Design Patterns:

## Singleton:

class Logger

{

private static Logger instance; // static instance which is private

private static object sync = new Object(); // object for locking

private Logger() { } // private instance of contructor

public string Name

{

get;

set;

}

public static Logger Instance // this would return the instance of the logger

{

get

{

lock (sync)

{

if (instance == null)

{

instance = new Logger();

}

}

return instance;

}

}

}

## Usage:

Logger log = Logger.Instance;

log.Name = "Name";

Logger log1 = Logger.Instance;

Console.WriteLine("Logger name = " + log1.Name);

## Multiton:

public class Camera

{

static Dictionary<int, Camera> cameras = new Dictionary<int, Camera>();

static object lockCamera = new object();

private Camera()

{

HardwareId = Guid.NewGuid();

}

public Guid HardwareId { get; private set; }

public static Camera GetCamera(int cameraCode)

{

lock (lockCamera)

{

if (!cameras.ContainsKey(cameraCode))

{

cameras.Add(cameraCode, new Camera());

}

}

return cameras[cameraCode];

}

}

## Usage:

Camera cam1a = Camera.GetCamera(1);

Camera cam1b = Camera.GetCamera(1);

Camera cam2a = Camera.GetCamera(2);

Camera cam2b = Camera.GetCamera(2);

Console.WriteLine(cam1a.HardwareId);

Console.WriteLine(cam1b.HardwareId);

Console.WriteLine(cam2a.HardwareId);

Console.WriteLine(cam2b.HardwareId);

## State Design Pattern:

public enum Button{ Home, Settings, Apps};

abstract class State

{

public abstract void Handle(Context context, Button btn);

}

class Context

{

private State stateInstance;

public Context(State state)

{

this.stateInstance = state;

}

public State state

{

get { return stateInstance; }

set

{

stateInstance = value;

Console.WriteLine("State: " + state.GetType().Name);

}

}

public void Request(Button btn)

{

state.Handle(this,btn);

}

}

class HomeState : State

{

public override void Handle(Context context, Button btn)

{

if (btn == Button.Settings)

context.state = new SettingsState();

else if (btn == Button.Apps)

context.state = new AppsState();

// else do nothing since we are at the home state

}

}

class SettingsState : State

{

public override void Handle(Context context, Button btn)

{

if (btn == Button.Home)

context.state = new HomeState();

else if (btn == Button.Apps)

context.state = new AppsState();

// else do nothing since we are at the Settings State

}

}

class AppsState : State

{

public override void Handle(Context context, Button btn)

{

if (btn == Button.Home)

context.state = new HomeState();

else if (btn == Button.Settings)

context.state = new SettingsState();

// else do nothing since we are at the Apps State

}

}

## Usage:

Context context = new Context(new HomeState());

context.Request(Button.Settings);

context.Request(Button.Home);

context.Request(Button.Apps);

context.Request(Button.Apps);

context.Request(Button.Home);

## Façade Design Pattern:

class SubSystemOne

{

public void MethodOne()

{

Console.WriteLine("Sub System one method");

}

}

class SubSystemTwo

{

public void MethodTwo()

{

Console.WriteLine("Sub System two method");

}

}

class SubSystemThree

{

public void MethodThree()

{

Console.WriteLine("Sub System three method");

}

}

class Facade

{

private SubSystemOne subSystemOne;

private SubSystemTwo subSystemTwo;

private SubSystemThree subSystemThree;

public Facade()

{

subSystemOne = new SubSystemOne();

subSystemTwo = new SubSystemTwo();

subSystemThree = new SubSystemThree();

}

// MethodA and MethodB are the new ones which are exposed from Facade

public void MethodA()

{

Console.WriteLine("\nMethodA() ---- ");

subSystemOne.MethodOne();

subSystemTwo.MethodTwo();

}

public void MethodB()

{

Console.WriteLine("\nMethodB() ---- ");

subSystemThree.MethodThree();

