# Домашно Design Patterns

Ще направя кратко описание на по един desing pattern(модел) от трите основни вида модели за дизайн.

Трите основни вида модели за дизайн са Creational, Structural и Behavioral. Всеки един от тях има отделни характеристики, поведение и начин на имплементиране в кода.

1.Creational desing patterns

За този вид модели съм избрал като пример за описание Singleton design pattern. Този модел е много често изпозлван при писането на код в големи проекти. Неговата отговорност е да гарантира, че даден клас има само една инстанция и да осигури глобален достъп до него.

Примерен код:

using System;

public class Singleton

{

private static Singleton instance;

private Singleton() {}

public static Singleton Instance

{

get

{

if (instance == null)

{

instance = new Singleton();

}

return instance;

}

}

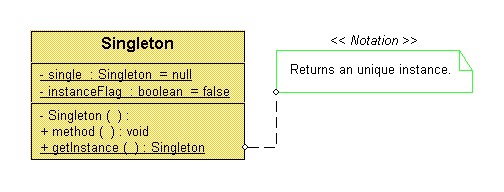
}

Този код има две основни предимства – тъй като инстанцията се създава вътре в метода, класът може да има допълнителна функционалност ( например да се инстанцира подклас, въпреки че това може да доведе до нежелани зависимости между тях. Друго предимство е, че инстанцията не се прави докато даден друг обект не я създаде – този подход е е познат като Lazy инстанциране. По този начин се избягват ненужни действия когато дадено прилижение се стартира.

Въпреки че Singleton е сравнително прост модел, има различни варианти и възможности за неговото изпълнение

Основният надостатък на Singleton е че не е подходящо за изпълнение в многонишкова среда. Ако отделните нишки на изпълнение се изпълнпват едновременно, може да се създаде повече от една инстанция, което да доведе до нарушаване на модела.

UML диаграма:



2.Structural desing patterns

За тези модели съм избрал като пример за описание Bridge design pattern. Основната цел на този модел е да разграничи една абстракция от нейното директно имплементиране, като целта е двете да бъдат абсолютно независими. Bridge използва енкапсулация и наследяване за да отделните отговорнисти в различни класове.

Когато даден клас се променя често, характеристиките на обектно-ориентираното програмиране стават много полезни, тъй като промените в кода на дадена програма могат да се направят лесно с минимални предварителни познания за самата програма. Bridge моделът може да се разглежда като като двуслойна абстракция.

Тъй като теоретично е доста сложно за обяснение ще дам пример. Когато имаме код от следния вид:

----Shape---

/ \

Rectangle Circle

/ \ / \

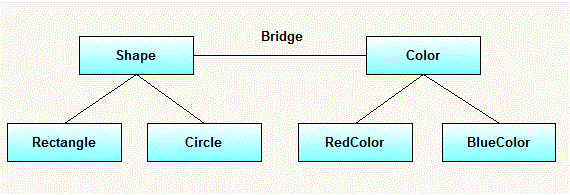
BlueRectangle RedRectangle BlueCircle RedCircle

След рефактори кодът става така:

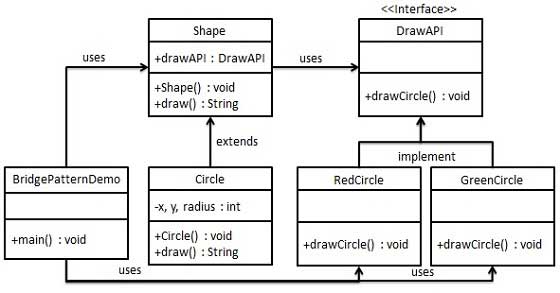
----Shape--- Color

/ \ / \

Rectangle(Color) Circle(Color) Blue Red



UML диаграма:



Примерен код:

using System;

namespace DoFactory.GangOfFour.Bridge.Structural

{

  /// <summary>

  /// MainApp startup class for Structural

  /// Bridge Design Pattern.

  /// </summary>

  class MainApp

  {

    /// <summary>

    /// Entry point into console application.

