Design Patterns

# Декоратор (Decorator)

1. Теория – декоратор е структорен шаблон за разработка. Използва се когато е необходимо да се добави допълнителна функционалност към отделен обек(инстанция на клас), независимо дали статично или динамично, без това да влияе на останалите инстанции от същият клас. Това се постига чрез създаването на нов декоратор клас, който обвива оригиналният клас. Този шаблон е проектиран по такъв начин, че множество декоратои могат да се наслоят един върху друг, като всеки път се добавя нова фунцкионалност. Този шаблон е алтернатива на subclassing, който добавя фунцкионалност на всички инстанции на оригиналния клас, докато декоратора, добавя функционалност само на индивидуални обекти. Входно/Изходните потоци в .NET Framework използват този шаблон.

## C:\Users\Aleks\Desktop\400px-Decorator_UML_class_diagram.svg.pngUML диаграма/Блок схема

## Примерен C# код/Имплементация:

using System;

using System.Collections.Generic;

namespace DoFactory.GangOfFour.Decorator.RealWorld

{

  /// <summary>

  /// MainApp startup class for Real-World

  /// Decorator Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      // Create book

      Book book = new Book("Worley", "Inside ASP.NET", 10);

      book.Display();

      // Create video

      Video video = new Video("Spielberg", "Jaws", 23, 92);

      video.Display();

      // Make video borrowable, then borrow and display

      Console.WriteLine("\nMaking video borrowable:");

      Borrowable borrowvideo = new Borrowable(video);

      borrowvideo.BorrowItem("Customer #1");

      borrowvideo.BorrowItem("Customer #2");

      borrowvideo.Display();

      // Wait for user

      Console.ReadKey();

    }

  }

  /// <summary>

  /// The 'Component' abstract class

  /// </summary>

  abstract class LibraryItem

  {

    private int \_numCopies;

    // Property

    public int NumCopies

    {

      get { return \_numCopies; }

      set { \_numCopies = value; }

    }

    public abstract void Display();

  }

  /// <summary>

  /// The 'ConcreteComponent' class

  /// </summary>

  class Book : LibraryItem

  {

    private string \_author;

    private string \_title;

    // Constructor

    public Book(string author, string title, int numCopies)

    {

      this.\_author = author;

      this.\_title = title;

      this.NumCopies = numCopies;

    }

    public override void Display()

    {

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

      Console.WriteLine(" Author: {0}", \_author);

      Console.WriteLine(" Title: {0}", \_title);

      Console.WriteLine(" # Copies: {0}", NumCopies);

    }

  }

  /// <summary>

  /// The 'ConcreteComponent' class

  /// </summary>

  class Video : LibraryItem

  {

    private string \_director;

    private string \_title;

    private int \_playTime;

    // Constructor

    public Video(string director, string title,

      int numCopies, int playTime)

    {

      this.\_director = director;

      this.\_title = title;

      this.NumCopies = numCopies;

      this.\_playTime = playTime;

    }

    public override void Display()

    {

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

      Console.WriteLine(" Director: {0}", \_director);

      Console.WriteLine(" Title: {0}", \_title);

      Console.WriteLine(" # Copies: {0}", NumCopies);

      Console.WriteLine(" Playtime: {0}\n", \_playTime);

    }

  }

  /// <summary>

  /// The 'Decorator' abstract class

  /// </summary>

  abstract class Decorator : LibraryItem

  {

    protected LibraryItem libraryItem;

    // Constructor

    public Decorator(LibraryItem libraryItem)

    {

      this.libraryItem = libraryItem;

    }

    public override void Display()

    {

      libraryItem.Display();

    }

  }

  /// <summary>

  /// The 'ConcreteDecorator' class

  /// </summary>

  class Borrowable : Decorator

  {

    protected List<string> borrowers = new List<string>();

    // Constructor

    public Borrowable(LibraryItem libraryItem)

      : base(libraryItem)

    {

    }

    public void BorrowItem(string name)

    {

      borrowers.Add(name);

      libraryItem.NumCopies--;

    }

    public void ReturnItem(string name)

    {

      borrowers.Remove(name);

      libraryItem.NumCopies++;

    }

    public override void Display()

    {

      base.Display();

      foreach (string borrower in borrowers)

      {

        Console.WriteLine(" borrower: " + borrower);

