ООП часть 2

SOLID. Шаблоны проектирования

ссылка на github репозиторий дисциплины

Ветров С. В.

Графические элементы

Тут текст пояснения, курсив (italic), но не просто наклонный

Содержание

1. Содержание
2. со
3. ссылками
4. на разделы

Ссылка на слайды по след теме
docs.google.com

Ссылка на слайды по предыдущей теме
docs.google.com

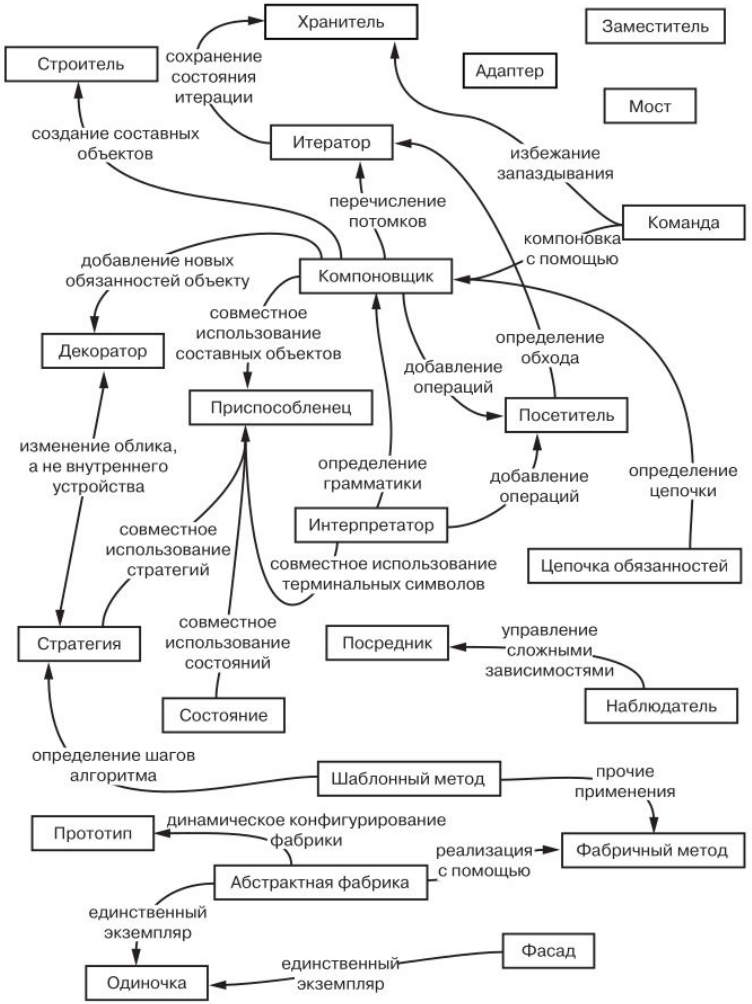
Содержание

1. Литература и источники
2. Отношения между классами (для повторения)
3. Важные [для понимания и применения ОО паттернов] концепции программирования
4. SOLID
5. Уровни проектирования
6. Паттерны проектирования
    a. Одиночка
    b. Стратегия
    c. Команда
    d. Фасад
    e. Адаптер
    f. Декоратор
    g. Наблюдатель
    h. Фабричный метод
    i. Абстрактная фабрика
    j. Итератор

Парадигмы программирования
SOLID
Организация модулей (компонентов)
Архитектура программных систем,
Чистая архитектура

Неформальное изложение

Банда четырёх (Gang of Four, GoF)



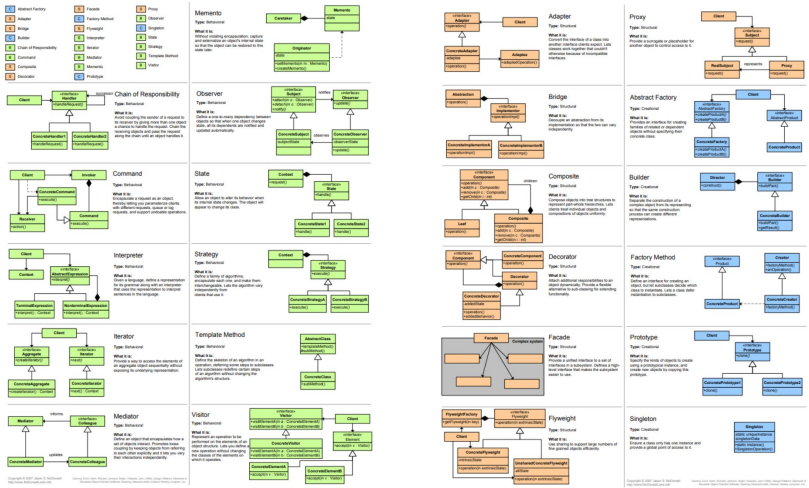
Много паттернов, более академическое изложение чем в предыдущей книге

Шпаргалка по паттернам проектирования:
mcdonaldland.info/files/designpatterns/designpatternscard.pdf

Перевод: habr.com/ru/articles/210288

– [часть 1](habr.com/ru/articles/210288) (поведенческие)

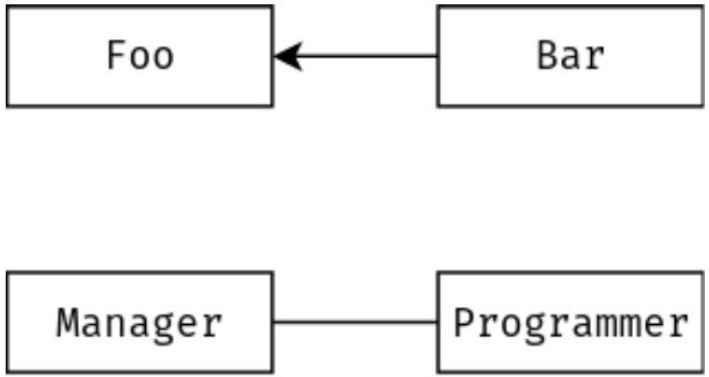
– [часть 2](habr.com/ru/articles/210288) (структурные и порождающие)



refactoring.guru/ru/design-patterns

Отношения между классами

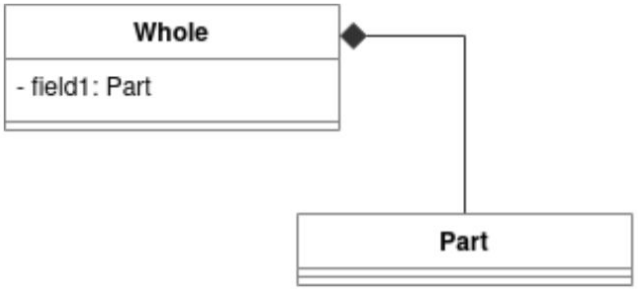
Ассоциация



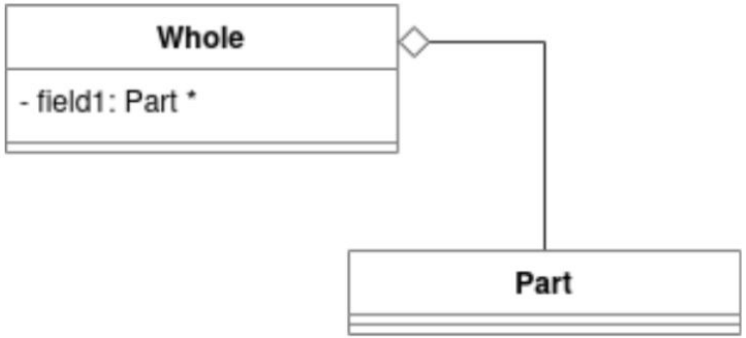

```
class Foo{
    // ...
};

class Bar{
    public:
        void baz( ) {
            Foo f;
            ...
        }
}
```

```
class Programmer{
    Manager *manager;
    // ...
};

class Manager{
    vector<Programmer*> programmers;
    // ...
}
```

Часть и целое: композиция \ агрегация

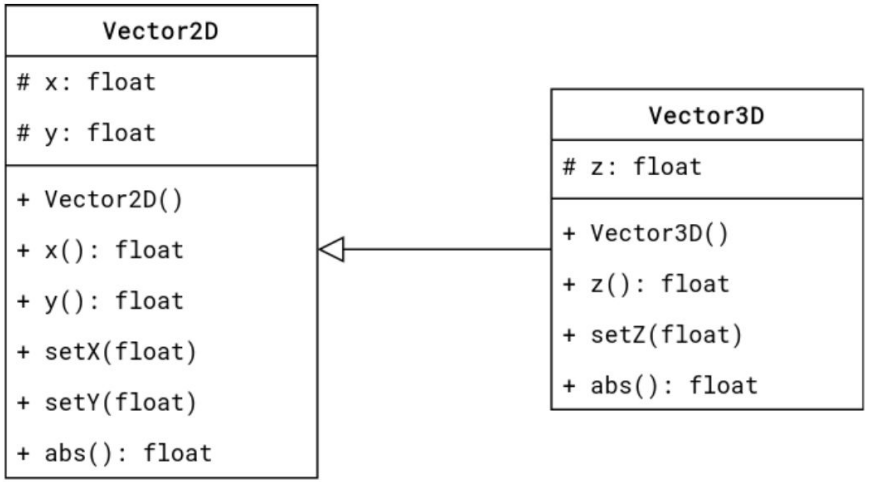


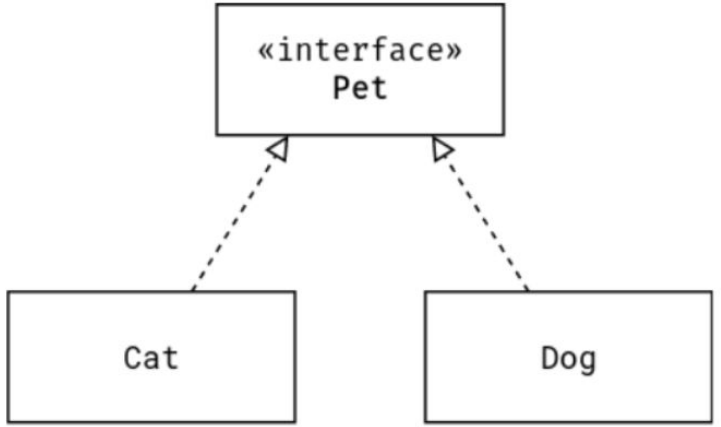

```
class Part{
    // ...
};

class Whole{
    Part filed1;
    // ...
};
```



