3. Personal online unified (phone, social and other) contact management system Basic required functional: 1. List of contacts 2. Contact CRUD Operations 3. Contact search functional

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон? Когда необходимо, чтобы для класса существовал только один экземпляр

Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

Классическая реализация данного шаблона проектирования на C# выглядит следующим образом:

class Singleton

{

    private static Singleton instance;

    private Singleton()

    {}

    public static Singleton getInstance()

    {

        if (instance == null)

            instance = new Singleton();

        return instance;

    }

}

В классе определяется статическая переменная - ссылка на конкретный экземпляр данного объекта и приватный конструктор. В статическом методе getInstance() этот конструктор вызывается для создания объекта, если, конечно, объект отсутствует и равен null.

Для применения паттерна Одиночка создадим небольшую программу. Например, на каждом компьютере можно одномоментно запустить только одну операционную систему. В этом плане операционная система будет реализоваться через паттерн синглтон:

class Program

{

    static void Main(string[] args)

    {

        Computer comp = new Computer();

        comp.Launch("Windows 8.1");

        Console.WriteLine(comp.OS.Name);

        // у нас не получится изменить ОС, так как объект уже создан

        comp.OS = OS.getInstance("Windows 10");

        Console.WriteLine(comp.OS.Name);

        Console.ReadLine();

    }

}

class Computer

{

    public OS OS { get; set; }

    public void Launch(string osName)

    {

        OS = OS.getInstance(osName);

    }

}

class OS

{

    private static OS instance;

    public string Name { get; private set; }

    protected OS(string name)

    {

        this.Name=name;

    }

    public static OS getInstance(string name)

    {

        if (instance == null)

            instance = new OS(name);

        return instance;

    }

}

**Синглтон и многопоточность**

При применении паттерна синглтон в многопоточным программах мы можем столкнуться с проблемой, которую можно описать следующим образом:

static void Main(string[] args)

{

    (new Thread(() =>

    {

        Computer comp2 = new Computer();

        comp2.OS = OS.getInstance("Windows 10");

        Console.WriteLine(comp2.OS.Name);

    })).Start();

    Computer comp = new Computer();

    comp.Launch("Windows 8.1");

    Console.WriteLine(comp.OS.Name);

    Console.ReadLine();

}

Здесь запускается дополнительный поток, который получает доступ к синглтону. Параллельно выполняется тот код, который идет запуска потока и кторый также обращается к синглтону. Таким образом, и главный, и дополнительный поток пытаются инициализровать синглтон нужным значением - "Windows 10", либо "Windows 8.1". Какое значение сиглтон получит в итоге, пресказать в данном случае невозможно.

Вывод программы может быть такой:

Windows 8.1

Windows 10

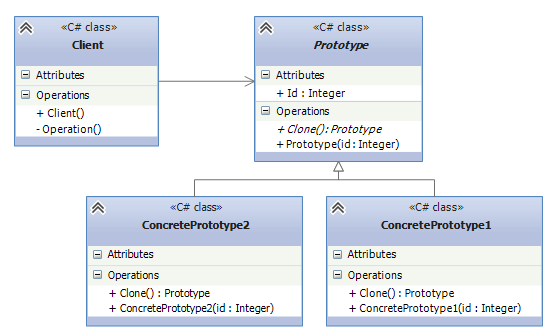
4. University HR recruitment system (roles to implement: HR, candidate) Basic required functional: 1. Roles based access to HR or candidate view 2. Simple form to create a self-info by the candidate with CRUD operations. 3. HR list of all applied candidates.

Паттерн Прототип (Prototype) позволяет создавать объекты на основе уже ранее созданных объектов-прототипов. То есть по сути данный паттерн предлагает технику клонирования объектов.

Когда использовать Прототип?

* Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения
* Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов из параллельной иерархии классов (как это делается, например, при использовании паттерна Абстрактная фабрика)
* Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.