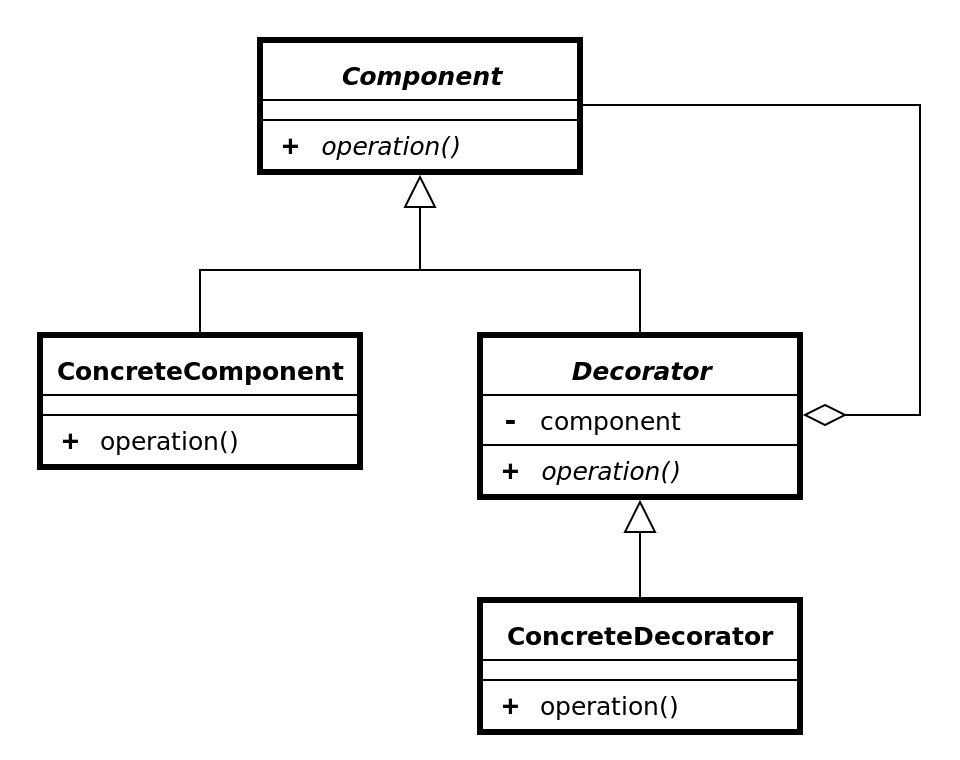
subSystemTwo.MethodTwo();

}

}

## Decorator or Adapter Design Pattern:

Allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from same class.



public abstract class Pizza

{

public abstract string GetName();

public abstract float GetPrice();

}

class PizzaBase : Pizza

{

private float fPrice = 10.00f;

private string szName = "Pizza Base";

public override float GetPrice()

{

return fPrice;

}

public override string GetName()

{

return szName;

}

}// there can be lot of different types of base concrete component

public class Decorator : Pizza

{

protected Pizza basePizza = null;

protected string szName = "Decorator";

protected float fPrice = 0.0f;

protected Decorator(Pizza objPizza)

{

basePizza = objPizza;

}

public override float GetPrice()

{

return fPrice + basePizza.GetPrice();

}

public override string GetName()

{

return string.Format("{0}, {1}", basePizza.GetName(), szName);

}

}// decorator class which will be used by other types of decorators

class OnionTopping : Decorator

{

public OnionTopping(Pizza baseComponent): base(baseComponent)

{

this.szName = "Onion";

this.fPrice = 2.0f;

}

}

class GreenPaperTopping : Decorator

{

public GreenPaperTopping(Pizza baseComponent)

: base(baseComponent)

{

this.szName = "Green Pepper";

this.fPrice = 2.0f;

}

}

class ChickenTopping : Decorator

{

public ChickenTopping(Pizza baseComponent)

: base(baseComponent)

{

this.szName = "Chicken";

this.fPrice = 3.0f;

}

}

PizzaBase basePizza = new PizzaBase();

OnionTopping onion = new OnionTopping(basePizza);

GreenPaperTopping greenPepper = new GreenPaperTopping(onion);

ChickenTopping chicken = new ChickenTopping(greenPepper);

Console.WriteLine("Item: {0}, Price: {1}", chicken.GetName(), chicken.GetPrice());

## Observer Pattern without Delegate:

class Observable

{

private string szValue;

// Property

public string ObservableValue

{

get { return szValue; }

set { szValue = value; }

}

private ArrayList observers = new ArrayList();

public void Subscribe(Observer observer)

{

observers.Add(observer);

}

public void Unsubscribe(Observer observer)

{

observers.Remove(observer);

}

public void Notify()

{

foreach (Observer o in observers)

{

o.Update();

}

}

}

class Observer

{

private string name;

private string observableName;

private Observable observerable;

// Constructor

public Observer(

Observable subject, string name)

{

this.observerable = subject;

this.name = name;

}

public void Update()

{

observableName = observerable.ObservableValue;

Console.WriteLine("Observer {0}'s new state is {1}",

name, observableName);

}

// Property

public Observable Subject

{

get { return observerable; }

set { observerable = value; }

}

}

## Usage:

// Observervable class instance

Observable s = new Observable();

// creates observers

Observer observer1 = new Observer(s, "X");

Observer observer2 = new Observer(s, "Y");

Observer observer3 = new Observer(s, "Z");

// Subscribe

s.Subscribe(observer1);

s.Subscribe(observer2);

s.Subscribe(observer3);

// Change Value and and notify observers

s.ObservableValue = "ABC";

s.Notify();

// Unsubscribe

s.Unsubscribe(observer3);

// Change Value again and then notify

s.ObservableValue = "DEF";

s.Notify();