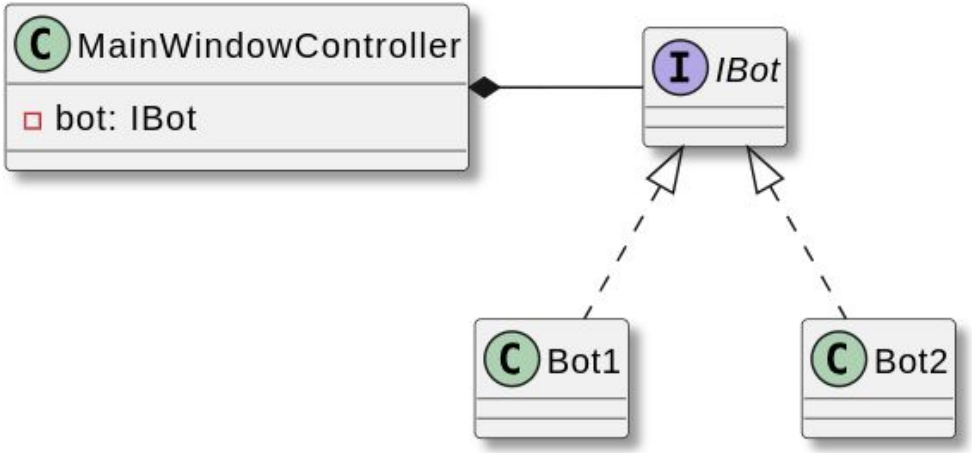

```
class Part{
    // ...
};

class Whole{
    Part *filed1;
    // ...

    public:
        Whole(){ filed1 = new Part();}

        ~Whole(){ delete filed1;}
};
```

Обобщение: наследование \ реализация





Пример: композиция или агрегация через Интерфейс




```
interface IBot{
    ... }
Bot1 implements IBot{
    ... }
Bot2 implements IBot{
    ... }

MainWindowController{
   private IBot bot;

   MainWindowController(){
       bot = new Bot1();
       // или
       bot = new Bot2(); }
}
```

Важные концепции программирования

Функциональный тип

- Анонимные функции в C++
- Стандартное описание функционального типа (указателя на функцию) в C++
- Переменные – указатели на функции, передача указателя на функцию в другую функцию
- Описание функционального типа в C++ используя std::functional
- Функция высшего порядка

github.com/VetrovSV/OOP/blob/master/CPP_OOP/220_func_pointers_etc.md

Функциональные типы в Java

- Callable
- Runnable
- Fucntion
- u gp.

github.com/ivtipm/BigDataLanguages/blob/main/Java/SAM.md

Ограничение через интерфейс

```
List<String> list = new LinkedList<>();

list.addFirst("1");
list.addLast("2");
list.removeFirst();
list.removeLast();

list.add("3");              // addLast
list.remove(0);

Queue<String> stack = new LinkedList<>();

stack.addFirst("0");
stack.add("1");     // addLast
stack.addLast("4");
stack.remove();     // removeFirst
stack.removeFirst();
stack.removeLast();
```

Статические поля

Статические поля

Избегайте создания статических полей, для решения проблемы передачи данных из одного объекта в другой.

Используйте для этого вызов обычных методов (сеттеров)

Статические методы

Рефлексия (reflection)

Рефлексия — способность программы исследовать и (иногда) изменять собственную структуру и поведение во время выполнения.

Например во время выполнения получать информацию о типах, о наборе полей и методов класса или даже изменять их.

Рефлексия (reflection) в C++

На момент 2025 года рефлексия в C++ представлена RTTY (Run-Time Type Information):
- dynamic_cast,
- typeid, std::type_info

```
struct Base { virtual ~Base(){} };

struct D1 : Base {};
struct D2 : Base {};

   srand( time(0) );
   Base *b;
   if ( rand() % 2 ) b = new D1();
   else              b = new D2();

   cout << "Тип объекта: " << typeid(*b).name();
```

Подробнее про оператор typeid
en.cppreference.com/w/cpp/language/typeid
также см. класс std::type_info; метаобъекты в фреймворке Qt

Рефлексия (reflection) в Java

```java
// Получение Class по имени
Class<?> cls = Class.forName("com.foo.MyClass");
// throws ClassNotFoundException;

// Создание экземпляра
Object obj = cls.getDeclaredConstructor().newInstance();
// throws NoSuchMethodException, InstantiationException, InvocationTargetException;;

// Доступ к полю
Field fld = cls.getDeclaredField("value");
// throws NoSuchFieldException, IllegalAccessException;

fld.setAccessible(true);
fld.set(obj, 42);
// Вызов метода
Method m = cls.getMethod("doSomething", String.class);
// throws NoSuchMethodException, IllegalAccessException;

Object result = m.invoke(obj, "hello");
```

*В Java почти все сущности являются объектами, за исключением примитивных типов. Классы тоже является объектами специального типа **Class**.*

У класса Class нет публичных конструкторов. Class – это generic тип. Методы Class предназначены для получения информации о классе (объекте типа Class). Например, можно узнать полное имя класса, какие у него аннотации, какие конструкторы и т.п.

DRY, KISS, YAGNI

DRY (Don’t Repeat Yourself)
Избегать дублирования знаний или кода, вынося общие части в единую абстракцию .

KISS (Keep It Simple, Stupid!)
Простота — ключ к надёжной и понятной архитектуре, избегать ненужных усложнений .

YAGNI (You Aren’t Gonna Need It)
Не реализовывать функционал заранее, пока нет реальной в нём необходимости .

SOLID

Что такое наследование?

Что такое виртуальный класс?

Что такое абстрактный класс? Интерфейс?

Какие требования налагает абстрактный класс на производные классы?

Можно ли в функцию принимающий тип X в параметре, передать переменную типа Y, если Y унаследован от X?

SOLID

SOLID – принципы объектно-ориентированного *проектирования*

– **S**ingle responsibility,

– **O**pen-closed,

– **L**iskov substitution,

– **I**nterface segregation

– **D**ependency inversion)

SOLID

S. Принцип единственной ответственности (The Single Responsibility Principle, SRP)

O. Принцип открытости/закрытости (The Open Closed Principle, OCP)

L. Принцип подстановки Барбары Лисков (The Liskov Substitution Principle, LSP)

I. Принцип разделения интерфейса (The Interface Segregation Principle, ISP)

D. Принцип инверсии зависимостей (The Dependency Inversion Principle, DIP)

S. Принцип единственной ответственности (The Single Responsibility Principle, SRP)

Каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс.
Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.

Принцип единственной ответственности

Применяется практически для любого масштаба: метод, класс, модуль.

Например, согласно этому принципу не стоит помещать бизнес-логику в класс окна приложения.

Принцип единственной ответственности

Несоблюдение принципа приводит к созданию *божественных объектов*.

Объект-бог (God object) антипаттерн объектно-ориентированного программирования, описывающий объект, который хранит в себе слишком много или делает слишком много

Этот швейцарский армейский нож не соблюдает принцип единственной ответственности

Принцип единственной ответственности

Буквальное и неразумное следование приводит – к увеличению числа классов и усложнению приложения.

Что нарушает принцип KISS

Пример

```
class Person {
    public name : string;
    public surname : string;
    public email : string;

    constructor(name : string, surname : string, email : string){
        this.surname = surname;
        this.name = name;
        if(this.validateEmail(email))
            this.email = email;
        else
            throw new Error("Invalid email!");
     }

     validateEmail(email : string) {
          var re =
     /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.([a-z]{2,6}(?:\.[a-z]{2})?$$)/i;
     return re.test(email); }

greet() {
     alert("Hi!");    }
```

Класс Person отвечает ещё и за проверку корректности адреса электронной почты. То есть выполняет несвойственную для себя задачу.

Для электронной почты должен быть создан отдельный класс

```
    class Email {
    public email : string;
    constructor(email : string){
        if(this.validateEmail(email)) { //...
        }
        else { throw new Error("Invalid email!"); }
    }
    validateEmail(email : string) {
        var re = /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.
        return re.test(email);
    }}

class Person {
    public name : string;
    public surname : string;
    public email : Email;
    // ...
    greet() {
        alert("Hi!");
    }}
```

Принцип открытости \ закрытости

Принцип подстановки Барбары Лисков

Принцип разделения интерфейсов

Принцип инверсии зависимостей

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций

Принцип инверсии зависимостей

Инверсия зависимости используется в фреймворках

Фреймворк управляет кодом программиста, а не программист управляет фреймворком

Фреимворк здесь – модуль верхнего уровня, код программиста – модуль нижнего уровня

Изменяя код нижнего уровня не приходится вносить изменения в фреймворк

Абстракция — это модель некоего объекта или явления реального мира, откидывающая незначительные детали, не играющие существенной роли в данном приближении

Уровень абстракции — это ступень приближения.

Пример. Проблема 1.

```
class Worker {
    public Worker(String name) { ... }
    public void work() {
        // ...working...    }
}

class Manager {
    Worker worker;

    Manager(){
        worker = new Worker("Robert");  }

    public void manage() {
        worker.work();  }
}
```

Проблема?
Классы Worker нарушают принцип инверсии зависимостей:
Manager зависит от Worker.

Если у класса Worker поменяется конструктор, например будет принимать не 1, а 2 параметра. То придётся изменять и класс Manager.