На языке UML отношения между классами при применении данного паттерна можно описать следующим образом:



Формальная структура паттерна на C# могла бы выглядеть следующим образом:

class Client

{

    void Operation()

    {

        Prototype prototype = new ConcretePrototype1(1);

        Prototype clone = prototype.Clone();

        prototype = new ConcretePrototype2(2);

        clone = prototype.Clone();

    }

}

abstract class Prototype

{

    public int Id { get; private set; }

    public Prototype(int id)

    {

        this.Id = id;

    }

    public abstract Prototype Clone();

}

class ConcretePrototype1 : Prototype

{

    public ConcretePrototype1(int id)

        : base(id)

    { }

    public override Prototype Clone()

    {

        return new ConcretePrototype1(Id);

    }

}

class ConcretePrototype2 : Prototype

{

    public ConcretePrototype2(int id)

        : base(id)

    { }

    public override Prototype Clone()

    {

        return new ConcretePrototype2(Id);

    }

}

### Участники

* **Prototype**: определяет интерфейс для клонирования самого себя, который, как правило, представляет метод Clone()
* **ConcretePrototype1** и **ConcretePrototype2**: конкретные реализации прототипа. Реализуют метод Clone()
* **Client**: создает объекты прототипов с помощью метода Clone()

Рассмотрим клонирование на примере фигур - прямоугольников и кругов:

class Program

{

    static void Main(string[] args)

    {

        IFigure figure = new Rectangle(30,40);

        IFigure clonedFigure = figure.Clone();

        figure.GetInfo();

        clonedFigure.GetInfo();

        figure = new Circle(30);

        clonedFigure=figure.Clone();

        figure.GetInfo();

        clonedFigure.GetInfo();

        Console.Read();

    }

}

interface IFigure

{

    IFigure Clone();

    void GetInfo();

}

class Rectangle: IFigure

{

    int width;

    int height;

    public Rectangle(int w, int h)

    {

        width = w;

        height = h;

    }

    public IFigure Clone()

    {

        return new Rectangle(this.width, this.height);

    }

    public void GetInfo()

    {

        Console.WriteLine("Прямоугольник длиной {0} и шириной {1}", height, width);

    }

}

class Circle : IFigure

{

    int radius;

    public Circle(int r)

    {

        radius = r;

    }

    public IFigure Clone()

    {

        return new Circle(this.radius);

    }

    public void GetInfo()

    {

        Console.WriteLine("Круг радиусом {0}", radius);

    }

}

Здесь в качестве прототипа используется интерфейс IFigure, который реализуется классами Circle и Rectangle.

Но в данном случае надо заметить, что фреймворк .NET предлагает функционал для копирования в виде метода **MemberwiseClone()**. Например, мы могли бы изменить реализацию метода Clone() в классах прямоугольника и круга следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4 | public IFigure Clone()  {      return this.MemberwiseClone() as IFigure;  } |

Причем данный метод был бы общим для обоих классов. И работа программы никак бы не изменилась.

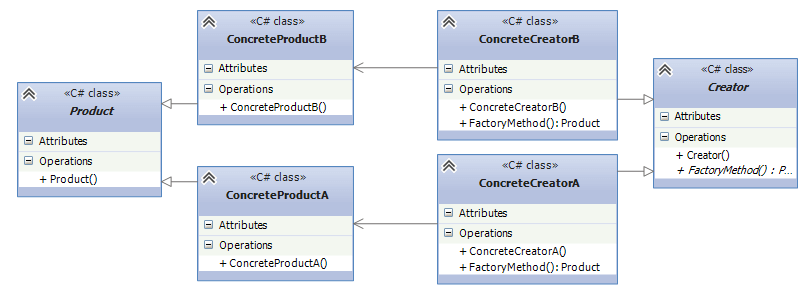
5. Street Banner management system (map integration is required to display banners' location) Basic required functional: 1. function to handle click on the point and get coordinates pass to backend service 2. Banner’s CRUD operations 3. Search by Banner Coordinates.

Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

### Когда надо применять паттерн

* Когда заранее неизвестно, объекты каких типов необходимо создавать
* Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
* Когда создание новых объектов необходимо делегировать из базового класса классам наследникам

На языке UML паттерн можно описать следующим образом:



Формальное определение паттерна на языке C# может выглядеть следующим образом:

abstract class Product

{}

class ConcreteProductA : Product

{}

class ConcreteProductB : Product

{}

abstract class Creator

{

    public abstract Product FactoryMethod();

}

class ConcreteCreatorA : Creator

{

    public override Product FactoryMethod() { return new ConcreteProductA(); }

}

class ConcreteCreatorB : Creator

{

    public override Product FactoryMethod() { return new ConcreteProductB(); }

}

### Участники

* Абстрактный класс **Product** определяет интерфейс класса, объекты которого надо создавать.
* Конкретные классы **ConcreteProductA** и **ConcreteProductB** представляют реализацию класса Product. Таких классов может быть множество
* Абстрактный класс **Creator** определяет абстрактный фабричный метод FactoryMethod(), который возвращает объект Product.
* Конкретные классы **ConcreteCreatorA** и **ConcreteCreatorB** - наследники класса Creator, определяющие свою реализацию метода FactoryMethod(). Причем метод FactoryMethod() каждого отдельного класса-создателя возвращает определенный конкретный тип продукта. Для каждого конкретного класса продукта определяется свой конкретный класс создателя.

Таким образом, класс Creator делегирует создание объекта Product своим наследникам. А классы ConcreteCreatorA и ConcreteCreatorB могут самостоятельно выбирать какой конкретный тип продукта им создавать.

Теперь рассмотрим на реальном примере. Допустим, мы создаем программу для сферы строительства. Возможно, вначале мы захотим построить многоэтажный панельный дом. И для этого выбирается соответствующий подрядчик, который возводит каменные дома. Затем нам захочется построить деревянный дом и для этого также надо будет выбрать нужного подрядчика:

class Program

{

    static void Main(string[] args)

    {

        Developer dev = new PanelDeveloper("ООО КирпичСтрой");

        House house2 = dev.Create();

        dev = new WoodDeveloper("Частный застройщик");

        House house = dev.Create();

        Console.ReadLine();

    }

}

// абстрактный класс строительной компании

abstract class Developer

{

    public string Name { get; set; }

    public Developer (string n)

    {

        Name = n;

    }

    // фабричный метод

    abstract public House Create();

}

// строит панельные дома

class PanelDeveloper : Developer

{

    public PanelDeveloper(string n) : base(n)

    { }

    public override House Create()

    {

        return new PanelHouse();

    }

}

// строит деревянные дома

class WoodDeveloper : Developer

{

    public WoodDeveloper(string n) : base(n)

    { }

    public override House Create()

    {

        return new WoodHouse();

    }

}

abstract class House

{ }

class PanelHouse : House

{

    public PanelHouse()

    {

        Console.WriteLine("Панельный дом построен");

    }

}

class WoodHouse : House

{

    public WoodHouse()

    {

        Console.WriteLine("Деревянный дом построен");

    }

}

В качестве абстрактного класса Product здесь выступает класс House. Его две конкретные реализации - PanelHouse и WoodHouse представляют типы домов, которые будут строить подрядчики. В качестве абстрактного класса создателя выступает Developer, определяющий абстрактный метод Create(). Этот метод реализуется в классах-наследниках WoodDeveloper и PanelDeveloper. И если в будущем нам потребуется построить дома какого-то другого типа, например, кирпичные, то мы можем с легкостью создать новый класс кирпичных домов, унаследованный от House, и определить класс соответствующего подрядчика. Таким образом, система получится легко расширяемой. Правда, недостатки паттерна тоже очевидны - для каждого нового продукта необходимо создавать свой класс создателя.