

ПАТТЕРНЫ

*Уметь рассказывать, что применяли на практике с примерами.
<https://refactoring.guru/ru/design-patterns>

Что такое «шаблон проектирования»?

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

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.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Если привести аналогии, то:

- алгоритм — это кулинарный рецепт с чёткими шагами
- паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

Польза паттернов

Вы можете вполне успешно работать, не зная ни одного паттерна. Но зная паттерны, вы получаете ещё один инструмент в свой личный набор профессионала.

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.

Но 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.

<https://github.com/enhorse/java-interview/blob/master/patterns.md>

Приведите ПРИМЕРЫ основных ШАБЛОНОВ проектирования.

Делегирование (Delegation pattern)

Сущность внешне выражает некоторое поведение, но в реальности передаёт ответственность за выполнение этого поведения связанному объекту.

Функциональный дизайн (Functional design)

Гарантирует, что каждая сущность имеет только одну обязанность и исполняет её с минимумом побочных эффектов на другие.

Неизменяемый интерфейс (Immutable interface). Создание неизменяемого объекта.

Интерфейс (Interface)

Общий метод структурирования сущностей, облегчающий их понимание.

Интерфейс-маркер (Marker interface)

В качестве атрибута (как пометки объектной сущности) применяется наличие или отсутствие реализации интерфейса-маркера. В современных языках программирования вместо этого применяются атрибуты или аннотации.

Контейнер свойств (Property container)

Позволяет добавлять дополнительные свойства сущности в контейнер внутри себя, вместо расширения новыми свойствами.

Канал событий (Event channel)

Создаёт централизованный канал для событий. Использует сущность-представитель для подписки и сущность-представитель для публикации события в канале.

Представитель существует отдельно от реального издателя или подписчика. Подписчик может получать опубликованные события от более чем одной сущности, даже если он зарегистрирован только на одном канале.

Назовите основные ХАРАКТЕРИСТИКИ шаблонов.

Из чего состоит паттерн?

Описания паттернов обычно очень формальны и чаще всего состоят из таких пунктов:

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

Такой формализм в описании позволил создать обширный каталог паттернов, проверив каждый из них на состоятельность.

Назовите ТРИ основные ГРУППЫ паттернов (классификация)

Паттерны отличаются **по уровню сложности, охвата и детализации** проектируемой системы. Кроме этого, их можно поделить на три группы, относительно решаемых проблем.

- **Порождающие** (беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей)

- **Структурные** (показывают различные способы построения связей между объектами)
- **Поведенческие** (заботятся об эффективной коммуникации между объектами)

Поведенческие решают задачи **эффективного и безопасного взаимодействия** между объектами программы.



Порождающие отвечают за удобное и безопасное **создание новых объектов** или даже целых семейств объектов.



Структурные отвечают за **построение удобных в поддержке иерархий** классов



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

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

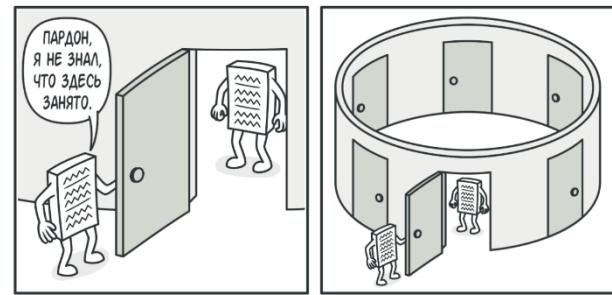
Расскажите про паттерн Одиночка (Singleton).

<https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples>

Одиночка / Singleton — это порождающий паттерн проектирования, который гарантирует, что у класса **есть только один экземпляр**, и предоставляет к нему **глобальную точку доступа**.

Признаки применения паттерна:

Одиночку можно определить по статическому создающему методу, который возвращает один и тот же объект.



Клиенты могут не подозревать, что работают с одним и тем же объектом.

Singleton **ограничивает создание экземпляра класса** и гарантирует, что в виртуальной машине Java **существует только ОДИН экземпляр класса**.

- Одноэлементный класс должен предоставить глобальную точку доступа для получения экземпляра класса.
- Singleton используется для ведения журнала, объектов драйверов, кэширования и пуль потоков.
- Singleton используется в других шаблонах проектирования: Abstract Factory, Builder, Prototype, Facade и т. д.
- Singleton используется в основных классах Java (например, java.lang.Runtime, java.awt.Desktop).

Из определения это кажется простым шаблоном проектирования, но, когда дело доходит до реализации, **возникает много проблем**.

Реализация шаблона Java Singleton

Для реализации одноэлементного шаблона у нас есть разные подходы, но все они имеют следующие общие концепции.

- **Private конструктор** для ограничения создания экземпляра класса из других классов Private static variable of the same class that is the only instance of the class.
- **Private static переменная** того же класса, который является единственным экземпляром класса.
- **Public static метод**, который возвращает экземпляр класса, это глобальная точка доступа для внешнего мира, чтобы получить экземпляр одноэлементного класса.

Вариант написания «одиночки», который создаётся при старте приложения.

```
// Загрузчик классов Class Loader один на всё приложение.  
// Вариант написания «одиночки», который создаётся при старте приложения.  
  
public class Singleton {  
  
    1 usage  
    private static class ClassHolder {  
        1 usage  
        private static Singleton instance = new Singleton();  
    }  
  
    1 usage  
    public Singleton() {}      // конструктор без параметров  
  
    1 usage  
    public static Singleton getInstance() {  
        return ClassHolder.instance;  
    }  
  
    public static void main(String[] args) {  
        Singleton object = Singleton.getInstance();  
        System.out.println(object.getClass().getClassLoader().getName());  
    }  
}
```

Обратите внимание на внутренний класс (*private inner static class*), который содержит экземпляр одноэлементного класса.

Когда загружается класс Singleton, класс ClassHolder не загружается в память, и только когда кто-то вызывает метод `getInstance()`, этот класс загружается и создает экземпляр класса singleton.

Это наиболее широко используемый подход для одноэлементного класса, поскольку он не требует синхронизации.

```
Singleton  
app  
  
Process finished with exit code 0
```

Вариант написания через аннотации Spring, см. видео → <https://youtu.be/61duchvKI6o?t=120>



Lazy injection – since 4.3

```
@Service  
@Lazy  
public class LazySingleton {  
  
    @Autowired  
    @Lazy  
    private LazySingleton lazySingleton;  
  
    @Autowired  
    public MainService(@Lazy LazySingleton lazySingleton) {  
        this.lazySingleton = lazySingleton;  
    }  
}
```



```
@Lazy  
private LazySingleton lazySingleton;  
  
@Autowired  
public MainService(LazySingleton lazySingleton) {  
    this.lazySingleton = lazySingleton;  
}
```

Почему СИНГЛОН ЭТО анти-паттерн

- Создаёт coupling
- Провоцирует неправильное написание кода
- Делает код менее гибким
- Невозможно делать моки без PowerMock
- Да и вообще это нарушает Single Responsibility

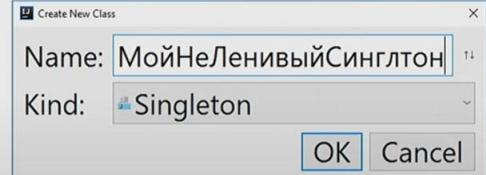


Lazy injection – since 4.3

```
@Service  
@Lazy  
public class LazySingleton {  
  
    @Autowired  
    @Lazy  
    private LazySingleton lazySingleton;  
  
    @Autowired  
    public MainService(@Lazy LazySingleton lazySingleton) {  
        this.lazySingleton = lazySingleton;  
    }  
  
    @Lazy  
    public MainService(LazySingleton lazySingleton) {  
        this.lazySingleton = lazySingleton;  
    }  
}
```

Сделано в JetBrains – eager singleton

```
public class Singleton {  
    private static Singleton ourInstance = new Singleton();  
  
    public static Singleton getInstance() {  
        return ourInstance;  
    }  
  
    private Singleton() {  
    }  
}
```



Почему СИНГЛОН называют анти-паттерном?

Нарушение принципа единственной ответственности SOLID

В ООП существует правило хорошего тона — «Принцип единственной ответственности» (Single Responsibility Principle, первая буква в аббревиатуре SOLID).

Согласно этому правилу, каждый класс должен отвечать лишь за один какой-то аспект. Но любой Singleton-класс **отвечает сразу за две вещи**: за то, что класс имеет лишь один объект, и за реализацию того, для чего этот класс вообще был создан.

Принцип единственной ответственности был создан не просто так — если класс отвечает за несколько действий, то, внося изменения в один аспект поведения класса, можно затронуть и другой, что может сильно усложнить разработку.

Так же разработку усложняет тот факт, что пере-использование (reusability) класса практически невозможно.

Поток-небезопасность

Один из популярных вариантов реализации Singleton содержит ленивую инициализацию. Это значит, что объект класса создаётся не в самом начале, а лишь когда будет получено первое обращение к нему.

Однако здесь начинаются проблемы с потоками, которые могут создавать несколько различных объектов. Происходит это примерно так:

- Первый поток обращается к `getInstance()`, когда объект ещё НЕ создан
- В это время второй тоже обращается к этому методу, пока первый ещё не успел создать объект, и сам создаёт его
- Первый поток создаёт ещё один, второй, экземпляр класса

Разумеется, можно просто пометить метод как **synchronized**, и эта проблема исчезнет.

Проблема заключается в том, что, сохраняя время на старте программы, мы теперь будем терять его каждый раз при обращении к Singleton'у из-за того, что метод синхронизирован, а **это очень дорого**, если к экземпляру приходится часто обращаться.

Единственный раз, когда свойство synchronized действительно требуется — **первое обращение к методу**. Есть два способа решить эту проблему.

Первый — пометить как synchronized не весь метод, а только блок, где создаётся объект. Не забывайте, что это нельзя использовать в версии Java ниже, чем 1.5, потому что там используется иная модель памяти. Также не забудьте пометить поле instance как volatile.

Второй путь — **использовать паттерн «Lazy Initialization Holder»**.

Это решение основано на том, что вложенные классы не инициализируются до первого их использования (как раз то, что нам нужно). См. код выше.

Трудности с тестированием «одиночки»

Один из главных минусов паттерна «Одиночка» — он сильно затрудняет юнит-тестирование. «Одиночка» привносит в программу **глобальное состояние**, поэтому нельзя просто взять и изолировать классы, которые полагаются на Singleton.

Поэтому, если нужно протестировать какой-то класс, то нужно вместе с ним тестировать и Singleton, но это ещё полбеды. Состояние «Одиночки» может меняться, что порождает следующие проблемы:

- Порядок тестов теперь имеет значение
- Тесты могут иметь нежелательные сторонние эффекты, порождённые Singleton'ом
- Вы не можете запускать несколько тестов параллельно
- Несколько вызовов одного и того же теста могут приводить к разным результатам.

На эту тему есть отличный доклад с «Google Tech Talks».

Загрузчик класса

Если говорить о Java, то обеспечение существования лишь одного экземпляра класса, которое так необходимо для Singleton, становится всё сложнее.

Проблема в том, что классическая реализация не проверяет, существует ли один экземпляр на JVM, он лишь удостоверяется, что существует один экземпляр на classloader. Если вы пишете небольшое клиентское приложение, в котором **используется лишь один classloader, то никаких проблем не возникнет** (см. код выше).

Однако если используете несколько загрузчиков класса или ваше приложение должно работать на сервере (где может быть запущено несколько экземпляров приложения в разных загрузчиках классов), то всё становится очень печально.

ВЫВОД

Несмотря на то, что паттерн Singleton очень известный и популярный, у него есть множество серьёзных недостатков. Чем дальше, тем больше этих недостатков выявляется, и оригинальные паттерны из книги GOF «Design Patterns» часто сегодня считаются анти-паттернами.

Тем не менее, сама идея иметь лишь один объект на класс по-прежнему имеет смысл, но **достаточно сложно реализовать ее правильно**.

⊕ Преимущества и недостатки

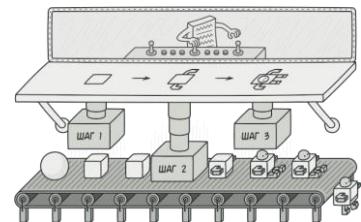
- ✓ Гарантирует наличие единственного экземпляра класса.
- ✓ Предоставляет к нему глобальную точку доступа.
- ✓ Реализует отложенную инициализацию объекта-одиночки.
- ✗ Нарушает принцип единственной ответственности класса.
- ✗ Маскирует плохой дизайн.
- ✗ Проблемы мультипоточности.
- ✗ Требует постоянного создания Mock-объектов при юнит-тестировании.

↔ Отношения с другими паттернами

- Фасад можно сделать Одиночкой, так как обычно нужен только один объект-фасад.
- Паттерн Легковес может напоминать Одиночку, если для конкретной задачи у вас получилось свести количество объектов к одному. Но помните, что между паттернами есть два кардинальных отличия:
 1. В отличие от Одиночки, вы можете иметь множество объектов-легковесов.
 2. Объекты-легковесы должны быть неизменяемыми, тогда как объект-одиночка допускает изменение своего состояния.
- Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.

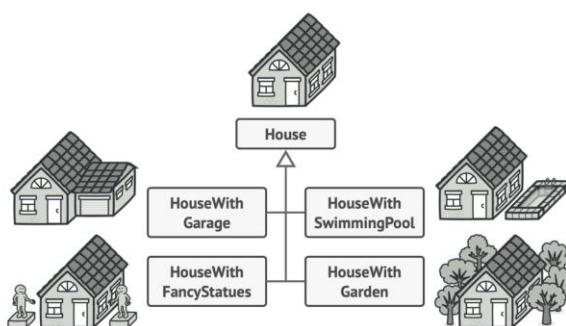
Расскажите про паттерн Строитель (Builder).

Строитель — это порождающий паттерн проектирования, который позволяет **создавать сложные объекты пошагово**. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

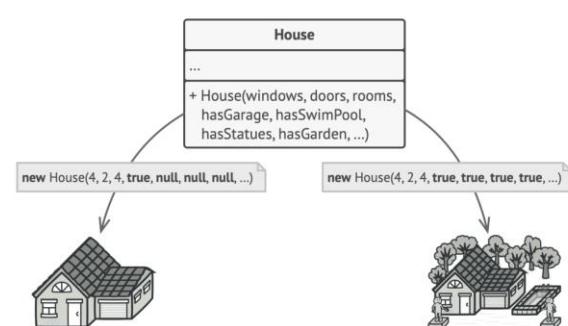


Проблема

Представьте сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации таких объектов обычно спрятан внутри монструозного конструктора с десятком параметров. Либо ещё хуже — распылён по всему клиентскому коду.



Создав кучу подклассов для всех конфигураций объектов, вы можете излишне усложнить программу.



Конструктор со множеством параметров имеет свой недостаток: не все параметры нужны большую часть времени.

Чтобы не плодить подклассы, вы можете подойти к решению с другой стороны. Вы можете создать гигантский конструктор Дома, принимающий уйму параметров для контроля над создаваемым продуктом. Действительно, это избавит вас от подклассов, но приведёт к другой проблеме.

Большая часть этих параметров будет простаивать, а вызовы конструктора будут выглядеть монструозно из-за длинного списка параметров. К примеру, далеко не каждый дом имеет бассейн, поэтому параметры, связанные с бассейнами, будут простаивать бесполезно в 99% случаев.

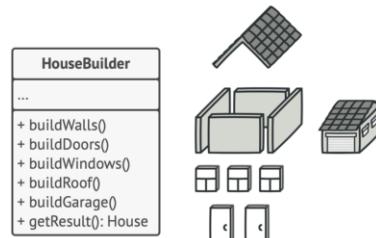
Решение

Паттерн Строитель предлагает **вынести конструирование объекта за пределы его собственного класса**, поручив это дело отдельным объектам, называемым строителями.

Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, `построитьСтены`, `вставитьДвери` и другие). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

Зачастую один и тот же шаг строительства может отличаться для разных вариаций производимых объектов. Например, деревянный дом потребует строительства стен из дерева, а каменный — из камня.

В этом случае вы можете создать несколько классов строителей, выполняющих одни и те же шаги по-разному. Используя этих строителей в одном и том же строительном процессе, вы сможете получать на выходе различные объекты.