Это зависимость модуля (класса) верхнего уровня (Manager) от модуля нижнего уровня (Worker)

Пример. Решение проблемы зависимости

```
class Worker {
   public Worker(String name) { ... }
   public void work() {
       // ...working…   }
}

class Manager {
   Worker worker;

   Manager(Worker w){
       worker = w; }
   // или
   public void setWorker(Worker w){
       worker = w; }

   public void manage() {
       worker.work();}
}
```

Теперь можно создать экземпляр класса Worker вовне класса Manager. Значит Manager больше не зависит от Worker:

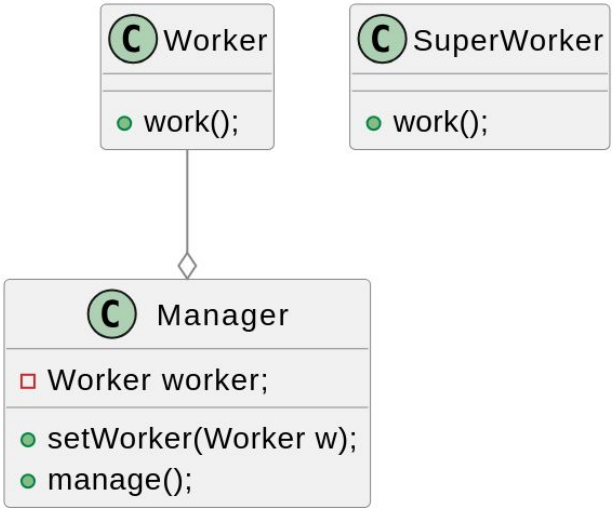
```
Worker worker = new Worker("Sam");
Manager manager = new Manager();

manager.setWorker ( worker );
```

Пример. Проблема 2

```
class Worker {
    public Worker(String name) { ... }
    public void work() {
        // ...working...  }
}

class Manager {
    Worker worker;

    Manager(Worker w){
        worker = w; }
    // или
    public void setWorker(Worker w){
        worker = w; }

    public void manage() {
        worker.work();}
}
```

```
class SuperWorker {
    public Worker(String name) { ... }
    public void work() {
        // ...super working... }
}
```

С классом SuperWorker класс Manager не работает... Вторая часть принципа не выполняется: модули не зависят от абстракций



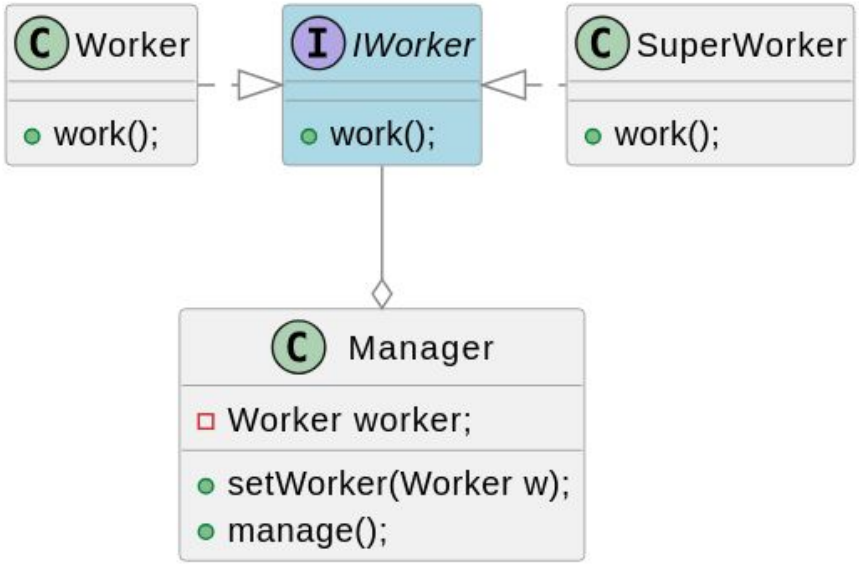
Пример. Решение

```
interface IWorker {
    public void work(); }
```

```
class Worker implements IWorker {
    public Worker(String name) { ... }
    public void work() {
        // ...working...   }
}
```

```
class SuperWorker implements IWorker {
    public Worker(String name) { ... }
    public void work() {
        // ...super working... }
}
```

```
class Manager {
    IWorker worker;

    Manager(IWorker  w){
        worker = w; }
    // или
    public void setWorker(IWorker  w){
        worker = w; }

    public void manage() {
        worker.work();}
}
```



Dependency injection (DI)

Такой подход называется Dependency injection (DI)

Dependency injection – процесс предоставления внешней зависимости программному компоненту.

Внешняя зависимость в примере – класс Worker

Компонент, добавляют дают внешнюю зависимость – класс Manager

Уровень абстракции здесь представлен интерфейсом IWorker

Шаблоны и архитектура

Иерархия программных структур

Системная архитектура
Охватывает организацию всей ИТ-инфраструктуры, взаимодействие между системами, серверами, клиентами и прочими компонентами (программами).
Пример — клиент-серверная архитектура. Оболочка для чата (клиент) отправляет запрос на сервер (например, ollama).

Архитектура приложения
Как структурировать код внутри одного приложения. Фокусируется на разделении приложения на модули или слои.
Пример: MVC, MVP, MVVM, Clean Architecture и подобных.

Паттерны (шаблоны) проектирования (design pattern)
Набор проверенных решений для типовых задач разработки (наблюдатель, фабрика, декоратор, и т.д.). Они работают на более детальном, локальном уровне (уровень классов) и помогают решать конкретные проблемы при реализации возможностей программ.

Паттерны проектирования (design patterns)

Шаблон проектирования

Шаблон проектирования или паттерн (design pattern) — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста

Далее будут рассмотрены паттерны проектирования, которые говорят как организовывать классы между собой для решения типовых задач.

Алгоритм — паттерн

Алгоритмы по своей сути также являются шаблонами, но не проектирования, а вычисления, так как решают вычислительные задачи

Антипаттерн (anti-pattern) это распространённый подход к решению класса часто встречающихся проблем, являющийся неэффективным, рискованным или непродуктивным

Использование паттернов

Достоинства

+ снижении сложности разработки за счёт готовых абстракций для решения целого класса проблем
+ унификация деталей решений: модулей, элементов проекта

Недостатки

- слепое следование выбранному шаблону может привести к усложнению программы
- необоснованное применение применение шаблона

BEGINNER MIND

"I need a pattern for Hello World."

Зачем изучать паттерны?

Они применяются в кодовой базе существующих программных продуктов, фреимворках и библиотеках.

Чтобы их эффективно использовать, без “костылей”, нужно понимать как они работают и почему устроены именно так.

Решать типовые проблемы с помощью типовых решений, понятных многим программистам.

