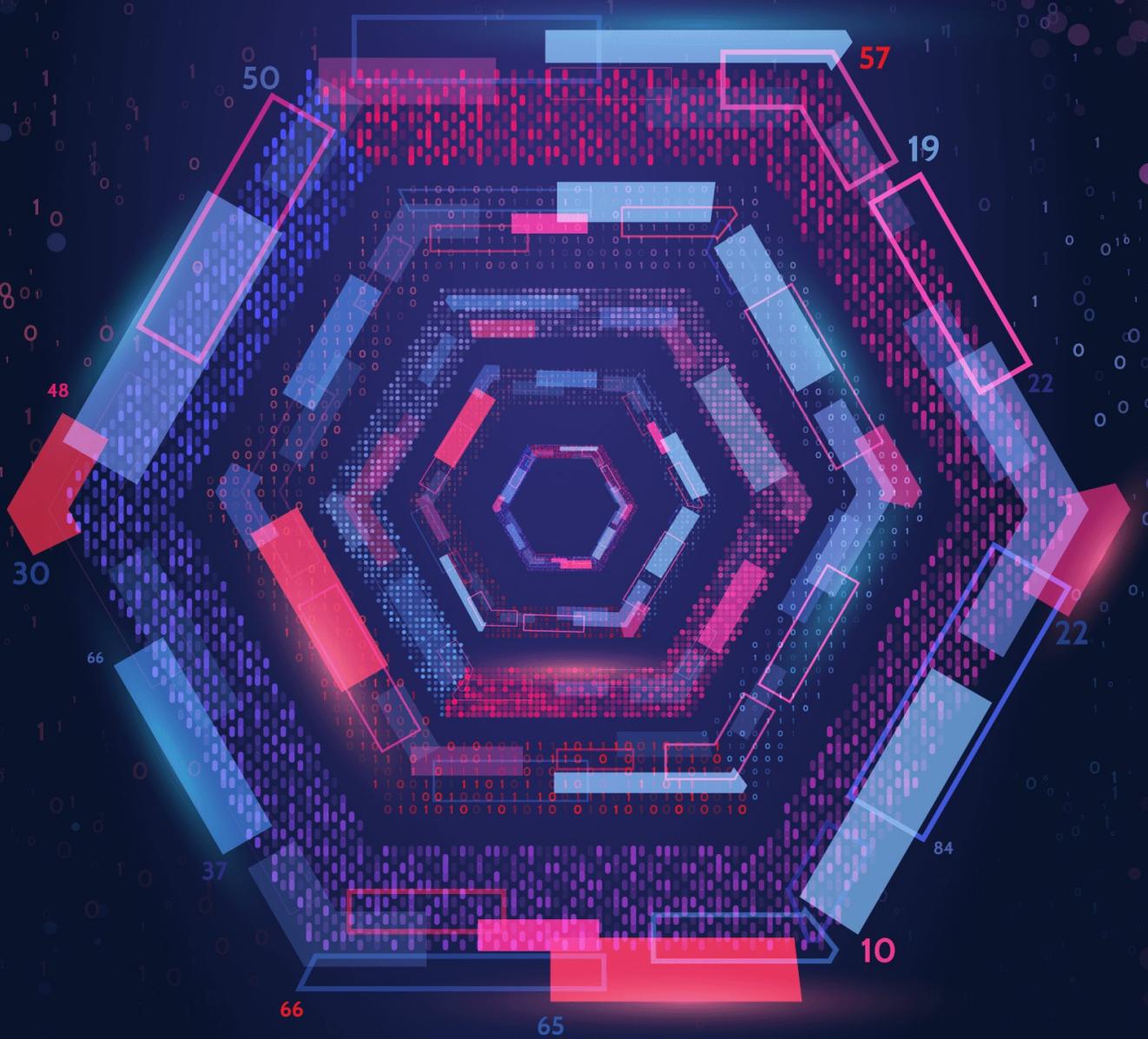


Погружение в

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ



Александр Швец

Погружение в

ПАТТЕРНЫ

ПРОЕКТИРОВАНИЯ

v2018-2.12

Книга куплена Владислав Романик
romanik.vladislav@gmail.com

Вместо копирайта

Привет! Меня зовут Александр Швец, я автор книги [Погружение в Паттерны](#), а также онлайн-курса [Погружение в Рефакторинг](#).



Эта книга предназначена для вашего личного пользования. Пожалуйста, не передавайте книгу третьим лицам, за исключением членов своей семьи. Если вы хотите поделиться книгой с друзьями или коллегами, то купите и подарите им легальную копию.

Все деньги, вырученные от продаж моих книг и курсов, идут на развитие [Refactoring.Guru](#). Это один из немногих ресурсов с авторским контентом, доступным на русском языке. Каждая lawfully приобретённая копия помогает проекту жить дальше и приближает момент выхода нового курса или книги.

© Александр Швец, Refactoring.Guru, 2018

support@refactoring.guru

Иллюстрации: Дмитрий Жарт

Редактор: Эльвира Мамонтова

*Посвящаю эту книгу своей жене, Марии, без которой я бы не смог довести дело до конца ещё
лет тридцать.*

Содержание

Содержание

Как читать эту книгу

ВВЕДЕНИЕ В ООП

Вспоминаем ООП

Краеугольные камни ООП

Отношения между объектами

ОСНОВЫ ПАТТЕРНОВ

Что такое паттерн?

Зачем знать паттерны?

ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ

Качества хорошей архитектуры

Базовые принципы проектирования

- § Инкапсулируйте то, что меняется
- § Программируйте на уровне интерфейса
- § Предпочитайте композицию наследованию

Принципы SOLID

- § S: Принцип единой ответственности
- § O: Принцип открытости/закрытости
- § L: Принцип подстановки Лисков
- § I: Принцип разделения интерфейса
- § D: Принцип инверсии зависимостей

КАТАЛОГ ПАТТЕРНОВ

Порождающие паттерны

- § Фабричный метод
- § Абстрактная фабрика

§ Строитель

§ Прототип

§ Одиночка

Структурные паттерны

§ Адаптер

§ Мост

§ Компоновщик

§ Декоратор

§ Фасад

§ Легковес

§ Заместитель

Поведенческие паттерны

§ Цепочка обязанностей

§ Команда

§ Итератор

§ Посредник

§ Снимок

§ Наблюдатель

§ Состояние

§ Стратегия

§ Шаблонный метод

§ Посетитель

Заключение

Небольшой совет

Если ваша электронная читалка поддерживает режим прокрутки, я рекомендую включить его.



