Design Pattern

# Singleton

Stellt sicher, dass von einer Klasse genau 1 Objekt vorhanden ist

Ein Bild, das Schwarz, Dunkelheit enthält.

Automatisch generierte Beschreibung

public class Singleton  
{  
 public string Name { get; set; }  
 private Singleton(){}  
 private static Singleton instance = null;  
  
 private static Singleton Instance  
 {  
 get  
 {  
 if (instance == null)  
 {  
 instance = new Singleton();  
 }  
 return instance;  
 }  
 }  
}

Singleton myinstance = Singleton.Instance;  
myinstance.Name="Florian";  
Console.WriteLine(myinstance.Name);

# Builder

Ein Objekt viele optionale Parameter

Objekterstellung leicht und flexibel und wenig Konstruktor Überladungen

Ein Bild, das Text, Diagramm, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

public class BuildingBuilder  
{  
 private readonly Building \_building;  
  
 public BuildingBuilder()  
 {  
 \_building = new Building();  
 }  
  
 public BuildingBuilder SetName(string name)  
 {  
 \_building.Name = name;  
 return this;  
 }  
  
 public BuildingBuilder SetFloors(int floors)  
 {  
 \_building.Floors = floors;  
 return this;  
 }  
  
 public BuildingBuilder SetHasElevator(bool hasElevator)  
 {  
 \_building.HasElevator = hasElevator;  
 return this;  
 }  
  
 public Building Build()  
 {  
 return \_building;  
 }  
}

public class Building  
{  
 public string Name { get; set; }  
 public int Floors { get; set; }  
 public bool HasElevator { get; set; }  
  
 public override string ToString()  
 {  
 return $"Building: {Name}, Floors: {Floors}, Has Elevator: {HasElevator}";  
 }  
  
 public Building GetSchool()  
 {  
 return new BuildingBuilder()  
 .SetName("Local School")  
 .SetHasElevator(true)  
 .SetFloors(3)  
 .Build();  
 }  
}

# Factory

Aufruf einer Methode statt Konstruktor erzeugt

Ein Bild, das Text, Screenshot, Diagramm, Reihe enthält.

Automatisch generierte Beschreibung

public class Building  
{

public string Name { get; set; }  
public int Floors { get; set; }  
public bool HasElevator { get; set; }

}

public class BuildingFactory  
{  
 public Building GetSchool()  
 {  
 return new Building  
 {  
 Name = "Local School",  
 Floors = 2,  
 HasElevator = false  
 };  
 }  
}

var factory = new BuildingFactory();  
var school= factory.GetSchool();

# Abstract Factory

Es definiert eine Schnittstelle zur Erzeugung einer Familie von Objekten, wobei die konkreten Klassen der zu instanziierende Objekte nicht näher festgelegt werden.

Ein Bild, das Text, Screenshot, Schrift enthält.

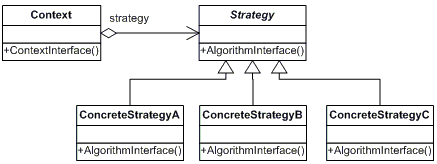
Automatisch generierte Beschreibung

public abstract class ABuildingFactory  
{  
 public abstract Building CreateSchool();  
}  
  
public class UrbanBuildingFactory : ABuildingFactory  
{  
 public override Building CreateSchool()  
 {  
 return new Building  
 {  
 Name = "Urban School",  
 Floors = 5,  
 HasElevator = true  
 };  
 }  
}

ABuildingFactory urbanFactory=new UrbanBuildingFactory();  
var urbanSchool=urbanFactory.CreateSchool();  
Console.WriteLine($"Urban School {urbanSchool}");

# Strategy

definiert eine Familie austauschbarer Algorithmen



* Stockenten fliegen und quaken laut.
* Holzenten können weder fliegen noch quaken.
* Gummienten quaken anders und fliegen nicht.

