Стратегия (шаблон проектирования)

Стратегия (***Strategy***) — поведенческий шаблон проектирования, предназначенный для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости. Это позволяет выбирать алгоритм путём определения соответствующего класса. Шаблон ***Strategy*** позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют.

Основные характеристики

Задача

По типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. Если используется правило, которое не подвержено изменениям, нет необходимости обращаться к шаблону «стратегия».

Мотивы

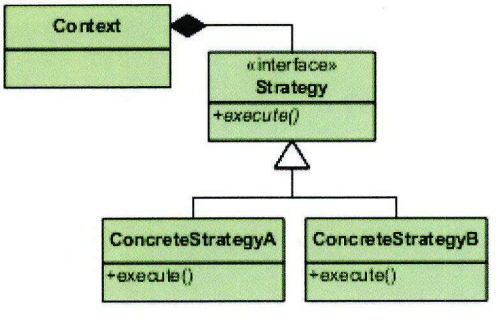
- Программа должна обеспечивать различные варианты алгоритма или поведения
- Нужно изменять поведение каждого экземпляра класса
- Необходимо изменять поведение объектов на стадии выполнения
- Введение интерфейса позволяет классам-клиентам ничего не знать о классах, реализующих этот интерфейс и инкапсулирующих в себе конкретные алгоритмы

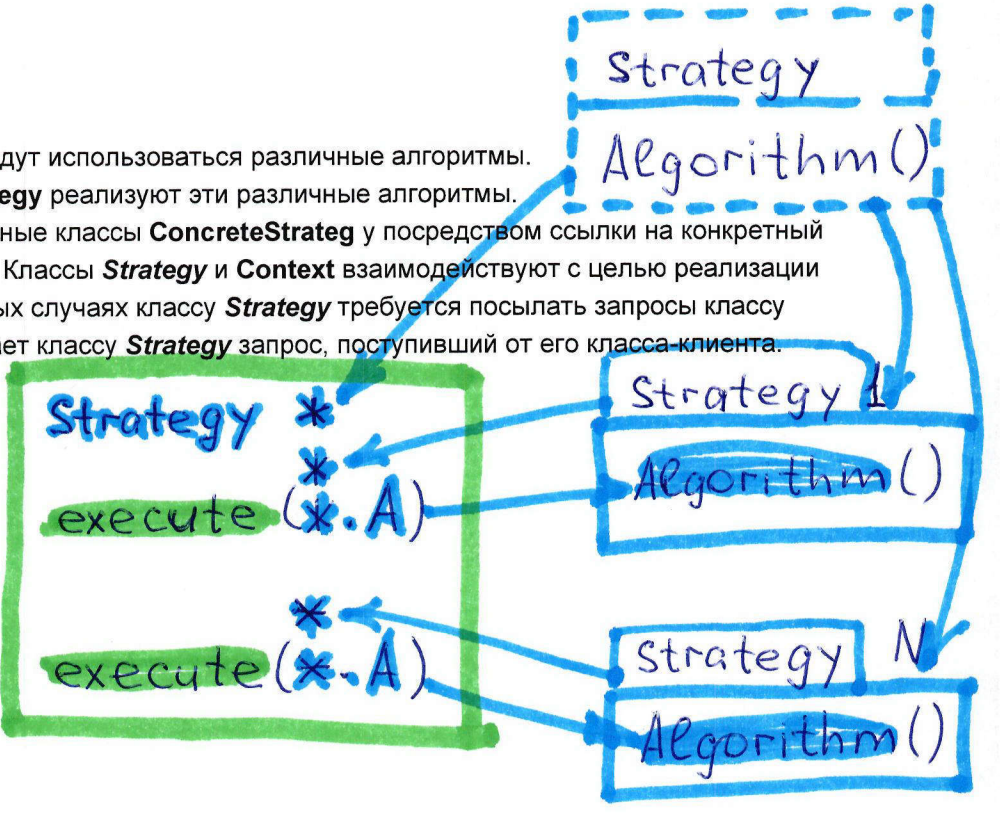
Способ решения

Отделение процедуры выбора алгоритма от его реализации. Это позволяет сделать выбор на основании контекста.

