СТРАТЕГИЯ

Также известен как: Strategy

Стратегия — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс. После чего, алгоритмы можно взаимозаменять прямо во время исполнения программы.

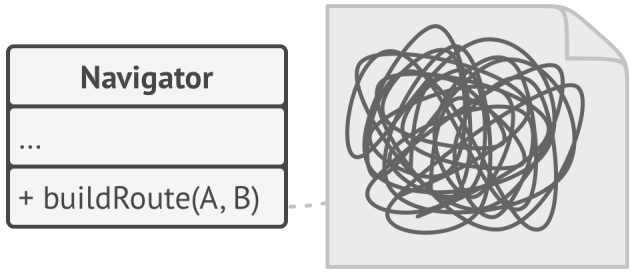
☹ Проблема

Вы решили написать приложение-навигатор для путешественников. Он должен показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.

Одной из самых востребованных функций был поиск и прокладка маршрута, поэтому вы планировали посвятить ей особое внимание. Пребывая в неизвестном ему городе, пользователь должен иметь возможность указать начальную точку и пункт назначения. А навигатор — проложит оптимальный путь.

Первая версия вашего навигатора могла прокладывать маршрут лишь по дорогам, поэтому отлично подходила для путешествий на автомобиле. Но, очевидно, не все ездят в отпуск на машине. Поэтому следующим шагом вы добавили в навигатор прокладку пеших маршрутов. Через некоторое время выяснилось, что некоторые люди предпочитают ездить по городу на общественном транспорте, поэтому вы добавили и такую опцию прокладки пути.

Но и это ещё не всё. В ближайшей перспективе вы хотели бы добавить прокладку маршрутов по велодорожкам. А в отдалённом будущем — интересные маршруты посещения достопримечательностей.



Код навигатора становится слишком раздутым.

Если с популярностью навигатора не было никаких проблем, то техническая часть вызывала вопросы и периодическую головную боль. С каждым новым алгоритмом, код основного класса навигатора увеличивался вдвое. В таком большом классе стало довольно трудно ориентироваться.

Любое изменение алгоритмов поиска, будь то исправление багов или добавление нового алгоритма, затрагивало основной класс. Это повышало риск сделать ошибку, случайно задев остальной работающий код.

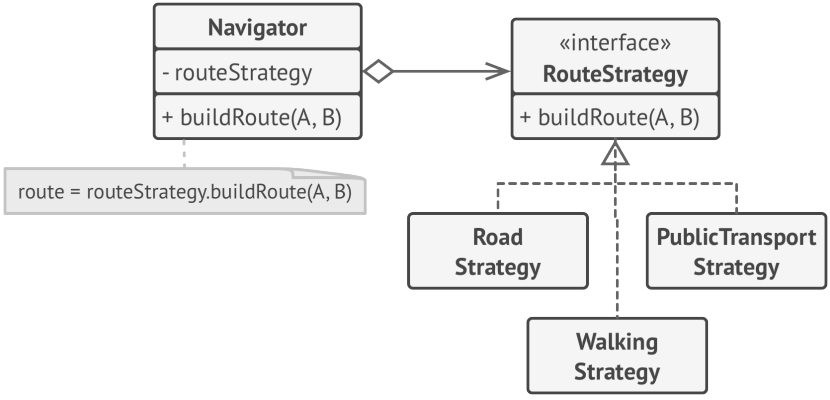
Кроме того, осложнялась командная работа с другими программистами, которых вы наняли после успешного релиза навигатора. Ваши изменения нередко затрагивали один и тот же код, создавая конфликты, которые требовали дополнительного времени на их разрешение.

☺ Решение

Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.

Вместо того чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет отыгрывать роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. А для смены алгоритма будет достаточно подставить в контекст другой объект-стратегию.

Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.

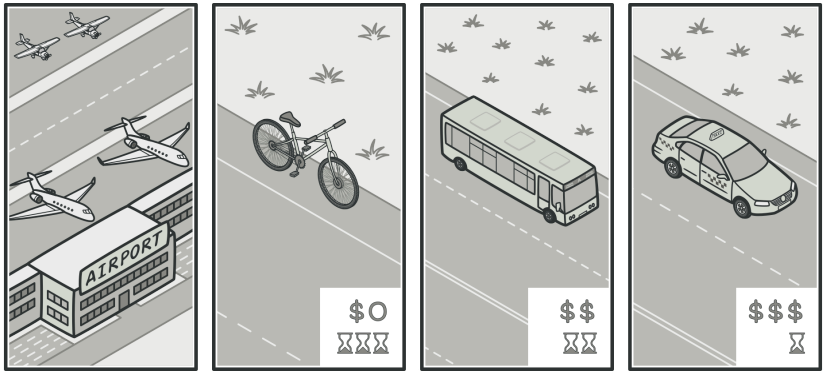


Стратегии постройки пути.

В нашем примере, каждый алгоритм поиска пути переедет в свой собственный класс с единственным методом, принимающим в параметрах начальную и конечную точку маршрута и возвращающий массив точек маршрута.

Хотя каждый класс будет прокладывать маршрут по-своему, для навигатора это не будет иметь никакого значения, так как его работа заключается только в отрисовке маршрута. Навигатору достаточно подать в стратегию данные о начале и конце маршрута, чтобы получить массив точек маршрута в оговорённом формате. Класснавигатора будет иметь метод изменения стратегии, позволяющий изменять стратегию поиска пути на лету. Им сможет воспользоваться клиентский код навигатора, например, кнопки-переключатели типов маршрутов в пользовательском интерфейсе.

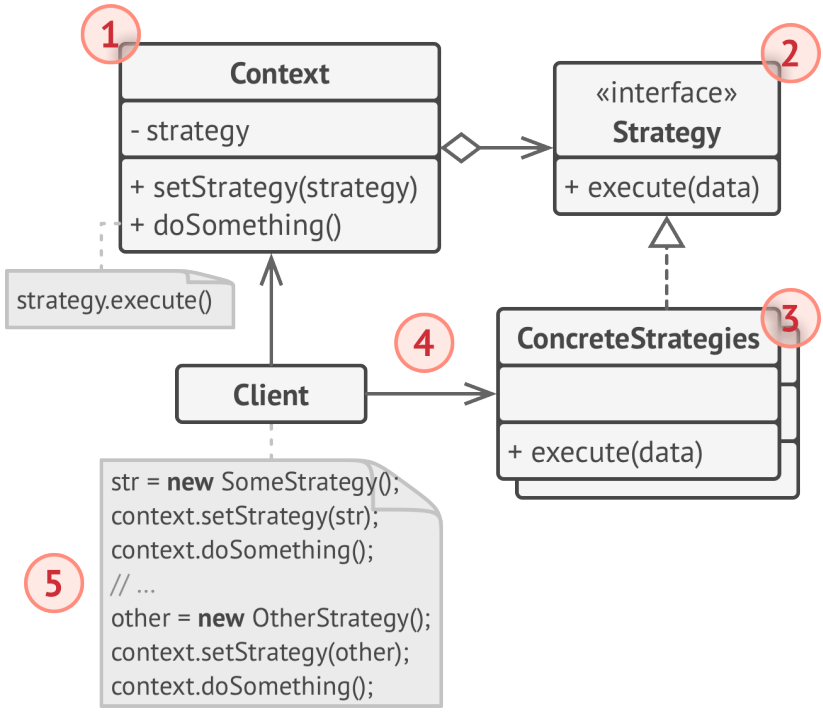
Аналогия из жизни



Различные стратегии попадания в аэропорт.

Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией. Вы выбираете конкретную стратегию в зависимости от контекста (например, наличия денег или времени до отлёта).

Структура



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

2. **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.

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

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

5. 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.

Псевдокод

В этом примере, контекст использует **Стратегию** для выполнения той или иной арифметической операции.

```
// Общий интерфейс всех стратегий.
interface Strategy is
  method execute(a, b)

// Каждая конкретная стратегия реализует общий интерфейс
// своим способом.
class ConcreteStrategyAdd implements Strategy is
  method execute(a, b) is
    return a + b

class ConcreteStrategySubtract implements Strategy is
  method execute(a, b) is
    return a - b

class ConcreteStrategyMultiply implements Strategy is
  method execute(a, b) is
    return a * b

// Контекст всегда работает со стратегиями через общий
// интерфейс. Он не знает какая именно стратегия ему подана.
class Context is
  private strategy: Strategy

  method setStrategy(Strategy strategy) is
    this.strategy = strategy

  method executeStrategy(int a, int b) is
    return strategy.execute(a, b)

// Конкретная стратегия выбирается на более высоком уровне,
// например, конфигуратором всего приложения. Готовый
// объект-стратегия подаётся в клиентский объект, а затем
// может быть заменён другой стратегией в любой момент
// на лету.
```

```
class ExampleApplication is
  method main() is
    Create context object.

    Read first number.
    Read last number.
    Read the desired action from user input.

    if (action == addition) then
      context.setStrategy(new ConcreteStrategyAdd())

    if (action == subtraction) then
      context.setStrategy(new ConcreteStrategySubtract())

    if (action == multiplication) then
      context.setStrategy(new ConcreteStrategyMultiply())

    result = context.executeStrategy(First number, Second number)

    Print result.
```

Применимость

Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.

Стратегия позволяет варьировать поведение объекта во время выполнения программы, подставляя в него различные объекты-поведения (например, отличающиеся балансом скорости и потребления ресурсов).

Когда у вас есть множество похожих классов, отличающихся только некоторым поведением.

Стратегия позволяет вынести отличающееся поведение в отдельную иерархию классов и свести первоначальные классы к одному, сделав его поведение настраиваемым.

Когда вы не хотите обнажать детали реализации алгоритмов для других классов.

Стратегия позволяет изолировать код, данные и зависимости алгоритмов от других объектов, скрыв из внутри собственных классов.

Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет вариацию алгоритма.

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.

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.

Преимущества и недостатки

- ✔ Горячая замена алгоритмов на лету.
- ✔ Изолирует код и данные алгоритмов от остальных классов.
- ✔ Уход от наследования к делегированию.

✔ Реализует *принцип открытости/закрытости*.

✖ Усложняет программу за счёт дополнительных классов.

✖ Клиент должен знать, в чём разница между стратегиями, чтобы выбрать подходящую.

⇄ Отношения с другими паттернами

- **Мост**, **Стратегия** и **Состояние** (а также слегка и **Адаптер**) имеют схожие структуры классов — все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны — это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.

- **Команда** и **Стратегия** похожи по духу, но отличаются масштабом и применением:

  - *Команду* используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.

  - С другой стороны, *Стратегия* описывает разные способы сделать одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.

- **Стратегия** меняет поведение объекта «изнутри», а **Декоратор** изменяет его «снаружи».

- **Шаблонный метод** использует наследование, чтобы расширять части алгоритма. **Стратегия** использует делегирование, чтобы изменять выполняемые алгоритмы на лету. *Шаблонный метод* работает на уровне классов. *Стратегия* позволяет менять логику отдельных объектов.

- **Состояние** можно рассматривать как надстройку над **Стратегией**. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в *Стратегии* эти объекты не знают друг о друге и никак не связаны. В *Состоянии* сами конкретные состояния могут переключать контекст.