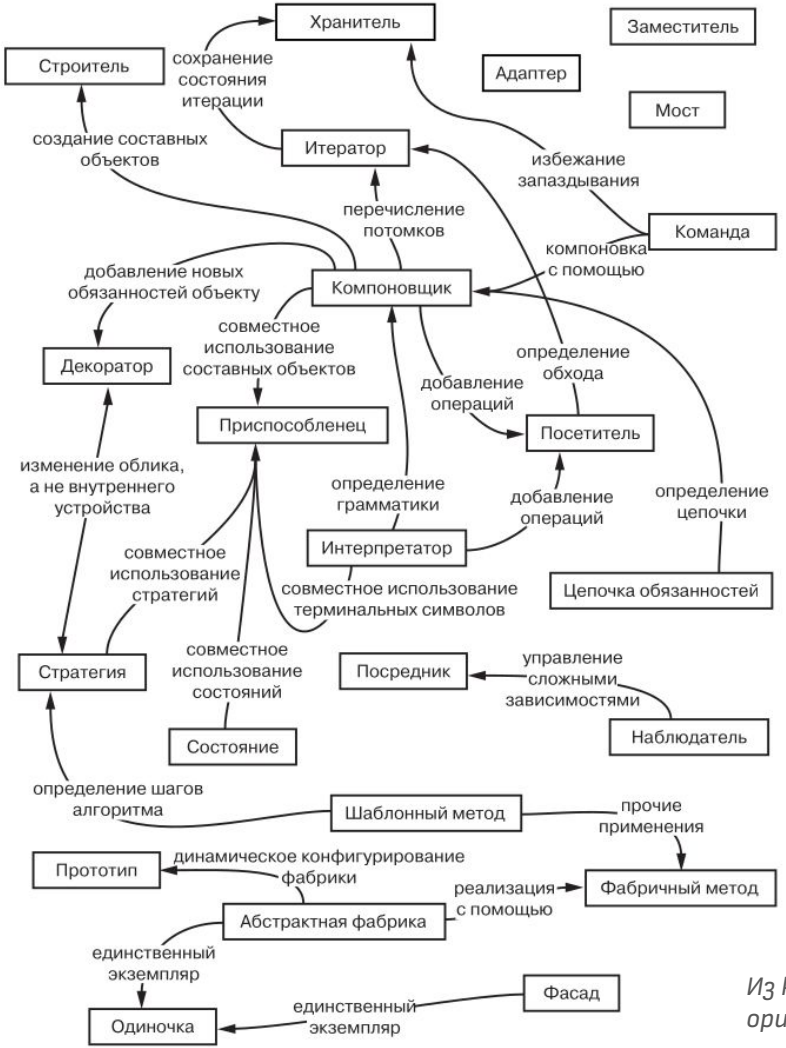
Некоторые паттерны

Виды паттернов

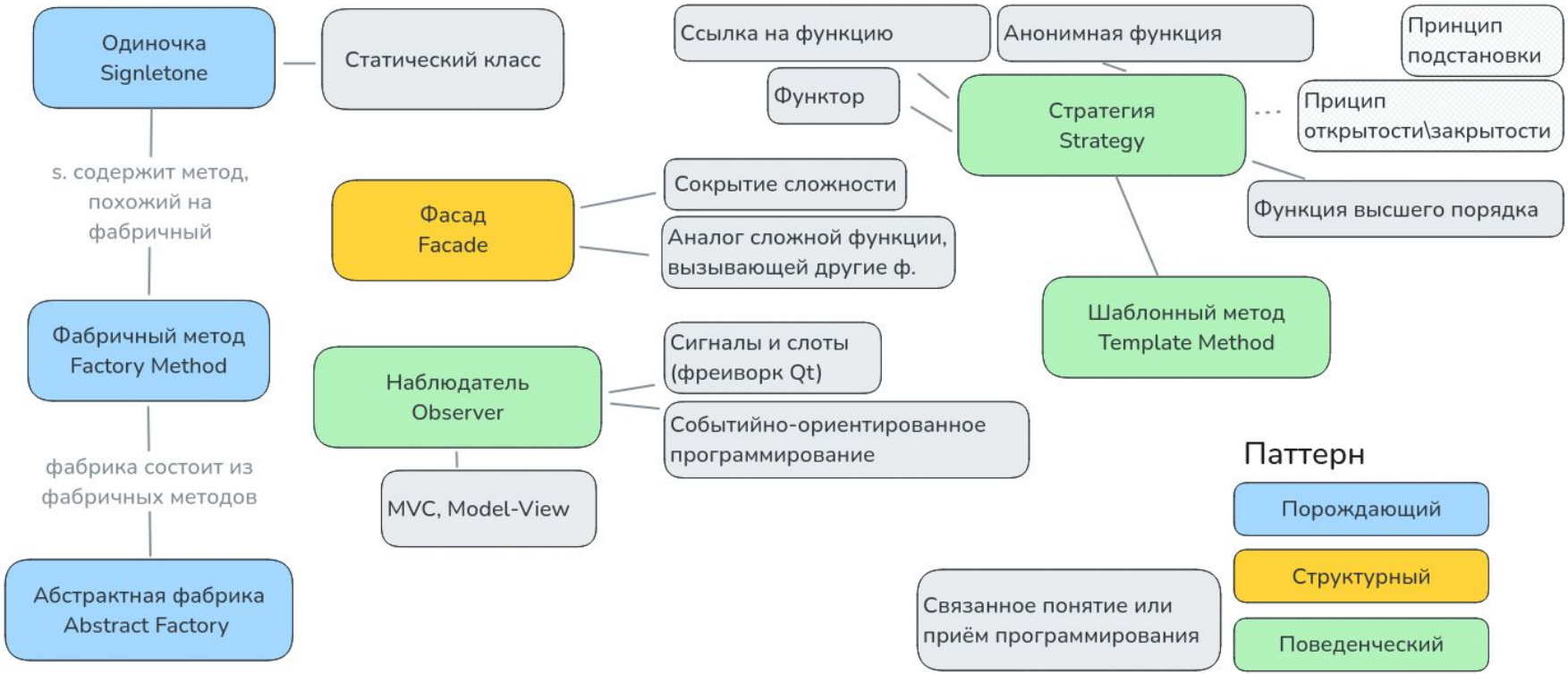
- B — поведенческие (behavioral);
- C — порождающие (creational);
- S — структурные (structural).

Шпаргалка по паттернам проектирования: mcdonaldland.info/files/designpatterns/designpatternscard.pdf

Перевод: habr.com/ru/articles/210288
– часть 1 (поведенческие)
– часть 2 (структурные и порождающие)



Из книги Паттерны объектно-ориентированного проектирования


Одиночка
Signletone
Статический класс
s. содержит метод, похожий на фабричный
Фабричный метод
Factory Method
фабрика состоит из фабричных методов
Абстрактная фабрика
Abstract Factory
Ссылка на функцию
Анонимная функция
Принцип подстановки
Функтор
Стратегия
Strategy
Прицип открытости\закрытости
Функция высшего порядка
Фасад
Facade
Сокрытие сложности
Аналог сложной функции, вызывающей другие ф.
Шаблонный метод
Template Method
Наблюдатель
Observer
Сигналы и слоты (фреиворк Qt)
Событийно-ориентированное программирование
MVC, Model-View
Паттерн
Порождающий
Структурный
Поведенческий
Связанное понятие или приём программирования

Одиночка
Singleton

Одиночка (Singleton)

Порождающий шаблон проектирования

Зачем?

Нужно во всей программы иметь *один и только один* экземпляр класса

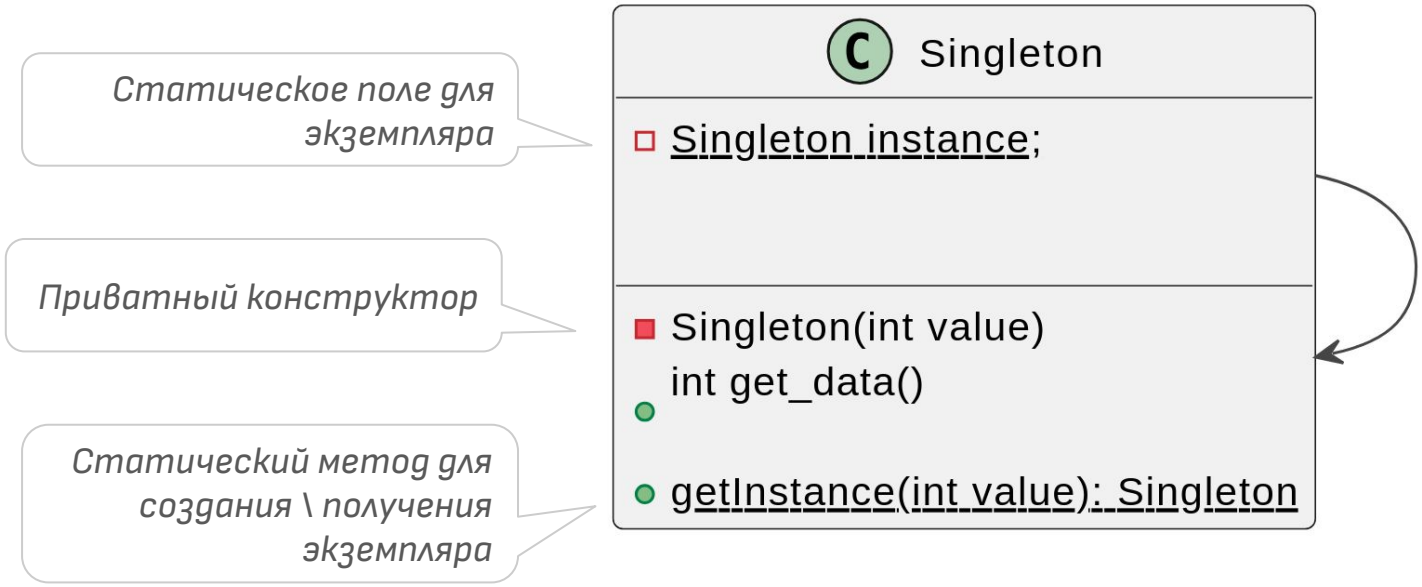
Экземпляр должен быть доступен *отовсюду*

Примеры
C++ — `std::cout`, `std::cerr`, `std::cin` — объекты для вывода и ввода данных в консоль
Могут быть использованы в любой программе

Java — `java.lang.Runtime#getRuntime()`

Одиночка также полезен для создания объектов для логирования, подключений к СУБД, счётчиков и т.п.

Одиночка (Singleton)



Пример реализации: Java

```
public final class Singleton {
    // хранит единственный экземпляр
    private static Singleton instance;

    public int some_data; // данные, для примера

    // Приватный конструктор:
    // его можно вызвать только внутри класса
    private Singleton(int value) {
        this.some_data = value;     }

    /** Возвращает экземпляр класса */
    public static Singleton getInstance(int value) {
        // если экземпляр ещё не создан, то создаём
        if (instance == null)   instance = new Singleton(value);
        return instance;
    }
}
```

```
// Ошибка: конструктор приватный!
Singleton s = new Singleton();

Singleton s1 =
      Singleton.getInstance(42);

// ...

Singleton s2 =
      Singleton.getInstance(42);

// Объекты s1 и s2 идентичны

System.out.println( s1.get_data() );
System.out.println( s2.get_data() );
System.out.println(
   Singleton.getInstance(42).get_data()
                  );
```

Одиночка (Singleton)

+ Простой паттерн
+ Всегда один экземпляр класса
+ Доступен отовсюду

- Могут быть проблемы типа неопределённости параллелизма
- Может быть удалён сборщиком мусора
- Своего рода глобальная переменная, к которой есть доступ отовсюду

Рекомендуется применять паттерн, если нет лучших способов решить задачу

Одиночка (Singleton) и статический класс

Статический класс

+ Ещё проще создавать чем Одиночку
+ Отлично подходит для констант

- Может фактически содержать глобальные переменные, к которым есть доступ отовсюду

- Для того, чтобы представлять состояние Одиночка подходит лучше чем статический класс

Стратегия
Strategy

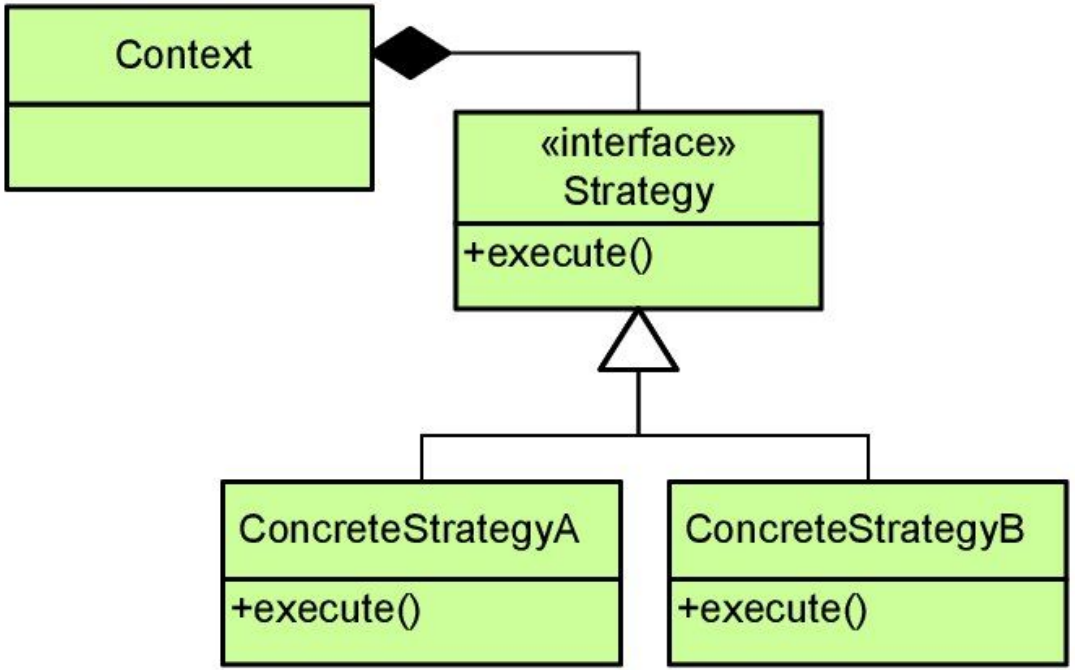
Стратегия (Strategy)

Стратегия
Strategy

Тип: Поведенческий

Что это:

Определяет группу алгоритмов, инкапсулирует их и делает взаимо-заменяемыми. Позволяет изменять алгоритм независимо от клиентов, его использующих.



wikipedia refactoring.guru/ru/design-patterns/strategy

Стратегия (Strategy)

+ Возможно замены алгоритмов во время работы программы
+ 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 состоит разница между стратегиями, чтобы выбрать подходящую.

