1. **Singleton**

Singleton zapewnia, że istnieje tylko jedna instancja danej klasy. Umożliwia globalny dostęp do tej instancji. Zapobiega wielokrotnemu tworzeniu obiektu.

using System;

namespace SingletonPrzyklad

{

public sealed class Logger

{

private static readonly Logger \_instance = new Logger();

public static Logger Instance

{

get { return \_instance; }

}

private Logger() { }

public void Log(string message)

{

Console.WriteLine("[Log] " + message);

}

}

class Program

{

static void Main(string[] args)

{

Logger.Instance.Log("Uruchomienie aplikacji");

}

}

}

1. **Adapter**

Adapter umożliwia współpracę klas o niekompatybilnych interfejsach. Opakowuje starą klasę w nowy interfejs. Przydaje się przy integracji ze starym kodem.

using System;

namespace AdapterPrzyklad

{

interface INowyOdtwarzacz

{

void OdtworzDzwiek();

}

class StaryOdtwarzacz

{

public void PlayOldSound()

{

Console.WriteLine("Odtwarzanie dźwięku (stary format).");

}

}

class OdtwarzaczAdapter : INowyOdtwarzacz

{

private StaryOdtwarzacz \_stary;

public OdtwarzaczAdapter(StaryOdtwarzacz stary)

{

\_stary = stary;

}

public void OdtworzDzwiek()

{

\_stary.PlayOldSound();

}

}

class Program

{

static void Main(string[] args)

{

INowyOdtwarzacz odtwarzacz = new OdtwarzaczAdapter(new StaryOdtwarzacz());

odtwarzacz.OdtworzDzwiek();

}

}

}

1. **Fasada**

Fasada upraszcza korzystanie ze złożonego systemu przez dostarczenie uproszczonego interfejsu. Ukrywa szczegóły implementacji. Pomaga organizować kod.

using System;

namespace FasadaPrzyklad

{

class Silnik

{

public void Start()

{

Console.WriteLine("Silnik uruchomiony.");

}

}

class SkrzyniaBiegow

{

public void ZmienNaPoczatek()

{

Console.WriteLine("Bieg wsteczny ustawiony.");

}

}

class SamochodFasada

{

private Silnik \_silnik;

private SkrzyniaBiegow \_skrzynia;

public SamochodFasada()

{

\_silnik = new Silnik();

\_skrzynia = new SkrzyniaBiegow();

}

public void UruchomSamochod()

{

\_skrzynia.ZmienNaPoczatek();

\_silnik.Start();

}

}

class Program

{

static void Main(string[] args)

{

SamochodFasada auto = new SamochodFasada();

auto.UruchomSamochod();

}

}

}

1. **Kompozyt**

Kompozyt pozwala traktować grupy obiektów tak samo jak pojedyncze. Struktura przypomina drzewo. Pomaga budować złożone struktury danych.

using System;

using System.Collections.Generic;

namespace KompozytPrzyklad

{

abstract class Komponent

{

public abstract void Wyswietl();

}

class Przycisk : Komponent

{

public override void Wyswietl()

{

Console.WriteLine("Przycisk");

}

}

class Panel : Komponent

{

private List<Komponent> \_komponenty;

public Panel()

{

\_komponenty = new List<Komponent>();

}

public void Dodaj(Komponent komponent)

{

\_komponenty.Add(komponent);

}

public override void Wyswietl()

{

Console.WriteLine("Panel:");

foreach (Komponent k in \_komponenty)

{

k.Wyswietl();

}

}

}

class Program

{

static void Main(string[] args)

{

Panel panel = new Panel();

panel.Dodaj(new Przycisk());

panel.Dodaj(new Przycisk());

panel.Wyswietl();

}

}

}

1. **Chain of Responsibility**

Tworzy łańcuch obiektów, przez który przekazywane jest żądanie. Każdy element decyduje, czy obsłużyć żądanie, czy przekazać dalej. Ułatwia dodawanie nowych kroków.

using System;

namespace ChainPrzyklad

{

abstract class Handler

{

protected Handler \_nastepny;

public void UstawNastepny(Handler handler)

{

\_nastepny = handler;

}

public abstract void Obsluz(string wiadomosc);

}

class EmailHandler : Handler

{

public override void Obsluz(string wiadomosc)

{

if (wiadomosc.Contains("@"))

{

Console.WriteLine("Obsłużono jako e-mail.");

}

else if (\_nastepny != null)

{

\_nastepny.Obsluz(wiadomosc);

}

}

}

class SmsHandler : Handler

{

public override void Obsluz(string wiadomosc)

{

Console.WriteLine("Obsłużono jako SMS.");

}

}

class Program

{

static void Main(string[] args)

{

Handler email = new EmailHandler();

Handler sms = new SmsHandler();

email.UstawNastepny(sms);

email.Obsluz("test@ex.com");

email.Obsluz("123456");

}

}

}

1. **Prosta Fabryka**

Tworzy obiekty na podstawie podanych parametrów. Ułatwia zarządzanie tworzeniem obiektów. Klient nie zna dokładnej klasy tworzonego obiektu.

using System;

namespace ProstaFabryka

{

public interface IPojazd

{

void Jedz();

}

public class Samochod : IPojazd

{

public void Jedz()

{

Console.WriteLine("Samochód jedzie");

}

}

public class Rower : IPojazd

{

public void Jedz()

{

Console.WriteLine("Rower jedzie");

}

}

public class PojazdFabryka

{

public IPojazd StworzPojazd(string typ)

{

if (typ == "samochod")

return new Samochod();

else if (typ == "rower")

return new Rower();

else

throw new ArgumentException("Nieznany typ pojazdu");

