Design Patterns and Principles - Solutions

# Exercise 1: Singleton Pattern

Java Project Name: SingletonPatternExample

Code:

public class Logger {

private static Logger \_instance;

private static readonly object \_lock = new object();

private Logger() {

Console.WriteLine("Logger initialized.");

}

public static Logger Instance {

get {

lock (\_lock) {

if (\_instance == null) {

\_instance = new Logger();

}

return \_instance;

}

}

}

public void Log(string message) {

Console.WriteLine("Log: " + message);

}

}

class Program {

static void Main(string[] args) {

Logger logger1 = Logger.Instance;

Logger logger2 = Logger.Instance;

logger1.Log("Singleton Pattern in C#");

Console.WriteLine("Same instance? " + (logger1 == logger2));

}

}

Output:

Logger initialized.

Log: Singleton Pattern in C#

Same instance? True

# Exercise 2: Factory Method Pattern

Java Project Name: FactoryMethodPattern

Code:

public interface IDocument {

void Open();

}

public class WordDocument : IDocument {

public void Open() => Console.WriteLine("Opening Word Document...");

}

public class PdfDocument : IDocument {

public void Open() => Console.WriteLine("Opening PDF Document...");

}

public abstract class DocumentFactory {

public abstract IDocument CreateDocument();

}

public class WordFactory : DocumentFactory {

public override IDocument CreateDocument() => new WordDocument();

}

public class PdfFactory : DocumentFactory {

public override IDocument CreateDocument() => new PdfDocument();

}

class Program {

static void Main(string[] args) {

DocumentFactory factory = new WordFactory();

IDocument doc1 = factory.CreateDocument();

doc1.Open();

factory = new PdfFactory();

IDocument doc2 = factory.CreateDocument();

doc2.Open();

}

}

Output:

Opening Word Document...

Opening PDF Document...

# Exercise 3: Builder Pattern

Java Project Name: BuilderPatternExample

Code:

public class Computer {

public string CPU { get; set; }

public string RAM { get; set; }

public string Storage { get; set; }

private Computer() {}

public override string ToString() {

return $"CPU: {CPU}, RAM: {RAM}, Storage: {Storage}";

}

public class Builder {

private readonly Computer \_computer = new Computer();

public Builder SetCPU(string cpu) {

\_computer.CPU = cpu;

return this;

}

public Builder SetRAM(string ram) {

\_computer.RAM = ram;

return this;

}

public Builder SetStorage(string storage) {

\_computer.Storage = storage;

return this;

}

public Computer Build() {

return \_computer;

}

}

}

class Program {

static void Main(string[] args) {

var computer = new Computer.Builder()

.SetCPU("Intel i7")

.SetRAM("16GB")

.SetStorage("512GB SSD")

.Build();

Console.WriteLine(computer);

}

}

Output:

CPU: Intel i7, RAM: 16GB, Storage: 512GB SSD

# Exercise 4: Adapter Pattern

Java Project Name: AdapterPatternExample

Code:

public interface IPaymentProcessor {

void Pay(string amount);

}

public class PayPalGateway {

public void MakePayment(string amount) {

Console.WriteLine($"Payment made using PayPal: {amount}");

}

}

public class PayPalAdapter : IPaymentProcessor {

private readonly PayPalGateway \_gateway = new PayPalGateway();

public void Pay(string amount) {

\_gateway.MakePayment(amount);

}

}

class Program {

static void Main(string[] args) {

IPaymentProcessor processor = new PayPalAdapter();

processor.Pay("1000");

}

}

Output:

Payment made using PayPal: 1000

# Exercise 5: Decorator Pattern

Java Project Name: DecoratorPatternExample

Code:

public interface INotifier {

void Send(string message);

}

public class EmailNotifier : INotifier {

public void Send(string message) {

Console.WriteLine($"Sending Email: {message}");

}

}

public class NotifierDecorator : INotifier {

protected INotifier \_notifier;

public NotifierDecorator(INotifier notifier) {

\_notifier = notifier;

}

public virtual void Send(string message) {

\_notifier.Send(message);

}

}

public class SMSNotifier : NotifierDecorator {

public SMSNotifier(INotifier notifier) : base(notifier) {}

public override void Send(string message) {

base.Send(message);

Console.WriteLine($"Sending SMS: {message}");

}

}

class Program {

static void Main(string[] args) {

INotifier notifier = new SMSNotifier(new EmailNotifier());

notifier.Send("Hello World!");

}

}

Output:

Sending Email: Hello World!

Sending SMS: Hello World!

# Exercise 6: Proxy Pattern

Java Project Name: ProxyPatternExample

Code:

public interface IImage {

void Display();

}

public class RealImage : IImage {

private string \_filename;

public RealImage(string filename) {

\_filename = filename;

LoadFromDisk();

}

private void LoadFromDisk() {

Console.WriteLine($"Loading {\_filename}");

}

public void Display() {

Console.WriteLine($"Displaying {\_filename}");

}

}

public class ProxyImage : IImage {

private RealImage \_realImage;

private string \_filename;

public ProxyImage(string filename) {

\_filename = filename;

}

public void Display() {

if (\_realImage == null) {

\_realImage = new RealImage(\_filename);

}

\_realImage.Display();

}

}

class Program {

static void Main(string[] args) {

IImage image = new ProxyImage("photo.jpg");

image.Display(); // Loads and displays

image.Display(); // Only displays

}

}

Output:

Loading photo.jpg

Displaying photo.jpg

Displaying photo.jpg

# **Exercise 7: Observer Pattern**

Java Project Name: ObserverPatternExample

Code:

public interface IObserver {

void Update(float price);

}

// IStock.cs

public interface IStock {

void Register(IObserver observer);

void Unregister(IObserver observer);

void NotifyObservers();

}

// StockMarket.cs

public class StockMarket : IStock {

private List<IObserver> \_observers = new();

private float \_price;

public void Register(IObserver observer) => \_observers.Add(observer);

public void Unregister(IObserver observer) => \_observers.Remove(observer);

public void SetPrice(float price) {

\_price = price;

NotifyObservers();

}

public void NotifyObservers() {

foreach (var observer in \_observers) {

observer.Update(\_price);

}

}

}

// MobileApp.cs

public class MobileApp : IObserver {

public void Update(float price) {

Console.WriteLine($"Mobile App: Stock price updated to {price}");

}

}

// WebApp.cs

public class WebApp : IObserver {

public void Update(float price) {

Console.WriteLine($"Web App: Stock price updated to {price}");

}

}

class Program {

static void Main(string[] args) {

StockMarket stock = new StockMarket();

IObserver mobile = new MobileApp();

IObserver web = new WebApp();

stock.Register(mobile);

stock.Register(web);

stock.SetPrice(101.5f);

}

}

Output:

Mobile App: Stock price updated to 101.5

Web App: Stock price updated to 101.5

# Exercise 8: Strategy Pattern

Java Project Name: StrategyPatternExample

Code:

public interface IPaymentStrategy {

void Pay(decimal amount);

}

public class CreditCardPayment : IPaymentStrategy {

public void Pay(decimal amount) {

Console.WriteLine($"Paid {amount} using Credit Card.");

}

}

public class PayPalPayment : IPaymentStrategy {

public void Pay(decimal amount) {

Console.WriteLine($"Paid {amount} using PayPal.");

}

}

public class PaymentContext {

private IPaymentStrategy \_strategy;

public void SetPaymentStrategy(IPaymentStrategy strategy) {

\_strategy = strategy;

}

public void PayAmount(decimal amount) {

\_strategy.Pay(amount);

}

}

class Program {

static void Main(string[] args) {

PaymentContext context = new PaymentContext();

context.SetPaymentStrategy(new CreditCardPayment());

context.PayAmount(500);

context.SetPaymentStrategy(new PayPalPayment());

context.PayAmount(750);

}

}

Output:

Paid 500 using Credit Card.

Paid 750 using PayPal.

# Exercise 9: Command Pattern

Java Project Name: CommandPatternExample

Code:

public interface ICommand {

void Execute();

}

// Receiver

public class Light {

public void TurnOn() => Console.WriteLine("Light is ON");

public void TurnOff() => Console.WriteLine("Light is OFF");

}

// Concrete Commands

public class LightOnCommand : ICommand {

private Light \_light;

public LightOnCommand(Light light) => \_light = light;

public void Execute() => \_light.TurnOn();

}

public class LightOffCommand : ICommand {

private Light \_light;

public LightOffCommand(Light light) => \_light = light;

public void Execute() => \_light.TurnOff();

}

// Invoker

public class RemoteControl {

private ICommand \_command;

public void SetCommand(ICommand command) {

\_command = command;

}

public void PressButton() {

\_command.Execute();

}

}

class Program {

static void Main(string[] args) {

Light light = new Light();

ICommand onCommand = new LightOnCommand(light);

ICommand offCommand = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.SetCommand(onCommand);

remote.PressButton();

remote.SetCommand(offCommand);

remote.PressButton();

}

}

Output:

Light is ON

Light is OFF

# Exercise 10: MVC Pattern

Java Project Name: MVCPatternExample

Code:

public class Student {

public string Name { get; set; }

public int ID { get; set; }

public string Grade { get; set; }

}

// StudentView.cs (View)

public class StudentView {

public void DisplayStudentDetails(Student student) {

Console.WriteLine($"Student: {student.Name}, ID: {student.ID}, Grade: {student.Grade}");

}

}

// StudentController.cs (Controller)

public class StudentController {

private Student \_student;

private StudentView \_view;

public StudentController(Student student, StudentView view) {

\_student = student;

\_view = view;

}

public void SetName(string name) => \_student.Name = name;

public void SetGrade(string grade) => \_student.Grade = grade;

public void UpdateView() => \_view.DisplayStudentDetails(\_student);

}

class Program {

static void Main(string[] args) {

Student student = new Student { Name = "Alice", ID = 1, Grade = "A" };

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.UpdateView();

controller.SetGrade("A+");

controller.UpdateView();

}

}

Output:

Student: Alice, ID: 1, Grade: A

Student: Alice, ID: 1, Grade: A+

# Exercise 11: Dependency Injection

Java Project Name: DependencyInjectionExample

Code:

public interface ICustomerRepository {

string FindCustomerById(int id);

}

public class CustomerRepository : ICustomerRepository {

public string FindCustomerById(int id) => $"Customer #{id}: John Doe";

}

// CustomerService.cs

public class CustomerService {

private ICustomerRepository \_repository;

public CustomerService(ICustomerRepository repository) {

\_repository = repository;

}

public void GetCustomer(int id) {

Console.WriteLine(\_repository.FindCustomerById(id));

}

}

class Program {

static void Main(string[] args) {

ICustomerRepository repo = new CustomerRepository();

CustomerService service = new CustomerService(repo);

service.GetCustomer(101);

}

}

Output:

Customer #101: John Doe