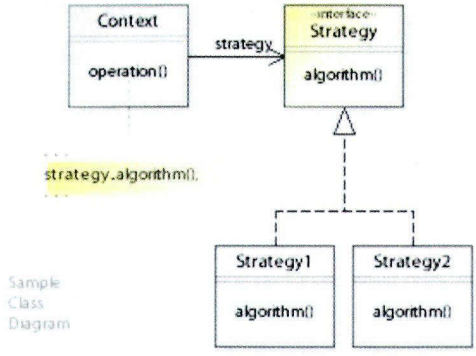
Участники

- Класс ***Strategy*** определяет, как будут использоваться различные алгоритмы.
- Конкретные классы **ConcreteStrategy** реализуют эти различные алгоритмы.
- Класс **Context** использует конкретные классы **ConcreteStrateg** у посредством ссылки на конкретный тип абстрактного класса ***Strategy***. Классы ***Strategy*** и **Context** взаимодействуют с целью реализации выбранного алгоритма (в некоторых случаях классу ***Strategy*** требуется посылать запросы классу **Context**). Класс **Context** пересылает классу ***Strategy*** запрос, поступивший от его класса-клиента.

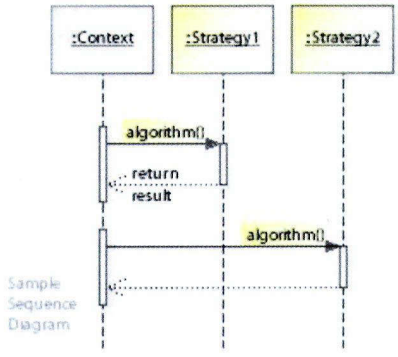




Strategy



Strategy




```
using System;

//////////////////////////////////////////////////////////////////////////
// Класс реализующий конкретную стратегию, должен наследовать этот интерфейс
// Класс контекста использует этот интерфейс для вызова конкретной стратегии
public interface IStrategy
{
void Algorithm();
}

//////////////////////////////////////////////////////////////////////////
// Реализаций может быть сколько угодно много.
// Первая конкретная реализация-стратегия.
public class ConcreteStrategyA : IStrategy
{
public void Algorithm()
{Console.WriteLine("ConcreteStrategyA():Algorithm()  run AAAAA.");
}}
//////////////////////////////////////////////////////////////////////////
// Вторая конкретная реализация-стратегия.
public class ConcreteStrategy2 : IStrategy
{
public void Algorithm()
{Console.WriteLine("ConcreteStrategy2():Algorithm() = run2");
}}
//////////////////////////////////////////////////////////////////////////
// Контекст, использующий стратегию для решения своей задачи.
public class Context
{
// Ссылка на интерфейс IStrategy позволяет автоматически переключаться
// между реализациями конкретной стратегии.
private IStrategy _strategy;

// Конструктор контекста. Инициализирует объект стратегией.
public Context(IStrategy strategy)
{_strategy = strategy; }

// Метод для установки стратегии Служит для замены во время выполнения.
public void SetStrategy(IStrategy strategy)
{_strategy = strategy;}

// Некоторая функциональность контекста, которая выбирает
// стратегию и использует её для решения своей задачи.
public void ExecuteOperation()
{_strategy.Algorithm();
}}
```

```
//////////////////////////////////////////////////////////////////////
// Класс приложения. В данном примере выступает как клиент контекста.
public static class Program
{
public static void Main()
{// Создаём контекст и инициализируем его первой стратегией.
Context context = new Context(new ConcreteStrategyA());
// Выполняем операцию контекста, которая использует первую стратегию.
context.ExecuteOperation();

// Заменяем в контексте первую стратегию второй.
context.SetStrategy(new ConcreteStrategy2());
// Выполняем операцию контекста, которая теперь использует вторую стратегию.
context.ExecuteOperation();
Console.ReadKey();
}}


//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
ConcreteStrategyA():Algorithm()  run AAAAA.
ConcreteStrategy2() :Algorithm() = run2
```