- В простых случаях можно заменить на передачу функции \ анонимной функции в метод класса Context или в функцию.

Стратегия (Strategy)

- В простых случаях можно заменить на передачу функции \ анонимной функции в метод класса Context или в функцию.
- Стоит использовать Стратегию, если: алгоритм (стратегия) должна хранить состояние, логировать действия и т.п.

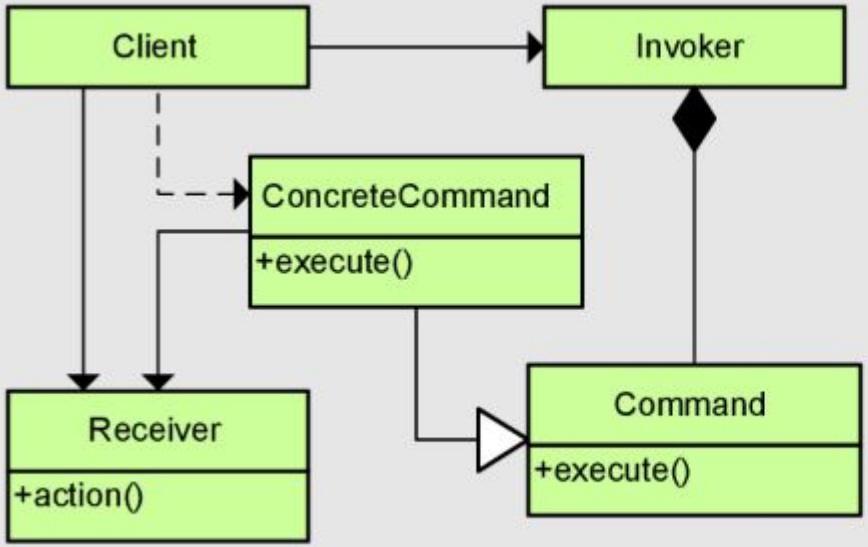
std::sort и компаратор

```
std::array<int, 10> s{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

std::sort(s.begin(), s.end());
print("sorted with the default operator<");

std::sort(s.begin(), s.end(), std::greater<int>());
print("sorted with the standard library compare function object");

struct {
    bool operator()(int a, int b) const { return a < b; }
} customLess;

std::sort(s.begin(), s.end(), customLess);
print("sorted with a custom function object");
```

Команда
Command

Команда (Command)

Отправитель. Вызывает действие (execute)



Команда
Command

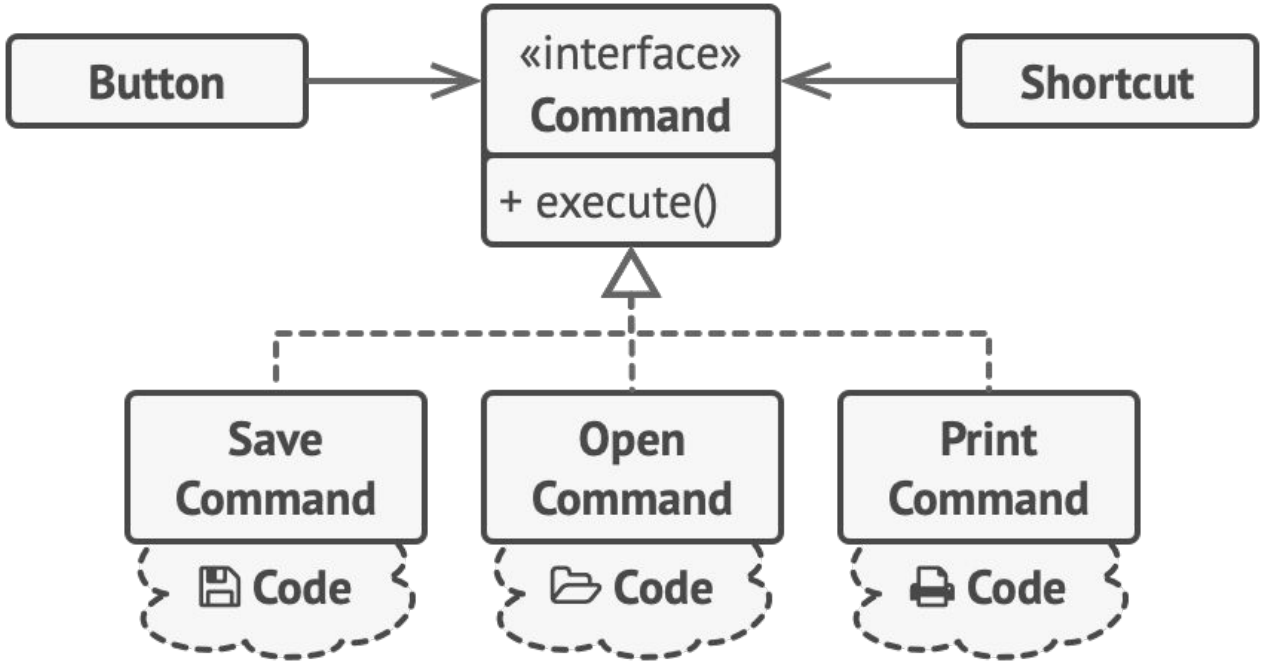
Тип: Поведенческий

Что это:

Инкапсулирует запрос в виде объекта, позволяя передавать их клиентам в качестве параметров, ставить в очередь, логировать а также поддерживает отмену операций.

Получатель. Содержит бизнес-логику. Получает команды, фактически выполняет действие. Иногда имеет смысл объединить конкретную команду и получателя

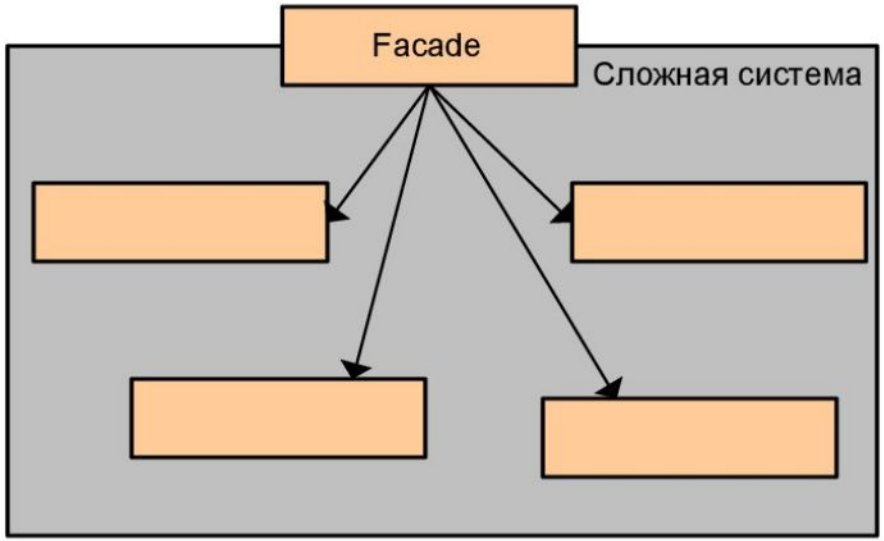
refactoring.guru



Фасад (Facade)

Фасад (Facade)

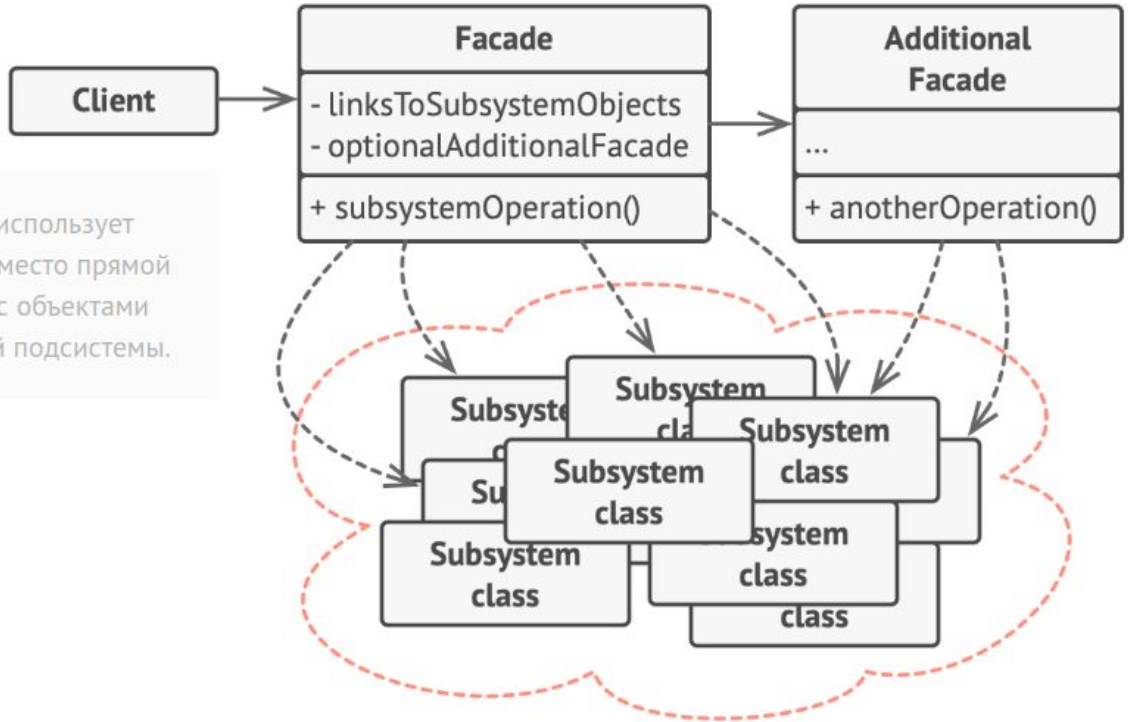
Фасад (Facade) — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.



