrika içi karar tek yerde,
// çağıranlar bilmez, umursamaz.
```

bir nesne istiyorum" der; nasıl oluşturulduğunu bilmesine gerek yoktur. Yeni bir tür eklemek yalnızca fabrika içinde bir değişiklik gerektirir.

> **Gerçek Dünya Analojisi**
>
> Bir fabrikayı düşünün: "Bana bir araba üret" dersiniz. Fabrika hangi parçaların nasıl birleştirileceğini bilir; siz sadece siparişi verirsiniz. Farklı modeller aynı siparişle üretilebilir.

5.3. Ne Zaman Kullanılır?

- Hangi sınıfın oluşturulacağı çalışma zamanında belirlendiğinde
- Nesne oluşturma mantığı karmaşık olduğunda
- Oluşturma kodunu tek bir yerde toplamak gerektiğinde
- Farklı türler aynı arayüzü paylaştığında

5.4. Katılımcılar

Rol	Açıklama
IUrun	Tüm ürünlerin ortak arayüzü
ConcreteUrun	Arayüzü gerçekleyen somut sınıflar
Fabrika	Türe göre doğru nesneyi oluşturan merkez

Not: Bu deneyde *Static Factory* kullanılacaktır. GoF'un tanımladığı *Factory Method* ve *Abstract Factory* pattern'ları daha gelişmiş yapılardır ve ayrı bir konu olarak ele alınır.

5.6. Kod Örneği — Bildirim Fabrikası

```
using System;

// Ortak arayüz
public interface IBildirim
{
    void Gonder(string alici, string mesaj);
    string Kanal { get; }
}

// Concrete sınıflar
public class EmailBildirim : IBildirim
{
    public string Kanal => "Email";
    public void Gonder(string alici, string mesaj)
        => Console.WriteLine($"  [Email → {alici}] {mesaj}");
}

public class SmsBildirim : IBildirim
{
    public string Kanal => "SMS";
    public void Gonder(string alici, string mesaj)
        => Console.WriteLine($"  [SMS → {alici}] {mesaj}");
}

public class PushBildirim : IBildirim
{
    public string Kanal => "Push";
    public void Gonder(string alici, string mesaj)
        => Console.WriteLine($"  [Push → {alici}] 🔔 {mesaj}");
}

// Fabrika — tek karar noktası
public static class BildirimFabrika
{
    public static IBildirim Olustur(string kanal)
    {
        switch (kanal.ToLower())
        {
            case "email": return new EmailBildirim();
            case "sms":   return new SmsBildirim();
            case "push":  return new PushBildirim();
            default:      throw new ArgumentException($"Bilinmeyen kanal: {kanal}");
        }
    }
}

class Program
{
    static void Main()
    {
        // Çağıran kod hangi sınıfın üretildiğini bilmez
        string[] kanallar = { "email", "sms", "push" };

        foreach (string k in kanallar)
        {
            IBildirim b = BildirimFabrika.Olustur(k);
            Console.WriteLine($"\n{b.Kanal} kanalı oluşturuldu:");
            b.Gonder("kullanici@test.com", "Siparişiniz kargoya verildi.");
        }

        // Hatalı kanal → Exception fırlatılır
        try { BildirimFabrika.Olustur("faks"); }
        catch (ArgumentException ex) { Console.WriteLine($"\nHata: {ex.Message}"); }
    }
}
```

6. PATTERN'LARIN KARŞILAŞTIRMASI

Pattern	Kategori	Çözdüğü Problem	Anahtar Yapı
Singleton	Creational	Yalnızca bir nesne örneğinin var olmasını garanti eder	`private` yapıcı, `static` alan
Factory	Creational	Nesne oluşturma kodunu tek bir merkeze toplar	`static` fabrika metodu, `interface`
Observer	Behavioral	Durum değişikliklerini ilgili nesnelere otomatik iletir	Gözlemci listesi, `foreach` bildirimi
Strategy	Behavioral	Algoritmayı çalışma zamanında değiştirmeyi sağlar	Strateji `interface`'i, bağlam sınıfı

7. DERSİN ÖĞRENME ÇIKTILARI

Bu deney çalışmasının sonunda aşağıdaki kazanımların elde edilmesi beklenmektedir:

- Design pattern kavramını, tarihsel arka planını ve yazılım geliştirmedeki yerini açıklayabilmek,
- Creational, Structural ve Behavioral kategorileri birbirinden ayırt ederek örnekleyebilmek,
- Singleton pattern ile tek nesne garantisini `sealed`, `static` ve `private` yapılarını kullanarak C#'ta uygulayabilmek,
- Observer pattern ile subject-observer ilişkisini kurabilmek; gözlemci ekleyip çıkardıkça sistemin davranışını gözlemleyebilmek,
- Strategy pattern ile birbirinin yerine geçebilir algoritmaları `interface` üzerinden tasarlayabilmek ve çalışma zamanında değiştirebilmek,
- Factory pattern ile nesne oluşturma sorumluluğunu merkezi bir yapıya devredebilmek ve yeni türlerin eklenmesini kolaylaştırabilmek,
- Birden fazla pattern'ı aynı projede anlamlı biçimde bir arada kullanarak genişletilebilir ve bakımı kolay sınıf yapıları oluşturabilmek.