}

}

class Program

{

static void Main(string[] args)

{

PojazdFabryka fabryka = new PojazdFabryka();

IPojazd pojazd = fabryka.StworzPojazd("samochod");

pojazd.Jedz();

}

}

}

1. **Fabryka**

Podklasy decydują, jaki obiekt ma być utworzony. Ułatwia rozszerzanie bez zmieniania istniejącego kodu. Wzorzec często używany z interfejsami.

using System;

namespace FabrykaMetoda

{

public abstract class Dokument

{

public abstract void Drukuj();

}

public class Pdf : Dokument

{

public override void Drukuj()

{

Console.WriteLine("Drukuję PDF");

}

}

public abstract class Drukarnia

{

public abstract Dokument UtworzDokument();

}

public class PdfDrukarnia : Drukarnia

{

public override Dokument UtworzDokument()

{

return new Pdf();

}

}

class Program

{

static void Main(string[] args)

{

Drukarnia drukarnia = new PdfDrukarnia();

Dokument dokument = drukarnia.UtworzDokument();

dokument.Drukuj();

}

}

}

1. **Fabryka abstrakcyjna**

Tworzy całe rodziny spójnych obiektów. Oddziela kod klienta od konkretnej platformy. Pomaga przy wielu środowiskach (np. Windows/Mac).

using System;

namespace FabrykaAbstrakcyjna

{

public interface IPrzycisk

{

void Rysuj();

}

public class WinPrzycisk : IPrzycisk

{

public void Rysuj()

{

Console.WriteLine("Rysuję przycisk Windows");

}

}

public class MacPrzycisk : IPrzycisk

{

public void Rysuj()

{

Console.WriteLine("Rysuję przycisk Mac");

}

}

public interface IUIPakietFabryka

{

IPrzycisk StworzPrzycisk();

}

public class WinFabryka : IUIPakietFabryka

{

public IPrzycisk StworzPrzycisk()

{

return new WinPrzycisk();

}

}

public class MacFabryka : IUIPakietFabryka

{

public IPrzycisk StworzPrzycisk()

{

return new MacPrzycisk();

}

}

class Program

{

static void Main(string[] args)

{

IUIPakietFabryka fabryka = new WinFabryka();

IPrzycisk przycisk = fabryka.StworzPrzycisk();

przycisk.Rysuj();

}

}

}

1. **MVC**

Model-View-Controller dzieli aplikację na 3 warstwy. Model przechowuje dane, View je wyświetla, Controller obsługuje logikę. Ułatwia zarządzanie dużym kodem.

using System;

namespace MVCPrzyklad

{

public class Model

{

public string Dane { get; set; }

}

public class Widok

{

public void Pokaz(string dane)

{

Console.WriteLine("Dane: " + dane);

}

}

public class Kontroler

{

private Model \_model;

private Widok \_widok;

public Kontroler(Model model, Widok widok)

{

\_model = model;

\_widok = widok;

}

public void AktualizujDane(string noweDane)

{

\_model.Dane = noweDane;

\_widok.Pokaz(\_model.Dane);

}

}

class Program

{

static void Main(string[] args)

{

Model model = new Model();

Widok widok = new Widok();

Kontroler kontroler = new Kontroler(model, widok);

kontroler.AktualizujDane("Hello MVC");

}

}

}

1. **Obserwator**

Obserwator reaguje na zmiany w obiekcie nadrzędnym. Subskrybenci są automatycznie powiadamiani. Wzorzec przydatny w GUI i systemach zdarzeń.

using System;

using System.Collections.Generic;

namespace ObserwatorPrzyklad

{

public interface IObserver

{

void Aktualizuj(string wiadomosc);

}

public class Uzytkownik : IObserver

{

private string \_nazwa;

public Uzytkownik(string nazwa)

{

\_nazwa = nazwa;

}

public void Aktualizuj(string wiadomosc)

{

Console.WriteLine(\_nazwa + " otrzymał: " + wiadomosc);

}

}

public class Powiadomienia

{

private List<IObserver> \_obserwatorzy;

public Powiadomienia()

{

\_obserwatorzy = new List<IObserver>();

}

public void Dodaj(IObserver observer)

{

\_obserwatorzy.Add(observer);

}

public void WyslijWiadomosc(string tekst)

{

foreach (IObserver obs in \_obserwatorzy)

{

obs.Aktualizuj(tekst);

}

}

}

class Program

{

static void Main(string[] args)

{

Powiadomienia powiadomienia = new Powiadomienia();

powiadomienia.Dodaj(new Uzytkownik("Jan"));

powiadomienia.Dodaj(new Uzytkownik("Anna"));

powiadomienia.WyslijWiadomosc("Nowa wiadomość!");

}

}

}

1. **Dependency Injection**

Zależności są przekazywane z zewnątrz, zamiast tworzenia ich w środku klasy. Ułatwia testowanie i wymianę komponentów. Wspiera luźne powiązanie.

using System;

namespace DIPrzyklad

{

public interface ILogger

{

void Loguj(string tekst);

}

public class KonsolaLogger : ILogger

{

public void Loguj(string tekst)

{

Console.WriteLine("[Log] " + tekst);

}

}

public class Serwis

{

private ILogger \_logger;

public Serwis(ILogger logger)

{

\_logger = logger;

}

public void Wykonaj()

{

\_logger.Loguj("Wykonywanie operacji");

}

}

class Program

{

static void Main(string[] args)

{

ILogger logger = new KonsolaLogger();

Serwis serwis = new Serwis(logger);

serwis.Wykonaj();

}

}

}