Пример

1 **Фасад** предоставляет быстрый доступ к определённой функциональности подсистемы. Он «знает», каким классам нужно переадресовать запрос, и какие данные для этого нужны.

2 **Дополнительный фасад** можно ввести, чтобы не «захламлять» единственный фасад разнородной функциональностью. Он может использоваться как клиентом, так и другими фасадами.



4 **Клиент** использует фасад вместо прямой работы с объектами сложной подсистемы.

3 **Сложная подсистема** состоит из множества разнообразных классов. Для того, чтобы заставить их что-то делать, нужно знать подробности устройства подсистемы, порядок инициализации объектов и так далее.

Классы подсистемы не знают о существовании фасада и работают друг с другом напрямую.

Изображение с сайта refactoring.guru/ru/design-patterns/facade

Фасад (Facade)

+ Упрощает взаимодействие с набором классов

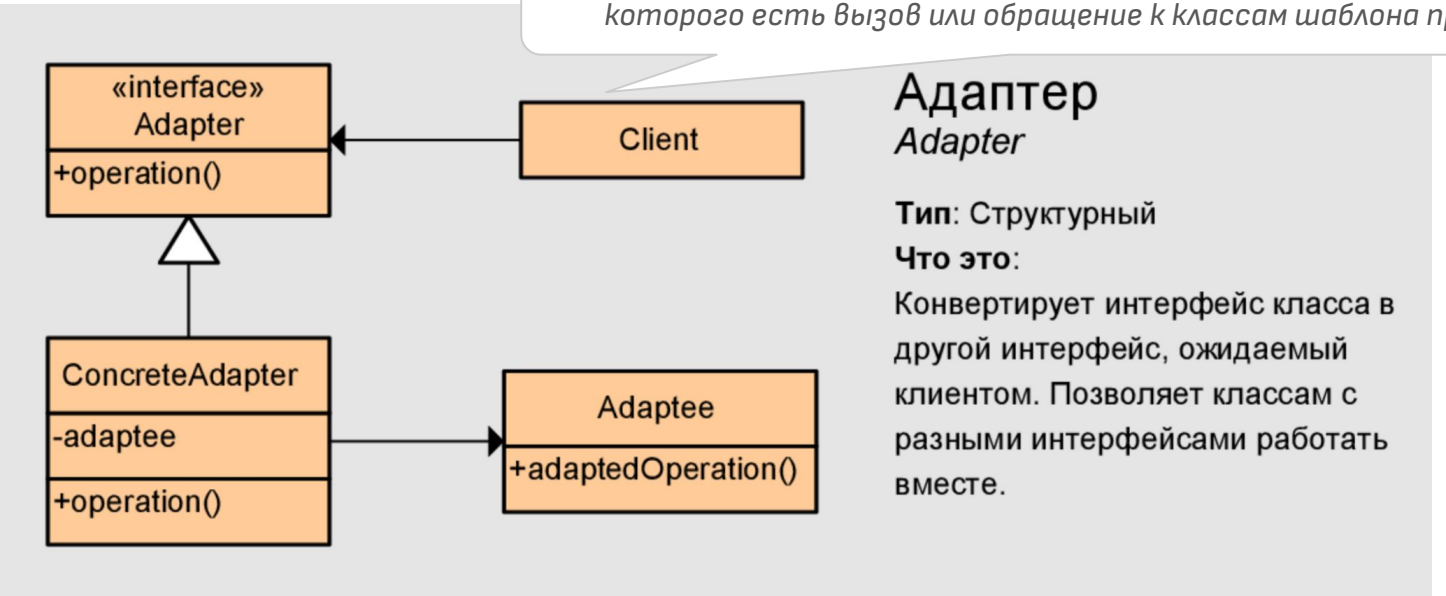
- Может стать божественным объектом

Адаптер (Adapter)

Адаптер (Adapter)

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Под клиентским классом обычно понимается произвольное место в коде, из которого есть вызов или обращение к классам шаблона проектирования



refactoring.guru

ru.wikipedia.org

Адаптер (Adapter). Пример. Несовместимые классы

```
/// Старый сенсор
public class FahrenheitSensor {

    // Принцип закрытости рекомендует
не менять класс

    /// Возвращает температуру в °F
    public double
getTemperatureInFahrenheit() {

        // Например, запрос к железу
или сторонней библиотеке ...

        return 75.0;

    }

}
```

```
/// Новый интерфейс, который требуется в
приложении
public interface CelsiusSensor {


    /// Возвращает температуру в °C
    double getTemperatureInCelsius();

}
```

Адаптер (Adapter). Пример.

```java
// Адаптер:
// 1. реализует нужный интерфейс
// 2. внутри использует FahrenheitSensor
public class TemperatureAdapter implements CelsiusSensor {

    private final FahrenheitSensor legacySensor;

    public TemperatureAdapter(FahrenheitSensor legacySensor) {
        this.legacySensor = legacySensor;
    }

    @Override
    public double getTemperatureInCelsius() {
        double f = legacySensor.getTemperatureInFahrenheit();
        // Формула перевода: (F - 32) × 5/9
        return (f - 32) * 5.0 / 9.0;
    }
}
```

Адаптер (Adapter). Пример. Использование в клиентском коде

```
public class Main {

    public static void main(String[] args) {

     FahrenheitSensor oldSensor = new FahrenheitSensor();

     // «Оборачиваем» его в адаптер
     CelsiusSensor sensor = new TemperatureAdapter(oldSensor);



     System.out.printf(
          "Температура: %.2f °C%n", sensor.getTemperatureInCelsius()  );

    }

}
```

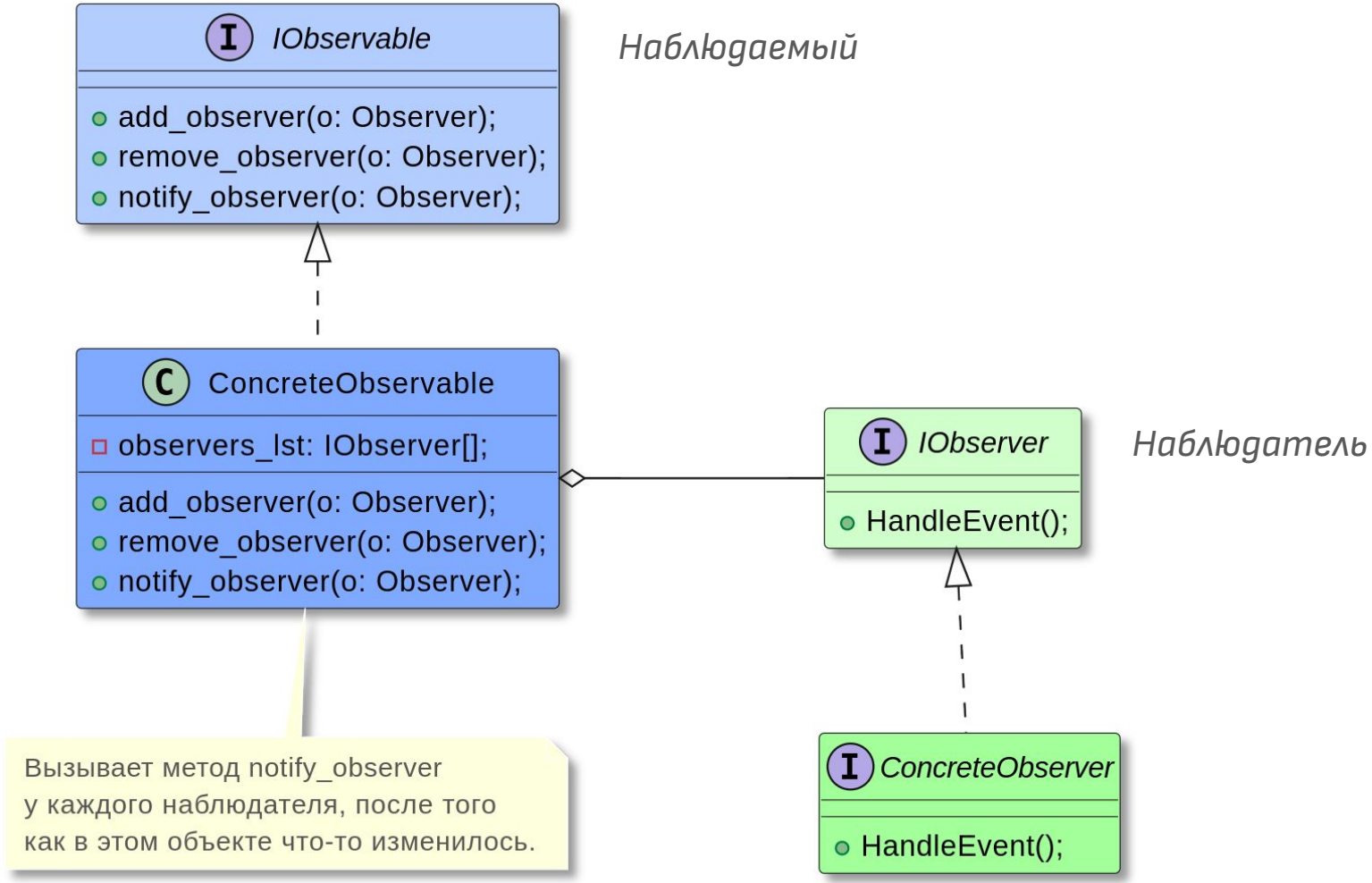
Паттерн Наблюдатель
Observer

Наблюдатель (Observer)

поведенческий шаблон проектирования. Также известен как “подчинённые” (Dependents).