      }

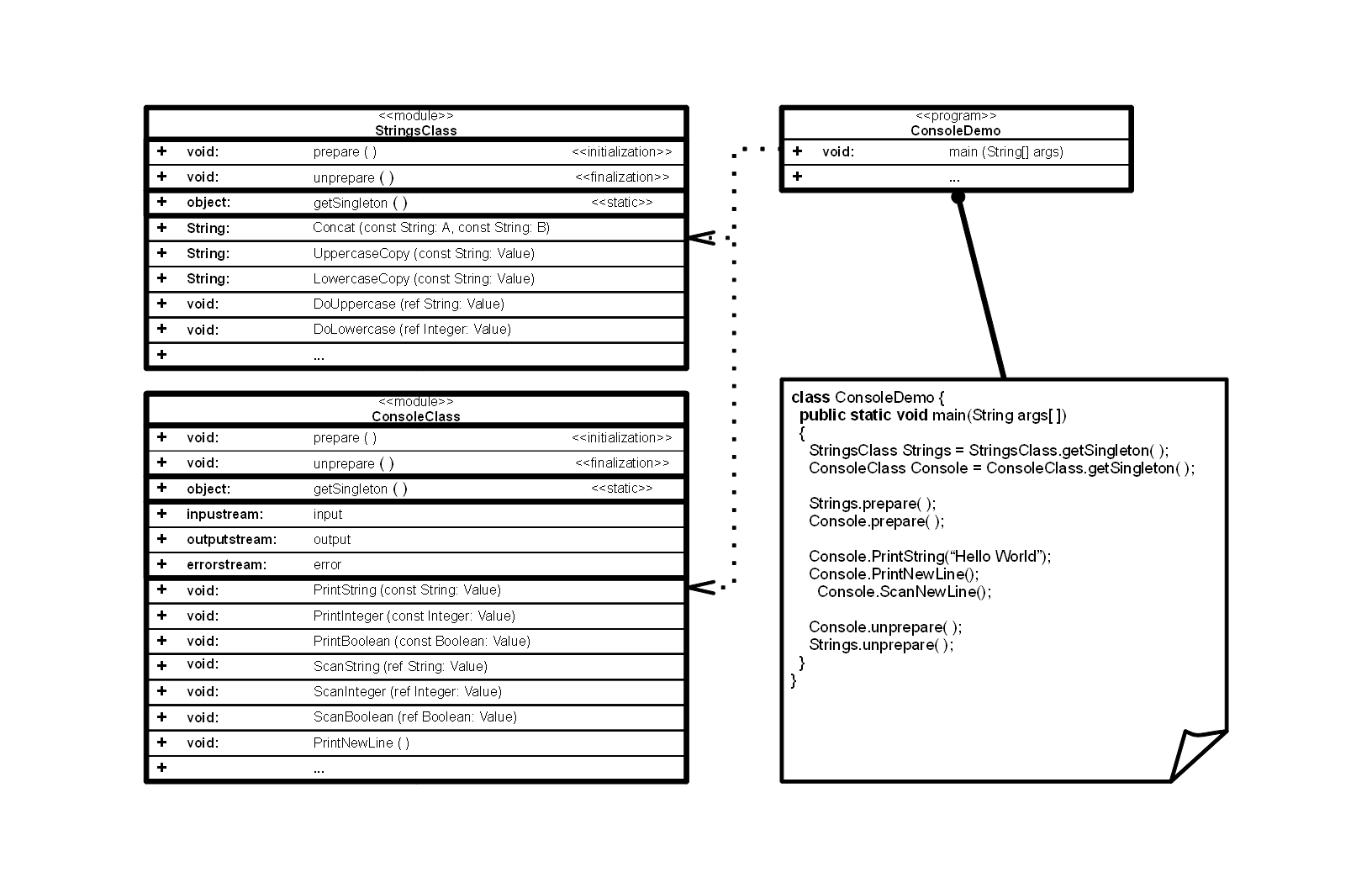
    }

  }

}

# Модул(Module pattern)

1. Теория – Модул е структорен шаблон за разработка. Използва се когато е необходима имплементация на идеята за софтуерните модули, в програмни езици, които не ги потдържат или ги потдържат само частично. Има няколко начина за имплементиране на този шаблон, които се изпозлват в зависимост от езика за програмиране: чрез Singleton шаблона, обектно ориентирани статични членове на клас и процедурни глобални функций. При разработката на софтуеър има няколко начина за организирането на кода в обособени компоненти, които изпълняват конкретни действия. Модулното програмиране е един от тези начини и е основно изпозлван при процедурното програмиране.