Es soll möglich sein, das Flug- und Quakverhalten der Enten dynamisch zu ändern, ohne den Code für jede Entenart zu duplizieren. Neue Entenarten und Verhaltensweisen sollen leicht hinzufügbar sein.

public interface IFlyBehavior  
{  
 void Fly();  
}  
public interface IQuackBehavior  
{  
 void Quack();  
}  
public class FlyWithWings : IFlyBehavior  
{  
 public void Fly()  
 {  
 Console.WriteLine("I'm flying with wings!");  
 }  
}  
public class FlyNoWay : IFlyBehavior  
{  
 public void Fly()  
 {  
 Console.WriteLine("I can't fly.");  
 }  
}  
public class Quack : IQuackBehavior  
{  
 public void Quack()  
 {  
 Console.WriteLine("Quack! Quack!");  
 }  
}  
public class MuteQuack : IQuackBehavior  
{  
 public void Quack()  
 {  
 Console.WriteLine("<< Silence >>");  
 }  
}  
public abstract class Duck  
{  
 protected IFlyBehavior flyBehavior;  
 protected IQuackBehavior quackBehavior;  
 protected Duck(IFlyBehavior flyBehavior, IQuackBehavior quackBehavior)  
 {  
 this.flyBehavior = flyBehavior;  
 this.quackBehavior = quackBehavior;  
 }  
 public void PerformFly()  
 {  
 flyBehavior.Fly();  
 }  
  
 public void PerformQuack()  
 {  
 quackBehavior.Quack();  
 }  
 public void Swim()  
 {  
 Console.WriteLine("All ducks float, even decoys!");  
 }  
  
 public abstract void Display();  
}  
*// Konkrete Entenklasse: Stockente*public class MallardDuck : Duck  
{  
 public MallardDuck()  
 {  
 flyBehavior = new FlyWithWings();  
 quackBehavior = new Quack();  
 }  
  
 public override void Display()  
 {  
 Console.WriteLine("I'm a real Mallard Duck.");  
 }  
}  
  
*// Konkrete Entenklasse: Holzente (kann nicht fliegen und quakt nicht)*public class DecoyDuck : Duck  
{  
 public DecoyDuck()  
 {  
 flyBehavior = new FlyNoWay();  
 quackBehavior = new MuteQuack();  
 }  
  
 public override void Display()  
 {  
 Console.WriteLine("I'm a wooden decoy duck.");  
 }  
}

# Command

Operationen in eine [Warteschlange](https://de.wikipedia.org/wiki/Warteschlange_(Datenstruktur)) zu stellen, [Logbucheinträge](https://de.wikipedia.org/wiki/Logdatei) zu führen und Operationen rückgängig zu machen. Objekte gekapselt

Ein Bild, das Text, Diagramm, Plan, Reihe enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

public interface ICommand  
{  
 void Execute();  
 void Undo();  
 }  
*// Konkreter Befehl: Licht einschalten* public class LightOnCommand : ICommand  
 {  
 private readonly Light \_light;  
  
 public LightOnCommand(Light light)  
 {  
 \_light = light;  
 }  
 public void Execute()  
 {  
 \_light.On();  
 }  
  
 public void Undo()  
 {  
 \_light.Off();  
 }  
 }  
*// Konkreter Befehl: Licht ausschalten* public class LightOffCommand : ICommand  
 {  
 private readonly Light \_light;  
  
 public LightOffCommand(Light light)  
 {  
 \_light = light;  
 }  
  
 public void Execute()  
 {  
 \_light.Off();  
 }  
  
 public void Undo()  
 {  
 \_light.On();  
 }  
 }  
*// Der Empfänger: Die Lichtklasse* public class Light  
 {  
 public void On()  
 {  
 Console.WriteLine("The light is on.");  
 }  
  
 public void Off()  
 {  
 Console.WriteLine("The light is off.");  
 }  
 }  
*// Der Invoker: Die Fernbedienung* public class RemoteControl  
 {  
 private ICommand \_command;  
  
 public void SetCommand(ICommand command)  
 {  
 \_command = command;  
 }  
  
 public void PressButton()  
 {  
 \_command.Execute();  
 }  
  
 public void PressUndo()  
 {  
 \_command.Undo();  
 }  
 }

# State

Kapselung unterschiedlicher, zustandsabhängiger Verhaltensweisen eines Objektes

Ein Bild, das Text, Diagramm, Reihe, Schrift enthält.