Создает механизм у класса, который позволяет получать экземпляру объекта этого класса оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними

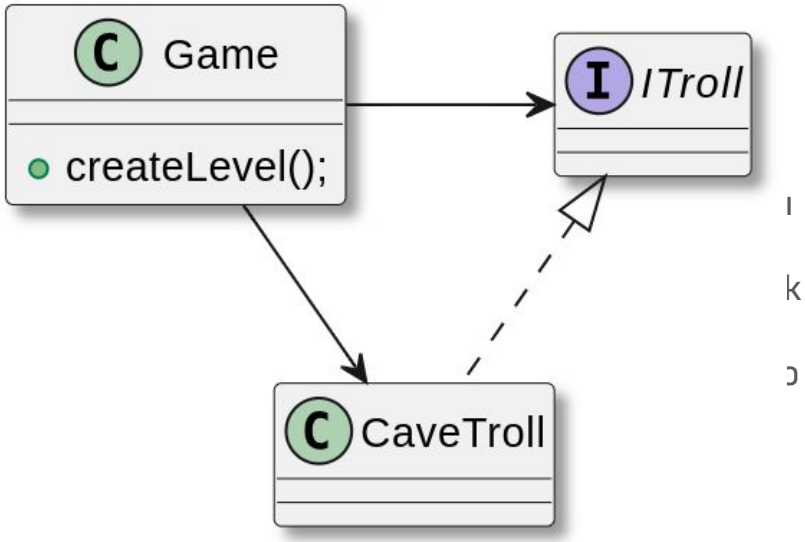
Примеры на разных языках программирования: wikipedia


IObservable
add_observer(o: Observer);
remove_observer(o: Observer);
notify_observer(o: Observer);
Наблюдаемый
ConcreteObservable
observers_lst: IObserver[];
add_observer(o: Observer);
remove_observer(o: Observer);
notify_observer(o: Observer);
IObserver
HandleEvent();
Наблюдатель
Вызывает метод notify_observer
у каждого наблюдателя, после того
как в этом объекте что-то изменилось.
ConcreteObserver
HandleEvent();

Фабричный метод
Factory Method

Проблема — сильная зависимость классов

```
interface ITroll{
    // ...
}

class CaveTroll implements ITroll{
    // ...
}

class Game{
    // ...

 void CreateLevel(){

    List<ITroll> enemies = new LinkedList<>();
    enemies.add( new CaveTroll());
    // тут могут быть и другие враги…
    }
}
```

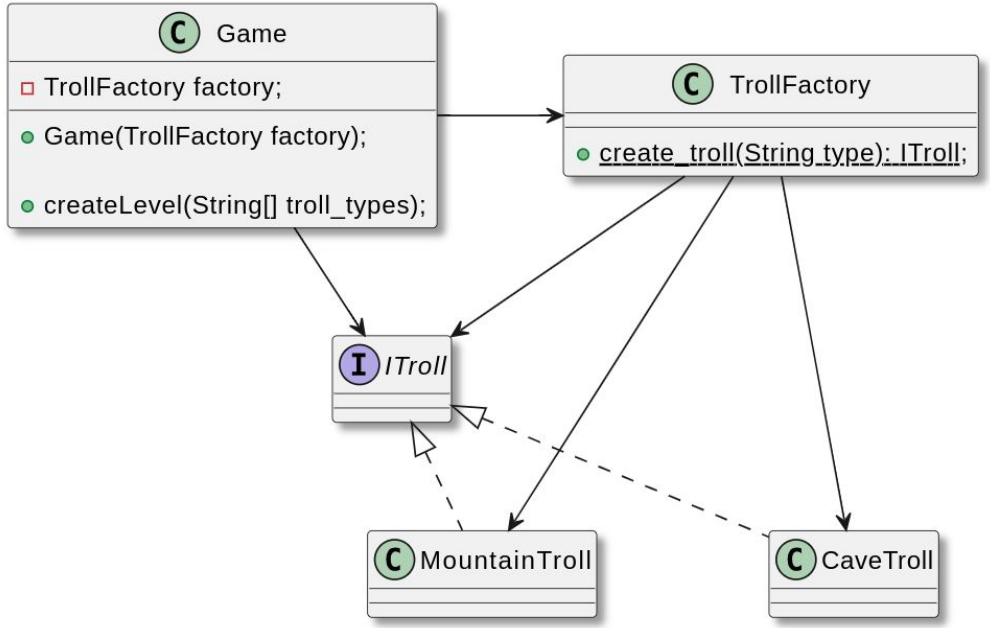


Как решить проблему?

1. Передавать в Game уже готовый список врагов
2. Передавать в Game специальный класс, который будет создавать врагов

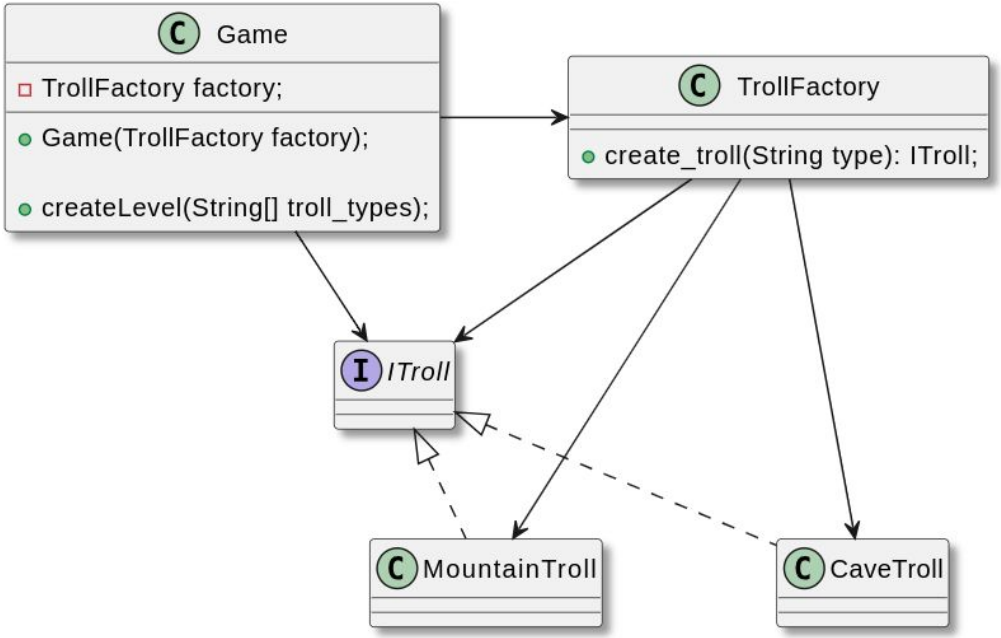
Решение – Фабричный метод. Factory Method

```
interface ITroll{ }

class CaveTroll implements ITroll{ }

class MountainTroll implements ITroll { }

class TrollFactory{
   public static ITroll create_troll(String type){
     // ...
     if ( ... )      return new CaveTroll();
     else if ( ... ) return new MountainTroll(); }
}

class Game{
   Game(){}

   void create_level(String[] troll_types){
       List<ITroll> enemies = new LinkedList<>();
       for( String type : troll_types )
           enemies.add( factory.create_troll( type ) );
   }
}
```



Решение – Фабричный метод. Factory Method

```
interface ITroll{ }

class CaveTroll implements ITroll{ }

class MountainTroll implements ITroll { }

class TrollFactory{
   public ITroll create_troll(String type){
     // ...
     if ( ... )      return new CaveTroll();
     else if ( ... ) return new MountainTroll(); }
}

class Game{
   TrollFactory factory;
   Game(TrollFactory factory_){
       this.factory = factory_; }

   void create_level(String[] troll_types){
     List<ITroll> enemies = new LinkedList<>();
     for( String type : troll_types )
       enemies.add( factory.create_troll( type ) );
   }
}
```



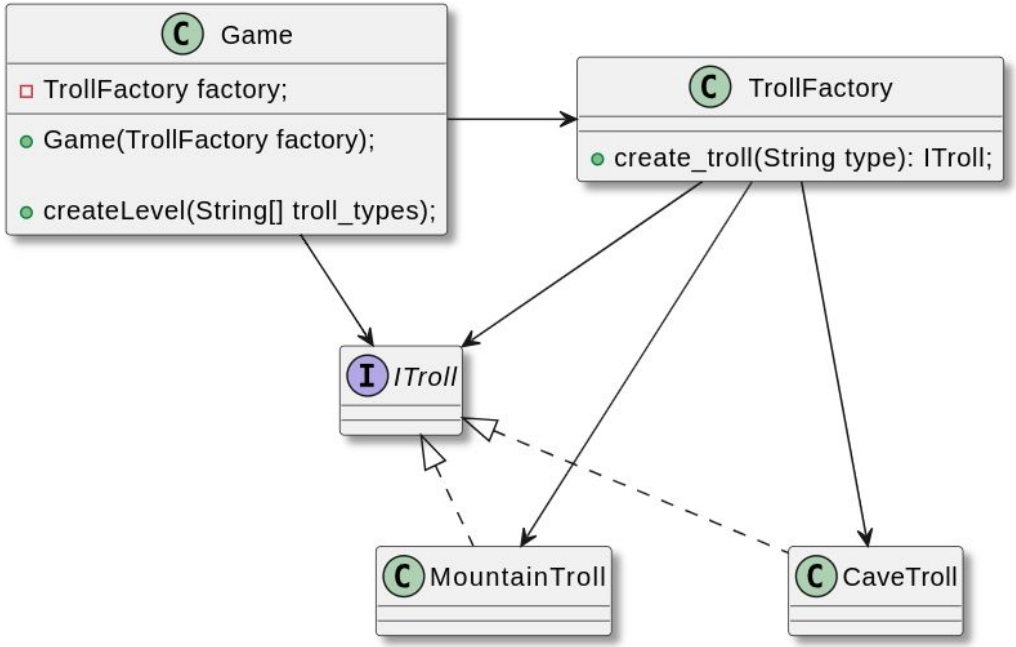
Фабричный метод. Factory Method

Код TrollFactory (метод create_troll) можно менять. При этом изменения в Game не потребуются.
Можно даже добавить новые подклассы ITroll.

Можно создавать наследников от TrollFactory, которые будут создавать другие подклассы ITroll

Реализуется принцип открытости/закрытости.

Недостатки: может получиться большое количество фабричный классов.