Книга содержит уйму иллюстраций и больших листингов кода, которые смотрятся не так хорошо при случайной разбивке на страницы.

Как читать эту книгу?

Эта книга состоит из описания 22-х классических паттернов проектирования, впервые открытых «Бандой Четырёх» ("Gang of Four" или просто GoF) в 1994 году.

Каждая глава книги посвящена только одному паттерну. Поэтому книгу можно читать как последовательно, от края до края, так и в произвольном порядке, выбирая только интересные в данный момент паттерны.

Многие паттерны связаны между собой, поэтому вы сможете с лёгкостью прыгать по связанным темам, используя ссылки, которых в книге предостаточно. В конце каждой главы приведены отношения текущего паттерна с остальными. Если вы видите там название паттерна, до которого ещё не добрались, то попросту читайте дальше – этот пункт будет повторён в другой главе.

Паттерны проектирования универсальны. Поэтому все примеры кода в этой книге приведены на псевдокоде, без привязки к конкретному языку программирования.

Перед изучением паттернов вы можете освежить память, пройдясь по [основным терминам объектного программирования](#). Параллельно я расскажу об UML-диаграммах, которых в этой книге множество. Если вы всё это уже знаете, смело приступайте к [изучению паттернов](#).

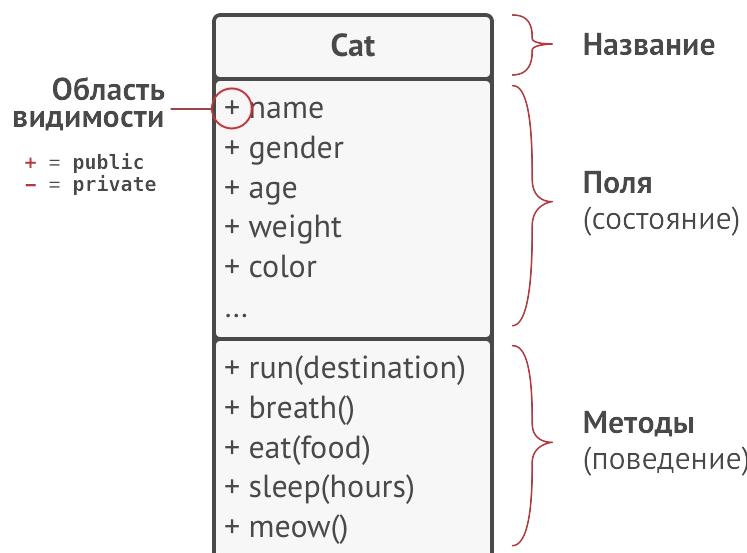
ВВЕДЕНИЕ В ООП

Вспоминаем ООП

Объектно-ориентированное программирование – это методология программирования, в которой все важные вещи представлены **объектами**, каждый из которых является экземпляром определенного **класса**, а классы образуют **иерархию наследования**.

Объекты, классы

Вы любите котиков? Надеюсь, да, потому что я попытаюсь объяснить все эти вещи на примерах с котами.



Это UML-диаграмма класса. В книге будет много таких диаграмм.

Итак, у вас есть кот Пушистик. Он является *объектом класса Кот*. Все коты имеют одинаковый набор свойств: имя, пол, возраст, вес, цвет, любимая еда и прочее. Кроме того, они ведут себя похожим образом: бегают, дышат, спят, едят и мурчат.

Мурка, кошка вашей подруги, тоже является экземпляром класса Кот. Она имеет такой же набор свойств и поведений, что и Пушистик, а отличается от него лишь значениями этих свойств: она другого пола, имеет другой окрас, вес и т. д.



Pushistik: Cat

```
name      = "Pushistik"  
gender    = "male"  
age       = 3  
weight   = 5.5  
color     = gray
```

Murka: Cat

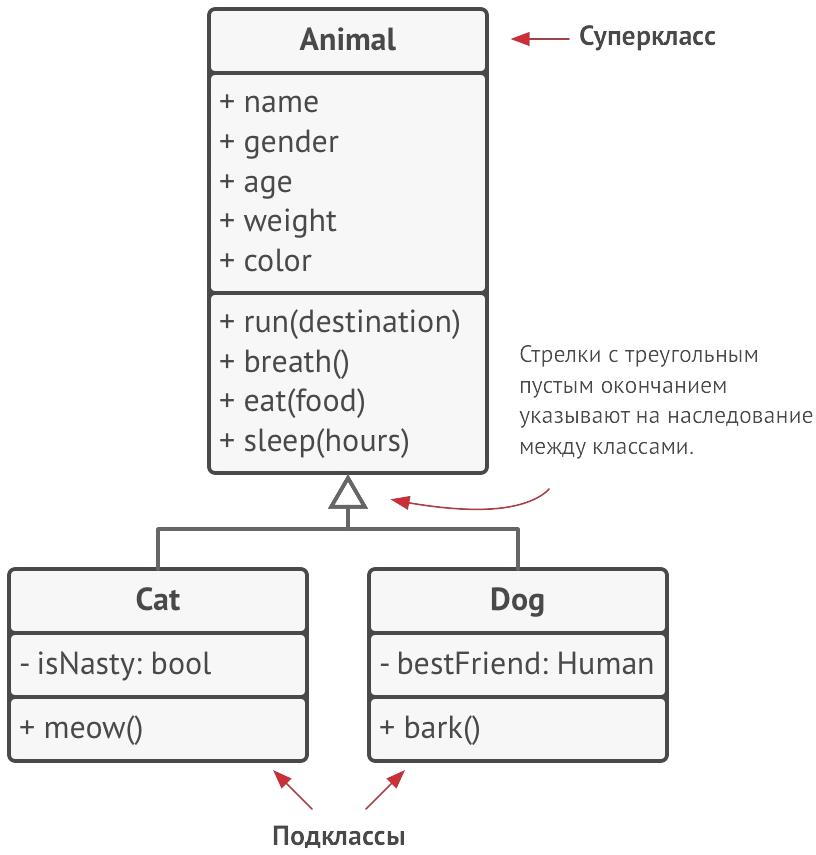
```
name      = "Murka"  
gender    = "female"  
age       = 1  
weight   = 3.5  
color     = white
```

Объекты – это экземпляры классов.

Итак, **класс** – это своеобразный «чертёж», по которому строятся **объекты** – экземпляры этого класса.