Применимость: **Стратегия** часто используется в Python-коде, особенно там, где нужно подменять алгоритм во время выполнения программы. Многие примеры стратегии можно заменить простыми ***lambda***-выражениями.

```
from __future__ import annotations
from abc import ABC, abstractmethod
from typing import List

##############################################
class Strategy(ABC):
    # Интерфейс Strategy(ABC)/Стратегии объявляет операции, общие для всех
    # поддерживаемых версий некоторого алгоритма.
    # Context()/Контекст использует этот интерфейс для вызова алгоритма, определённого
    # ConcreteStrategyA(Strategy)/Конкретными Стратегиями.
    @abstractmethod
    def do_algorithm(self, data: List):
        pass

# ConcreteStrategy(Strategy)/Конкретные Стратегии реализуют алгоритм,
# следуя базовому интерфейсу def do_algorithm(self, data: List) Strategy(ABC)/Стратегии.
# Этот интерфейс делает их взаимозаменяемыми в Context()/Контексте.
##############################
class ConcreteStrategyA(Strategy):
    def do_algorithm(self, data: List) -> List:
        return sorted(data)
##############################
class ConcreteStrategyB(Strategy):
    def do_algorithm(self, data: List) -> List:
        return reversed(sorted(data))
```

```python
##################################################
class Context():
    """Контекст определяет интерфейс, представляющий интерес для клиентов."""
    def __init__(self, strategy: Strategy) -> None:
        #Обычно Context()/Контекст принимает стратегию через конструктор, а также
        #предоставляет сеттер для её изменения во время выполнения.
        self._strategy = strategy

    @property
    def strategy(self) -> Strategy:
        #Context()/Контекст хранит ссылку на один из объектов Strategy()/Стратегии.
        #Context()/Контекст не знает конкретного класса стратегии.
        # Он должен работать со всеми стратегиями через
        # интерфейс Strategy()/Стратегии = do_algorithm(List).
        return self._strategy

    @strategy.setter
    def strategy(self, strategy: Strategy) -> None:
        #Обычно Context/Контекст позволяет заменить объект
        #Strategy(ABC)/Стратегии во время выполнения.
        self._strategy = strategy

    def do_some_business_logic(self) -> None:
        # Вместо того, чтобы самостоятельно реализовывать множественные версии алгоритма,
        # Context/Контекст делегирует некоторую работу объекту Strategy()/Стратегии.
        print("class Context(): def do_some_business_logic()==_strategy.do_algorithm(a.b.c.d.e)")
        result = self._strategy.do_algorithm(["a", "b", "c", "d", "e"])
        print(",".join(result))

##################################################
if __name__ == "__main__":
    # Клиентский код выбирает конкретную стратегию и передаёт её в контекст.
    # Context(ConcreteStrategyA()) , Context(ConcreteStrategyB())
    # Клиент должен знать о различиях между стратегиями, чтобы сделать правильный выбор.

    context = Context(ConcreteStrategyA())
    print("Context(ConcreteStrategyA()): return sorted(data)")
    context.do_some_business_logic()
    print()

    print("ConcreteStrategyB():  reversed ")
    context.strategy = ConcreteStrategyB()
    context.do_some_business_logic()
```

```
Context(ConcreteStrategyA()): return sorted(data)
class Context(): def do_some_business_logic() == _strategy.do_algorithm(a.b.c.d.e)
a,b,c,d,e

ConcreteStrategyB():  reversed
class Context(): def do_some_business_logic() == _strategy.do_algorithm(a.b.c.d.e)
e,d,c,b,a
```