Фабричный метод. Factory Method

refactoring.guru/ru/design-patterns/factory-method

ru.wikipedia.org

Фабричный метод. Factory Method

Примеры из библиотек Java:

Фабричный метод
```
public static Calendar getInstance()
```

возвращает объект конкретного подкласса (обычно GregorianCalendar, но может быть и BuddhistCalendar, JapaneseImperialCalendar и т.д.), исходя из текущей зоны и локали. Клиент работает с Calendar без знания о конкретном классе-реализации

```
Calendar cal = Calendar.getInstance();
```

https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html?utm_source=chatgpt.com

Абстрактная фабрика
Abstract Factory

Итератор
Iterator

Итератор – поведенческий паттерн проектирвания

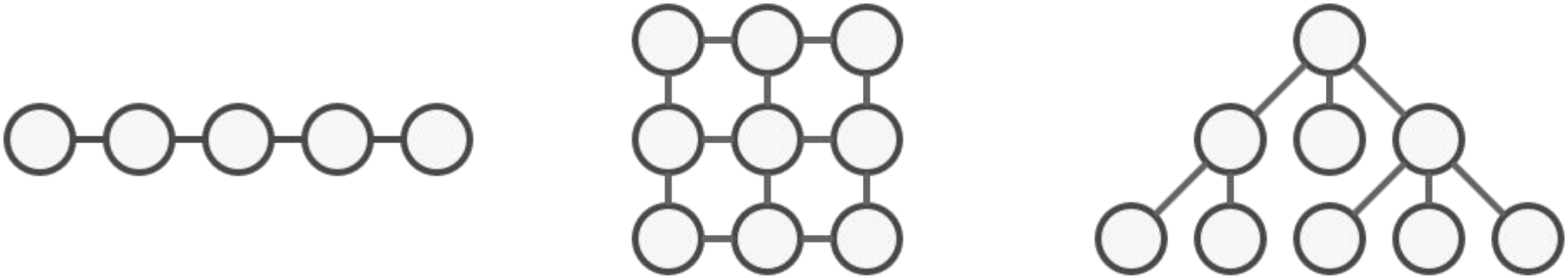
Итератор (iterator) — интерфейс, предоставляющий доступ к элементам коллекции (массива или контейнера) и навигацию (обычно перемещение вперед) без раскрытия внутреннего представления коллекции.

Итераторы позволяют перебрать все элементы коллекции (дерева, списка и т.п.). При этом инкапсулируя алгоритм обхода.

Итератор можно остановить в любой момент, что, например, трудно сделать при обходе дерева не редактируя код процедуры обхода.

Итераторы позволяют однажды прервав перебор элементов вернутся к нему с того же места. Что нельзя сделать в рекурсивных алгоритмах.

Итератор (iterator)

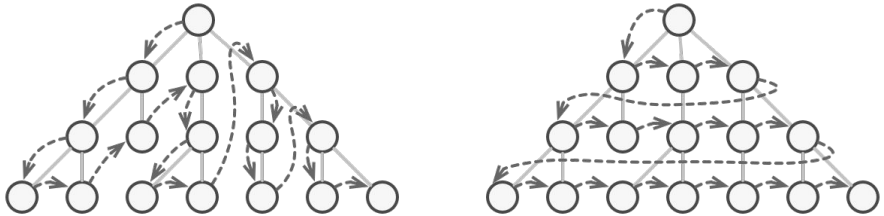


Обход всех элементов этих коллекций – это разные алгоритмы, которые стоит инкапсулировать в реализации одного и того же интерфейса.

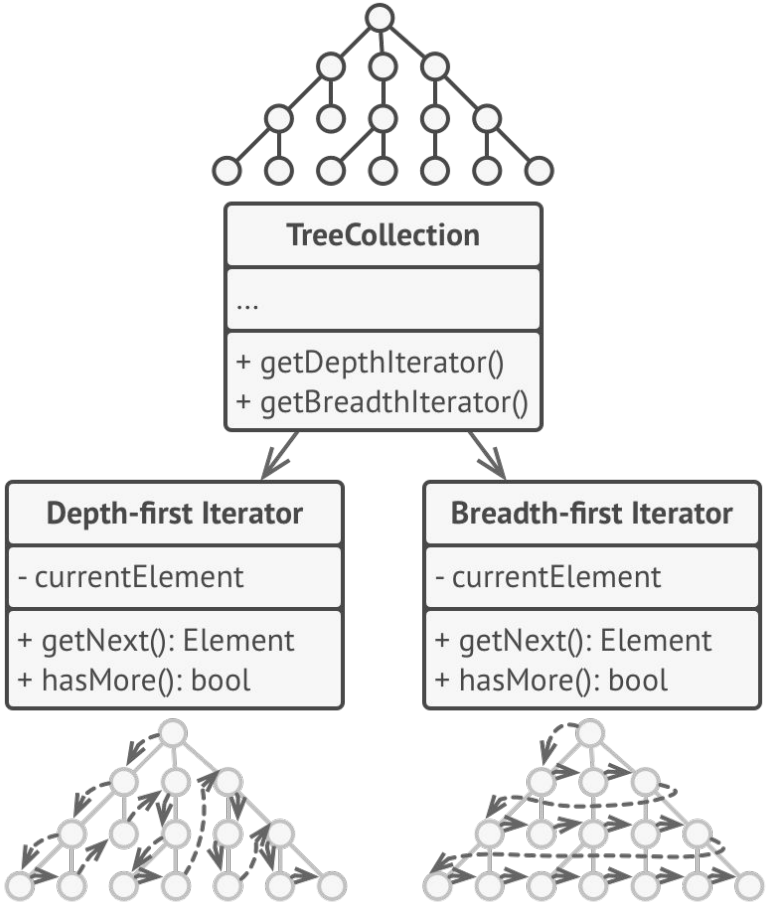
Который будет иметь типичные операции: выдать текущий элемент; перейти к следующему.

Изображение с сайта: refactoring.guru/ru/design-patterns/iterator

Итератор (iterator)

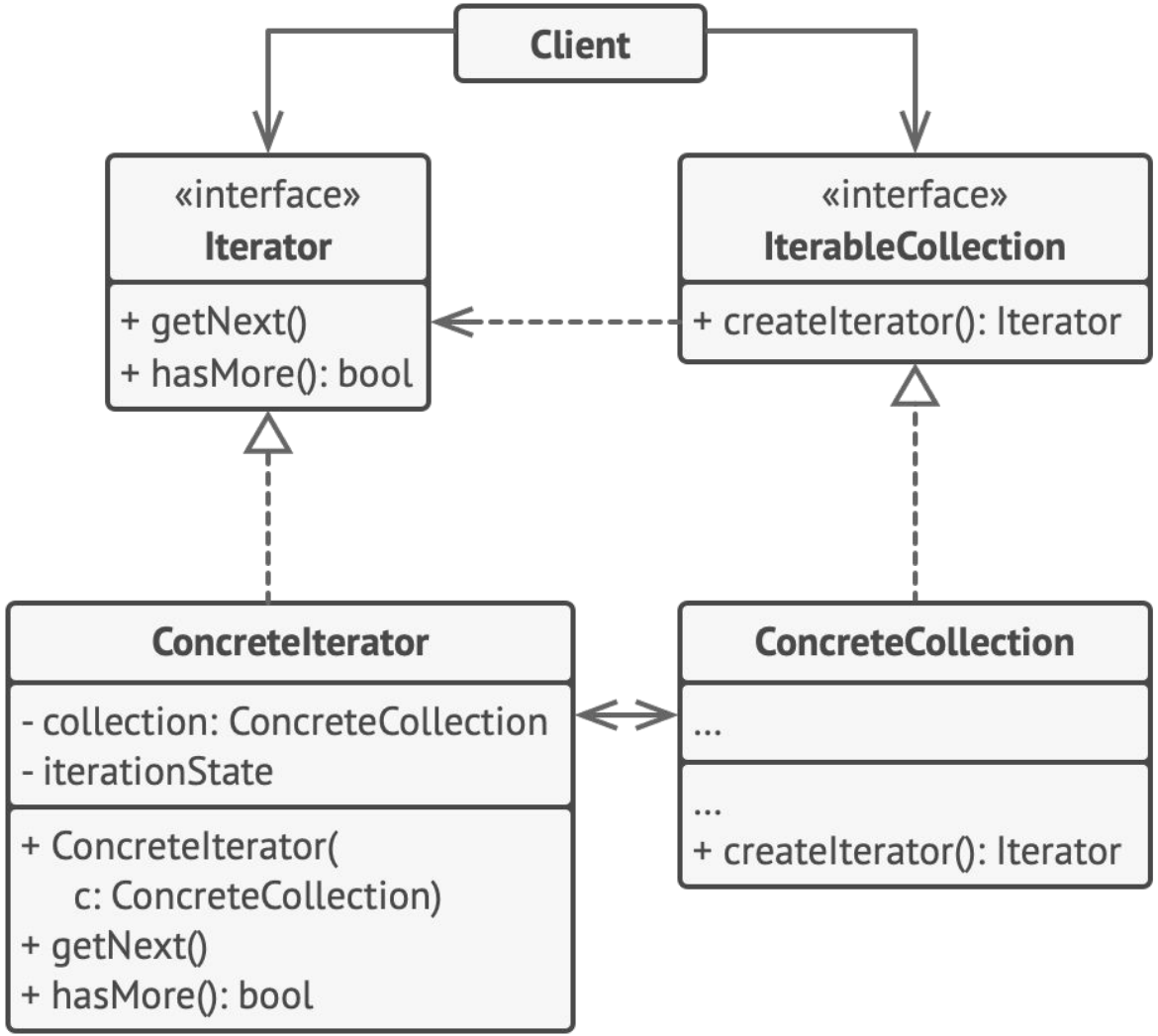


Даже для одной и той же коллекции
могут быть разные алгоритмы обхода



Изображения с сайта: refactoring.guru/ru/design-patterns/iterator

Итератор (iterator)



Подробности:

Слайды по САОД – 2. Итераторы

refactoring.guru/ru/design-patterns/iterator

Ссылки

- Принципы, определяющие связность компонентов
- GRASP (от англ. General Responsibility Assignment Software Patterns

Вопросы