Иерархии классов

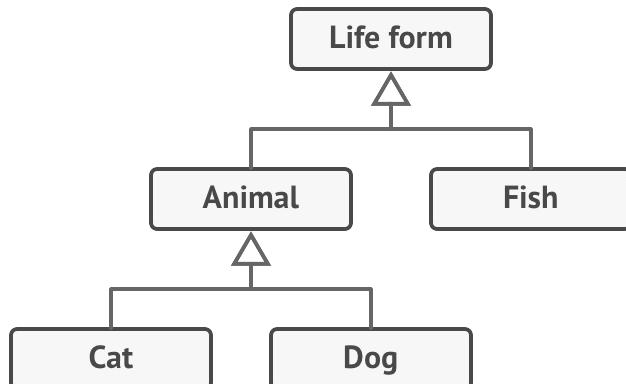
Идём дальше. У вашего соседа есть собака Жучка. Как известно, и собаки, и коты имеют много общего: имя, пол, возраст, цвет есть не только у котов, но и у собак. Да и бегать, дышать, спать и есть могут не только коты. Получается, эти свойства и поведения присущи общему классу `Животных`.



UML-диаграмма иерархии классов. Все классы на этой диаграмме являются частью иерархии **Животных**.

Такой родительский класс принято называть **суперклассом**, а его потомков — **подклассами**. Подклассы наследуют свойства и поведения своего родителя, поэтому в них содержится лишь то, чего нет в суперклассе. Например, только коты могут мурчать, а собаки — лаять.

Мы можем пойти дальше и выделить ещё более общий класс живых **Организмов**, который будет родительским и для **Животных**, и для **Рыб**. Такую «пирамиду» классов обычно называют **иерархией**. Класс **Котов** унаследует всё как из **Животных**, так из **Организмов**.



Классы на UML-диаграмме можно упрощать, если важно показать отношения между ними.

Следует упомянуть, что подклассы могут переопределять поведение методов, которые им достались от суперкласса. При этом они могут как полностью заменить поведение метода, так и просто добавить что-то к результату выполнения родительского метода.

Краеугольные камни ООП

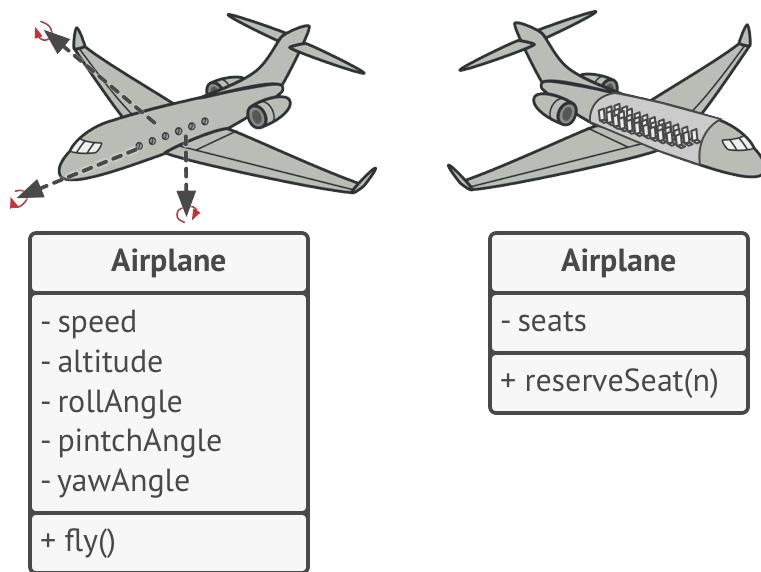
ООП имеет четыре главные концепции, которые отличают его от остальных методологий программирования.



Абстракция

Когда вы пишете программу, используя ООП, вы представляете её части через объекты реального мира. Но объекты в программе не повторяют в точности их реальные аналоги, да и это редко бывает нужно. Вместо этого объекты программы всего лишь *моделируют* свойства и поведения реальных объектов, важные в конкретном контексте, а остальные — игнорируют.

Так, например, класс `Самолёт` будет актуален как для программы тренажёра пилотов, так и для программы бронирования авиабилетов, но в первом случае будут важны детали пилотирования самолёта, а во втором — лишь расположение и занятость мест внутри самолёта.



Разные модели одного и того же реального объекта.

Абстракция – это модель некоего объекта или явления реального мира, в которой опущены незначительные детали, не играющие существенной роли в данном контексте.

Инкапсуляция

Когда вы заводите автомобиль, вам достаточно повернуть ключи зажигания или нажать кнопку. Вам не нужно вручную соединять провода под капотом, поворачивать коленчатый вал и поршни, запуская такт двигателя. Все эти детали скрыты под капотом автомобиля. Вам доступен только простой интерфейс: ключ зажигания, руль и педали. Таким образом, мы приходим к определению *интерфейса – публичной (public) части объекта, доступной остальным объектам*.

Инкапсуляция – это способность объектов скрывать часть своего состояния и поведения от других объектов, предоставляя внешнему миру только определённый интерфейс взаимодействия с собой.

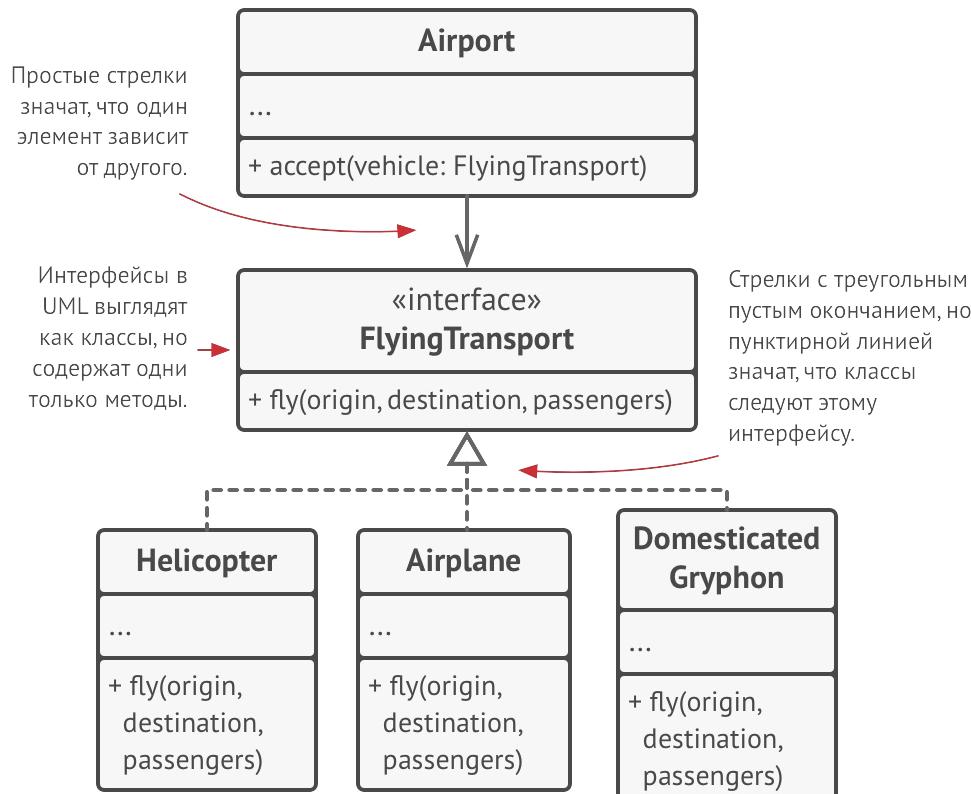
Например, вы можете *инкапсулировать* что-то внутри класса, сделав его *приватным* (*private*) и скрыв доступ к этому полю или методу для объектов других классов. Чуть более свободный, *защищённый* (*protected*) режим видимости сделает это поле или метод доступным в подклассах.

На идеях абстракции и инкапсуляции построены механизмы **интерфейсов** и **абстрактных классов/методов** большинства объектных языков программирования.

Многих путает, что словом «интерфейс» называют и публичную часть объекта, и конструкцию `interface` из большинства языков программирования.

В объектных языках программирования, с помощью механизма *интерфейсов*, которые обычно объявляют через ключевое слово `interface`, можно явно описывать «контракты» взаимодействия объектов.

Например, вы создали интерфейс `ЛетающийТранспорт` с методом `лететь` (откуда, куда, пассажиры), а затем описали методы класса `Аэропорт` так, чтобы они принимали любые объекты с этим интерфейсом. Теперь вы можете быть уверены, что любой объект, реализующий интерфейс – будь то `Самолёт`, `Вертолёт` или `ДрессированныйГрифон` – сможет работать с `Аэропортом`.

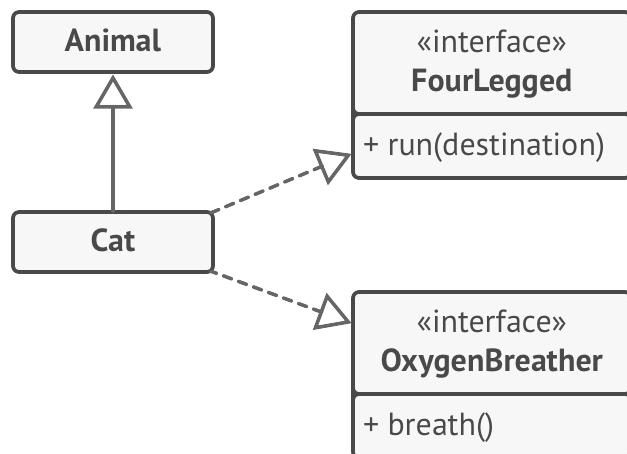


UML-диаграмма реализации и использования интерфейса.

Вы можете как угодно менять код классов, реализующих интерфейс, не беспокоясь о том, что `Аэропорт` перестанет быть с ними совместимым.

Наследование

Наследование – это возможность создания новых классов на основе существующих. Главная польза от наследования – повторное использование существующего кода. Расплата за наследование проявляется в том, что подклассы всегда следуют интерфейсу родительского класса. Вы не можете исключить из подкласса метод, объявленный в его родителе.

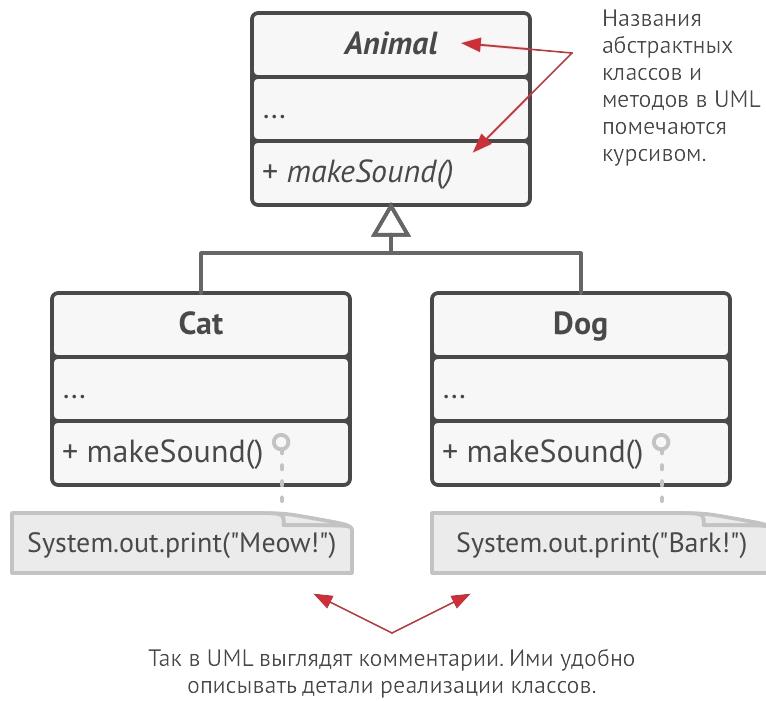


UML-диаграмма единичного наследования против реализации множества интерфейсов.

В большинстве объектных языков программирования класс может иметь только одного родителя. Но, с другой стороны, класс может реализовывать несколько интерфейсов одновременно.

Полиморфизм

Вернёмся к примерам с животными. Практически все `Животные` умеют издавать звуки, поэтому мы можем сделать их общий метод издавания звуков *абстрактным*. Все подклассы должны будут переопределить и реализовать такой метод по-своему.



Так в UML выглядят комментарии. Ими удобно описывать детали реализации классов.

UML-диаграмма единичного наследования против реализации множества интерфейсов.

Теперь представьте, что мы поместили нескольких собак и котов в здоровенный мешок. Затем мы будем с закрытыми глазами вытаскивать их по одному из мешка. Вытянув зверушку, мы не знаем точно, какого она класса. Но если её погладить, зверушка точно издаст какой-то звук, зависящий от её конкретного класса.

```

1 bag = [new Cat(), new Dog()];
2
3 foreach (Animal a : bag)
4     a.makeSound()
5
6 // Meow!
7 // Bark!
  
```

Здесь программе неизвестен конкретный класс объекта в переменной `a`, но благодаря специальному механизму, называемому *полиморфизмом*, будет запущен тот метод издания звуков, который соответствует реальному классу объекта.

Полиморфизм – это способность программы выбирать различные реализации при вызове операций с одним и тем же названием.

Для лучшего понимания *полиморфизм* можно рассматривать как способность объектов «притворяться» чем-то другим. В приведённом выше примере собаки и коты притворялись абстрактными животными.

Отношения между объектами

Кроме *наследования* и *реализации*, есть ещё несколько видов отношений между объектами, о которых мы ещё не говорили.



Ассоциация в UML-диаграммах. Профессор взаимодействует со студентами.

Ассоциация – это когда один объект использует другой либо зависит от него. В UML ассоциация обозначается простой стрелкой, которая направлена в сторону зависимости. Двухсторонняя ассоциация между объектами вполне допустима.



Композиция в UML-диаграммах. Университет состоит из кафедр.

Композиция – это отношение «часть-целое» между двумя объектами, когда один из них включает в себя другой. Особенность этого отношения заключается в том, что компонент может существовать только как часть контейнера. В UML композиция обозначается линией со стрелкой на одном конце и заполненным ромбом на другом конце. Ромб направлен в сторону контейнера, а стрелка – в сторону включаемого объекта.



Агрегация в UML-диаграммах. Кафедра содержит профессоров.

Агрегация – это менее строгий вариант композиции, когда один объект просто имеет ссылку на другой объект. Здесь контейнер не управляет жизненным циклом компонента. Компонент может существовать отдельно от контейнера. В UML агрегация изображается так же, как и композиция, но с пустым ромбом.

ОСНОВЫ ПАТТЕРНОВ

Что такое паттерн?

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, а паттерн – инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

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.

Классификация паттернов

Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы. Проводя аналогию со строительством, вы можете повысить безопасность перекрёстка, поставив светофор, а можете заменить перекрёсток целой автомобильной развязкой с подземными переходами.

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

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

Кроме того, паттерны отличаются и предназначением. В этой книге будут рассмотрены три основные группы паттернов:

- **Порождающие паттерны** беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.
- **Структурные паттерны** показывают различные способы построения связей между объектами.
- **Поведенческие паттерны** заботятся об эффективной коммуникации между объектами.

Кто придумал паттерны?

По определению, паттерны не придумывают, а, скорее, «открывают». Это не какие-то супер-оригинальные решения, а наоборот, часто встречающиеся, типовые решения одной и той же проблемы.

Концепцию паттернов впервые описал Кристофер Александер¹ в книге «Язык шаблонов. Города. Здания. Строительство». В книге описан «язык» для проектирования окружающей среды, единицы которого – шаблоны (или паттерны, что ближе к оригинальному

термину *patterns*) – отвечают на архитектурные вопросы: какой высоты сделать окна, сколько этажей должно быть в здании, какую площадь в микрорайоне отвести под деревья и газоны.

Идея показалась заманчивой четвёрке авторов: Эриху Гамме, Ричарду Хелму, Ральфу Джонсону, Джону Влиссидесу. В 1995 году они написали книгу «Design Patterns: Elements of Reusable Object-Oriented Software»², в которую вошли 23 паттерна, решающие различные проблемы объектно-ориентированного дизайна. Название книги было слишком длинным, чтобы кто-то смог всерьёз его запомнить. Поэтому вскоре все стали называть её «book by the gang of four», то есть «книга от банды четырёх», а затем и вовсе «GOF book».

С тех пор были найдены десятки других объектных паттернов. «Паттерновый» подход стал популярен и в других областях программирования, поэтому сейчас можно встретить всевозможные паттерны и за пределами объектного проектирования.

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.

ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ

Качества хорошей архитектуры

Прежде чем перейти к изучению конкретных паттернов, давайте поговорим о самом процессе проектирования, о том, к чему надо стремиться, и о том, чего надо избегать.

Повторное использование кода

Не секрет, что стоимость и время разработки – это наиболее важные метрики при разработке любых программных продуктов. Чем меньше оба этих показателя, тем конкурентнее продукт будет на рынке и тем больше прибыли получит разработчик.

Повторное использование программной архитектуры и кода – это один из самых распространённых способов снижения стоимости разработки. Логика проста: вместо того, чтобы разрабатывать что-то по второму разу, почему бы не использовать прошлые наработки в новом проекте?

Идея выглядит отлично на бумаге, но, к сожалению, не всякий код можно приспособить к работе в новых условиях. Слишком тесные связи между компонентами, зависимость кода от конкретных классов, а не более абстрактных интерфейсов, вшитые в код операции, которые невозможно расширить – всё это уменьшает гибкость вашей архитектуры и препятствует её повторному использованию.

На помощь приходят паттерны проектирования, которые ценой усложнения кода программы повышают гибкость её частей, упрощая дальнейшее повторное использование.

Приведу цитату Эрика Гаммы, одного из первооткрывателей паттернов, о повторном использовании кода и роли паттернов в нём.

Существует три уровня повторного использования кода. На самом нижнем уровне находятся классы: полезные библиотеки классов, контейнеры, а также «команды» классов, вроде контейнеров/итераторов.

Фреймворки стоят на самом верхнем уровне. В них важна только архитектура. Они определяют ключевые абстракции для решения определённых бизнес-задач, представленных в виде классов и отношений между ними.

Возьмите JUnit, это маленький фреймворк. Он имеет всего несколько классов: `Test`, `TestCase` и

`TestSuite`. Обычно фреймворк имеет гораздо больший охват, чем один класс. Вы должны вклиниться в

фреймворк, расширив какой-то из его классов. Всё работает по так называемому голливудскому принципу «не звоните нам, мы сами вам перезвоним». Фреймворк позволяет вам задать какое-то своё поведение, а затем сам вызывает его, когда приходит черёд что-то делать. То же происходит и в JUnit. Он обращается к вашему классу, когда нужно выполнить тест, но всё остальное происходит внутри фреймворка.

Есть ещё средний уровень. Это то, где я вижу паттерны. Паттерны проектирования и меньше, и более абстрактные, чем фреймворки. Они, на самом деле, просто описание того, как парочка классов относится и взаимодействует друг с другом. Уровень повторного использования повышается, когда вы двигаетесь в направлении от конкретных классов к паттернам, а затем к фреймворкам.

Что ещё замечательно в этом среднем уровне так это то, что паттерны – это менее рискованный способ повторного использования, чем фреймворки. Разработка фреймворка – это крайне рисковая и дорогая инвестиция. В то же время, паттерны позволяют вам повторно использовать идеи и концепции в отрыве от конкретного кода.

Расширяемость

Изменения часто называют главным врагом программиста.

- Вы придумали идеальную архитектуру интернет-магазина, но через месяц пришлось добавить интерфейс для заказов по телефону.
- Вы выпустили видеоигру под Windows, но затем понадобилась поддержка macOS.
- Вы сделали интерфейсный фреймворк с квадратными кнопками, но клиенты начали просить круглые.

У каждого программиста есть дюжина таких историй. Есть несколько причин, почему так происходит.

Во-первых, все мы понимаем проблему лучше в процессе её решения. Нередко к концу работы над первой версией программы, мы уже готовы полностью её переписать, так как стали лучше понимать некоторые аспекты, которые не были очевидны вначале. Сделав вторую версию, вы начинаете понимать проблему ещё лучше, вносите ещё изменения и

так далее – процесс не останавливается никогда, ведь не только ваше понимание, но и сама проблема может измениться со временем.

Во-вторых, изменения могут прийти извне. У вас есть идеальный клиент, который с первого раза сформулировал то, что ему надо, а вы в точности это сделали. Прекрасно! Но вот выходит новая версия операционной системы, в которой ваша программа не работает. Чертыхаясь, вы лезете в код, чтобы внести кое-какие изменения.

Можно посмотреть на это с оптимистичной стороны: если кто-то просит вас что-то изменить в программе, значит, она всё ещё кому-то нужна.

Вот почему даже мало-мальски опытный программист проектирует архитектуру и пишет код с учётом будущих изменений.

Базовые принципы проектирования

Что такое хороший дизайн? По каким критериям его оценивать и каких правил придерживаться при разработке? Как обеспечить достаточный уровень гибкости, связанности, управляемости, стабильности и понятности кода?

Всё это правильные вопросы, но для каждой программы ответ будет чуточку отличаться. Давайте рассмотрим универсальные принципы проектирования, которые помогут вам формулировать ответы на эти вопросы самостоятельно.

Кстати, большинство паттернов, приведённых в этой книге, основаны именно на перечисленных ниже принципах.

Инкапсулируйте то, что меняется

Определите аспекты программы, класса или метода, которые меняются чаще всего, и отделите их от того, что остаётся постоянным.

Этот принцип преследует единственную цель – уменьшить последствия, вызываемые изменениями.

Представьте, что ваша программа – это корабль, а изменения – коварные мины, которые его подстерегают. Натыкаясь на мину, корабль заполняется водой и тонет.

Зная это, вы можете разделить корабль на независимые секции, проходы между которыми можно наглухо задраивать. Теперь при встрече с миной корабль останется на плаву. Вода затопит лишь одну секцию, оставив остальные без изменений.

Изолируя изменчивые части программы в отдельных модулях, классах или методах, вы уменьшаете количество кода, который затронут последующие изменения. Следовательно, вам нужно будет потратить меньше усилий на то, чтобы привести программу в рабочее состояние, отладить и протестировать изменившийся код. Где меньше работы, там меньше стоимость разработки. А где меньше стоимость, там и преимущество перед конкурентами.

Пример инкапсуляции на уровне метода

Представьте, что вы разрабатываете интернет-магазин. Где-то внутри вашего кода может существовать метод `getOrderTotal`, который рассчитывает финальную сумму заказа, учитывая размер налога.

Мы можем предположить, что код вычисления налогов, скорее всего, будет часто меняться. Во-первых, логика начисления налога зависит от страны, штата и даже города, в котором находится покупатель. К тому же размер налога непостоянен: власти могут менять его, когда вздумается.

Из-за этих изменений вам постоянно придётся трогать метод `getOrderTotal`, который, по правде, не особо интересуется *деталями* вычисления налогов.

```
1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     if (order.country == "US")
7         total += total * 0.07 // US sales tax
8     else if (order.country == "EU"):
9         total += total * 0.20 // European VAT
10
11    return total
```

ДО: правила вычисления налогов смешаны с основным кодом метода.

Вы можете перенести логику вычисления налогов в собственный метод, скрыв детали от оригинального метода.

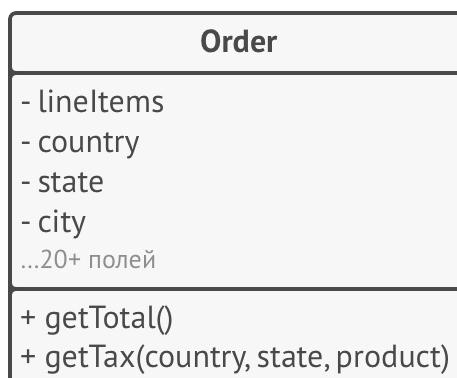
```
1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     total += total * getTaxAmount(order.country)
7
8     return total
9
10 method getTaxAmount(country) is
11     if (country == "US")
12         return 0.07 // US sales tax
13     else if (country == "EU"):
14         return 0.20 // European VAT
15     else
16         return 0
```

ПОСЛЕ: размер налога можно получить, вызывав один метод.

Теперь изменения налогов будут изолированы в рамках одного метода. Более того, если логика вычисления налогов станет ещё более сложной, вам будет легче извлечь этот метод в собственный класс.

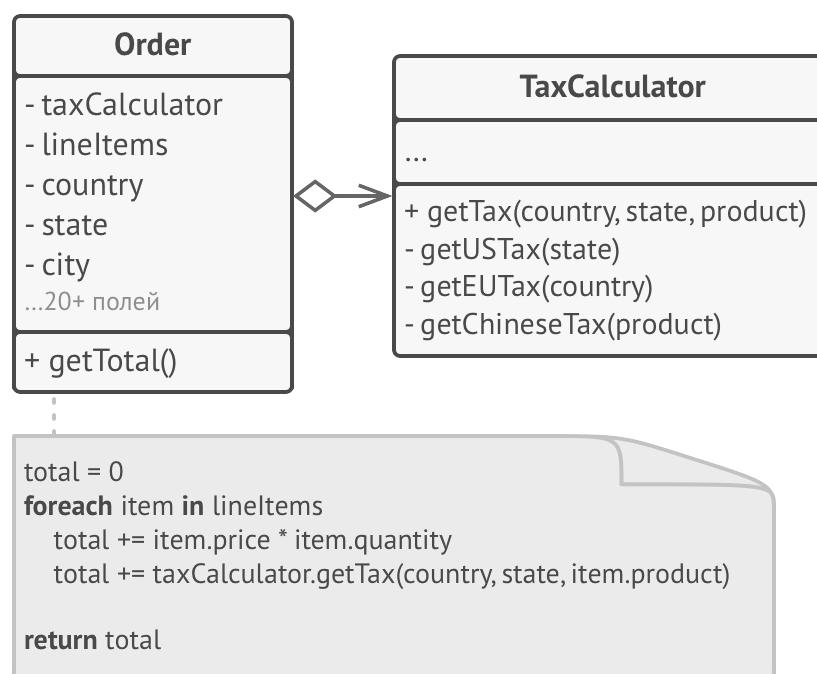
Пример инкапсуляции на уровне класса

Извлечь логику налогов в собственный класс? Если логика налогов стала слишком сложной, то почему бы и нет?



ДО: вычисление налогов в классе заказов.

Объекты заказов станут делегировать вычисление налогов отдельному объекту-калькулятору налогов.



ПОСЛЕ: вычисление налогов скрыто в классе заказов.

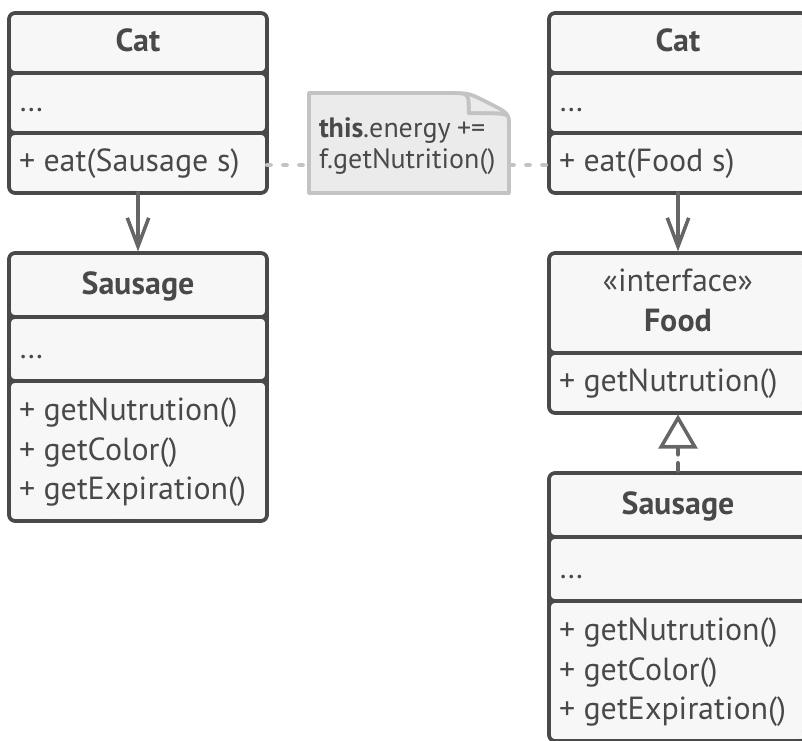
Программируйте на уровне интерфейса

Программируйте на уровне интерфейса, а не на уровне реализации. Код должен зависеть от абстракций, а не конкретных классов.

Гибкость архитектуры выражается в том, что её можно было бы расширять, не ломая существующий код. Для примера вернёмся к классу котов. Класс `Кот`, который ест только сардельки, будет менее гибким, чем тот, который может есть любую еду. Но при этом последнего можно будет кормить и сардельками, ведь они являются едой.

Когда вам нужно наладить взаимодействие между двумя объектами разных классов, вы можете начать с того, что попросту сделаете один класс зависимым от другого. Что говорить, зачастую я и сам так делаю. Но есть и другой, более гибкий, способ.

1. Определите, что именно нужно одному объекту от другого, какие методы он вызывает.
2. Затем опишите эти методы в отдельном интерфейсе.
3. Сделайте так, чтобы класс-зависимость следовал этому интерфейсу. Скорее всего, нужно будет только добавить этот интерфейс в описание класса.
4. Теперь вы можете и сделать второй класс зависимым от интерфейса, а не конкретного класса.



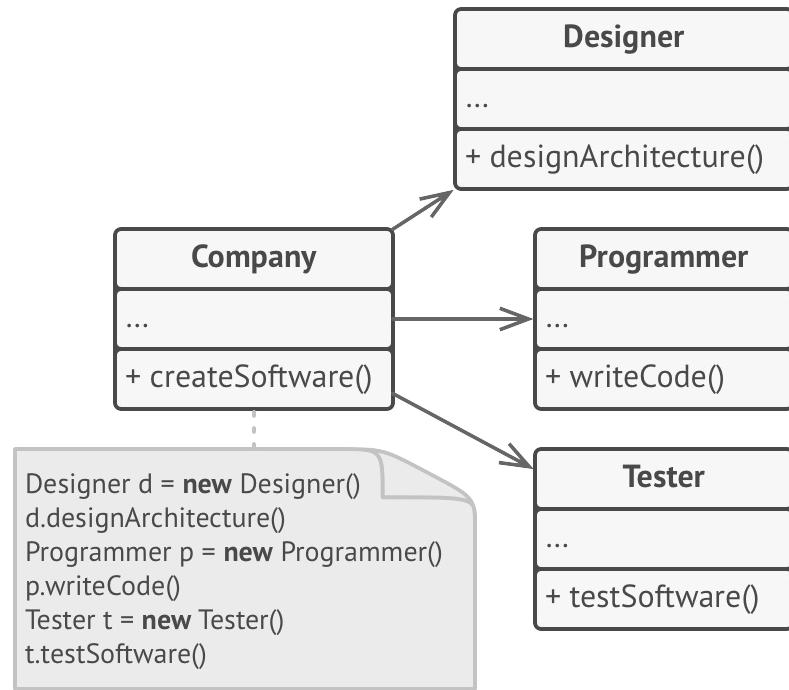
До и после извлечения интерфейса.

Код справа более гибкий, чем тот, что слева, но и более сложный.