Строитель позволяет создавать сложные объекты пошагово. Промежуточный результат всегда остаётся защищён.

Код, который вызывает шаги строительства, должен работать со строителями через **общий интерфейс**, чтобы их можно было свободно взаимо-заменять.

Директор

Вы можете пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый директором. В этом случае директор будет задавать порядок шагов строительства, а строитель — выполнять их.

Директор полезен, если у вас есть несколько способов конструирования продуктов, отличающихся порядком и наличием шагов конструирования. В этом случае вы сможете объединить всю эту логику в одном классе.

Такая структура классов полностью скроет от клиентского кода процесс конструирования объектов. Клиенту останется только привязать желаемого строителя к директору, а затем получить у строителя готовый результат.
Структура и псевдокод по ссылке: <https://refactoring.guru/ru/design-patterns/builder>

Когда применять?

1. Когда хотите избавиться от «телескопического конструктора».

Допустим, у вас есть один конструктор с десятью опциональными параметрами. Его неудобно вызывать, поэтому вы создали ещё десять конструкторов с меньшим количеством параметров. Всё, что они делают — это переадресуют вызов к базовому конструктору, подавая какие-то значения по умолчанию в параметры, которые пропущены в них самих.

Паттерн Строитель позволяет собирать объекты пошагово, вызывая только те шаги, которые вам нужны.

2. Когда код должен создавать разные представления какого-то объекта (например, деревянные и железобетонные дома).

Строитель можно применить, если создание нескольких представлений объекта состоит из **одинаковых этапов**, которые отличаются в деталях.

Интерфейс строителей определит все возможные этапы конструирования. Каждому представлению будет соответствовать собственный класс-строитель. А порядок этапов строительства будет задавать класс-директор.

3. Когда нужно собирать сложные составные объекты, например, деревья Компоновщика.

Строитель конструирует объекты пошагово, а не за один проход. Более того, шаги строительства можно выполнять рекурсивно. А без этого не построить древовидную структуру, вроде Компоновщика.

Строитель не позволяет посторонним объектам иметь доступ к конструируемому объекту, пока тот не будет полностью готов. Это предохраняет клиентский код от получения **незаконченных «битых» объектов**.

Шаги реализации

- Убедитесь в том, что создание разных представлений объекта можно свести к общим шагам.
- Опишите эти шаги в общем интерфейсе строителей.
- Для каждого из представлений объекта-продукта создайте по одному классу-строителю и реализуйте их методы строительства.
- Подумайте о создании класса директора. Его методы будут создавать различные конфигурации продуктов, вызывая разные шаги одного и того же строителя.
- Клиентский код должен будет создавать и объекты строителей, и объект директора.

Перед началом строительства клиент должен связать определённого строителя с директором. Это можно сделать либо через конструктор, либо через сеттер, либо подав строителя напрямую в строительный метод директора.

- Результат строительства можно вернуть из директора, но только если метод возврата продукта удалось поместить в общий интерфейс строителей.
Иначе вы жёстко привяжете директора к конкретным классам строителей.

⊕ Преимущества и недостатки

- | | |
|--|---|
| <ul style="list-style-type: none">✓ Позволяет создавать продукты пошагово.✓ Позволяет использовать один и тот же код для создания различных продуктов.✓ Изолирует сложный код сборки продукта от его основной бизнес-логики. | <ul style="list-style-type: none">✗ Усложняет код программы из-за введения дополнительных классов.✗ Клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата. |
|--|---|

↔ Отношения с другими паттернами

- Многие архитектуры начинаются с применения **Фабричного метода** (более простого и расширяемого через подклассы) и эволюционируют в сторону **Абстрактной фабрики**, **Прототипа** или **Строителя** (более гибких, но и более сложных).
- **Строитель** концентрируется на построении сложных объектов шаг за шагом. **Абстрактная фабрика** специализируется на создании семейств связанных продуктов. **Строитель** возвращает продукт только после выполнения всех шагов, а **Абстрактная фабрика** возвращает продукт сразу же.
- **Строитель** позволяет пошагово сооружать дерево **Компоновщика**.
- Паттерн **Строитель** может быть построен в виде **Моста**: директор будет играть роль абстракции, а строители — реализации.
- **Абстрактная фабрика**, **Строитель** и **Прототип** могут быть реализованы при помощи **Одиночки**.

Расскажите про паттерн Фабричный метод (Factory Method).

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для **создания объектов в суперклассе**, позволяя подклассам изменять тип создаваемых объектов.

Проблема

Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик.

Большая часть существующего кода **жёстко привязана** к классам Грузовиков. Чтобы добавить в программу классы морских Судов, понадобится переполотить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу придётся повторить.

В итоге вы получите ужасающий код, наполненный условными операторами, которые выполняют то или иное действие, в зависимости от класса транспорта.

Решение

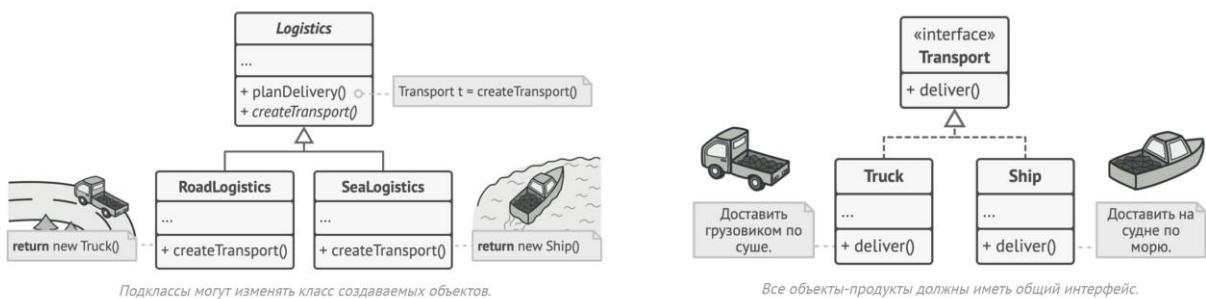
Паттерн Фабричный метод предлагает создавать объекты не напрямую, используя оператор new, а через **вызов особого фабричного метода**. Не пугайтесь, объекты всё равно будут создаваться при помощи new, но делать это будет фабричный метод.

Чтобы эта система заработала, все возвращаемые **объекты должны иметь общий интерфейс**. Подклассы смогут производить объекты различных классов, следующих одному и тому же интерфейсу.

Например, классы Грузовик и Судно реализуют **интерфейс Транспорт с методом доставить**.

Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а суда — по морю.

Фабричный метод в классе Дорожной Логистики вернёт объект-грузовик, а класс Морской Логистики — объект-судно.



Структура и псевдокод по ссылке: <https://refactoring.guru/ru/design-patterns/factory-method>

Когда применять?

1. Когда заранее неизвестны типы и зависимости объектов, с которыми должен работать ваш код.

Фабричный метод отделяет код производства продуктов от остального кода, который эти продукты использует. Благодаря этому, код производства можно расширять, не трогая основной. Так, чтобы добавить поддержку нового продукта, вам нужно создать новый подкласс и определить в нём фабричный метод, возвращая оттуда экземпляр нового продукта.

2. Когда вы хотите дать возможность пользователям расширять части вашего фреймворка или библиотеки.

Пользователи могут расширять классы вашего фреймворка через наследование. Но как сделать так, чтобы фреймворк создавал объекты из этих новых классов, а не из стандартных?

Решением будет дать пользователям возможность расширять не только желаемые компоненты, но и классы, которые создают эти компоненты. А для этого создающие классы должны иметь конкретные создающие методы, которые можно определить.

3. Когда вы хотите экономить системные ресурсы, повторно используя уже созданные объекты, вместо порождения новых.

Такая проблема обычно возникает при работе с тяжёлыми ресурсоёмкими объектами, такими, как подключение к базе данных, файловой системе и т. д.

Представьте, сколько действий нужно совершить, чтобы повторно использовать существующие объекты:

- Сначала вам следует создать общее хранилище, чтобы хранить в нём все создаваемые объекты.
- При запросе нового объекта нужно будет заглянуть в хранилище и проверить, есть ли там неиспользуемый объект. А затем вернуть его клиентскому коду.
- Но если свободных объектов нет — создать новый, не забыв добавить его в хранилище. Весь этот код нужно куда-то поместить, чтобы не засорять клиентский код.

Самым удобным местом был бы конструктор объекта, ведь все эти проверки нужны только при создании объектов. Но, увы, конструктор всегда создаёт новые объекты, он не может вернуть существующий экземпляр.

Значит, нужен другой метод, который бы отдавал как существующие, так и новые объекты. Им и станет фабричный метод.

Шаги реализации

- Приведите все создаваемые продукты к общему интерфейсу
- В классе, который производит продукты, создайте пустой фабричный метод. В качестве возвращаемого типа укажите общий интерфейс продукта.
- Найдите все участки кода класса, создающие продукты. Поочерёдно замените эти участки вызовами фабричного метода, перенося в него код создания различных продуктов.
- В фабричный метод, возможно, придётся добавить несколько параметров, контролирующих, какой из продуктов нужно создать.

На этом этапе фабричный метод, скорее всего, будет выглядеть удручающе. В нём будет жить большой условный оператор, выбирающий класс создаваемого продукта. Это можно исправить.

- Для каждого типа продуктов заведите подкласс и переопределите в нём фабричный метод. Переместите туда код создания соответствующего продукта из суперкласса.
- Если создаваемых продуктов слишком много для существующих подклассов создателя, вы можете подумать о введении параметров в фабричный метод, которые позволят возвращать различные продукты в пределах одного подкласса.

Например, у вас есть класс Почта с подклассами АвиаПочта и НаземнаяПочта, а также классы продуктов Самолёт, Грузовик и Поезд. Авиа соответствует Самолётам, но для НаземнойПочты есть сразу два продукта. Вы могли бы создать новый подкласс почты для поездов, но проблему можно решить и по-другому. Клиентский код может передавать в фабричный метод НаземнойПочты аргумент, контролирующий тип создаваемого продукта.

- Если после всех перемещений фабричный метод стал пустым, можете сделать его абстрактным. Если в нём что-то осталось — не беда, это будет его реализацией по умолчанию.

⊕ Преимущества и недостатки

- ✓ Избавляет класс от привязки к конкретным классам продуктов.
- ✓ Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- ✓ Упрощает добавление новых продуктов в программу.
- ✓ Реализует принцип открытости/закрытости.
- ✗ Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

↔ Отношения с другими паттернами

- Многие архитектуры начинаются с применения Фабричного метода (более простого и расширяемого через подклассы) и эволюционируют в сторону Абстрактной фабрики, Прототипа или Строителя (более гибких, но и более сложных).
- Классы Абстрактной фабрики чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе Прототипа.
- Фабричный метод можно использовать вместе с Итератором, чтобы подклассы коллекций могли создавать подходящие им итераторы.
- Прототип не опирается на наследование, но ему нужна сложная операция инициализации. Фабричный метод, наоборот, построен на наследовании, но не требует сложной инициализации.
- Фабричный метод можно рассматривать как частный случай Шаблонного метода. Кроме того, Фабричный метод нередко бывает частью большого класса с Шаблонными методами.

Расскажите про паттерн **Абстрактная фабрика (Abstract Factory)**.

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет **создавать семейства связанных объектов, не привязываясь к конкретным классам** создаваемых объектов.

Проблема

Представьте, что вы пишете симулятор мебельного магазина. Ваш код содержит семейство зависимых продуктов. Скажем, Кресло + Диван + Столик.

Несколько вариаций этого семейства. Например, продукты Кресло, Диван и Столик представлены в трёх разных стилях: Ар-деко, Викторианском и Модерн.

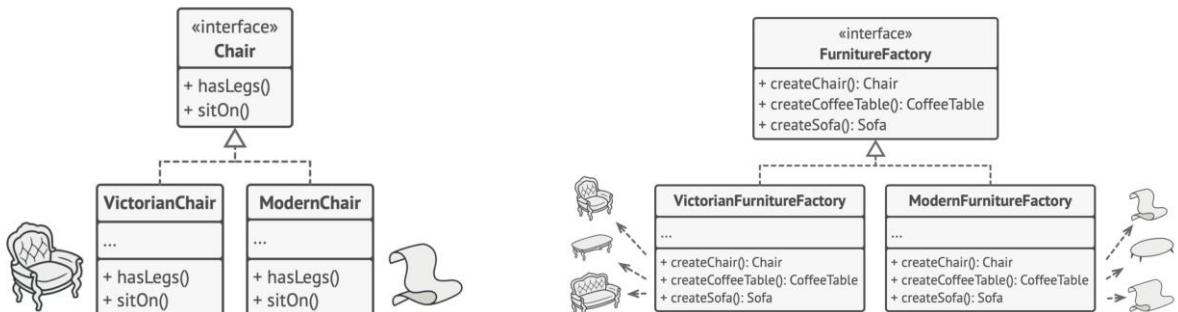


Нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель.

Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейств в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

Решение

Для начала паттерн Абстрактная фабрика предлагает **выделить общие интерфейсы** для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.



Все вариации одного и того же объекта должны жить в одной иерархии классов.

Конкретные фабрики соответствуют определённой вариации семейства продуктов.

Далее вы создаёте **абстрактную фабрику — общий интерфейс**, который содержит методы создания всех продуктов семейства (например, создать Кресло, создать Диван и создать Столик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые выделили ранее — Кресла, Диваны и Столики.

Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны **создать свою собственную фабрику**, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, ФабрикаМодерн будет возвращать только КреслаМодерн, ДиваныМодерн и СтоликиМодерн.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы.

Это позволит подавать в классы любой тип фабрики и производить любые продукты, ничего не ломая.



Для клиентского кода должно быть безразлично, с какой фабрикой работать.

Осталось прояснить последний момент: **кто создаёт объекты конкретных фабрик**, если клиентский код работает только с интерфейсами фабрик?

Обычно программа создаёт конкретный объект фабрики **при запуске**, причём тип фабрики выбирается, исходя из параметров окружения или конфигурации.

Когда применять?

1. Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.

Абстрактная фабрика скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, поскольку их общий интерфейс был заранее определён.

2. Когда в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов.

В хорошей программе каждый класс отвечает только за одну вещь. Если класс имеет слишком много фабричных методов, они способны затуманить его основную функцию.

Поэтому имеет смысл вынести всю логику создания продуктов в отдельную иерархию классов, применив абстрактную фабрику.

Шаги реализации

- Создайте таблицу соотношений типов продуктов к вариациям семейств продуктов.
- Сведите все вариации продуктов к общим интерфейсам.
- Определите интерфейс абстрактной фабрики. Он должен иметь фабричные методы для создания каждого из типов продуктов.
- Создайте классы конкретных фабрик, реализовав интерфейс абстрактной фабрики. Этих классов должно быть столько же, сколько и вариаций семейств продуктов.
- Измените код инициализации программы так, чтобы она создавала определённую фабрику и передавала её в клиентский код.
- Замените в клиентском коде участки создания продуктов через конструктор вызовами соответствующих методов фабрики.

⊕ Преимущества и недостатки

- | | |
|---|--|
| <ul style="list-style-type: none">✓ Гарантирует сочетаемость создаваемых продуктов.✓ Избавляет клиентский код от привязки к конкретным классам продуктов.✓ Выделяет код производства продуктов в одно место, упрощая поддержку кода.✓ Упрощает добавление новых продуктов в программу.✓ Реализует <u>принцип открытости/закрытости</u>. | <ul style="list-style-type: none">✗ Усложняет код программы из-за введения множества дополнительных классов.✗ Требует наличия всех типов продуктов в каждой вариации. |
|---|--|

↔ Отношения с другими паттернами

- Многие архитектуры начинаются с применения Фабричного метода (более простого и расширяемого через подклассы) и эволюционируют в сторону Абстрактной фабрики, Прототипа или Строителя (более гибких, но и более сложных).
 - Строитель концентрируется на построении сложных объектов шаг за шагом. Абстрактная фабрика специализируется на создании семейств связанных продуктов. Строитель возвращает продукт только после выполнения всех шагов, а Абстрактная фабрика возвращает продукт сразу же.
 - Классы Абстрактной фабрики чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе Прототипа.
 - Абстрактная фабрика может быть использована вместо Фасада для того, чтобы скрыть платформо-зависимые классы.
 - Абстрактная фабрика может работать совместно с Мостом. Это особенно полезно, если у вас есть абстракции, которые могут работать только с некоторыми из реализаций. В этом случае фабрика будет определять типы создаваемых абстракций и реализаций.
- Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.