Изпълнение – част от кода трябва да бъде с публичен достъп, като той разкрива фунцкионалаността, която може да се ползва. Вътрешната част(която е private), представлява логиката, която се изпълнява при извикване на някой от публичните методи. Всичките входящи данни, се предоставят като параметри при извикването на публичните методи.

## UML диаграма/Блок схема:

## Примерен C# код/Имплементация:

using System;

using System.IO;

using System.Text;

namespace Consoles {

public InputStream input = null;

public OutputStream output = null;

public ErrorStream error = null;

public sealed class MainModule {

private static Singleton = null;

// ...

public MainModule () {

// does nothing on purpose !!!

}

// ...

public MainModule getSingleton() {

if (MainModule.Singleton == null)

{

MainModule.Singleton = new MainModule();

}

return MainModule.Singleton;

}

// ...

public void prepare() {

//System.WriteLine("console::prepare();");

this.input = new InputStream();

this.output = new OutputStream();

this.error = new ErrorStream();

}

public void unprepare() {

this.output = null;

this.input = null;

this.error = null;

//System.WriteLine("console::unprepare();");

}

// ...

public void printNewLine() {

System.Console.WriteLine("");

}

public void printString(String Value) {

System.Console.Write(Value);

}

public void printInteger(Integer Value) {

System.Console.Write(Value);

}

public void printBoolean(Boolean Value) {

System.Console.Write(Value);

}

public void ScanNewLine() {

// to-do: ...

}

public void ScanString(String Value) {

// to-do: ...

}

public void ScanInteger(Integer Value) {

// to-do: ...

}

public void ScanBoolean(Boolean Value) {

// to-do: ...

}

// ...

}

}

class ConsoleDemo {

public static Consoles.MainModule Console = null;

public static void prepare()

{

Console = Consoles.MainModule.getSingleton();

Console.prepare();

}

public static void unprepare()

{

Console.unprepare();

}

public static void execute()

{

Console.PrintString("Hello World");

Console.PrintNewLine();

Console.ScanNewLine();

}

public static void main()

{

prepare();

execute(args);

unprepare();

}

}

# Прототип (Prototype)

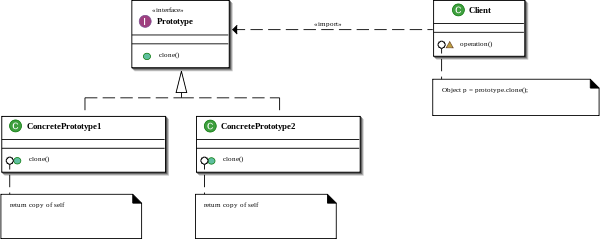
1. Теория – Прототип шаблонът се използва при разработката на софтуеър когато типът на обекта, който трябва да бъде създаден се определя от прототипна инстанция, която е клонирана за да се получи нов обект. Този шаблон се използва, когато:

* Трябва да се избегнат подкласове(subclasses), както прави Abstract Factory шаблон.
* Трябва да се избегне цената за създаване на нов обект по стандартният начин(използвайки new), когато е изключително скъпо(откъм ресурси) за самата програма.

За да се декларира този шаблон трябва да се декларира абстрактен базов клас, който има виртуален метод clone(). Всеки клас който има нужда от поломорфен конструктор онаследява този абстрактен базов клас и имплементира този clone() метод.

Вместо да се създава нова инстанция с new, се вика този clone() метод, и посредством някой друг шаблон (например Factory) , който чрез подаване на параметри създава копие на оригиналният обект.

## UML диаграма/Блок схема



## Примерен C# код/Имплементация:

//Note: In this example ICloneable interface (defined in .Net Framework) acts as Prototype

class ConcretePrototype : ICloneable

{

public int X { get; set; }

public ConcretePrototype(int x)

{

this.X = x;

}

public void PrintX()

{

Console.WriteLine("Value :" + X);

}

public object Clone()

{

return this.MemberwiseClone();

}

}

/\*\*

\* Client code

\*/

public class PrototypeTest

{

public static void Main()

{

var prototype = new ConcretePrototype(1000);

for (int i = 1; i < 10; i++)

{

ConcretePrototype tempotype = prototype.Clone() as ConcretePrototype;

// Usage of values in prototype to derive a new value.

tempotype.X \*= i;

tempotype.PrintX();

}

Console.ReadKey();

}

}

/\*

\*\*Code output\*\*

Value :1000

Value :2000

Value :3000

Value :4000

Value :5000

Value :6000

Value :7000

Value :8000

Value :9000

\*/