В программировании нам часто приходится использовать что-то, что уже было создано другими людьми, что столкнулись с похожей проблемой. Например, если говорить о C#, то можно вспомнить менеджер пакетов NuGet. Но речь сегодня пойдёт о другой вещи, которая крайне широко используется на сегодняшний день при проектировании программ разного назначения, написанных на разных языках программирования – о паттернах проектирования, а точнее, о конкретном паттерне, называемом одиночкой (или же Singleton’ом).

Кратко пробежимся по определению паттерна — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это абстрактное описание решения, реализация которого может различаться от проекта к проекту.

Как мы уже знаем, паттерны делятся по предназначению на 3 типа:

* Порождающие – создают объект
* Структурные – меняют структуру взаимодействия между классами
* Поведенческие – определяют поведение, способ взаимодействия между классами

Рассмотрим поподробнее один из порождающих паттернов, называемый, как я уже говорил, Singleton’ом.

Singleton - порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Приведу аналогию из жизни: правительство государства. В государстве может быть только одно официальное правительство. Вне зависимости от того, кто конкретно заседает в правительстве, оно имеет глобальную точку доступа «Правительство страны N».

А для чего нам может понадобиться Singleton? Есть 2 главные причины:

* Если нам нужен 1 и только 1 экземпляр класса (например, какой-то общий ресурс вроде базы данных, которую мы создадим только в 1 экземпляре и больше не будем создавать, а только обращаться к уже существующему для изменения/получения данных)
* К этому объекту нужно иметь глобальную точку доступа

В таком случае, как же нам создать подобный единственный и неповторимый объект?

Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

Структура одиночки выглядит следующим образом: (показал UML-диаграмму одиночки)

Ну а реализации подобного паттерна могут быть следующими (код показал, рассказал):

class Singleton

{

    private static Singleton instance;

    private Singleton()

    {}

    public static Singleton getInstance()

    {

        if (instance == null)

            instance = new Singleton();

        return instance;

    }

}

Это была общая реализация. А теперь давайте разберём нашу аналогию, которую я приводил в определении термина Singleton. (объяснил пример)

class Country

{

public static Country instance;

public string goverment { get; private set; }

private Country(string name)

{

goverment = name;

}

public static Country GetGoverment (string goverment)

{

if (instance == null)

{

instance = new Country(goverment);

}

return instance;

}

}

class Program

{

static void Main ()

{

Country goverm1 = Country.GetGoverment("Правительство Аргентины");

Country goverm2 = Country.GetGoverment("Правительство ЮАР");

System.Console.WriteLine(goverm1.goverment);

System.Console.WriteLine(goverm2.goverment);

}

}

Понятное дело, у данного паттерна имеются как достоинства, так и недостатки. Выделим основные:

1. Достоинства:
   1. Гарантирует наличие единственного экземпляра класса.
   2. Предоставляет к нему глобальную точку доступа.
   3. Реализует отложенную инициализацию объекта-одиночки.
2. Недостатки:
   1. Нарушает принцип единственной ответственности класса.
   2. Маскирует плохой дизайн.
   3. Проблемы мультипоточности (решаемые)

Собственно, поэтому одиночка хоть и является очень широко распространённым и удобным паттерном, его всё ещё необходимо использовать с умом.