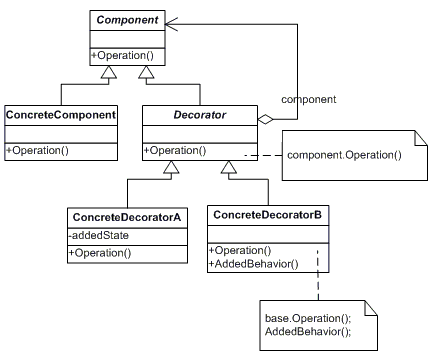
Automatisch generierte Beschreibung

*// Interface für die Zustände*public interface ICarState  
{  
 void StartDriving();  
 void StopDriving();  
}  
*// Zustand: Parken*public class ParkedState : ICarState  
{  
 private Car \_car;

public ParkedState(Car car)  
 {  
 \_car = car;  
 }  
 public void StartDriving()  
 {  
 Console.WriteLine("Car starts driving.");  
 \_car.SetState(\_car.DrivingState); *// Wechsel zum Fahrzustand* }  
 public void StopDriving()  
 {  
 Console.WriteLine("Car is already parked.");  
 }  
}  
*// Zustand: Fahren*public class DrivingState : ICarState  
{  
 private Car \_car;

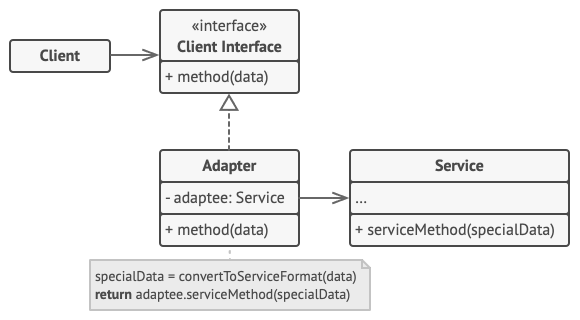
public DrivingState(Car car)  
 {  
 \_car = car;  
 }  
 public void StartDriving()  
 {  
 Console.WriteLine("Car is already driving.");  
 }  
 public void StopDriving()  
 {  
 Console.WriteLine("Car stops and is now parked.");  
 \_car.SetState(\_car.ParkedState); *// Wechsel zum Parkzustand* }  
}public class Car  
{  
 public ICarState ParkedState { get; }  
 public ICarState DrivingState { get; }  
  
 private ICarState \_currentState;  
  
 public Car()  
 {  
 ParkedState = new ParkedState(this);  
 DrivingState = new DrivingState(this);  
  
 \_currentState = ParkedState; *// Initialer Zustand: Parken* }  
  
 public void SetState(ICarState state)  
 {  
 \_currentState = state;  
 }  
  
 public void StartDriving()  
 {  
 \_currentState.StartDriving();  
 }  
  
 public void StopDriving()  
 {  
 \_currentState.StopDriving();  
 }

# Decorator

Alternative zur Unterklassenbildung, um eine Klasse um zusätzliche Funktionen zu "dekorieren"

public interface ICar  
{  
 string GetDescription();  
 double GetCost();  
}  
  
*// Konkretes Auto (Basisfunktionalität)*public class BasicCar : ICar  
{  
 public string GetDescription()  
 {  
 return "Basic Car";  
 }  
  
 public double GetCost()  
 {  
 return 10000; *// Grundpreis des Autos* }  
}  
  
*// Abstrakter Dekorator*public abstract class CarDecorator : ICar  
{  
 protected ICar \_car;  
  
 public CarDecorator(ICar car)  
 {  
 \_car = car;  
 }  
  
 public virtual string GetDescription()  
 {  
 return \_car.GetDescription();  
 }  
  
 public virtual double GetCost()  
 {  
 return \_car.GetCost();  
 }  
}  
  
*// Dekorator: Klimaanlage hinzufügen*public class AirConditioning : CarDecorator  
{  
 public AirConditioning(ICar car) : base(car) { }  
  
 public override string GetDescription()  
 {  
 return \_car.GetDescription() + ", Air Conditioning";  
 }  
  
 public override double GetCost()  
 {  
 return \_car.GetCost() + 1500; *// Zusatzkosten für Klimaanlage* }  
}  
  
*// Dekorator: Musikanlage hinzufügen*public class SoundSystem : CarDecorator  
{  
 public SoundSystem(ICar car) : base(car) { }  
  
 public override string GetDescription()  
 {  
 return \_car.GetDescription() + ", Sound System";  
 }  
  
 public override double GetCost()  
 {  
 return \_car.GetCost() + 800; *// Zusatzkosten für das Soundsystem* }  
}