Расскажите про паттерн Прототип (Prototype).

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

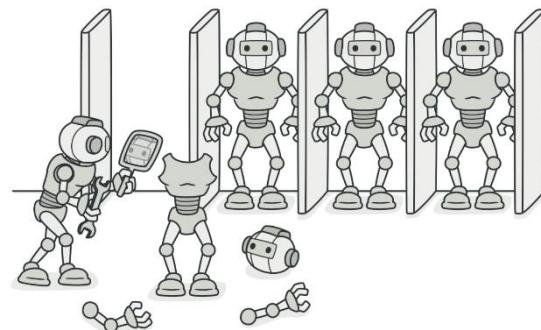
Проблема

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.

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



Решение

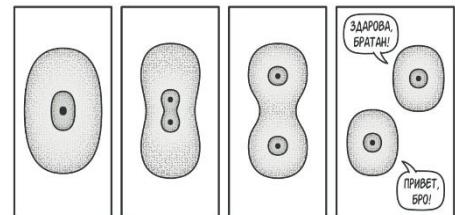
Паттерн Прототип **поручает** создание копий самим копируемым объектам. Он вводит **общий интерфейс** для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет **всего один метод** `clone()`.

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** (отсюда 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 приготовленного прототипа.

Наиболее понятный и близкий пример паттерна — деление клеток. После митозного деления клеток образуются две совершенно идентичные клетки. **Оригинальная клетка отыгрывает роль прототипа**, принимая активное участие в создании нового объекта.



Пример деления клетки.

Структура и псевдокод по ссылке → <https://refactoring.guru/ru/design-patterns/prototype>

Кода применять?

1. Когда ваш код не должен зависеть от классов копируемых объектов.

Такое часто бывает, если ваш код работает с объектами, поданными извне через какой-то общий интерфейс. Вы не можете привязаться к их классам, даже если бы хотели, поскольку их конкретные классы неизвестны.

Паттерн прототип предоставляет клиенту общий интерфейс для работы со всеми прототипами. Клиенту не нужно зависеть от всех классов копируемых объектов, а только от интерфейса клонирования.

2. Когда имеете уйму подклассов, которые отличаются начальными значениями полей. Кто-то мог создать все эти классы, чтобы иметь возможность легко порождать объекты с определённой конфигурацией.

Паттерн прототип предлагает использовать набор прототипов, вместо создания подклассов для описания популярных конфигураций объектов.

Таким образом, вместо порождения объектов из подклассов, вы будете копировать существующие объекты-прототипы, в которых уже настроено внутреннее состояние. **Это позволит избежать взрывного роста количества классов в программе и уменьшить её сложность.**

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, объявленных в рамках текущего класса, а затем передать выполнение родительскому конструктору, чтобы тот позаботился о полях, объявленных в суперклассе.

Если ваш язык программирования не поддерживает перегрузку методов, то вам не удастся создать несколько версий конструктора. В этом случае копирование значений можно проводить и в другом методе, специально созданном для этих целей. 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.

⊕ Преимущества и недостатки

- | | |
|---|---|
| <ul style="list-style-type: none">✓ Позволяет клонировать объекты, не привязываясь к их конкретным классам.✓ Меньше повторяющегося кода инициализации объектов.✓ Ускоряет создание объектов.✓ Альтернатива созданию подклассов для конструирования сложных объектов. | <ul style="list-style-type: none">✗ Сложно клонировать составные объекты, имеющие ссылки на другие объекты. |
|---|---|

↔ Отношения с другими паттернами

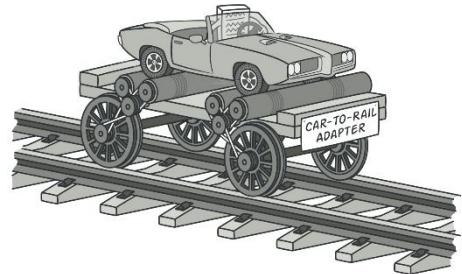
- Многие архитектуры начинаются с применения [Фабричного метода](#) (более простого и расширяемого через подклассы) и эволюционируют в сторону [Абстрактной фабрики](#), [Прототипа](#) или [Строителя](#) (более гибких, но и более сложных).
- Классы [Абстрактной фабрики](#) чаще всего реализуются с помощью [Фабричного метода](#), хотя они могут быть построены и на основе [Прототипа](#).
- Если [Команды](#) нужно копировать перед вставкой в историю выполненных команд, вам может помочь [Прототип](#).
- Архитектура, построенная на [Компоновщиках](#) и [Декораторах](#), часто может быть улучшена за счёт внедрения [Прототипа](#). Он позволяет клонировать сложные структуры объектов, а не собирать их заново.
- [Прототип](#) не опирается на наследование, но ему нужна сложная операция инициализации. [Фабричный метод](#), наоборот, построен на наследовании, но не требует сложной инициализации.
- [Снимок](#) иногда можно заменить [Прототипом](#), если объект, состояние которого требуется сохранять в истории, довольно простой, не имеет активных ссылок на внешние ресурсы либо их можно легко восстановить.
- [Абстрактная фабрика](#), [Строитель](#) и [Прототип](#) могут быть реализованы при помощи [Одиночки](#).

Расскажите про паттерн Адаптер (Adapter).

Мы изучали FileInputStream (поток байт). А также мы изучали FileInputStream Reader/Writer. Так вот последние – это настоящие Адаптеры (паттерн Адаптер), т.к. они преобразует поток байт в поток символов.

И еще про паттерн Адаптер (из того, что мы программировали в задачах): контроллер принимает http-запрос в виде json (или другие сырье форматы) и далее с помощью библиотеки Джексон конвертирует их в объекты java. Это тоже яркий пример паттерна Адаптер.

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



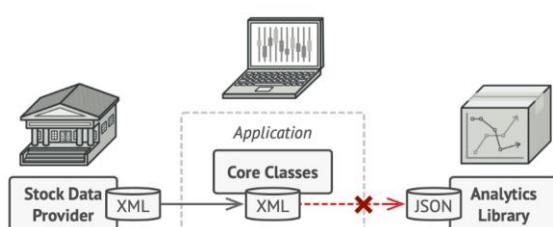
Проблема

Представьте, что вы делаете приложение для торговли на бирже. Ваше приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует красивые графики.

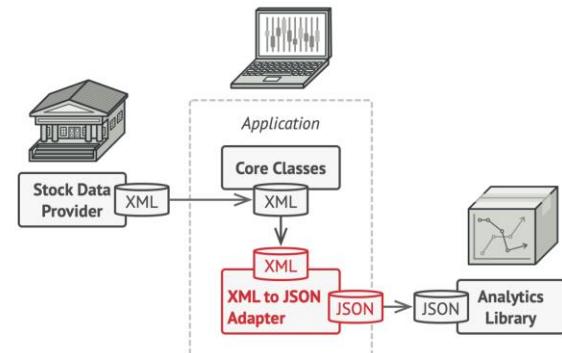
В какой-то момент вы решаете улучшить приложение, применив стороннюю библиотеку аналитики. Но вот беда — библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением.

Вы смогли бы переписать библиотеку, чтобы та поддерживала формат XML.

Но, во-первых, это может нарушить работу существующего кода, который уже зависит от библиотеки. А во-вторых, у вас может просто не быть доступа к её исходному коду.



Подключить стороннюю библиотеку не выйдет из-за несовместимых форматов данных.



Программа может работать со сторонней библиотекой через адаптер.

Решение

Вы можете создать адаптер. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

- Адаптер имеет интерфейс, который совместим с одним из объектов.
- Поэтому этот объект может свободно вызывать методы адаптера.
- Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

Иногда возможно создать двухсторонний адаптер, который работает в обе стороны.

Таким образом, в приложении биржевых котировок вы могли бы создать класс XML_To_JSON_Adapter, который бы оборачивал объект того или иного класса библиотеки аналитики. Ваш код посыпал бы адаптеру запросы в формате XML, а адаптер сначала транслировал входящие данные в формат JSON, а затем передавал бы их методам обёрнутого объекта аналитики.

Структура и псевдокод по ссылке → <https://refactoring.guru/ru/design-patterns/adapter>

Когда применять?

1. Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.

Адаптер позволяет создать объект-прокладку, который будет превращать вызовы приложения в формат, понятный стороннему классу.

2. Когда нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс нельзя.

Можно создать ещё один уровень подклассов и добавить в них недостающую функциональность, но при этом придётся дублировать один и тот же код в обеих ветках подклассов.

Более элегантным решением было бы поместить недостающую функциональность в адаптер и приспособить его для работы с суперклассом. Такой адаптер сможет работать со всеми подклассами иерархии. Это решение будет сильно напоминать **паттерн Декоратор**.

Шаги реализации

- Убедитесь, что у вас есть два класса с несовместимыми интерфейсами:
 - полезный сервис — служебный класс, который вы не можете изменять (он либо сторонний, либо от него зависит другой код);
 - один или несколько клиентов — существующих классов приложения, несовместимых с сервисом из-за неудобного или несовпадающего интерфейса.
- Опишите клиентский интерфейс, через который классы приложения смогли бы использовать класс сервиса.
- Создайте класс адаптера, реализовав этот интерфейс.
- Поместите в адаптер поле, которое будет хранить ссылку на объект сервиса. Обычно это поле заполняют объектом, переданным в конструктор адаптера. В случае простой адаптации этот объект можно передавать через параметры методов адаптера.
- Реализуйте все методы клиентского интерфейса в адаптере.
Адаптер должен делегировать основную работу сервису.
- Приложение должно использовать адаптер только через клиентский интерфейс. Это позволит легко изменять и добавлять адаптеры в будущем.

⊕ Преимущества и недостатки

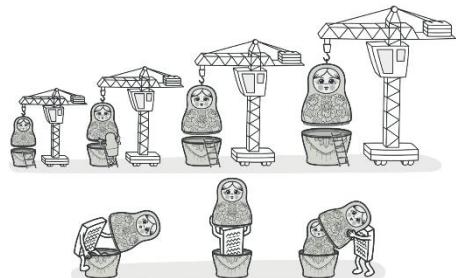
- ✓ Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.
- ✗ Усложняет код программы из-за введения дополнительных классов.

⇨ Отношения с другими паттернами

- **Мост** проектируют загодя, чтобы развивать большие части приложения отдельно друг от друга. **Адаптер** применяется постфактум, чтобы заставить несовместимые классы работать вместе.
- **Адаптер** меняет интерфейс существующего объекта. **Декоратор** улучшает другой объект без изменения его интерфейса. Причём **Декоратор** поддерживает рекурсивнуюложенность, чего не скажешь об **Адаптере**.
- **Адаптер** представляет классу альтернативный интерфейс. **Декоратор** предоставляет расширенный интерфейс. **Заместитель** предоставляет тот же интерфейс.
- **Фасад** задаёт новый интерфейс, тогда как **Адаптер** повторно использует старый. Адаптер оборачивает только один класс, а **Фасад** оборачивает целую подсистему. Кроме того, **Адаптер** позволяет двум существующим интерфейсам работать сообща, вместо того, чтобы задать полностью новый.
- **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.

Расскажите про паттерн Декоратор (Decorator).

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оберчивая их в **полезные «обёртки»**.



Проблема

Вы работаете над библиотекой оповещений, которую можно подключать к разным программам, чтобы получать уведомления о важных событиях.

Основной библиотеки является класс `Notifier` с методом `send`, который принимает на вход строку-сообщение и высылает её всем администраторам по электронной почте. Сторонняя программа должна создать и настроить этот объект, указав кому отправлять оповещения, а затем использовать его каждый раз, когда что-то случается.

В какой-то момент стало понятно, что одних email оповещений пользователям мало. Некоторые из них хотели бы получать извещения о критических проблемах через SMS. Другие хотели бы получать их в виде сообщений Facebook. Корпоративные пользователи хотели бы видеть сообщения в Slack.



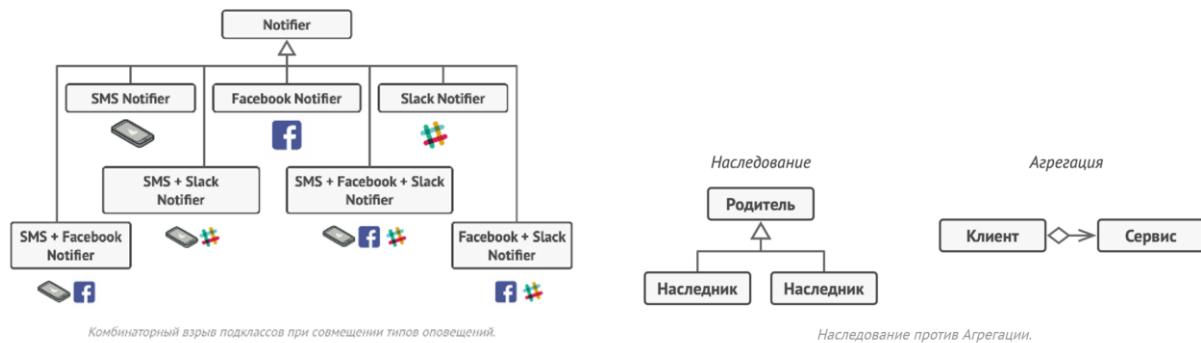
Сначала вы добавили каждый из этих типов оповещений в программу, унаследовав их от базового класса `Notifier`. Теперь пользователь выбирал один из типов оповещений, который и использовался в дальнейшем.

Но затем кто-то резонно спросил, почему нельзя выбрать несколько типов оповещений сразу? Ведь если вдруг в вашем доме начался пожар, вы бы хотели получить оповещения по всем каналам, не так ли?

Вы попытались реализовать все возможные комбинации подклассов оповещений.

После того, как вы добавили первый десяток классов, стало ясно, что такой подход **невероятно раздувает код программы**.

Нужен другой способ комбинирования поведения объектов, который не приводит к взрыву количества подклассов.



Решение

Наследование — это первое, что приходит в голову многим программистам, когда нужно расширить какое-то существующее поведение. Но механизм наследования имеет несколько досадных проблем.

- **Он статичен.** Вы не можете изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс.
- **Он не разрешает наследовать поведение нескольких классов одновременно.** Из-за этого приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.

Одним из способов обойти эти проблемы является замена наследования агрегацией либо композицией. Это когда один объект содержит ссылку на другой и делегирует ему работу, вместо того чтобы самому наследовать его поведение.

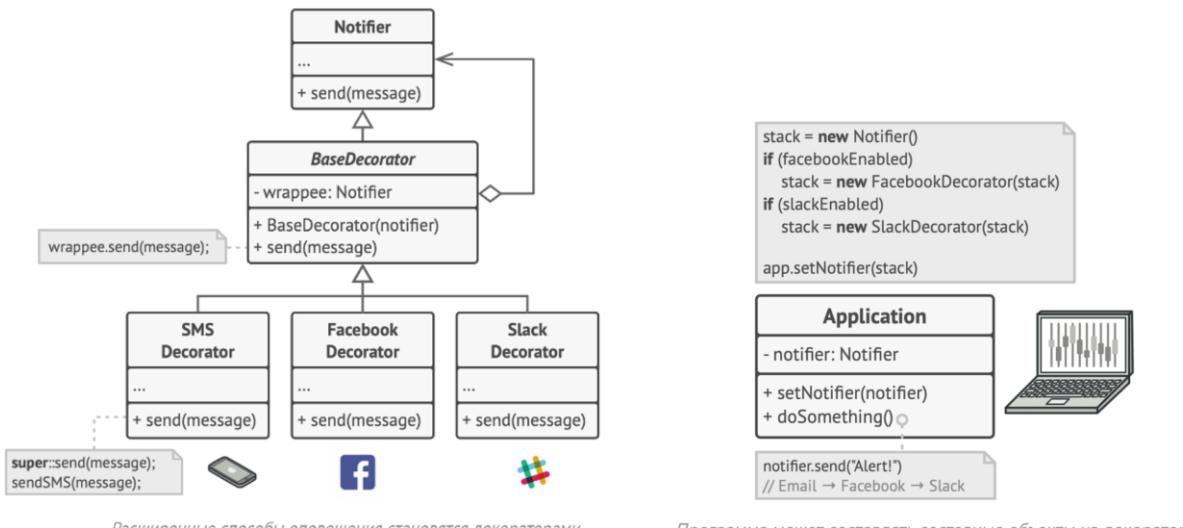
*Композиция — это более строгий вариант агрегации, при котором компоненты не могут жить без контейнера.

