|  |
| --- |
| Пермский филиал федерального государственного автономного образовательного учреждения высшего образования «Национальный исследовательский университет  «Высшая школа экономики»  Факультет экономики, менеджмента и бизнес-информатики |
| Образовательная программа бакалавриата «Программная инженерия» |
| **Лабораторная работа № 3**  **«ПОведенческий паттерн проектирования Стратегия (Strategy)»**   |  |  | | --- | --- | |  | Выполнила:  студентка группы ПИ-15-1  Жигалова Светлана Андреевна | |  | Проверил:  преподаватель кафедры информационных технологий в бизнесе  Кычкин Алексей Владимирович | |

Пермь, 2018 год

Оглавление

[1. Назначение и цель применения паттерна 3](#_Toc506841098)

[2. UML-диаграмма паттерна и пояснение основных элементов 3](#_Toc506841099)

[3. Области применения 4](#_Toc506841100)

[4. Особенности паттерна 4](#_Toc506841101)

[5. Реализация паттерна 4](#_Toc506841102)

[Шаги реализации 4](#_Toc506841103)

[Пример реализации 5](#_Toc506841104)

# Назначение и цель применения паттерна

Паттерн Стратегия (Strategy) представляет собой поведенческий шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

# UML-диаграмма паттерна и пояснение основных элементов



***Контекст*** хранит ссылку на объект конкретной стратегии, работая с этим объектом через общий интерфейс стратегий.

*Стратегия* определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма.

Для контекста не важно, какая именно вариация алгоритма будет выбрана, так как все они имеют одинаковый интерфейс.

***Конкретные стратегии*** реализуют различные вариации алгоритма.

Во время выполнения программы, контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.

Обычно, клиент должен создать объект конкретной стратегии и передать его в контекст: либо через конструктор, либо в какой-то другой решающий момент, используя сеттер. Благодаря этому, контекст не знает о том, какая именно стратегия сейчас выбрана.

# Области применения

1. Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
2. Стратегия позволяет варьировать поведение объекта во время выполнения программы, подставляя в него различные объекты-поведения (например, отличающиеся балансом скорости и потребления ресурсов).
3. Когда у вас есть множество похожих классов, отличающихся только некоторым поведением.
4. Стратегия позволяет вынести отличающееся поведение в отдельную иерархию классов и свести первоначальные классы к одному, сделав его поведение настраиваемым.
5. Когда вы не хотите обнажать детали реализации алгоритмов для других классов.
6. Стратегия позволяет изолировать код, данные и зависимости алгоритмов от других объектов, скрыв из внутри собственных классов.
7. Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет вариацию алгоритма.
8. Стратегия помещает каждую лапу такого оператора в отдельный класс-стратегию. Затем контекст получает определённый объект-стратегию от клиента и делегирует ему работу. Если вдруг понадобится сменить алгоритм, в контекст можно подать другую стратегию.

# Особенности паттерна

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Горячая замена алгоритмов на лету. | Усложняет программу за счёт дополнительных классов. |
| Изолирует код и данные алгоритмов от остальных классов. | Клиент должен знать, в чём разница между стратегиями, чтобы выбрать подходящую. |
| Уход от наследования к делегированию. |  |
| Реализует *принцип открытости/закрытости*. |

# Реализация паттерна

## Шаги реализации

1. Определите алгоритм, который подвержен частым изменениям. Также подойдёт алгоритм, имеющий несколько вариаций, которые выбираются во время выполнения программы.
2. Создайте интерфейс стратегий, описывающий этот алгоритм. Он должен быть общим для всех вариантов алгоритма.
3. Поместите вариации алгоритма в собственные классы, которые реализуют этот интерфейс.
4. В классе контекста создайте поле для хранения ссылки на текущий объект-стратегию, а также метод для её изменения. Убедитесь в том, что контекст работает с этим объектом только через общий интерфейс стратегий.
5. Клиенты контекста должны подавать в него соответствующий объект-стратегию, когда хотят, чтобы контекст вёл себя определённым образом.

## Пример реализации

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

public interface IStrategy

{

    void Algorithm();

}

public class ConcreteStrategy1 : IStrategy

{

    public void Algorithm()

    {}

}

public class ConcreteStrategy2 : IStrategy

{

    public void Algorithm()

    {}

}

public class Context

{

    public IStrategy ContextStrategy { get; set; }

    public Context(IStrategy \_strategy)

    {

        ContextStrategy = \_strategy;

    }

    public void ExecuteAlgorithm()

    {

        ContextStrategy.Algorithm();

    }

}

**Участники**

Как видно из диаграммы, здесь есть следующие участники:

* Интерфейс IStrategy, который определяет метод Algorithm(). Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
* Классы ConcreteStrategy1 и ConcreteStrategy, которые реализуют интерфейс IStrategy, предоставляя свою версию метода Algorithm(). Подобных классов-реализаций может быть множество.
* Класс Context хранит ссылку на объект IStrategy и связан с интерфейсом IStrategy отношением агрегации.

В данном случае объект IStrategy заключена в свойстве ContextStrategy, хотя также для нее можно было бы определить приватную переменную, а для динамической установки использовать специальный метод.

Конкретный пример реализации.

Существуют различные легковые машины, которые используют разные источники энергии: электричество, бензин, газ и так далее. Есть гибридные автомобили. В целом они похожи и отличаются преимущественно видом источника энергии. Не говоря уже о том, что мы можем изменить применяемый источник энергии, модифицировав автомобиль. И в данном случае вполне можно применить паттерн стратегию:

class Program

{

    static void Main(string[] args)

    {

        Car auto = new Car(4, "Volvo", new PetrolMove());

        auto.Move();

        auto.Movable = new ElectricMove();

        auto.Move();

        Console.ReadLine();

    }

}

interface IMovable

{

    void Move();

}

class PetrolMove : IMovable

{

    public void Move()

    {

        Console.WriteLine("Перемещение на бензине");

    }

}

class ElectricMove : IMovable

{

    public void Move()

    {

        Console.WriteLine("Перемещение на электричестве");

    }

}

class Car

{

    protected int passengers; // кол-во пассажиров

    protected string model; // модель автомобиля

    public Car(int num, string model, IMovable mov)

    {

        this.passengers = num;

        this.model = model;

        Movable = mov;

    }

    public IMovable Movable { private get; set; }

    public void Move()

    {

        Movable.Move();

    }

}

В данном случае в качестве IStrategy выступает интерфейс IMovable, определяющий метод Move(). А реализующий этот интерфейс семейство алгоритмов представлено классами ElectricMove и PetroleMove. И данные алгоритмы использует класс Car.