# Adapter



public interface IVehicle  
{  
 void Drive();  
}  
  
*// Klasse: Auto, implementiert die IVehicle-Schnittstelle*public class Car : IVehicle  
{  
 public void Drive()  
 {  
 Console.WriteLine("Driving a car.");  
 }  
}  
  
*// Klasse: Elektrofahrrad (unterschiedliche Schnittstelle, kein IVehicle)*public class ElectricBike: IElectric  
{  
 public void Pedal()  
 {  
 Console.WriteLine("Pedaling the electric bike.");  
 }  
}  
  
*// Der Adapter, der das Elektrofahrrad an die IVehicle-Schnittstelle anpasst*public class ElectricBikeAdapter : IVehicle  
{  
 private ElectricBike \_electricBike;  
  
 public ElectricBikeAdapter(ElectricBike electricBike)  
 {  
 \_electricBike = electricBike;  
 }  
  
 public void Drive() => \_electricBike.Pedal();  
}

# Extension Method

Klasse oder Schnittstelle erweitert, ohne Teil der Implementierung der erweiterten Klasse zu sein

public static class CarExtensions  
{  
 *// Erweiterungsmethode für ICar zur Berechnung der jährlichen Inspektionskosten* public static double AnnualInspection(this ICar car)  
 {  
 const double **inspectionCost** = 500;  
 Console.WriteLine($"Adding annual inspection cost of {**inspectionCost**}");  
 return car.GetCost() + **inspectionCost**;  
 }  
}

Observer Pattern:

Ein Bild, das Text, Screenshot, Reihe, Diagramm enthält.

Automatisch generierte Beschreibung

public interface IObserver  
{  
 void Update(string message);  
}  
  
public interface IPublisher  
{  
 void Subscribe(IObserver observer);  
 void Unsubscribe(IObserver observer);  
 void Notify(string message);  
}  
  
public class ConcretePublisher: IPublisher  
{  
 private readonly List<IObserver> \_observers = new List<IObserver>();  
  
 public void Subscribe(IObserver observer)  
 {  
 \_observers.Add(observer);  
 Console.WriteLine("Subscribed");  
 }  
  
 public void Unsubscribe(IObserver observer)  
 {  
 \_observers.Remove(observer);  
 Console.WriteLine("Unsubscribed");  
 }  
  
 public void Notify(string message)  
 {  
 foreach (var observer in \_observers)  
 {  
 Console.WriteLine($"Notifying {observer}");  
 }  
 }  
}  
  
public class Observer : IObserver  
{  
 private readonly string \_name;  
  
 public Observer(string name)  
 {  
 \_name = name;  
 }  
   
 public void Update(string message)  
 {  
 Console.WriteLine($"{\_name} received: {message}");  
 }  
}

Command-Pattern:

public interface Command  
{  
 void Execute();  
}  
  
public class Fan  
{  
 public void Start()  
 {  
 Console.WriteLine("Fan started");  
 }  
  
 public void Stop()  
 {  
 Console.WriteLine("Fan stopped");  
 }  
}  
  
public class Light  
{  
 public void TurnOn()  
 {  
 Console.WriteLine("Light turned on");  
 }  
  
 public void TurnOff()  
 {  
 Console.WriteLine("Light turned off");  
 }  
}  
  
public class LightOnCommand : Command  
{  
 private Light \_light;  
  
 public LightOnCommand(Light light)  
 {  
 \_light = light;  
 }  
  
 public void Execute()  
 {  
 \_light.TurnOn();  
 }  
}  
  
public class LightOffCommand : Command  
{  
 private Light \_light;  
  
 public LightOffCommand(Light light)  
 {  
 \_light = light;  
 }  
  
 public void Execute()  
 {  
 \_light.TurnOff();  
 }  
}  
  
public class FanOnCommand : Command  
{  
 private Fan \_fan;  
  
 public FanOnCommand(Fan fan) => \_fan = fan;  
  
 public void Execute()  
 {  
 \_fan.Start();  
 }  
}  
  
public class FanOffCommand : Command  
{  
 private Fan \_fan;  
  
 public FanOffCommand(Fan fan) => \_fan = fan;  
  
 public void Execute()  
 {  
 \_fan.Stop();  
 }  
}