Как раз на этом принципе построен паттерн Декоратор.

Декоратор имеет альтернативное название — обёртка. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу.

В примере с оповещениями оставим в базовом классе простую отправку по электронной почте, а расширенные способы отправки сделаем декораторами.



Расширенные способы оповещения становятся декораторами.

Программа может составлять составные объекты из декораторов.

Сторонняя программа, выступающая клиентом, во время первичной настройки будет заворачивать объект оповещений в те обёртки, которые соответствуют желаемому способу оповещения.

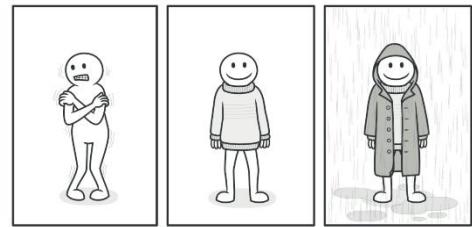
Последняя обёртка в списке и будет тем объектом, с которым клиент будет работать в остальное время. Для остального клиентского кода, по сути, ничего не изменится, ведь **все обёртки имеют точно такой же интерфейс**, что и базовый класс оповещений.

Таким же образом можно изменять не только способ доставки оповещений, но и форматирование, список адресатов и так далее. К тому же клиент может «до-обернуть» объект любыми другими обёртками, когда ему захочется.

Любая одежда — это аналог Декоратора. Применяя Декоратор, **вы не меняете первоначальный класс и не создаёте дочерних классов**.

Так и с одеждой — надевая свитер, вы не перестаёте быть собой, но **получаете новое свойство** — защиту от холода.

Вы можете пойти дальше и **надеть сверху ещё один декоратор** — плащ, чтобы защититься и от дождя.



Одежду можно надевать слоями, получая комбинированный эффект.

Структур и псевдокод тут → <https://refactoring.guru/ru/design-patterns/decorator>

Когда применять?

1. Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.

Объекты помещают в обёртки, имеющие дополнительные поведения. Обёртки и сами объекты имеют одинаковый интерфейс, поэтому клиентам без разницы, с чем работать — с обычным объектом данных или с обёрнутым.

2. Когда нельзя расширить обязанности объекта с помощью наследования.

Во многих языках программирования есть ключевое слово `final`, которое может заблокировать наследование класса. Расширить такие классы можно только с помощью Декоратора.

Шаги реализации

- Убедитесь, что в вашей задаче есть один основной компонент и несколько опциональных дополнений или надстроек над ним.
- Создайте интерфейс компонента, который описывал бы общие методы как для основного компонента, так и для его дополнений.
- Создайте класс конкретного компонента и поместите в него основную бизнес-логику.
- Создайте базовый класс декораторов. Он должен иметь поле для хранения ссылки на вложенный объект-компонент. Все методы базового декоратора должны делегировать действие вложенному объекту.
- И конкретный компонент, и базовый декоратор должны следовать одному и тому же интерфейсу компонента.
- Теперь создайте классы конкретных декораторов, наследуя их от базового декоратора. Конкретный декоратор должен выполнять свою добавочную функцию, а затем (или перед этим) вызывать эту же операцию обёрнутого объекта.
- Клиент берёт на себя ответственность за конфигурацию и порядок обёртывания объектов.

⊕ Преимущества и недостатки

- ✓ Большая гибкость, чем у наследования.
- ✓ Позволяет добавлять обязанности на лету.
- ✓ Можно добавлять несколько новых обязанностей сразу.
- ✓ Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.
- ✗ Трудно конфигурировать многократно обёрнутые объекты.
- ✗ Обилие крошечных классов.

↔ Отношения с другими паттернами

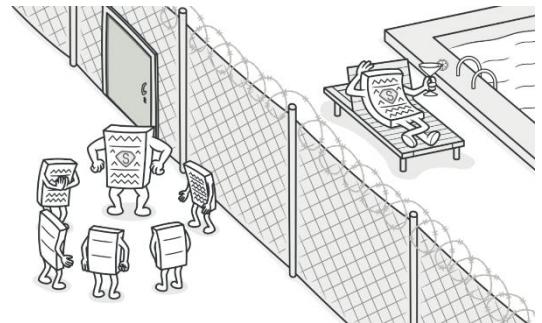
- **Адаптер** меняет интерфейс существующего объекта. **Декоратор** улучшает другой объект без изменения его интерфейса. Причём **Декоратор** поддерживает рекурсивную вложенность, чего не скажешь об **Адаптере**.
- **Адаптер** предоставляет классу альтернативный интерфейс. **Декоратор** предоставляет расширенный интерфейс. **Заместитель** предоставляет тот же интерфейс.
- **Цепочка обязанностей** и **Декоратор** имеют очень похожие структуры. Оба паттерна базируются на принципе рекурсивного выполнения операции через серию связанных объектов. Но есть и несколько важных отличий.
- **Компоновщик** и **Декоратор** имеют похожие структуры классов из-за того, что оба построены на рекурсивной вложенности. Она позволяет связать в одну структуру бесконечное количество объектов.
- Архитектура, построенная на **Компоновщиках** и **Декораторах**, часто может быть улучшена за счёт внедрения **Прототипа**. Он позволяет клонировать сложные структуры объектов, а не собирать их заново.
- **Стратегия** меняет поведение объекта «изнутри», а **Декоратор** изменяет его «снаружи».
- **Декоратор** и **Заместитель** имеют схожие структуры, но разные назначения. Они похожи тем, что оба построены на принципе композиции и делегируют работу другим объектам. Паттерны отличаются тем, что **Заместитель** сам управляет жизнью сервисного объекта, а обёртывание **Декораторов** контролируется клиентом.

Расскажите про паттерн Заместитель (Proxy).

Заместитель — это структурный паттерн проектирования, который 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.

См. также ниже инфо про паттерны Spring.

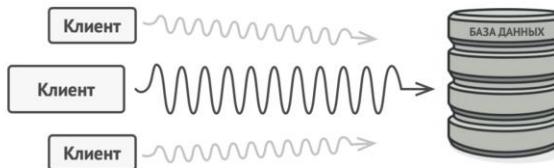


Проблема

Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.

Мы могли бы создавать этот объект не в самом начале программы, а только тогда, когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода.

В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.



Запросы к базе данных могут быть очень медленными.



Заместитель «притворяется» базой данных, ускоряя работу за счёт ленивой инициализации и кэширования повторяющихся запросов.

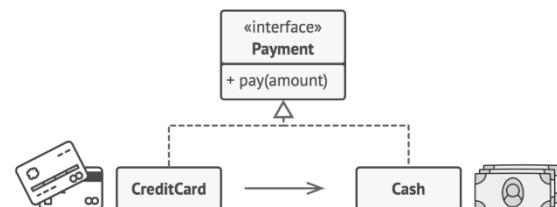
Решение

Паттерн Заместитель предлагает создать **новый класс-дублёр, имеющий тот же интерфейс**, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.

Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одноковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.

Платёжная карточка — это **заместитель пачки наличных**. И карточка, и наличные имеют **общий интерфейс** — ими можно оплачивать товары.

Для покупателя польза в том, что не надо таскать с собой тонны наличных, а владелец магазина рад, что ему не нужно делать дорогостоящую инкассацию наличности в банк — деньги поступают к нему на счёт напрямую.



Структура и псевдокод по ссылке →
<https://refactoring.guru/ru/design-patterns/proxy>

Платёжной картой можно расплачиваться, как и наличными.

Когда применять?

- Ленивая инициализация** (виртуальный прокси). Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных.

Вместо того, чтобы грузить данные сразу после старта программы, можно сэкономить ресурсы и создать объект тогда, когда он действительно понадобится.

- Защита доступа** (защищающий прокси). Когда в программе есть разные типы пользователей, и вам хочется защищать объект от неавторизованного доступа. Например, если ваши объекты — это важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).

Прокси может проверять доступ при каждом вызове и передавать выполнение служебному объекту, если доступ разрешён.

- Локальный запуск сервиса** (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.

В этом случае заместитель транслирует запросы клиента в вызовы по сети в протоколе, понятном удалённому сервису.

- Логирование запросов** (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту.

Заместитель может сохранять историю обращения клиента к сервисному объекту.

5. **Кеширование объектов** («умная» ссылка). Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.

Заместитель может **подсчитывать количество ссылок на сервисный объект**, которые были отданы клиенту и остаются активными. Когда все ссылки освобождаются, можно будет освободить и сам сервисный объект (например, закрыть подключение к базе данных).

Кроме того, Заместитель может отслеживать, не менял ли клиент сервисный объект. Это позволит **использовать объекты повторно и экономить ресурсы**, особенно если речь идёт о больших прожорливых сервисах.

Шаги реализации

- Определите интерфейс, который бы сделал заместитель и оригинальный объект взаимозаменяемыми.
- Создайте класс заместителя. Он должен содержать ссылку на сервисный объект. Чаще всего, сервисный объект создаётся самим заместителем. В редких случаях заместитель получает готовый сервисный объект от клиента через конструктор.
- Реализуйте методы заместителя в зависимости от его предназначения. В большинстве случаев, проделав какую-то полезную работу, методы заместителя должны передать запрос сервисному объекту.
- Подумайте о введении фабрики, которая решала бы, какой из объектов создавать — заместитель или реальный сервисный объект. Но, с другой стороны, эта логика может быть помещена в создающий метод самого заместителя.
- Подумайте, не реализовать ли вам ленивую инициализацию сервисного объекта при первом обращении клиента к методам заместителя.

⊕ Преимущества и недостатки

- | | |
|---|--|
| <p>✓ Позволяет контролировать сервисный объект незаметно для клиента.</p> <p>✓ Может работать, даже если сервисный объект ещё не создан.</p> <p>✓ Может контролировать жизненный цикл служебного объекта.</p> | <p>✗ Усложняет код программы из-за введения дополнительных классов.</p> <p>✗ Увеличивает время отклика от сервиса.</p> |
|---|--|

↔ Отношения с другими паттернами

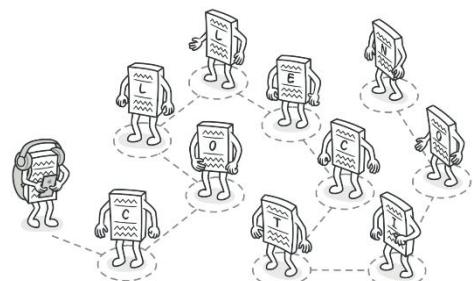
- **Адаптер** предоставляет классу альтернативный интерфейс. **Декоратор** предоставляет расширенный интерфейс. **Заместитель** предоставляет тот же интерфейс.
- **Фасад** похож на **Заместитель** тем, что замещает сложную подсистему и может сам её инициализировать. Но в отличие от **Фасада**, **Заместитель** имеет тот же интерфейс, что его служебный объект, благодаря чему их можно взаимозаменять.
- **Декоратор** и **Заместитель** имеют схожие структуры, но разные назначения. Они похожи тем, что оба построены на принципе композиции и делегируют работу другим объектам. Паттерны отличаются тем, что **Заместитель** сам управляет жизнью сервисного объекта, а обёртывание **Декораторов** контролируется клиентом.

Расскажите про паттерн Итератор (Iterator).

Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Проблема

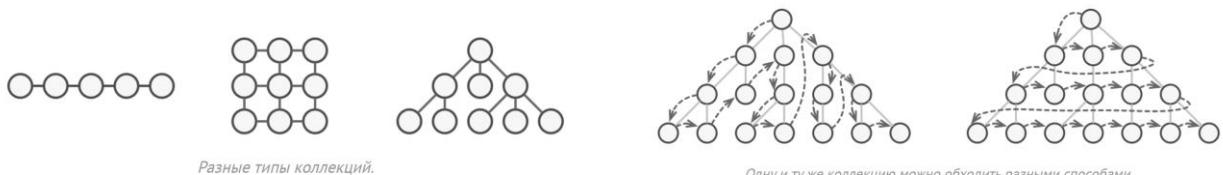
Коллекции — самая распространённая структура данных, которую вы можете встретить в программировании. Это набор объектов, собранный в одну кучу по каким-то критериям.



Большинство коллекций выглядят как обычный список элементов. Но есть и экзотические коллекции, построенные на основе деревьев, графов и других сложных структур данных.

Но как бы ни была структурирована коллекция, пользователь должен иметь возможность последовательно обходить её элементы, чтобы проделывать с ними какие-то действия.

Но каким способом следует перемещаться по сложной структуре данных? Например, сегодня может быть достаточным обход дерева в глубину, но завтра потребуется возможность перемещаться по дереву в ширину. А на следующей неделе и того хуже — понадобится обход коллекции в случайном порядке.

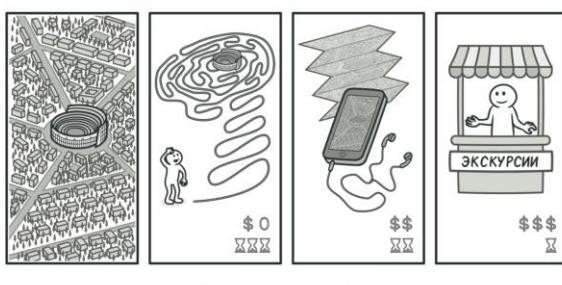


Добавляя всё новые алгоритмы в код коллекции, вы понемногу размываете её основную задачу, которая заключается в эффективном хранении данных. Некоторые алгоритмы могут быть и вовсе слишком «заточены» под определённое приложение и смотреться дико в общем классе коллекции.

Решение

Идея паттерна Итератор состоит в том, чтобы **вынести поведение** обхода коллекции из самой коллекции в **отдельный класс**.

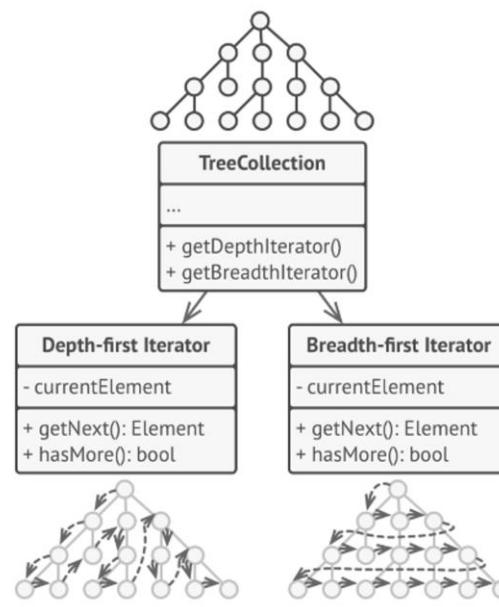
Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом. К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя существующий код коллекции.



Варианты прогулок по Риму.

Рим выступает коллекцией достопримечательностей, а ваш мозг, навигатор или гид — итератором по коллекции.

Вы, как клиентский код, можете выбрать один из итераторов, отталкиваясь от решаемой задачи и доступных ресурсов.



Итераторы содержат код обхода коллекции. Одну коллекцию могут обходить сразу несколько итераторов.

Структура и псевдокод по ссылке → <https://refactoring.guru/ru/design-patterns/iterator>

Когда применять?

1. Когда у вас есть сложная структура данных, и вы хотите скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).

Итератор предоставляет клиенту всего несколько простых методов перебора элементов коллекции. Это не только упрощает доступ к коллекции, но и защищает её данные от неосторожных или злоумышленных действий.

2. Когда нужно иметь несколько вариантов обхода одной и той же структуры данных.

Нетривиальные алгоритмы обхода структуры данных могут иметь довольно объёмный код. Этот код будет захламлять всё вокруг — будь то сам класс коллекции или часть бизнес-логики программы. Применив итератор, вы можете выделить код обхода структуры данных в собственный класс, упростив поддержку остального кода.

3. Когда хочется иметь единий интерфейс обхода различных структур данных.

Итератор позволяет вынести реализации различных вариантов обхода в подклассы. Это позволит легко взаимо-заменять объекты итераторов, в зависимости от того, с какой структурой данных приходится работать.

Шаги реализации

- Создайте общий интерфейс итераторов. Обязательный минимум — это операция получения следующего элемента коллекции. Но для удобства можно предусмотреть и другое. Например, методы для получения предыдущего элемента, текущей позиции, проверки окончания обхода и прочие.
- Создайте интерфейс коллекции и опишите в нём метод получения итератора. Важно, чтобы сигнатура метода возвращала общий интерфейс итераторов, а не один из конкретных итераторов.
- Создайте классы конкретных итераторов для тех коллекций, которые нужно обходить с помощью паттерна. Итератор должен быть привязан только к одному объекту коллекции. Обычно эта связь устанавливается через конструктор.
- Реализуйте методы получения итератора в конкретных классах коллекций. Они должны создавать новый итератор того класса, который способен работать с данным типом коллекции. Коллекция должна передавать ссылку на собственный объект в конструктор итератора.
- В клиентском коде и в классах коллекций не должно оставаться кода обхода элементов. Клиент должен получать новый итератор из объекта коллекции каждый раз, когда ему нужно перебрать её элементы.

⊕ Преимущества и недостатки

- | | |
|---|--|
| <ul style="list-style-type: none">✓ Упрощает классы хранения данных.✓ Позволяет реализовать различные способы обхода структуры данных.✓ Позволяет одновременно перемещаться по структуре данных в разные стороны. | <ul style="list-style-type: none">✗ Не оправдан, если можно обойтись простым циклом. |
|---|--|

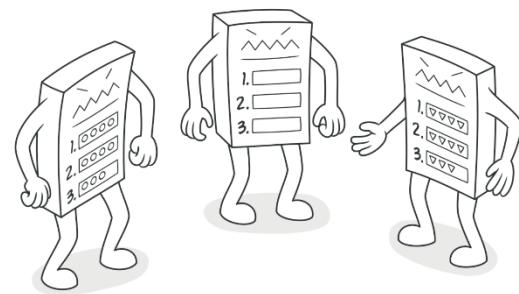
↔ Отношения с другими паттернами

- Вы можете обходить дерево Компоновщика, используя Итератор.
- Фабричный метод можно использовать вместе с Итератором, чтобы подклассы коллекций могли создавать подходящие им итераторы.
- Снимок можно использовать вместе с Итератором, чтобы сохранить текущее состояние обхода структуры данных и вернуться к нему в будущем, если потребуется.
- Посетитель можно использовать совместно с Итератором. Итератор будет отвечать за обход структуры данных, а Посетитель — за выполнение действий над каждым её компонентом.

Расскажите про паттерн Шаблонный метод (Template Method).

Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы.

Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



Проблема

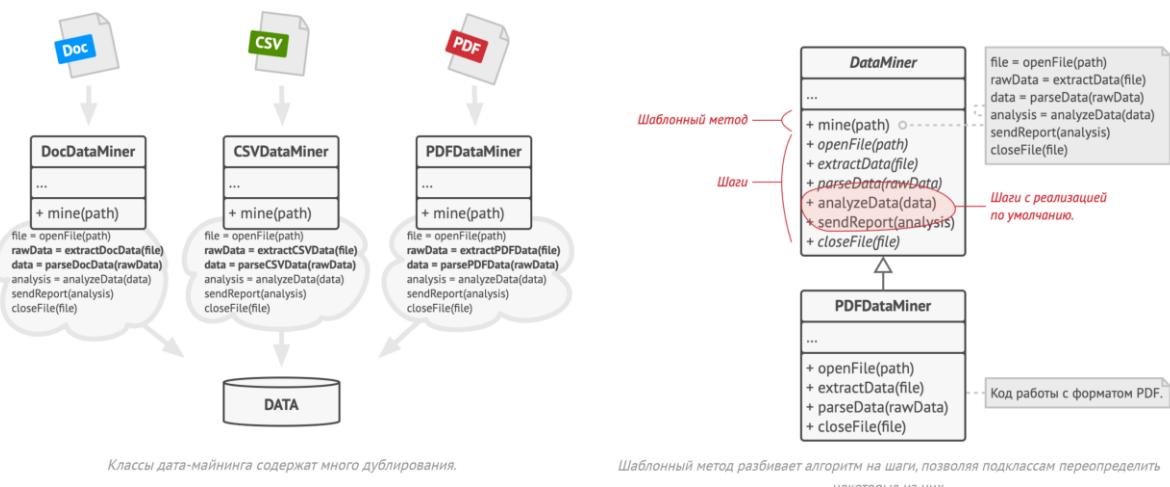
Вы пишете программу для дата-майнинга в офисных документах. Пользователи будут загружать в неё документы в разных форматах (PDF, DOC, CSV), а программа должна извлекать из них полезную информацию.

В первой версии вы ограничились только обработкой DOC-файлов. В следующей версии добавили поддержку CSV. А через месяц прикрутили работу с PDF-документами.

Решение

Паттерн Шаблонный метод предлагает:

- разбить алгоритм на последовательность шагов
- описать эти шаги в отдельных методах
- вызывать их в одном шаблонном методе друг за другом



Это позволит подклассам переопределять некоторые шаги алгоритма, оставляя без изменений его структуру и остальные шаги, которые для этого подкласса не так важны.

В нашем примере с дата-майнингом мы можем создать **общий базовый класс для всех трёх алгоритмов**. Этот класс будет состоять из шаблонного метода, который последовательно вызывает шаги разбора документов.

Для начала шаги шаблонного метода можно сделать **абстрактными**. Из-за этого все подклассы должны будут реализовать каждый из шагов по-своему. В нашем случае все подклассы и так содержат реализацию каждого из шагов, поэтому ничего дополнительного делать не нужно.

По-настоящему важным является следующий этап. Теперь мы можем **определить общее для всех классов поведение и вынести его в суперкласс**. В нашем примере шаги открытия, считывания и закрытия могут отличаться для разных типов документов, поэтому останутся абстрактными. А вот одинаковый для всех типов документов код обработки данных переедет в базовый класс.

Как видите, у нас получилось **два вида шагов: абстрактные**, которые каждый подкласс обязательно должен реализовать, а также **шаги с реализацией по умолчанию**, которые можно переопределять в подклассах, но не обязательно.

Но есть и **третий тип шагов — ХУКИ** (крючки): их не обязательно переопределять, но они не содержат никакого кода, выглядя как обычные методы. Шаблонный метод останется рабочим, даже если ни один подкласс не переопределит такой хук.

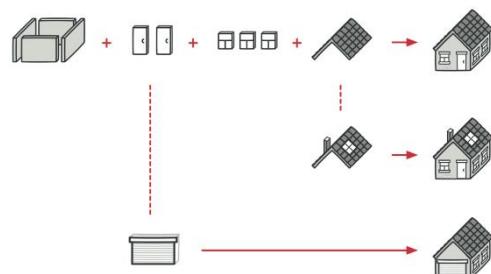
Хук даёт подклассам дополнительные точки «вклинивания» в шаблонный метод.

Строители используют подход, похожий на шаблонный метод при строительстве типовых домов.

У них есть **основной архитектурный проект**, в котором расписаны шаги строительства: заливка фундамента, постройка стен, перекрытие крыши, установка окон и так далее.

Но, несмотря на стандартизацию каждого этапа, **строители могут вносить небольшие изменения на любом из этапов**, чтобы сделать дом чутьоки непохожим на другие.

Структура и псевдокод по ссылке →
<https://refactoring.guru/ru/design-patterns/template-method>



Проект типового дома могут немного изменить по желанию клиента.

Когда применять?

1. Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.

Шаблонный метод позволяет подклассам расширять определённые шаги алгоритма через наследование, не меняя при этом структуру алгоритмов, объявленную в базовом классе.

2. Когда у вас есть несколько классов, делающих одно и то же с незначительными отличиями. Если вы редактируете один класс, то приходится вносить такие же правки и в остальные классы.

Паттерн шаблонный метод предлагает создать для похожих классов общий суперкласс и оформить в нём главный алгоритм в виде шагов. Отличающиеся шаги можно переопределить в подклассах. Это позволит убрать дублирование кода в нескольких классах с похожим поведением, но отличающихся в деталях.

Шаги реализации

- Изучите алгоритм и подумайте, можно ли его разбить на шаги. Прикиньте, какие шаги будут стандартными для всех вариаций алгоритма, а какие — изменяющимися.
- Создайте абстрактный базовый класс. Определите в нём шаблонный метод. Этот метод должен состоять из вызовов шагов алгоритма. Имеет смысл сделать шаблонный метод финальным, чтобы подклассы не могли переопределить его (если ваш язык программирования это позволяет).
- Добавьте в абстрактный класс методы для каждого из шагов алгоритма. Вы можете сделать эти методы абстрактными или добавить какую-то реализацию по

умолчанию. В первом случае все подклассы должны будут реализовать эти методы, а во втором — только если реализация шага в подклассе отличается от стандартной версии.

- Подумайте о введении в алгоритм хуков. Чаще всего, хуки располагают между основными шагами алгоритма, а также до и после всех шагов.
- Создайте конкретные классы, унаследовав их от абстрактного класса. Реализуйте в них все недостающие шаги и хуки.

⊕⊕ Преимущества и недостатки

- | | |
|--|---|
| <p>✓ Облегчает повторное использование кода.</p> | <p>✗ Вы жёстко ограничены скелетом существующего алгоритма.</p> |
| <p>✗ Вы можете нарушить <u>принцип подстановки Барбары Лисков</u>, изменяя базовое поведение одного из шагов алгоритма через подкласс.</p> | |
| <p>✗ С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.</p> | |

↔ Отношения с другими паттернами

- Фабричный метод можно рассматривать как частный случай Шаблонного метода. Кроме того, Фабричный метод нередко бывает частью большого класса с Шаблонными методами.
- 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.

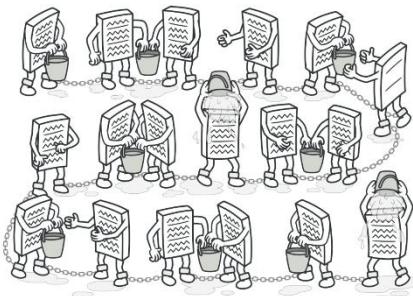
Расскажите про паттерн Цепочка обязанностей (Chain of Responsibility).

BeanPostProcessor позволяет настраивать бины ДО того, как они попадают в контейнер.

Задействован паттерн Chain of Responsibility (цепочка обязанностей) – поведенческий шаблон проектирования, предназначенный для организации в системе уровней ответственности. Также этот паттерн используется при построении цепочки фильтров в Spring Security (процесс аутентификации).

Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков.

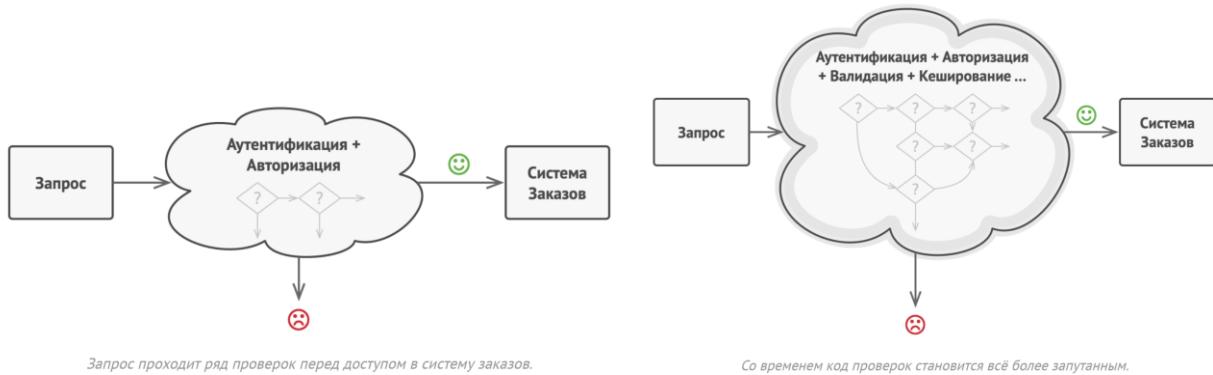
Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.



Проблема

Представьте, что вы делаете систему приёма онлайн-заказов. Вы хотите ограничить к ней доступ так, чтобы только авторизованные пользователи могли создавать заказы. Кроме того, определённые пользователи, владеющие правами администратора, должны иметь полный доступ к заказам.

Вы быстро сообразили, что эти проверки нужно выполнять последовательно. Ведь пользователя можно попытаться «залогинить» в систему, если его запрос содержит логин и пароль. Но если такая попытка не удалась, то проверять расширенные права доступа попросту не имеет смысла.



На протяжении следующих нескольких месяцев вам пришлось добавить ещё несколько таких последовательных проверок.

- Кто-то резонно заметил, что неплохо бы проверять данные, передаваемые в запросе перед тем, как вносить их в систему — вдруг запрос содержит данные о покупке несуществующих продуктов.
- Кто-то предложил блокировать массовые отправки формы с одним и тем же логином, чтобы предотвратить подбор паролей ботами.
- Кто-то заметил, что форму заказа неплохо бы доставать из кеша, если она уже была однажды показана.

С каждой новой «фичей» код проверок, выглядящий как большой клубок условных операторов, всё больше и больше раздувался.

При изменении одного правила приходилось трогать код всех проверок. А для того, чтобы применить проверки к другим ресурсам, пришлось продублировать их код в других классах. Поддерживать такой код стало не только очень хлопотно, но и затратно. И вот в один прекрасный день вы получаете задачу рефакторинга...

Решение

Как и многие другие поведенческие паттерны, Цепочка обязанностей базируется на том, чтобы превратить отдельные поведения в объекты. В нашем случае каждая проверка переедет в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы.

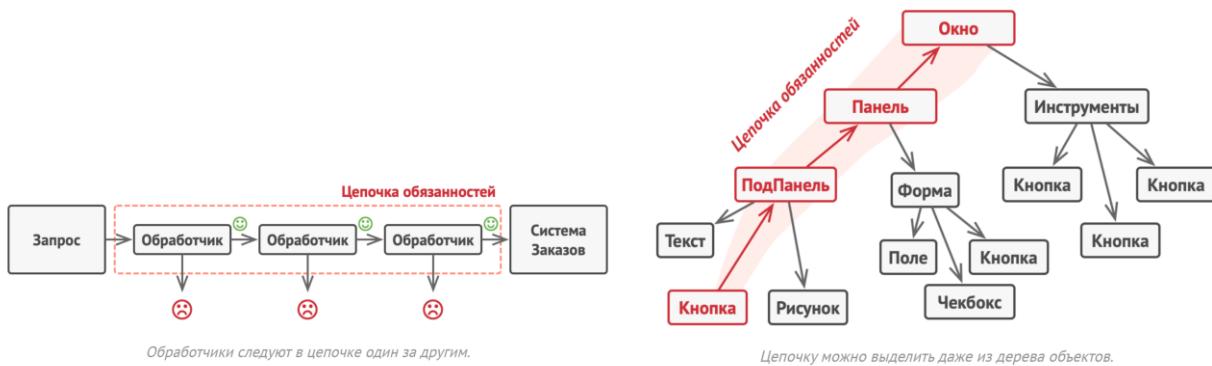
А теперь по-настоящему важный этап. **Паттерн предлагает связать объекты обработчиков в одну цепь.** Каждый из них будет иметь ссылку на следующий обработчик в цепи. Таким образом, при получении запроса обработчик сможет не только сам что-то с ним сделать, но и передать обработку следующему объекту в цепочке.

Передавая запросы в первый обработчик цепочки, вы можете быть уверены, что все объекты в цепи смогут его обработать. При этом длина цепочки не имеет никакого значения. И последний штрих. Обработчик не обязательно должен передавать запрос дальше, причём эта особенность может быть использована по-разному.

В примере с фильтрацией доступа обработчики прерывают дальнейшие проверки, если текущая проверка не прошла. Ведь нет смысла тратить попусту ресурсы, если и так понятно, что с запросом что-то не так.

Но есть и другой подход, при котором обработчики прерывают цепь только когда они могут обработать запрос. В этом случае запрос движется по цепи, пока не найдётся обработчик, который может его обработать. Очень часто такой подход используется для передачи событий, создаваемых классами графического интерфейса в результате взаимодействия с пользователем.

Например, когда пользователь кликает по кнопке, программа выстраивает цепочку из объекта этой кнопки, всех её родительских элементов и общего окна приложения на конце. Событие клика передаётся по этой цепи до тех пор, пока не найдётся объект, способный его обработать. Этот пример примечателен ещё и тем, что цепочку всегда можно выделить из древовидной структуры объектов, в которую обычно и свёрнуты элементы пользовательского интерфейса.



Очень важно, чтобы все объекты цепочки имели общий интерфейс. Обычно каждому конкретному обработчику достаточно знать только то, что следующий объект в цепи имеет метод выполнить. Благодаря этому связи между объектами цепочки будут более гибкими. Кроме того, вы сможете формировать цепочки на лету из разнообразных объектов, не привязываясь к конкретным классам.

Структура и псевдокод по ссылке → <https://refactoring.guru/ru/design-patterns/chain-of-responsibility>

Когда применять?

1. Когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно, какие конкретно запросы будут приходить и какие обработчики для них понадобятся.

С помощью Цепочки обязанностей вы можете связать потенциальных обработчиков в одну цепь и при получении запроса поочерёдно спрашивать каждого из них, не хочет ли он обработать запрос.

2. Когда важно, чтобы обработчики выполнялись один за другим в строгом порядке.

Цепочка обязанностей позволяет запускать обработчиков последовательно один за другим в том порядке, в котором они находятся в цепочке.

3. Когда набор объектов, способных обработать запрос, должен задаваться динамически.

В любой момент вы можете вмешаться в существующую цепочку и переназначить связи так, чтобы убрать или добавить новое звено.

Шаги реализации

- Создайте интерфейс обработчика и опишите в нём основной метод обработки.

Продумайте, в каком виде клиент должен передавать данные запроса в обработчик. Самый гибкий способ — превратить данные запроса в объект и передавать его целиком через параметры метода обработчика.

- Имеет смысл создать абстрактный базовый класс обработчиков, чтобы не дублировать реализацию метода получения следующего обработчика во всех конкретных обработчиках.

Добавьте в базовый обработчик поле для хранения ссылки на следующий объект цепочки. Установливайте начальное значение этого поля через конструктор. Это сделает объекты обработчиков неизменяемыми. Но если программа предполагает динамическую перестройку цепочек, можете добавить и сеттер для поля.

Реализуйте базовый метод обработки так, чтобы он перенаправлял запрос следующему объекту, проверив его наличие. Это позволит полностью скрыть поле-ссылку от подклассов, дав им возможность передавать запросы дальше по цепи, обращаясь к родительской реализации метода.

- Один за другим создайте классы конкретных обработчиков и реализуйте в них методы обработки запросов. При получении запроса каждый обработчик должен решить:
 - Может ли он обработать запрос или нет?
 - Следует ли передать запрос следующему обработчику или нет?
- Клиент может собирать цепочку обработчиков самостоятельно, опираясь на свою бизнес-логику, либо получать уже готовые цепочки извне. В последнем случае цепочки собираются фабричными объектами, опираясь на конфигурацию приложения или параметры окружения.
- Клиент может посыпать запросы любому обработчику в цепи, а не только первому. Запрос будет передаваться по цепочке до тех пор, пока какой-то обработчик не откажется передавать его дальше, либо когда будет достигнут конец цепи.
- Клиент должен знать о динамической природе цепочки и быть готов к таким случаям:
 - Цепочка может состоять из единственного объекта.
 - Запросы могут не достигать конца цепи.
 - Запросы могут достигать конца, оставаясь необработанными.

⊕ Преимущества и недостатки

- | | |
|--|--|
| <ul style="list-style-type: none">✓ Уменьшает зависимость между клиентом и обработчиками.✓ Реализует принцип единственной обязанности.✓ Реализует принцип открытости/закрытости. | <ul style="list-style-type: none">✗ Запрос может остаться никем не обработанным. |
|--|--|

- Цепочка обязанностей и Декоратор имеют очень похожие структуры. Оба паттерна базируются на принципе рекурсивного выполнения операции через серию связанных объектов. Но есть и несколько важных отличий.

Обработчики в Цепочке обязанностей могут выполнять произвольные действия, независимые друг от друга, а также в любой момент прерывать дальнейшую передачу по цепочке. С другой стороны Декораторы расширяют какое-то определённое действие, не ломая интерфейс базовой операции и не прерывая выполнение остальных декораторов.

↔ Отношения с другими паттернами

- Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:
 - Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
 - Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.
 - Посредник убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
 - Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
 - Цепочку обязанностей часто используют вместе с Компоновщиком. В этом случае запрос передаётся от дочерних компонентов к их родителям.
 - Обработчики в Цепочке обязанностей могут быть выполнены в виде Команд. В этом случае множество разных операций может быть выполнено над одним и тем же контекстом, коим является запрос.
- Но есть и другой подход, в котором сам запрос является Командой, посланной по цепочке объектов. В этом случае одна и та же операция может быть выполнена над множеством разных контекстов, представленных в виде цепочки.

Какие паттерны используются в Spring Framework?

Singleton - Creating beans with default scope.

Factory - Bean Factory classes

Prototype - Bean scopes

Adapter - Spring Web and Spring MVC

Proxy - Spring Aspect Oriented Programming support

Template Method - JdbcTemplate, HibernateTemplate etc

Front Controller - Spring MVC DispatcherServlet

Data Access Object - Spring DAO support Dependency Injection and Aspect Oriented Programming

Паттерн Команда (Command) — поведенческий шаблон проектирования, используемый при ООП, для обработки команды в виде объекта и представляющий ДЕЙСТВИЕ. То есть Объект команды заключает в себе само действие и его параметры.

Используется, например, при инъекции прототипа в синглтон
<https://itsobes.ru/JavaSobes/kak-rabotaet-inyeckiya-prototipa-v-singlton/>

Еще пример из кора-1: могут ли быть приватные конструкторы? Для чего они?

Приватный конструктор запрещает создание экземпляра класса вне методов самого класса, например, что гарантировать существование только одного объекта определённого класса, предположим какого-то ресурса, например БД.

Паттерн Singleton (Одиночка): объект в программе должен быть один и к нему есть глобальный доступ.

BeanFactory – это интерфейс, центральный игрок, отвечает за создание и хранение всех объектов (синглтонов).

BeanFactory — это реализация **паттерна Фабрика**, его функциональность покрывает создание бинов. Так как эта фабрика знает многое об объектах приложения, то она может создавать связи между объектами на этапе создания экземпляра.

Паттерн Заместитель (Proxy) предоставляет объект-заместитель, который управляет доступом к другому объекту. То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его (перехватывать все вызовы).

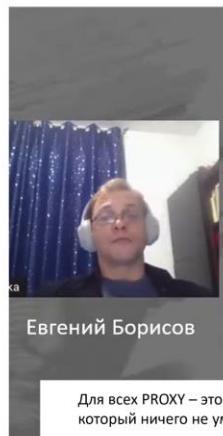
В Spring прокси просто обращивает bean, он может добавить логику до и после выполнения методов.

Spring AOP использует либо динамические прокси JDK (на основе интерфейса), либо CGLIB (на основе класса) для создания прокси для данного целевого объекта.

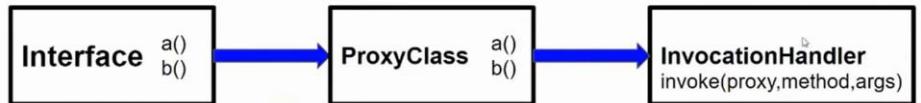
Динамические прокси-серверы JDK предпочтительнее, когда у вас есть выбор.

Если целевой объект для проксирования реализует хотя бы один **интерфейс**, то будет использоваться динамический прокси JDK. Все интерфейсы, реализованные целевым типом, будут проксированы.

<https://youtu.be/DKNDU7OiyJs?t=541>



Как работает dynamic proxy?

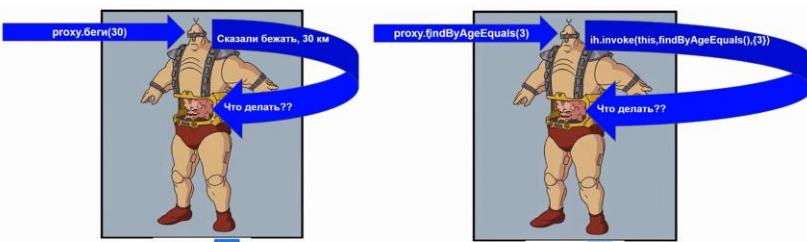


1. Пишем интерфейс

2. Создаём Proxy Class с помощью Java библиотеки, который будет **имплементировать те же самые методы**

3. Реализовывать эти методы будет InvocationHandler, который мы при создании Proxy передаём ему. Тут будет один метод, кот. принимает сигнатуру метода, кот. вызвали у прокси класса, аргументы, которые ему передали и сам объект прокси.

Для всех PROXY – это умный, классный объект, который офигенно работает, а по факту это **ТУПОЙ** объект, который ничего не умеет делать сам, а за него думает **InvocationHandler**.



Если целевой объект **не реализует** интерфейсов, будет создан прокси-сервер CGLIB.

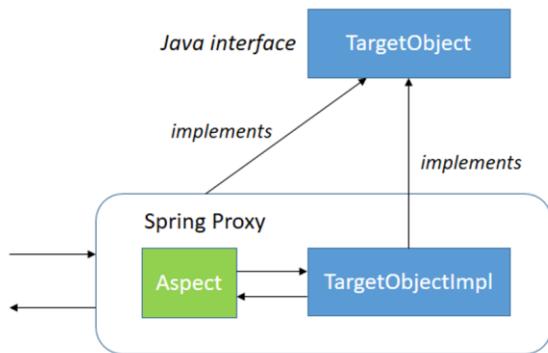
Если вы хотите принудительно использовать проксирование CGLIB (например, проксировать каждый метод, определенный для целевого объекта, а не только те, которые реализованы его интерфейсами), вы можете это сделать.

Чтобы принудительно использовать прокси CGLIB, установите для атрибута **proxy-target-class** элемента **<aop:config>** значение **true**.

Spring AOP Process

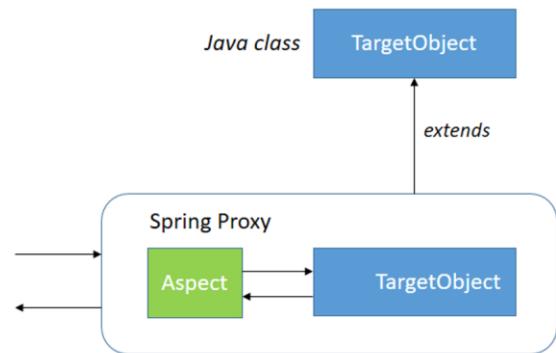
JDK Proxy (interface based)

JDK dynamic proxy - Spring AOP по умолчанию использует JDK dynamic proxy, которые позволяют проксировать любой интерфейс (или набор интерфейсов). Если целевой объект реализует хотя бы один интерфейс, то будет использоваться динамический прокси JDK.

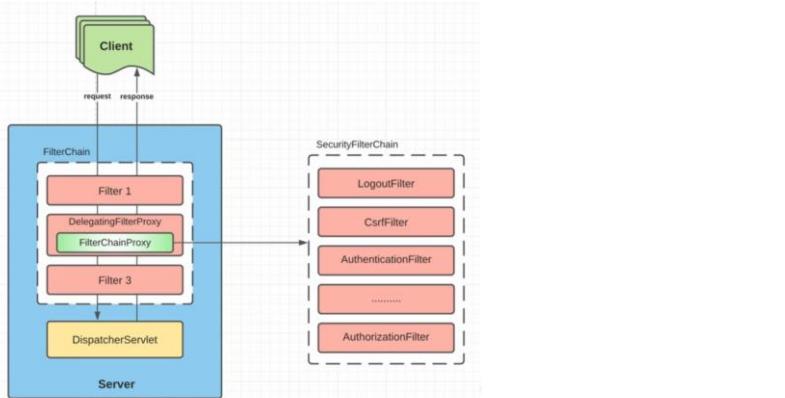


CGLib Proxy (class based)

CGLIB-прокси - используется по умолчанию, если бизнес-объект НЕ реализует ни одного интерфейса.



Цепочка фильтров в Spring Security.



Chain of responsibility

```

@Service
public class MainHandler {
    @Autowired
    private List<Handler> handlers;
    Choose Bean
    *handler1 (Handler1.java) spring-patterns
    *handler2 (Handler2.java) spring-patterns
    *handler3 (Handler3.java) spring-patterns

    public void handle(DataObject t) {
        handlers.forEach(handler -> handler.handle(t));
    }
}

```

We love you, Switch ...

```

public DistribHandler resolveValue(FieldValue value) {
    switch (value.getValueOfAc)
        switch (documentObject.getDocumentType())
            case PDF_STORAGE:
                fileContainer = new PdfRecordFile();
                getPdffromStorage(fileContainer, d);
                break;
            case PDF_SRC:
                fileContainer = new PdfRecordFile();
                fillObjectsForPdf(fileContainer, d);
                break;
            case PDF_WS:
                fileContainer = new WsPdfRecon();
                getPdffromPdfWs(fileContainer, d);
                break;
            case LS:
                fileContainer = new PdfRecordFile();
                getPdffromListDocument(j, fileC);
                break;
            case IMAGE:
                fileContainer = new PdfRecordFile();
                getPdffromImageDocument(j, fileC);
                break;
            case FORM:
                fileContainer = new PdfRecordFile();
                getPdffromFormDocument(fileC);
                break;
        }
    ...
}

switch (value.getNumericalValue() {
    case 1:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", emailRequest.getSubject());
        break;
    case 2:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", emailRequest.getSubject());
        break;
    case 3:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", emailRequest.getSubject());
        break;
    case 4:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", emailRequest.getSubject());
        break;
    case 5:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", emailRequest.getSubject());
        break;
    case 6:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
    case 7:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
    case 8:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
    case 9:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
    case 10:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
    case 11:
        text = emailRequest.getSubject();
        break;
    case 12:
        if (resources.getBrandKey(j).isBiluhYashirQ) {
            text = format("({0} {1} {2} {3} {4} {5}", brandHebName);
        } else {
            text = format("({0} {1} {2} {3} {4} {5}", brandHebName);
        }
        break;
    case 13:
        text = format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
    case 14:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
    case 15:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
    case 16:
        text = MessageFormat.format("({0} {1} {2} {3} {4} {5}", brandHebName);
        break;
}

```

Какие паттерны используются в Hibernate?

В хибере есть Session Factory и она является самой настоящей фабрикой по производству Sessions

Domain Model – объектная модель предметной области, включающая в себя как поведение, так и данные.

Data Mapper – слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя.

Proxy применяется для ленивой загрузки.

Factory используется в SessionFactory

Что такое Dependency Injection?

Dependency Injection (внедрение зависимости) – набор паттернов и принципов разработки программного обеспечения, которые позволяют писать слабо связанный код.

В полном соответствии с принципом единой ответственности объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

Что такое «антипаттерн»? Какие антипаттерны знаете?

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

Poltergeists (полтергейсты) — это классы с ограниченной ответственностью и ролью в системе, чьё единственное предназначение — передавать информацию в другие классы.

Их эффективный жизненный цикл непродолжителен. Полтергейсты нарушают стройность архитектуры программного обеспечения, создавая избыточные (лишние) абстракции, они чрезмерно запутаны, сложны для понимания и трудны в сопровождении. Обычно такие классы задумываются как **классы-контроллеры**, которые существуют только для вызова методов других классов, зачастую в предопределенной последовательности.

Признаки появления и последствия антипаттерна:

- Избыточные межклассовые связи.
- Временные ассоциации.
- Классы без состояния (содержащие только методы и константы).
- Временные объекты и классы (с непродолжительным временем жизни).
- Классы с единственным методом, который предназначен только для создания или вызова других классов посредством временной ассоциации.
- Классы с именами методов в стиле «управления», такие как startProcess.

Типичные причины

1. Отсутствие объектно-ориентированной архитектуры (архитектор не понимает объектно-ориентированной парадигмы).
2. Неправильный выбор пути решения задачи.
3. Предположения об архитектуре приложения на этапе анализа требований (до объектно-ориентированного анализа) могут также вести к проблемам на подобии этого антипаттерна.

Внесенная сложность (Introduced complexity): Необязательная сложность дизайна. Вместо одного простого класса выстраивается целая иерархия интерфейсов и классов. Типичный пример «Интерфейс - Абстрактный класс - Единственный класс реализующий интерфейс на основе абстрактного».

Инверсия абстракции (Abstraction inversion): Сокрытие части функциональности от внешнего использования, в надежде на то, что никто не будет его использовать.

Неопределённая точка зрения (Ambiguous viewpoint): Представление модели без спецификации её точки рассмотрения.

Большой комок грязи (Big ball of mud): Система с нераспознаваемой структурой.

Божественный объект (God object): Концентрация слишком большого количества функций в одной части системы (классе).

Затычка на ввод данных (Input kludge): Забывчивость в спецификации и выполнении поддержки возможного неверного ввода.

Раздувание интерфейса (Interface bloat): Разработка интерфейса очень мощным и очень сложным для реализации.

Волшебная кнопка (Magic pushbutton): Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса. Например, написание прикладной логики в обработчиках нажатий на кнопку.

Перестыковка (Re-Coupling): Процесс внедрения ненужной зависимости.

Дымоход (Stovepipe System): Редко поддерживаемая сборка плохо связанных компонентов.

Состояние гонки (Race hazard): непредвидение возможности наступления событий в порядке, отличном от ожидаемого.

Членовредительство (Mutilation): Излишнее «затачивание» объекта под определенную очень узкую задачу таким образом, что он не способен будет работать с никакими иными, пусть и очень схожими задачами.

Сохранение или смерть (Save or die): Сохранение изменений лишь при завершении приложения.

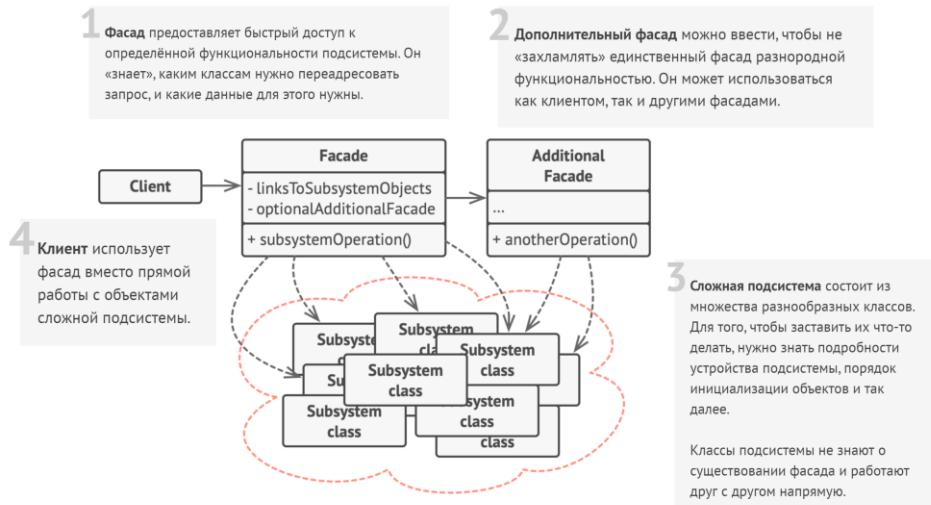
Еще что использовали на практике?

Фасад

Проблема: имеем циклические зависимости в Сервис-слое (зависимости через сеттеры и ленивая загрузка не очень хороший вариант – это долго и громоздко). **Что делать?** Сервисов в рабочем проекте бывает много и почти каждый из них зависит друг от друга. Есть и циклические зависимости. <https://refactoring.guru/ru/design-patterns/facade>

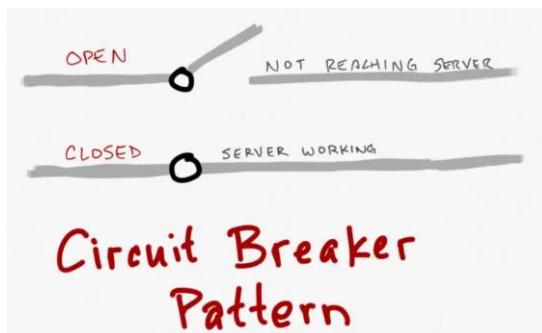
Решение: добавление дополнительного слоя Фасад (-ы) между слоем контроллеров и сервисов. Тогда любые контроллеры будут обращаться к фасаду, а каждый нужный фасад внедряет в себя нужные ему сервисы через конструктор.

Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.



Предохранитель

Стратегии обработки ошибок: Circuit Breaker pattern (автоматический выключатель, предохранитель)



Микросервисная архитектура — это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными.

Это воплощение паттернов High Cohesion и Low Coupling. Так что микросервис обязан быть независимым компонентом.

Проблема: есть некоторое приложение, которое содержит множество микросервисов. Все они общаются между собой, один из них получает данные из базы, передает другому, тот в свою очередь обрабатывает их, передает полученный id следующему сервису и так далее по цепочке.

Если в нашем приложении в каком-либо блоке произошла какая-либо ошибка (или БД), один из сервисов, который передает результат по цепочке через другие сервисы отвалился и все стало очень плохо.

Варианты решения:

1. Idempotency Key (ключ идемпотентности)

Каждый наш запрос из сервиса А мы снабжаем ключом, подписываем, говорим сервису В что вот этот запрос наш и у него такой вот идентификатор.

Для параллельной отправки запросов подойдут не все запросы. Мы должны быть уверены, что:

- + есть несколько реплик сервиса В,
- + запрос идемпотентен (то свойство, когда мы можем безопасно переотправлять запрос, т.к. мы получим один и тот же результат).

Идемпотентными запросами обладают GET, HEAD, PUT, DELETE.

- + используем Idempotency key,
- + требуется какая-либо доработка стороннего сервиса, т.к. может оказаться, что сторонний сервис (или БД) и мы никак не можем на него повлиять. В таком случае потребуется что-либо придумать на клиентской стороне, чтобы уведомить пользователя о данной проблеме.

2. Переотправка запросов (Retry pattern)

Мы будем переотправлять запрос до тех пор, пока он не окажется успешным.

Сколько же раз стоит повторять ?

Перед тем как повторять отправку, можно проверить, что это за ошибка. Если пользователь ввел неверные данные карты, то сколько бы мы не пробовали раз повторить, то все равно не получится совершить удачный запрос. Если исчерпали какое-то кол-во попыток, то не стоит пробовать отправлять дальше до бесконечности. Например такую ошибку можно сразу показать. Если возникла ошибка timeout exception или too many request, тогда можем попытаться отправить снова.

Здесь лучше дать серверу некоторое время чтобы он восстановил свою работу (условно 1–2 секунды или более) и снова повторить запрос.



Стратегии ожидания бывает следующие:

без ожидания (no delay), когда сразу без паузы повторяем отправку запроса
с константным значением (constant), когда устанавливаем строго заданный лимит
с линейным значением (linear)
с экспоненциальным значением (exponencial)

В каких случаях стоит использовать Retry паттерн ?

Когда в вашем приложении при работе с удаленным сервисом могут возникнуть временные ошибки. Эти ошибки имеют кратковременный характер и высока вероятность того, что следующие запросы будут завершены успешно (Временная недоступность сервиса или тайм-ауты из-за пиковой нагрузки на сервис).

Когда не стоит использовать данный паттерн ?

Когда ошибки имеют долговременный характер, и приложение будет бесполезно тратить ресурсы на попытки повторить операции (в этом случае стоит задуматься об использовании Circuit Breaker)

Для обработки ошибок связанных с бизнес-логикой приложения

Если сервис слишком часто сигнализирует о том, что он “занят”, то скорее всего он требует больше ресурсов

3. Circuit Breaker pattern (предохранитель)

В Spring обычно берут реализацию из Netflix стека, которая называется **Hystrix**.

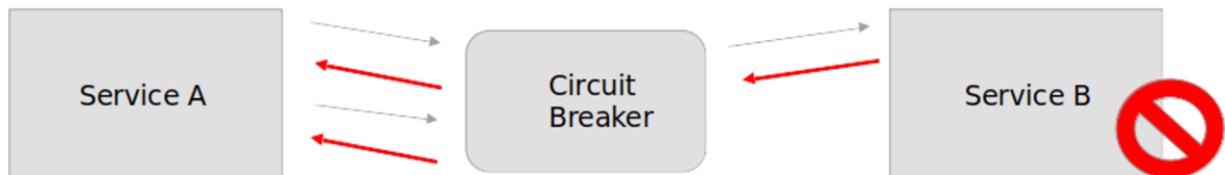
Hystrix — это библиотека задержек и отказоустойчивости, это имплементация паттерна **Circuit Breaker**.



Как сказано из официальной документации: Hystrix — это библиотека, которая помогает вам контролировать взаимодействие между этими распределенными сервисами, добавляя терпимость к задержкам и логику отказоустойчивости.

Есть готовое решение для подобного рода проблем:

Паттерн Circuit Breaker предотвращает попытки приложения выполнить операцию, которая скорее всего завершится неудачно, что позволяет продолжить работу дальше не тратя важные ресурсы, пока известно, что проблема не устранена. Приложение должно быстро принять сбой операции и обработать его.



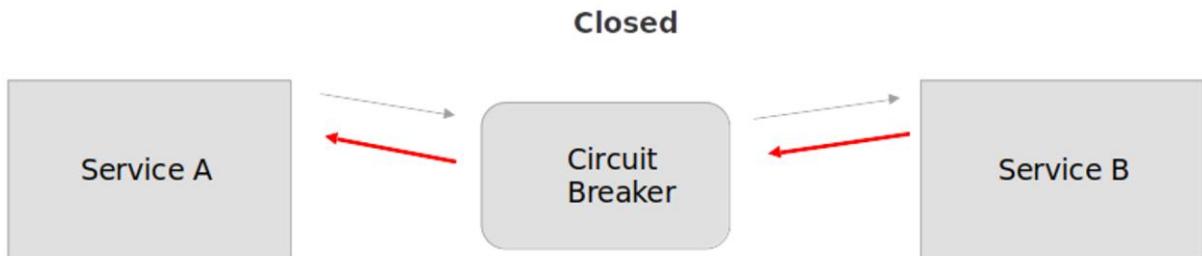
Он также позволяет приложению определять, была ли устранена неисправность. Если проблема устранена, приложение может попытаться вызвать операцию снова.

Circuit Breaker выступает как прокси-сервис между приложением и удаленным сервисом. Прокси-сервис мониторит последние возникшие ошибки, для определения, можно ли выполнить операцию или просто сразу вернуть ошибку.

Если на сервисе В что-то пошло не так, то он возвращает сервису А ошибку и запоминает, что там есть ошибки, и просто последующие запросы не отправит на сервис В. Мы не тратим ресурсы сервера в таком случае.

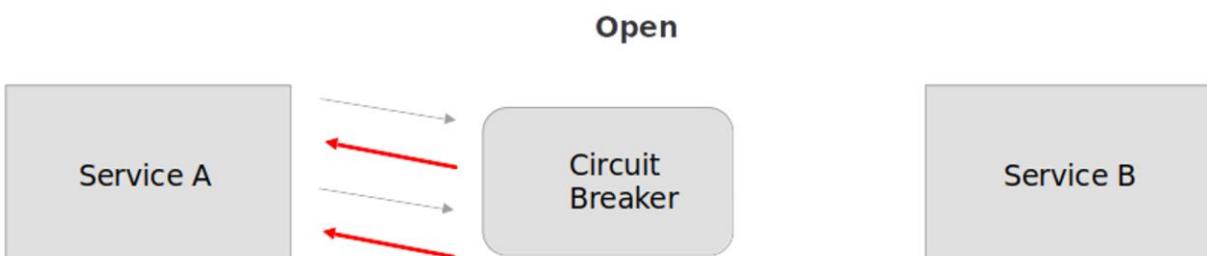
У него есть 3 состояния:

Closed: запрос от приложения направляется напрямую к сервису. Счетчик ошибок = 0 и приложение спокойно функционирует и шлет запросы направо и налево. Все счастливы.



Прокси-сервис увеличивает счетчик ошибок, если операция завершилась неуспешно. Если количество ошибок за некоторый промежуток времени превышает заранее заданный порог значений, то прокси-сервис переходит в состояние Open и запускает таймер времени ожидания. Когда таймер истекает, он переходит в состояние Half-Open. Назначение таймера — дать сервису время для решения проблемы, прежде чем разрешить приложению попытаться выполнить операцию еще раз.

2) Open: запрос от приложения немедленно завершает с ошибкой и исключение возвращается в приложение.



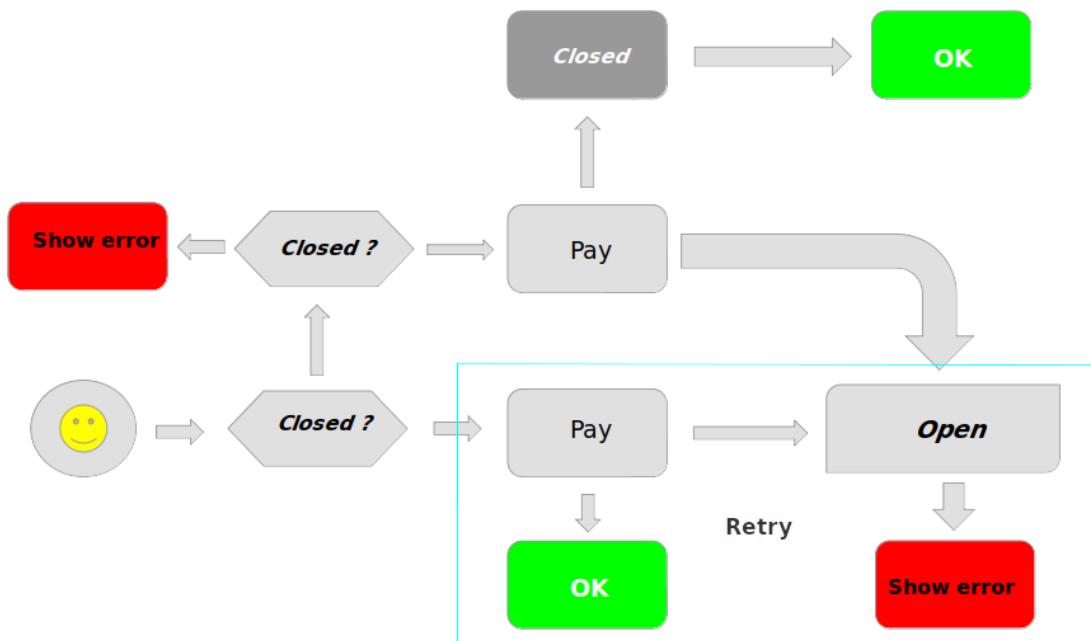
3) Half-Open: ограниченному количеству запросов от приложения разрешено обратиться к сервису. Если эти запросы успешны, то считаем что предыдущая ошибка исправлена и прокси-сервис переходит в состояние Closed (счетчик ошибок сбрасывается на 0). Если любой из запросов завершился ошибкой, то считается, что ошибка все еще существует, тогда прокси-сервис возвращается в состояние Open и перезапускает таймер, чтобы дать системе дополнительное время на восстановление после сбоя.

Состояние Half-Open помогает предотвратить быстрый рост запросов к сервису. Т.к. после начала работы сервиса, некоторое время он может быть способен обрабатывать ограниченное число запросов до полного восстановления.

Шаблон Circuit Breaker обеспечивает стабильность, пока система восстанавливается после сбоя и снижает влияние на производительность.

Благодаря этому можно поддерживать определенный показатель времени отклика системы, быстро отклоняя запрос на операцию, которая, скорее всего, завершится со сбоем, вместо того чтобы ждать, пока не истечет время ожидания операции или ждать в течение неопределенного времени (так как операция никогда не возвратится).

Схематично работа паттерна Circuit Breaker выглядит подобным образом.



Возьмем пример с оплатой картой:

проверяем, в каком состоянии circuit breaker

если закрыт (Closed), то отправляем запрос на сервер, делаем попытку оплаты, все хорошо и все счастливы

если произошла ошибка, то переводим в состояние Open, запускаем таймер, и показываем ошибку

следующий раз, когда будем слать запрос, состояние уже не Closed, а Open, то мы проверяем тот самый таймер (таймер — это то время, которое мы даем серверу на восстановление). Если он не истек, т.е. эта условная скажем минута еще не прошла, то мы на клиенте завершаемся с ошибкой, которая была в предыдущем запросе.

Если таймер истек — пробуем оплатить, все хорошо — переводим в состояние Closed, выключаем таймер и завершаем оплату заказа. Если все плохо — то возвращаемся на шаг Open.

Когда стоит использовать?

Для предотвращения попыток обращения к сервису или разделяемым ресурсам, когда **вероятность возникновения ошибки высока и эти ошибки имеют продолжительный характер**.

Когда не стоит использовать?

Для обращения к приватным ресурсам приложения — это даст только дополнительный overhead

Как замена обработки исключений бизнес-логики приложения

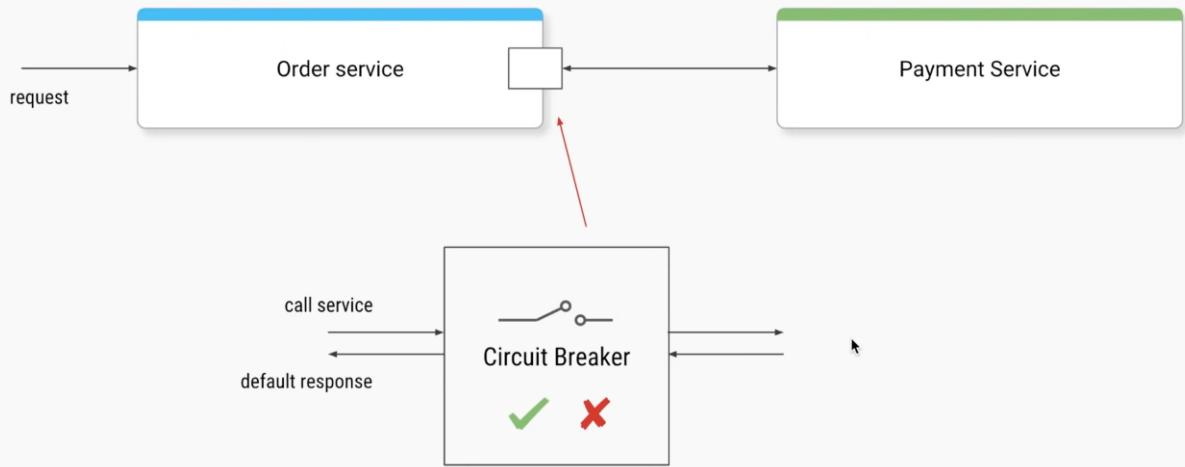
```

@RestController
public class OrderController {

    @PostMapping("/pay")
    public ResponseEntity<Payment> makePayment() {
        try {
            Payment response = paymentService.makePayment();
            return ResponseEntity.ok(response);
        } catch (IOException e) {
            return ResponseEntity.status(INTERNAL_SERVER_ERROR).build();
        }
    }
}

```

*thread freed even
during exception*



Defsys
Tech

```

// create once
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("paymentService");
// decorate remote call with circuit-breaker
Supplier<String> supplier = CircuitBreaker.decorateSupplier(circuitBreaker, this::makePayment);

// make the call
Try<String> result = Try.ofSupplier(supplier);
if (result.isSuccess()) {
    return result.get();
} else {
    return "default-response";
}

```

*decorates our application logic
with circuit-breaker logic*

Decorate Runnable/Callable/Supplier/Consumer

```
// create once
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("paymentService");

// decorate remote call with circuit-breaker
Callable<String> callable = CircuitBreaker.decorateCallable(circuitBreaker, this::makePayment);

// submit the decorated callable
Future<String> future = Executors.newSingleThreadExecutor().submit(callable);

try {
    return future.get();
} catch (CircuitBreakerOpenException ex) {
    return "default-response";
}
// ...

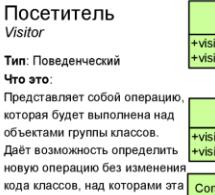
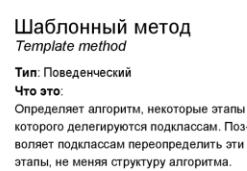
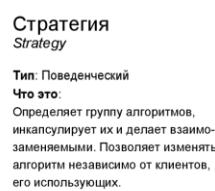
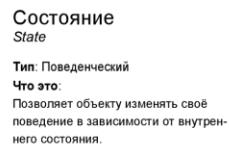
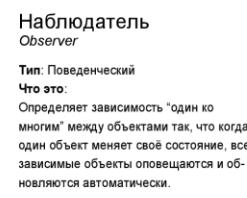
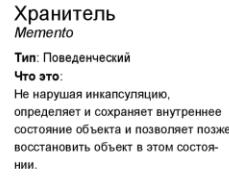
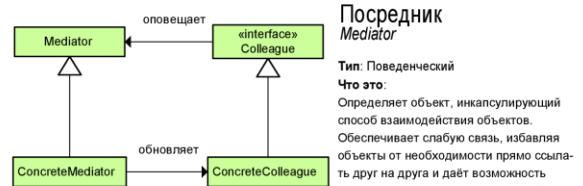
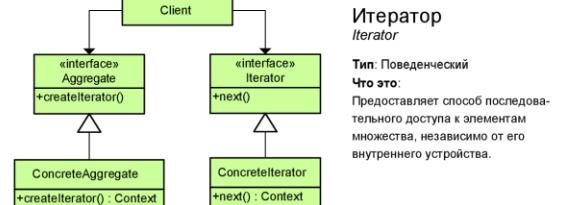
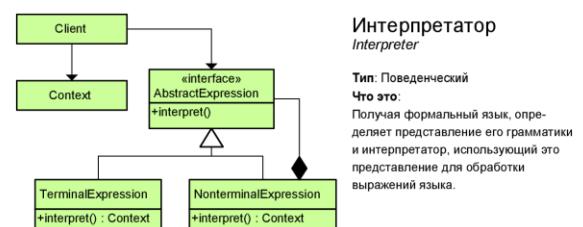
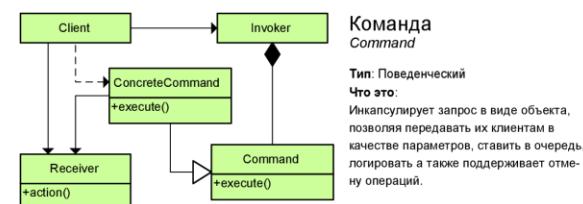
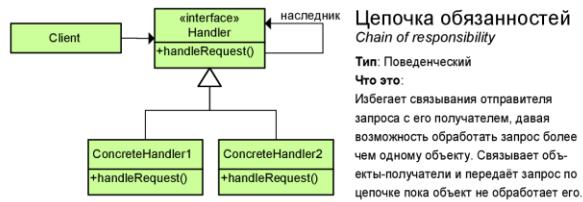
```



Сводная таблица паттернов.

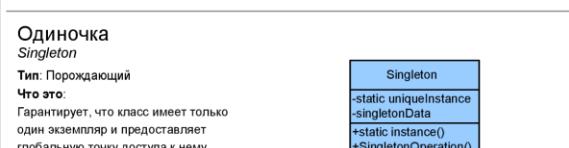
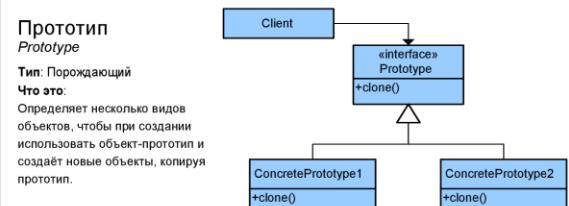
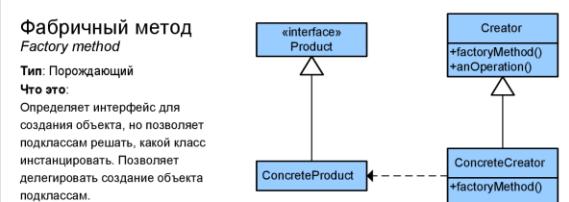
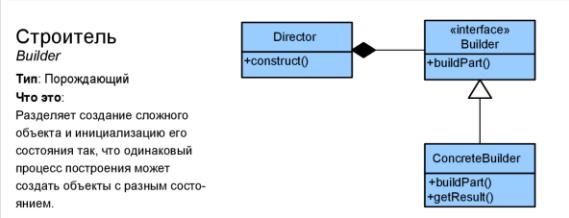
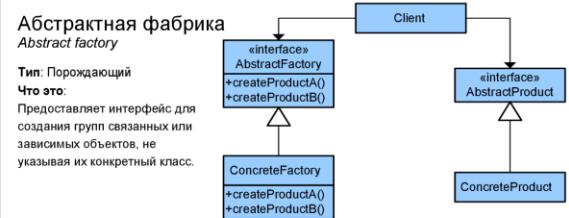
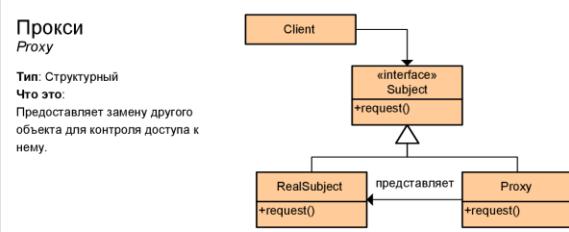
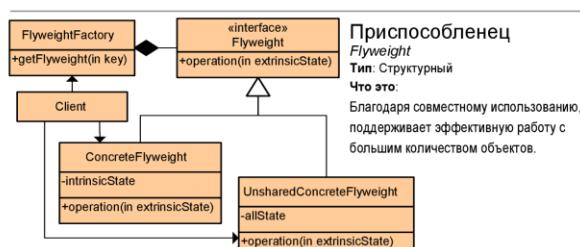
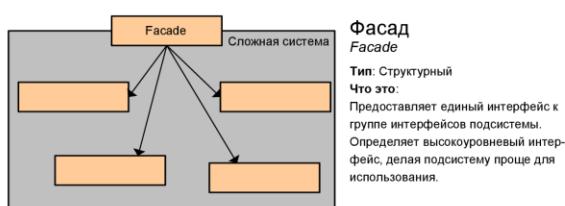
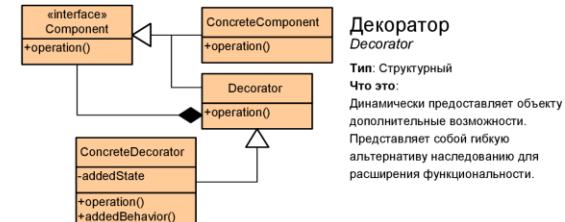
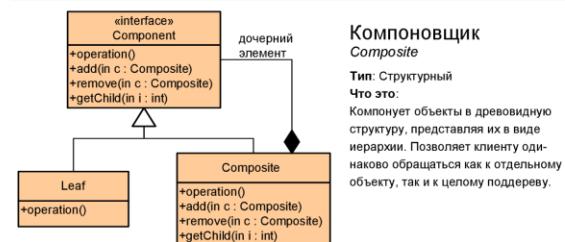
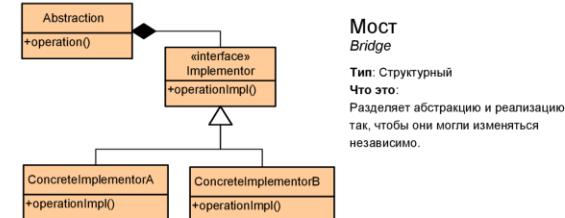
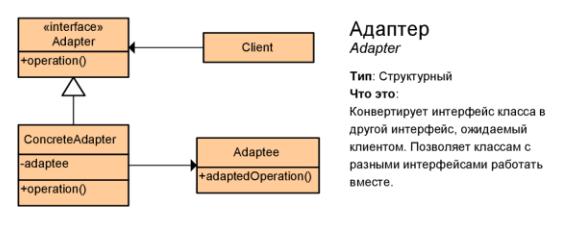
Название	Описание	Где используется в Java
ПОРОЖДАЮЩИЕ ПАТТЕРНЫ		
Абстрактная фабрика (Abstract factory)	Класс, который представляет собой интерфейс для создания других классов.	newInstance() из javax.xml.parsers.DocumentBuilderFactory
Строитель (Builder)	Класс, который представляет собой интерфейс для создания сложного объекта.	StringBuilder
Фабричный метод (Factory method)	Делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.	метод <code>toString()</code> у Object
Прототип (Prototype)	Определяет интерфейс создания объекта через клонирование другого объекта вместо создания через конструктор	метод <code>clone()</code> у Object.
Одиночка (Singleton)	Класс, который может иметь только один экземпляр	метод <code>getDesktop()</code> у Desktop
СТРУКТУРНЫЕ ПАТТЕРНЫ		
Адаптер (Adapter)	Объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.	метод <code>asList()</code> у Arrays
Мост (Bridge)	Разделяет реализацию и абстракцию, дает возможность изменять их свободно друг от друга. Делает конкретные классы независимыми от классов реализации интерфейса.	метод <code>newSetFromMap()</code> у Collections
Компоновщик (Composite)	Объект, который объединяет в себе объекты, подобные ему самому.	метод <code>add(Component)</code> у java.awt.Container.
Декоратор (Decorator)	Класс, расширяющий функциональность другого класса без использования наследования.	java.io.InputStream, OutputStream, Reader и Writer.
Паттерн Фасад (Facade)	Объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое.	javax.faces.context.ExternalContext, который используется внутри ServletContext, HttpSession, HttpServletRequest, HttpServletResponse т.д.

Приспособленец (Flyweight)	Объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковыми. Вместо создания большого количества похожих объектов, объекты используются повторно. Экономит память.	пул строк, а также метод valueOf(int) у java.lang.Integer (также на Boolean, Byte, Character, Short, Long и BigDecimal)
Заместитель (Proxy)	Объект, который является посредником между двумя другими объектами, и который реализует/ограничивает доступ к объекту, к которому обращаются через него.	javax.persistence.PersistenceContext.
ПОВЕДЕНИЧЕСКИЕ ПАТТЕРНЫ		
Цепочка обязанностей (Chain of responsibility)	Позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами.	метод log() у java.util.logging.Logger.
Команда (Command)	Позволяет инкапсулировать различные операции в отдельные объекты.	все реализации java.lang.Runnable.
Интерпретатор (Interpreter)	Решает часто встречающуюся, но подверженную изменениям, задачу.	все подклассы java.text.Format.
Итератор (Iterator)	Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящих в состав агрегации.	все реализации java.util.Iterator.
Посредник (Mediator)	Обеспечивает взаимодействие множества объектов, формируя при этом слабую связность и избавляя объекты от необходимости явно ссылаться друг на друга.	метод execute() у java.util.concurrent.Executor
Хранитель (Memento)	Позволяет, не нарушая инкапсуляцию, зафиксировать и сохранить внутренние состояния объекта так, чтобы позднее восстановить его в этих состояниях.	все реализации java.io.Serializable
Наблюдатель (Observer)	Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.	все реализации java.util.EventListener (практически во всем Swing)
Состояние (State)	Используется в тех случаях, когда во время выполнения программы объект должен менять свое поведение в зависимости от своего состояния.	метод execute() у javax.faces.lifecycle.LifeCycle (контролируется FacesServlet, поведение зависит от текущей фазы (состояния) жизненного цикла JSF).
Стратегия (Strategy)	Определяет ряд алгоритмов позволяя взаимодействовать между ними. Алгоритм стратегии может быть изменен во время выполнения программы.	метод compare() java.util.Comparator, выполненный среди других методов sort() у Collections.
Шаблонный метод (Template method)	Определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.	Определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
Посетитель (Visitor)	Описывает операцию, которая выполняется над объектами других классов. При изменении класса Visitor нет необходимости изменять обслуживаемые классы.	java.nio.file.FileVisitor и SimpleFileVisitor



Copyright © 2007 Jason S. McDonald
http://www.McDonaldLand.info
Перевод Vlad Lastname для http://www.habrahabr.ru/

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison Wesley Longman, Inc.



Copyright © 2007 Jason S. McDonald
http://www.McDonaldLand.info
Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison Wesley Longman, Inc.
Перевод Vlad Lastname для http://www.habrahbr.ru/

Каталог примеров (Java код) <https://refactoring.guru/ru/design-patterns/java>