**Принцип открытости/закрытости** (Open/Closed Principle) можно сформулировать так:

**Сущности программы должны быть открыты для расширения, но закрыты для изменения**.

Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.

Рассмотрим простейший пример - класс повара:

class Cook

{

    public string Name { get; set; }

    public Cook(string name)

    {

        this.Name = name;

    }

    public void MakeDinner()

    {

        Console.WriteLine("Чистим картошку");

        Console.WriteLine("Ставим почищенную картошку на огонь");

        Console.WriteLine("Сливаем остатки воды, разминаем варенный картофель в пюре");

        Console.WriteLine("Посыпаем пюре специями и зеленью");

        Console.WriteLine("Картофельное пюре готово");

    }

}

И с помощью метода MakeDinner любой объект данного класса сможет сделать картофельного пюре:

Cook bob = new Cook("Bob");

bob.MakeDinner();

Однако одного умения готовить картофельное пюре для повара вряд ли достаточно. Хотелось бы, чтобы повар мог приготовить еще что-то. И в этом случае мы подходим к необходимости изменения функционала класса, а именно метода MakeDinner. Но в соответствии с рассматриваемым нами принципом классы должны быть открыты для расширения, но закрыты для изменения. То есть, нам надо сделать класс Cook отрытым для расширения, но при этом не изменять.

Для решения этой задачи мы можем воспользоваться паттерном Стратегия. В первую очередь нам надо вынести из класса и инкапсулировать всю ту часть, которая представляет изменяющееся поведение. В нашем случае это метод MakeDinner. Однако это не всегда бывает просто сделать. Возможно, в классе много методов, но на начальном этапе сложно определить, какие из них будут изменять свое поведение и как изменять. В этом случае, конечно, надо анализировать возможные способы изменения и уже на основании анализа делать выводы. То есть, все, что подается изменению, выносится из класса и инкапсулируется во вне - во внешних сущностях.

Итак, изменим класс Cook следующим образом:  
class Cook

{

    public string Name { get; set; }

    public Cook(string name)

    {

        this.Name = name;

    }

    public void MakeDinner(IMeal meal)

    {

        meal.Make();

    }

}

interface IMeal

{

    void Make();

}

class PotatoMeal : IMeal

{

    public void Make()

    {

        Console.WriteLine("Чистим картошку");

        Console.WriteLine("Ставим почищенную картошку на огонь");

        Console.WriteLine("Сливаем остатки воды, разминаем варенный картофель в пюре");

        Console.WriteLine("Посыпаем пюре специями и зеленью");

        Console.WriteLine("Картофельное пюре готово");

    }

}

class SaladMeal : IMeal

{

    public void Make()

    {

        Console.WriteLine("Нарезаем помидоры и огурцы");

        Console.WriteLine("Посыпаем зеленью, солью и специями");

        Console.WriteLine("Поливаем подсолнечным маслом");

        Console.WriteLine("Салат готов");

    }

}

Теперь приготовление еды абстрагировано в интерфейсе IMeal, а конкретные способы приготовления определены в реализациях этого интерфейса. А класс Cook делегирует приготовление еды методу Make объекта IMeal.

// Использование класса:

|  |  |
| --- | --- |
| 1  2  3  4 | Cook bob = new Cook("Bob");  bob.MakeDinner(new PotatoMeal());  Console.WriteLine();  bob.MakeDinner(new SaladMeal()); |

Теперь класс Cook закрыт от изменений, зато мы можем легко расширить его функциональность, определив дополнительные реализации интерфейса IMeal.

Другим распространенным способом применения принципа открытости/закрытости представляет паттерн Шаблонный метод. Переделаем предыдущую задачу с помощью этого паттерна:

abstract class MealBase

{

    public void Make()

    {

        Prepare();

        Cook();

        FinalSteps();

    }

    protected abstract void Prepare();

    protected abstract void Cook();

    protected abstract void FinalSteps();

}

class PotatoMeal : MealBase

{

    protected override void Cook()

    {

        Console.WriteLine("Ставим почищенную картошку на огонь");

        Console.WriteLine("Варим около 30 минут");

        Console.WriteLine("Сливаем остатки воды, разминаем варенный картофель в пюре");

    }

    protected override void FinalSteps()

    {

        Console.WriteLine("Посыпаем пюре специями и зеленью");

        Console.WriteLine("Картофельное пюре готово");

    }

    protected override void Prepare()

    {

        Console.WriteLine("Чистим и моем картошку");

    }

}

class SaladMeal : MealBase

{

    protected override void Cook()

    {

        Console.WriteLine("Нарезаем помидоры и огурцы");

        Console.WriteLine("Посыпаем зеленью, солью и специями");

    }

    protected override void FinalSteps()

    {

        Console.WriteLine("Поливаем подсолнечным маслом");

        Console.WriteLine("Салат готов");

    }

    protected override void Prepare()

    {

        Console.WriteLine("Моем помидоры и огурцы");

    }

}

Теперь абстрактный класс MealBase определяет шаблонный метод Make, отдельные части которого реализуются классами наследниками.

Пусть класс Cook теперь принимает набор MealBase в виде меню:

class Cook

{

    public string Name { get; set; }

    public Cook(string name, )

    {

        this.Name = name;

    }

    public void MakeDinner(MealBase[] menu)

    {

        foreach (MealBase meal in menu)

            meal.Make();

    }

}

В данном случае расширение класса опять же производится за счет наследования классов, которые определяют требуемый функционал.

И применение классов:

MealBase[] menu = new MealBase[] { new PotatoMeal(), new SaladMeal() };

Cook bob = new Cook("Bob");

bob.MakeDinner(menu);