    /// </summary>

    static void Main()

    {

      Abstraction ab = new RefinedAbstraction();

      // Set implementation and call

      ab.Implementor = new ConcreteImplementorA();

      ab.Operation();

      // Change implemention and call

      ab.Implementor = new ConcreteImplementorB();

      ab.Operation();

      // Wait for user

      Console.ReadKey();

    }

  }

  /// <summary>

  /// The 'Abstraction' class

  /// </summary>

  class Abstraction

  {

    protected Implementor implementor;

    // Property

    public Implementor Implementor

    {

      set { implementor = value; }

    }

    public virtual void Operation()

    {

      implementor.Operation();

    }

  }

  /// <summary>

  /// The 'Implementor' abstract class

  /// </summary>

  abstract class Implementor

  {

    public abstract void Operation();

  }

  /// <summary>

  /// The 'RefinedAbstraction' class

  /// </summary>

  class RefinedAbstraction : Abstraction

  {

    public override void Operation()

    {

      implementor.Operation();

    }

  /// <summary>

  /// The 'ConcreteImplementorA' class

  /// </summary>

  class ConcreteImplementorA : Implementor

  {

    public override void Operation()

    {

      Console.WriteLine("ConcreteImplementorA Operation");

    }

  }

  /// <summary>

  /// The 'ConcreteImplementorB' class

  /// </summary>

  class ConcreteImplementorB : Implementor

  {

    public override void Operation()

    {

      Console.WriteLine("ConcreteImplementorB Operation");

    }

  }

}

3.Behavioral desing patterns

Моделът, който ще разгледам и опиша накратко тук е Memento design pattern. Основното свойство на този модел е без да се нарушава енкапсулацията да може да се разшири вътрешното състояние на даден обекта, по такъв начин, че обектът да може да се възстанови до същото състояние на по-късен етап.

Memento съхранява вътрешното състояние на обекта, който е първоизточник. Този модел може да съхранява толкова информация от състоянието на родителския клас, колкото е необходимо по преценка на създадения клас. Memento осигурява защита срещу достъп до обекти, различни от тези на автора.

Memento има два интерфейса – Caretaker и Originator. Caretaker е „тесен“ интерфейс - може само да предава Memento към други обекти. Originator е негова противоположност. Той осигурява „широк“ интерфейс, който осигурява достъп до всички данни, необходими за възстановяването на предишно състояние. В идеалния случай, само родителският клас може да осигурява връзка към вътрешното състояние на Memento.

Примерен код:

|  |
| --- |
| // Memento pattern -- Structural example |
| using System;    namespace DoFactory.GangOfFour.Memento.Structural  {    /// <summary>    /// MainApp startup class for Structural    /// Memento Design Pattern.    /// </summary>    class MainApp    {      /// <summary>      /// Entry point into console application.      /// </summary>      static void Main()      {        Originator o = new Originator();        o.State = "On";        // Store internal state        Caretaker c = new Caretaker();        c.Memento = o.CreateMemento();        // Continue changing originator        o.State = "Off";        // Restore saved state        o.SetMemento(c.Memento);        // Wait for user        Console.ReadKey();      }    }    /// <summary>    /// The 'Originator' class    /// </summary>    class Originator    {      private string \_state;      // Property      public string State      {        get { return \_state; }        set        {          \_state = value;          Console.WriteLine("State = " + \_state);        }      }      // Creates memento      public Memento CreateMemento()      {        return (new Memento(\_state));      }      // Restores original state      public void SetMemento(Memento memento)      {        Console.WriteLine("Restoring state...");        State = memento.State;      }    }    /// <summary>    /// The 'Memento' class    /// </summary>    class Memento    {      private string \_state;      // Constructor      public Memento(string state)      {        this.\_state = state;      }      // Gets or sets state      public string State      {        get { return \_state; }      }    }    /// <summary>    /// The 'Caretaker' class    /// </summary>    class Caretaker    {      private Memento \_memento;      // Gets or sets memento      public Memento Memento      {        set { \_memento = value; }        get { return \_memento; }      }    }  } |

UML диаграма:

