

Паттерны проектирования

Лекция

Что такое паттерн?

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

Паттерны это не алгоритмы. И если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

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.

Если привести аналогии, то алгоритм — это кулинарный рецепт с чёткими шагами, а паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

Описание паттернов.

- проблема, которую решает паттерн;
- мотивации к решению проблемы способом, который предлагает паттерн;
- структуры классов, составляющих решение;
- пример на одном из языков программирования;
- особенностей реализации в различных контекстах;
- связей с другими паттернами.

Зачем знать паттерны?



Вы можете вполне успешно работать, не зная ни одного паттерна. Более того, вы могли уже не раз реализовать какой-то из паттернов, даже не подозревая об этом.

Осознанное владение инструментом как раз и отличает профессионала от любителя. Вы можете забить гвоздь молотком, а можете и дрелью, если сильно постараетесь. Но профессионал знает, что главная фишка дрели совсем не в этом. Итак, зачем же знать паттерны?

Проверенные решения. Вы тратите меньше времени, используя готовые решения, вместо повторного изобретения велосипеда. До некоторых решений вы смогли бы додуматься и сами, но многие могут быть для вас открытием.

Стандартизация кода. Вы делаете меньше просчётов при проектировании, используя типовые унифицированные решения, так как все скрытые проблемы в них уже давно найдены.

Общий программистский словарь. Вы произносите название паттерна, вместо того, чтобы час объяснять другим программистам, какой крутой дизайн вы придумали и какие классы для этого нужны.

Классификация паттернов

Самые низкоуровневые и простые паттерны — **идиомы**. Они не универсальны, поскольку применимы только в рамках одного языка программирования.

Самые универсальные — **архитектурные паттерны**, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных её элементов.



Порождающие паттерны

Отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

Фабричный метод — определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Абстрактная фабрика — позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Строитель — позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



Прототип - позволяет копировать объекты, не вдаваясь в подробности их реализации.

Одиночка — гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Структурные паттерны

Адаптер - позволяет объектам с несовместимыми интерфейсами работать вместе.

Мост — разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Компоновщик — позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

Декоратор — позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Фасад — предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

Заместитель — позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

Отвечают за построение удобных в поддержке иерархий классов.



Легковес - позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Поведенческие паттерны

Посетитель — позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Итератор — даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Снимок — позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Решают задачи эффективного и безопасного взаимодействия между объектами программы.

Наблюдатель

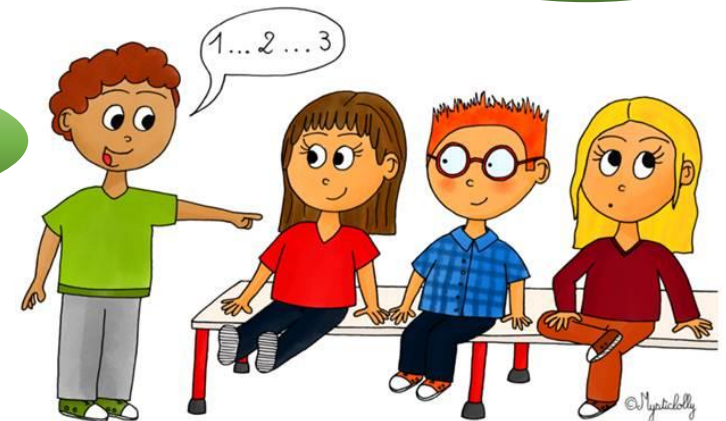
Снимок

Посетитель

Состояние



Итератор



Поведенческие паттерны

Цепочка обязанностей — позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Команда — превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Посредник — позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Стратегия — определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Решают задачи эффективного и безопасного взаимодействия между объектами программы.

Команда

Стратегия

Цепочка
обязанностей

Шаблонный
метод



Посредник



Реализация паттерна «Стратегия». I

Суть паттерна

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. Оно должно показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.



Одной из самых востребованных функций являлся поиск и прокладывание маршрутов. Пребывая в неизвестном ему городе, пользователь должен иметь возможность указать начальную точку и пункт назначения, а навигатор — проложит оптимальный путь.



Развитие проекта «Навигатор»

Реализация паттерна «Стратегия». II

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.

Реализация паттерна «Стратегия». III

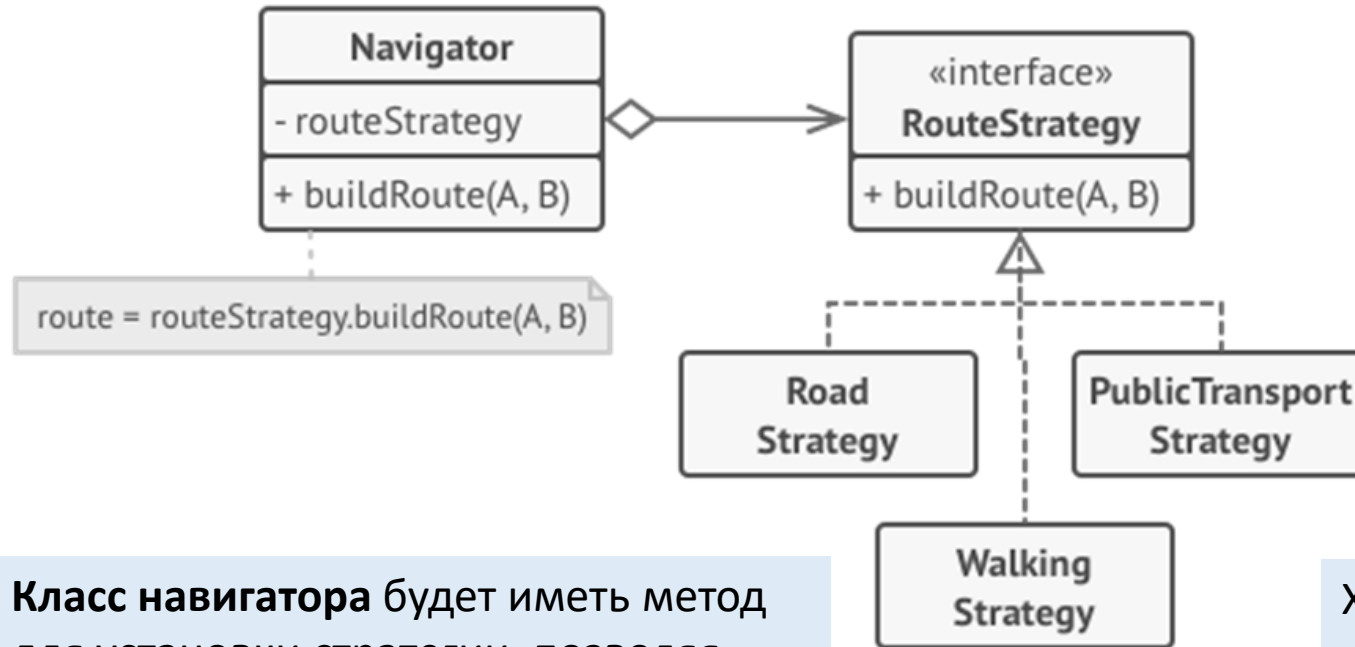
Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.

Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам 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.



Реализация паттерна «Стратегия». IV



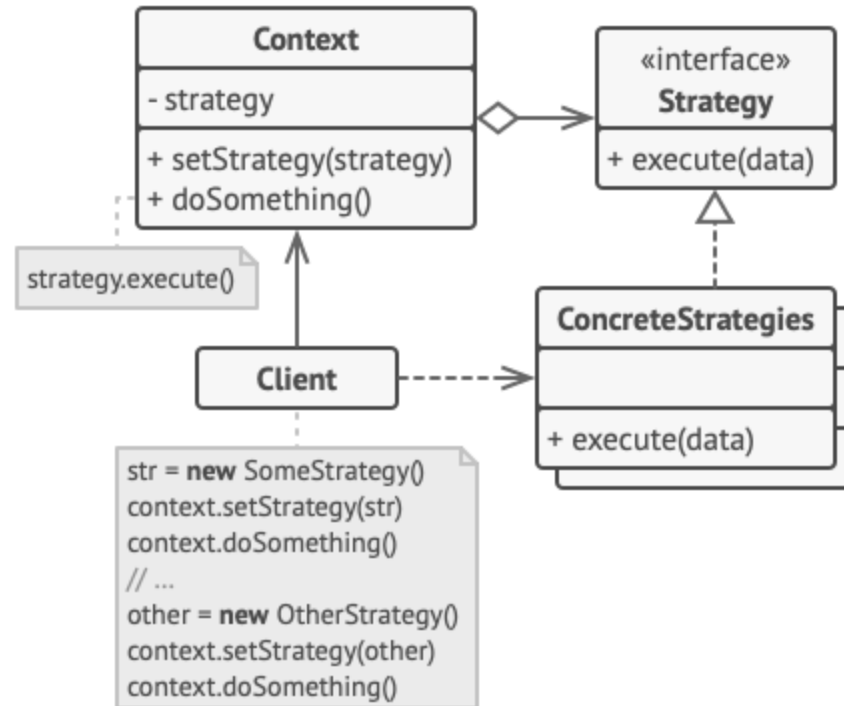
Класс навигатора будет иметь метод для установки стратегии, позволяя изменять стратегию поиска пути на лету. Такой метод пригодится клиентскому коду навигатора, например, переключателям типов маршрутов в пользовательском интерфейсе.

В нашем примере каждый алгоритм поиска пути перейдет в свой собственный класс. В этих классах будет определён лишь **один метод, принимающий в параметрах координаты начала и конца пути**, а 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.

Реализация паттерна «Стратегия». V

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



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

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

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

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

5. **Клиент** должен создать объект конкретной стратегии и передать его в конструктор контекста. Кроме этого, клиент должен иметь возможность заменить стратегию на лету, используя сеттер. Благодаря этому, контекст не будет знать о том, какая именно стратегия сейчас выбрана.

Реализация паттерна «Стратегия». VI

interface Strategy is
method execute(a, b)

Общий
интерфейс всех
стратегий

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

Каждая конкретная
стратегия реализует
общий интерфейс своим
способом.

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

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)

Реализация паттерна «Стратегия». VII

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

```
// 1. Создать объект контекста.  
// 2. Получить первое число (n1).  
// 3. Получить второе число (n2).  
// 4. Получить желаемую операцию.  
// 5. Затем, выбрать стратегию:
```

```
if (action == addition) then  
    context.setStrategy(new ConcreteStrategyAdd())  
  
if (action == subtraction) then  
    context.setStrategy(new ConcreteStrategySubtract())  
  
if (action == multiplication) then  
    context.setStrategy(new ConcreteStrategyMultiply())  
  
// 6. Выполнить операцию с помощью стратегии:  
result = context.executeStrategy(n1, n2)  
  
// 7. Вывести результат на экран.
```


Реализация паттерна «Стратегия». VIII

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.

Реализация паттерна «Стратегия». IX

Пример структуры паттерна

```
class Strategy
{
public:
    virtual ~Strategy() = default;
    virtual std::string doAlgorithm(std::string_view data) const = 0;
};
```

Интерфейс Стратегии объявляет операции, общие для всех поддерживаемых версий некоторого алгоритма.

Контекст использует этот интерфейс для вызова алгоритма, определённого Конкретными Стратегиями.

Реализация паттерна «Стратегия». X

```
class Context{
private:
    std::unique_ptr<Strategy> strategy_;
public:
    explicit Context(std::unique_ptr<Strategy> &&strategy = {}) :
        strategy_(std::move(strategy)){}
    void set_strategy(std::unique_ptr<Strategy> &&strategy){
        strategy_ = std::move(strategy);
    }
    void doSomeBusinessLogic() const {
        if (strategy_) {
            std::cout << "Context: Sorting data using the strategy
                          (not sure how it'll do it)\n";
            std::string result = strategy_->doAlgorithm("aecbd");
            std::cout << result << "\n";
        } else {
            std::cout << "Context: Strategy isn't set\n";
        }
    }
};
```

Контекст определяет интерфейс, представляющий интерес для клиентов.

@var Strategy Контекст хранит ссылку на один из объектов Стратегии. Контекст не знает конкретного класса стратегии. Он должен работать со всеми стратегиями через интерфейс Стратегии.

Обычно Контекст принимает стратегию через конструктор, а также предоставляет сеттер для её изменения во время выполнения.

Обычно Контекст позволяет заменить объект Стратегии во время выполнения.

Вместо того, чтобы самостоятельно реализовывать множественные версии алгоритма, Контекст делегирует некоторую работу объекту Стратегии.

Реализация паттерна «Стратегия». XI

```
class ConcreteStrategyA : public Strategy{
public:
    std::string doAlgorithm(std::string_view data) const override{
        std::string result(data);
        std::sort(std::begin(result), std::end(result));
        return result;
    }
};
```

Конкретные Стратегии реализуют алгоритм, следуя базовому интерфейсу Стратегии. Этот интерфейс делает их взаимозаменяемыми в Контексте.

```
class ConcreteStrategyB : public Strategy{
public:
    std::string doAlgorithm(std::string_view data) const override {
        std::string result(data);
        std::sort(std::begin(result), std::end(result), std::greater<>());
        return result;
    }
};
```

Реализация паттерна «Стратегия». XII

Клиентский код выбирает конкретную стратегию и передаёт её в контекст. Клиент должен знать о различиях между стратегиями, чтобы сделать правильный выбор.

```
int main(){
    clientCode();
    return 0;
}
```

```
void clientCode()
{
    Context context(std::make_unique<ConcreteStrategyA>());
    std::cout << "Client: Strategy is set to normal sorting.\n";
    context.doSomeBusinessLogic();

    std::cout << "\n";

    std::cout << "Client: Strategy is set to reverse sorting.\n";
    context.set_strategy(std::make_unique<ConcreteStrategyB>());
    context.doSomeBusinessLogic();
}
```

Результат выполнения

Client: Strategy is set to normal sorting.

Context: Sorting data using the strategy (not sure how it'll do it)

abcde

Client: Strategy is set to reverse sorting.

Context: Sorting data using the strategy (not sure how it'll do it)

edcba

Реализация паттерна «Прототип». I

Суть паттерна

Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Проблема

У вас есть объект, который нужно скопировать. Как это сделать? Нужно создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый.

Прекрасно! Но есть нюанс. Не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной, а значит — недоступной для остального кода программы.



Но есть и другая проблема. Копирующий код станет зависим от классов копируемых объектов. Ведь, чтобы перебрать все поля объекта, нужно привязаться к его классу. Из-за этого вы не сможете копировать объекты, зная только их интерфейсы, а не конкретные классы.

Реализация паттерна «Прототип». II

Решение

Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет всего один метод **clone**.

Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. Так получится скопировать даже приватные поля, так как большинство языков программирования разрешает доступ к приватным полям любого объекта текущего класса.



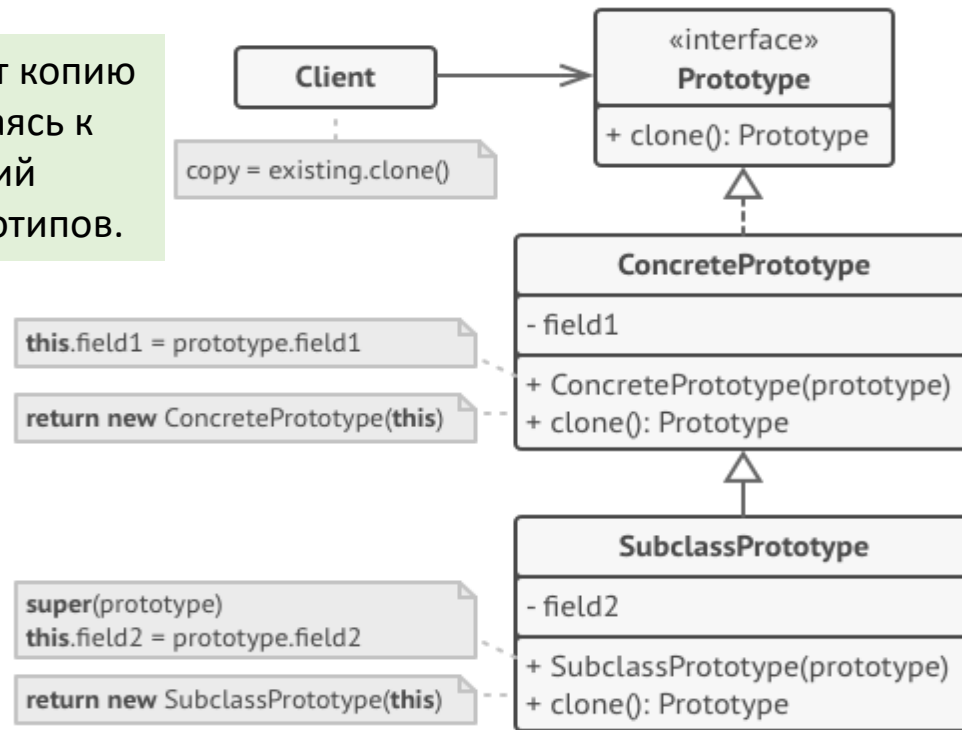
Объект, который копируют, называется **прототипом** (откуда и название паттерна). Когда объекты программы содержат сотни полей и тысячи возможных конфигураций, прототипы могут служить своеобразной альтернативой созданию подклассов.

В этом случае все возможные прототипы заготавливаются и настраиваются на этапе инициализации программы. Потом, когда программе нужен новый объект, она создаёт копию из подготовленного прототипа.

Реализация паттерна «Прототип». III

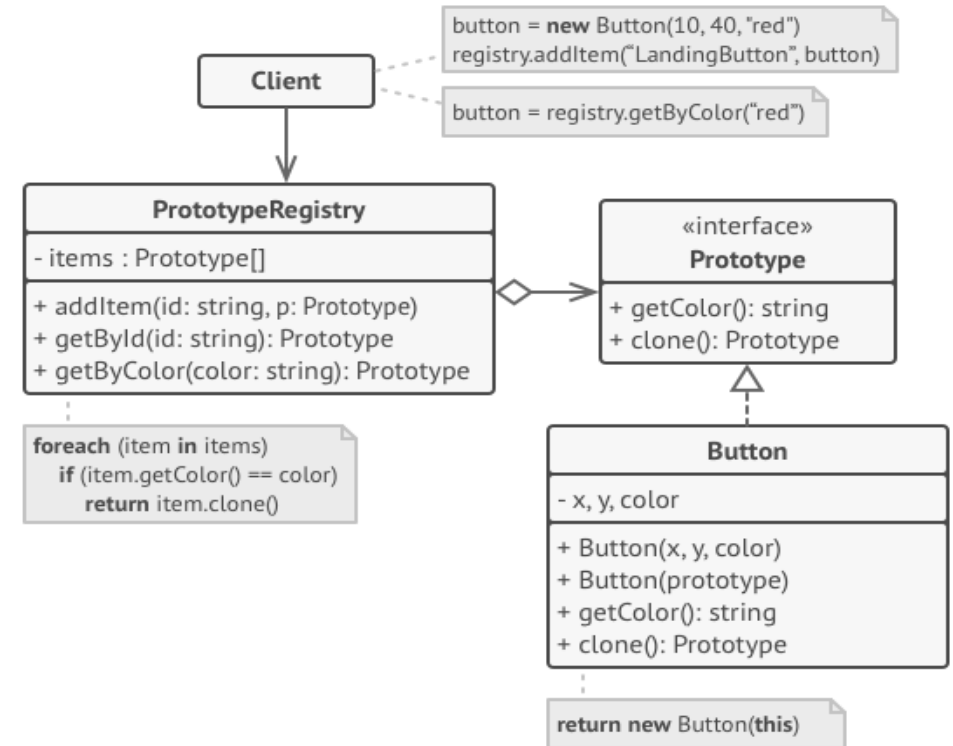
Структура

3. Клиент создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.



1. Интерфейс прототипов описывает операции клонирования. В большинстве случаев — это единственный метод **clone**.

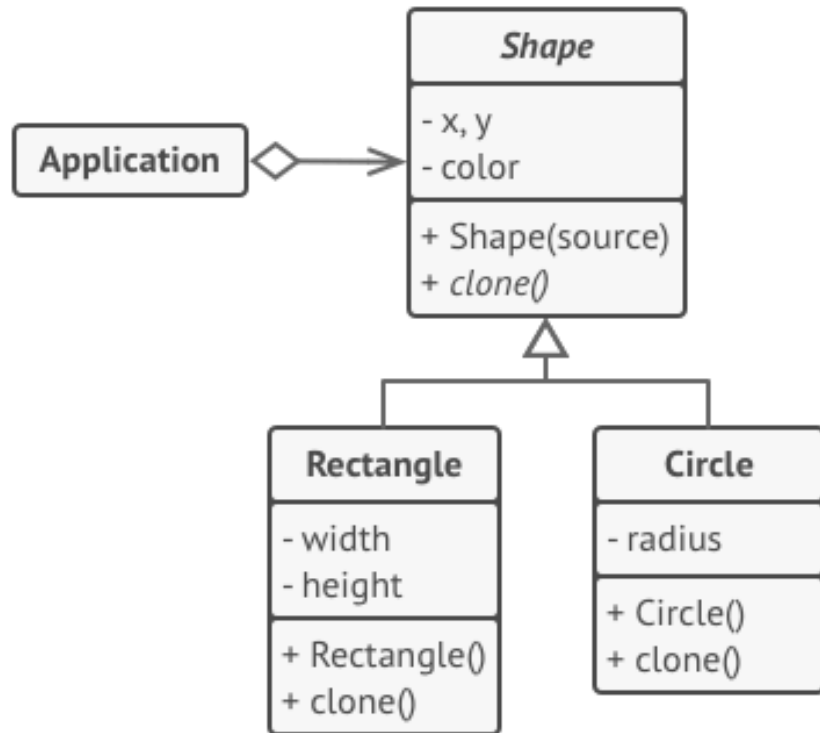
2. Конкретный прототип реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту.



1. Хранилище прототипов облегчает доступ к часто используемым прототипам, храня набор предварительно созданных эталонных, готовых к копированию объектов. Простейшее хранилище может быть построено с помощью хеш-таблицы вида имя-прототипа → прототип. Но для удобства поиска прототипы можно маркировать и другими критериями, а не только условным именем.

Реализация паттерна «Прототип». IV

В этом примере Прототип позволяет производить точные копии объектов геометрических фигур, не привязываясь к их классам.



Все фигуры реализуют интерфейс клонирования и предоставляют метод для воспроизводства самой себя. Подклассы используют метод клонирования родителя, а затем копируют собственные поля в получившийся объект.

// Базовый прототип.

abstract class Shape is

field X: int

field Y: int

field color: string

// Обычный конструктор.

constructor Shape() is

// ...

// Конструктор прототипа.

constructor Shape(source: Shape) is

this()

this.X = source.X

this.Y = source.Y

this.color = source.color

// Результатом операции клонирования всегда будет объект из иерархии классов Shape.

abstract method clone():Shape

Реализация паттерна «Прототип». V

class Rectangle extends Shape is

field width: int

field height: int

constructor Rectangle(source: Rectangle) is

// Вызов родительского конструктора нужен,
чтобы скопировать потенциальные приватные
поля, объявленные в родительском классе.

super(source)

this.width = source.width

this.height = source.height

method clone():Shape is

return new Rectangle(this)

Конкретный прототип. Метод клонирования создаёт новый объект текущего класса, передавая в его конструктор ссылку на собственный объект. Благодаря этому операция клонирования получается атомарной — пока не выполнится конструктор, нового объекта ещё не существует. Но как только конструктор завершит работу, мы получим полностью готовый объект-клон, а не пустой объект, который нужно ещё заполнить.

class Circle extends Shape is

field radius: int

constructor Circle(source: Circle) is

super(source)

this.radius = source.radius

method clone():Shape is

return new Circle(this)

Реализация паттерна «Прототип». VI

// Где-то в клиентском коде.

class Application is

field shapes: array of Shape

constructor Application() is

Circle circle = new Circle()

circle.X = 10

circle.Y = 10

circle.radius = 20

shapes.add(circle)

Circle anotherCircle = circle.clone()

shapes.add(anotherCircle)

// anotherCircle будет содержать точную копию circle.

Rectangle rectangle = new Rectangle()

rectangle.width = 10

rectangle.height = 20

shapes.add(rectangle)

method businessLogic() is

// Плюс Прототипа в том, что вы можете клонировать набор объектов, не зная их конкретные классы.

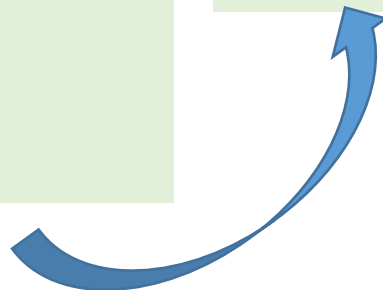
Array shapesCopy = new Array of Shapes.

// Например, мы не знаем, какие конкретно объекты находятся внутри массива shapes, так как он объявлен с типом Shape. Но благодаря полиморфизму, мы можем клонировать все объекты «вслепую». Будет выполнен метод clone того класса, которым является этот объект.

foreach (s in shapes) do

shapesCopy.add(s.clone())

// Переменная shapesCopy будет содержать точные копии элементов массива shapes.



Реализация паттерна «Прототип». VII

Шаги реализации

- Создайте интерфейс прототипов с единственным методом `clone`. Если у вас уже есть иерархия продуктов, метод клонирования можно объявить непосредственно в каждом из её классов.
- Добавьте в классы будущих прототипов альтернативный конструктор, принимающий в качестве аргумента объект текущего класса. Этот конструктор должен скопировать из поданного объекта значения всех полей, текущего класса, а затем передать выполнение родительскому конструктору, чтобы тот позаботился о полях, объявленных в суперклассе. Конструктор удобнее тем, что позволяет клонировать объект за один вызов.
- Метод клонирования обычно состоит всего из одной строки: вызова оператора `new` с конструктором прототипа. Все классы, поддерживающие клонирование, должны явно определить метод **`clone`**, чтобы использовать собственный класс с оператором `new`. В обратном случае результатом клонирования станет объект родительского класса. Опционально, создайте центральное хранилище прототипов. В нём удобно хранить вариации объектов, возможно, даже одного класса, но по-разному настроенных.
- Вы можете разместить это хранилище либо в новом фабричном классе, либо в фабричном методе базового класса прототипов. Такой фабричный метод должен на основании входящих аргументов искать в хранилище прототипов подходящий экземпляр, а затем вызывать его метод клонирования и возвращать полученный объект.
- Наконец, нужно избавиться от прямых вызовов конструкторов объектов, заменив их вызовами фабричного метода хранилища прототипов.

Реализация паттерна «Прототип». VIII

пример структуры
паттерна

Все классы—Прототипы имеют общий интерфейс. Поэтому вы можете копировать объекты, не обращая внимания на их конкретные типы и всегда быть уверенными, что получите точную копию. Клонирование совершается самим объектом-прототипом, что позволяет ему скопировать значения всех полей, даже приватных.

```
using std::string;
```

```
enum Type {  
    PROTOTYPE_1 = 0, PROTOTYPE_2  
};
```

```
class Prototype {  
protected:  
    string m_prot_name;  
    float m_field_;  
  
public:  
    Prototype() {}  
    Prototype(string prot_name)  
        : m_prot_name_(prot_name) {  
    }  
    virtual ~Prototype() {}  
    virtual Prototype *Clone() const = 0;  
    virtual void Method(float field) {  
        this->m_field_ = field;  
        std::cout << "Call Method from " << prototype_name_ << "  
with field : " << field << std::endl;  
    }  
};
```

Реализация паттерна «Прототип». IX

```
class ConcretePrototype1 : public Prototype {
private:
    float m_field1_;
public:
    ConcretePrototype1(string prot_name, float field) : Prototype(prot_name),
        m_field1_(field) {
    }
    //Отметим что метод Clone вернет указатель на новый объект класса ConcretePrototype1,
    //поэтому клиент, (который вызвал метод Clone) отвечает за освобождение памяти. По
    //возможности используйте умные указатели здесь.
    Prototype *Clone() const override {
        return new ConcretePrototype1(*this);
    }
};
```

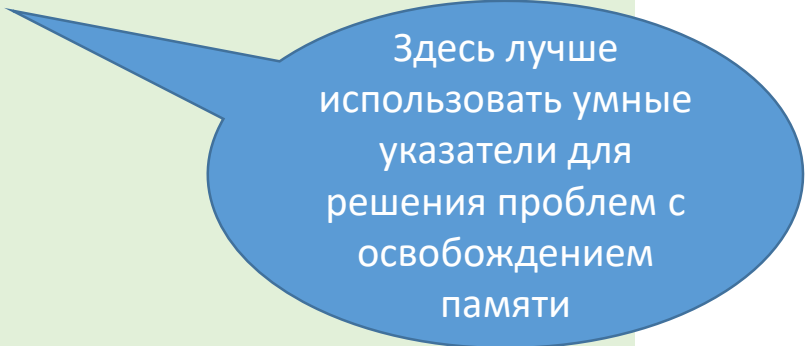
ConcretePrototype1 является подклассом классу Prototype и включает в себя метод Clone. В этом примере все переменные члены класса Prototype являются локальными и находятся в стеке. Если вы используете указатели необходимо позаботиться о реализации Конструктора копирования с реализацией глубокого копирования.

Реализация паттерна «Прототип». X

```
class ConcretePrototype2 : public Prototype {  
private:  
    float m_field2_;  
  
public:  
    ConcretePrototype2(string prot_name, float field)  
        : Prototype(prot_name), m_field2_(field) {  
    }  
    Prototype *Clone() const override {  
        return new ConcretePrototype2(*this);  
    }  
};
```

Реализация паттерна «Прототип». XI

```
class PrototypeFactory {  
private:  
    std::unordered_map<Type, Prototype *, std::hash<int>> m_prots_;  
public:  
    PrototypeFactory() {  
        m_prots_[Type::PROTOTYPE_1] = new ConcretePrototype1("PROTOTYPE_1 ", 50.f);  
        m_prots_[Type::PROTOTYPE_2] = new ConcretePrototype2("PROTOTYPE_2 ", 60.f);  
    }  
    ~PrototypeFactory() {  
        delete m_prots_[Type::PROTOTYPE_1];  
        delete m_prots_[Type::PROTOTYPE_2];  
    }  
    Prototype *CreatePrototype(Type type) {  
        return prototypes_[type]->Clone();  
    }  
};
```



Здесь лучше
использовать умные
указатели для
решения проблем с
освобождением
памяти

Реализация паттерна «Прототип». XII

```
void Client(PrototypeFactory &prototype_factory) {  
    std::cout << "Let's create a Prototype 1\n";  
  
    Prototype *prototype = prototype_factory.CreatePrototype(Type::PROTOTYPE_1);  
    prototype->Method(90);  
    delete prototype;  
    std::cout << "\n";  
    std::cout << "Let's create a Prototype 2 \n";  
    prototype = prototype_factory.CreatePrototype(Type::PROTOTYPE_2);  
    prototype->Method(10);  
    delete prototype;  
}
```

```
int main() {  
    PrototypeFactory *prototype_factory = new PrototypeFactory();  
    Client(*prototype_factory);  
    delete prototype_factory;  
    return 0;  
}
```

Результат выполнения

Let's create a Prototype 1
Call Method from PROTOTYPE_1 with field : 90

Let's create a Prototype 2
Call Method from PROTOTYPE_2 with field : 10

На сегодня всё!