## Observer Design Pattern with Delegates:

class Observable

{

private string szValue;

// Property

public string ObservableValue

{

get { return szValue; }

set { szValue = value; }

}

public delegate void StatusUpdate(string value);

public event StatusUpdate OnStatusUpdate = null;

public void Subscribe(Observer observer)

{

//For way 2 lets assign attach the observers with subjects

OnStatusUpdate += new StatusUpdate(observer.Update);

//observers.Add(observer);

}

public void Unsubscribe(Observer observer)

{

//For way 2 lets assign detach the observers with subjects

OnStatusUpdate -= new StatusUpdate(observer.Update);

}

public void Notify()

{

//For way 2 lets notify the observers with change

if (OnStatusUpdate != null)

{

OnStatusUpdate(szValue);

}

}

}

class Observer

{

private string name;

private string observableName;

private Observable observerable;

// Constructor

public Observer(

Observable subject, string name)

{

this.observerable = subject;

this.name = name;

}

public void Update(string szValue)

{

observableName = observerable.ObservableValue;

Console.WriteLine("Observer {0}'s new state is {1}",

name, szValue);

}

// Property

public Observable Subject

{

get { return observerable; }

set { observerable = value; }

}

}

## Usage:

// Observervable class instance

Observable s = new Observable();

// creates observers

Observer observer1 = new Observer(s, "X");

Observer observer2 = new Observer(s, "Y");

Observer observer3 = new Observer(s, "Z");

// Subscribe

s.Subscribe(observer1);

s.Subscribe(observer2);

s.Subscribe(observer3);

// Change Value and and notify observers

s.ObservableValue = "ABC";

s.Notify();

// Unsubscribe

s.Unsubscribe(observer3);

// Change Value again and then notify

s.ObservableValue = "DEF";

s.Notify();

## Simple Factory Design Pattern:

interface IConnection

{

void Connect();

void Disconnect();

}

public class ConnectionWiFi: IConnection

{

public void Connect()

{

Console.WriteLine("Inside Wifi connection");

}

public void Disconnect()

{

Console.WriteLine("Inside Wifi Disconnect");

}

public void WiFiOperation()

{

Console.WriteLine("WiFi specific operation");

}

}

public class Connection3G : IConnection

{

public void Connect()

{

Console.WriteLine("Inside 3G connection");

}

public void Disconnect()

{

Console.WriteLine("Inside 3G Disconnect");

}

public void ThreeGOperation()

{

Console.WriteLine("3G specific operation");

}

}

class ConnectionFactory

{

public static IConnection GetConnection(int i)

{

switch (i)

{

case 1:

ConnectionWiFi wifi = new ConnectionWiFi();

wifi.Connect();

wifi.Disconnect();

wifi.WiFiOperation();

return wifi;

case 2:

Connection3G threeG = new Connection3G();

threeG.Connect();

threeG.Disconnect();

threeG.WiFiOperation();

return threeG;

default :

return null;

};

}

}

## Factory Method Design Pattern:

public interface IConnection

{

void Connect();

void Disconnect();

}

public abstract class BaseConnectionFactory

{

public IConnection GetConnection()

{

IConnection myConn = this.CreateConnection();

myConn.Connect();

myConn.Disconnect();

return myConn;

}

public abstract IConnection CreateConnection();

}

public class ConnectionWiFiFactory : BaseConnectionFactory

{

public override IConnection CreateConnection()

{

ConnectionWiFi objWiFi = new ConnectionWiFi();

objWiFi.WiFiOperation();

return objWiFi;

}

}

public class Connection3GFactory : BaseConnectionFactory

{

public override IConnection CreateConnection()

{

Connection3G obj3G = new Connection3G();

obj3G.ThreeGOperation();

return obj3G;

}

}

Usage:

//Client Code

BaseConnectionFactory c = new Connection3GFactory();// Or new ConnectionWiFiFactory();

IConnection objCust = c.GetConnection();

## Strategy Pattern:

abstract class Strategy

{

public abstract void AlgoInterface();

}

class ConcreteStrategyA : Strategy

{

public override void AlgoInterface()

{

Console.WriteLine("Strategy A function called");

}

}

class ConcreteStrategyB : Strategy

{

public override void AlgoInterface()

{

Console.WriteLine("Strategy B function called");

}

}

class ConcreteStrategyC : Strategy

{

public override void AlgoInterface()

{

Console.WriteLine("Strategy C function called");

}

}

class Context

{

private Strategy strategy;

public Context(Strategy strategy)

{

this.strategy = strategy;

}

public void AlgoInterface()

{

strategy.AlgoInterface();

}

}

## Usage:

Context context;

// Three contexts following different strategies

context = new Context(new ConcreteStrategyA());

context.AlgoInterface();

context = new Context(new ConcreteStrategyB());

context.AlgoInterface();

context = new Context(new ConcreteStrategyC());

context.AlgoInterface();