Проделав всё это вы, скорее всего, не получите немедленной выгоды. Но зато в будущем вы сможете использовать альтернативные реализации классов, не изменяя использующий их код.

Пример

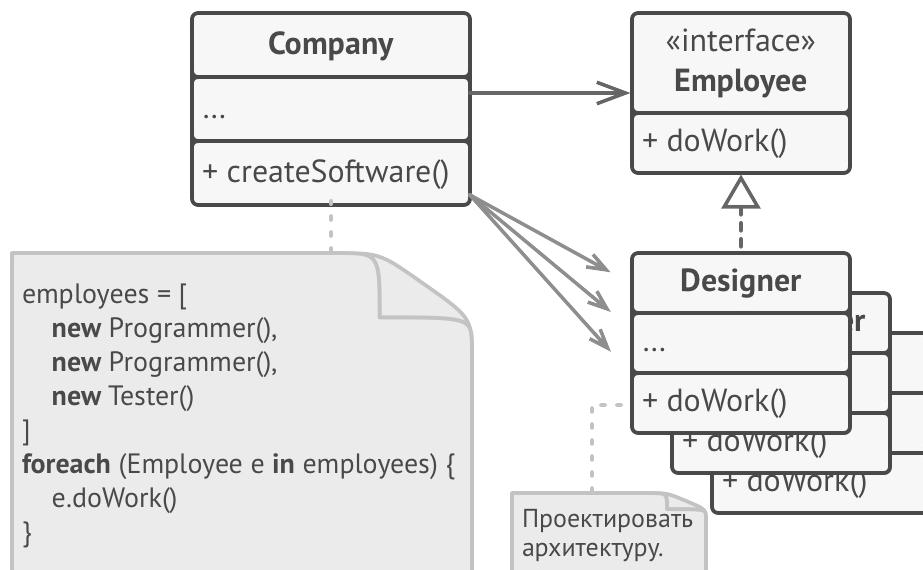
Давайте рассмотрим ещё один пример, где работа на уровне интерфейса оказывается более удачной, чем привязка к конкретным классам. Представьте, что вы делаете симулятор софтверной компании. У вас есть различные классы работников, которые делают ту или иную работу внутри компании.



ДО: классы жёстко связаны.

Вначале класс компании жёстко привязан к конкретным классам работников. Несмотря на то, что каждый тип работников делает разную работу, мы можем свести их методы работы к одному виду, выделив для всех классов общий интерфейс.

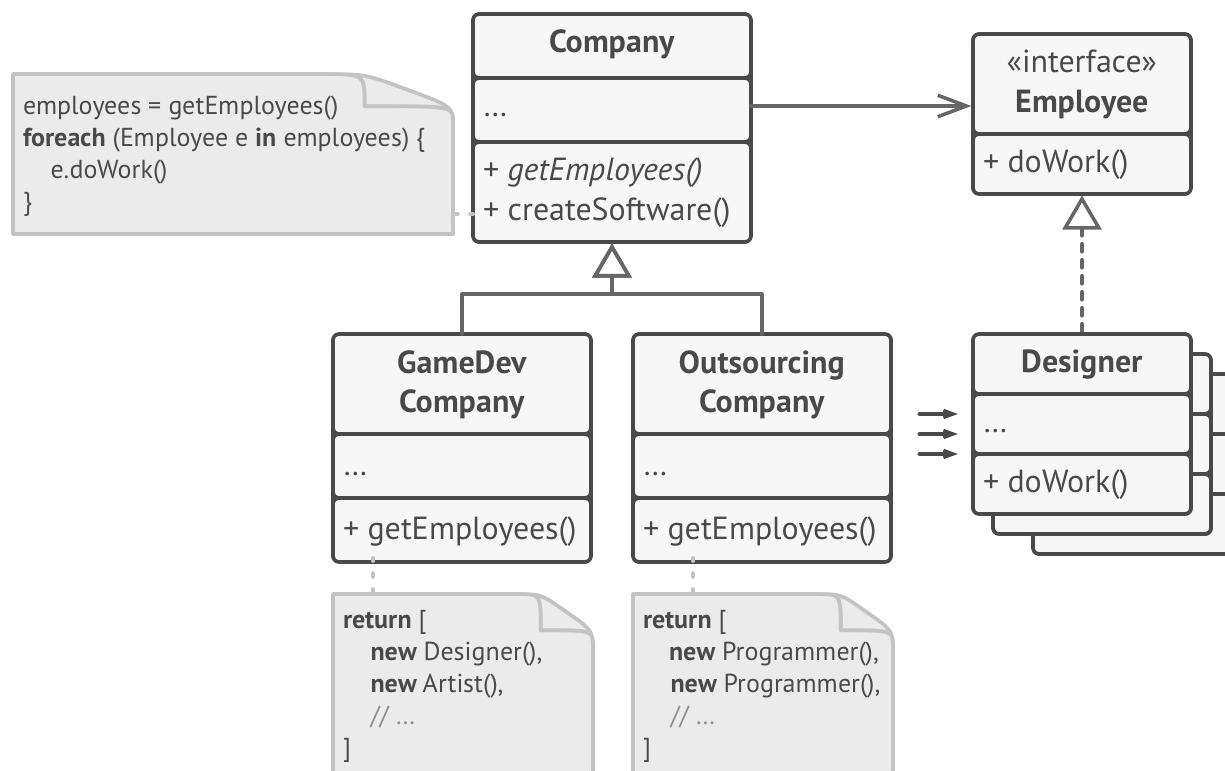
Сделав это, мы сможем применить полиморфизм в классе компании, трактуя всех работников единообразно через интерфейс `Employee`.



ЛУЧШЕ: полиморфизм помог упростить код, но основной код компании всё ещё зависит от конкретных классов сотрудников.

Тем не менее, класс компании всё ещё остаётся жёстко привязанным к конкретным классам работников. Это не очень хорошо, особенно, если предположить, что нам понадобится реализовать несколько видов компаний. Все эти компании будут отличаться тем, какие конкретно работники в них нужны.

Мы можем сделать метод получения сотрудников в базовом классе компании *абстрактным*. Конкретные компании должны будут сами позаботиться о создании объектов сотрудников. А значит, каждый тип компаний сможет иметь собственный набор сотрудников.



ПОСЛЕ: основной код класса компании стал независимым от классов сотрудников. Конкретных сотрудников создают конкретные классы компаний.

После этого изменения код класса компании стал окончательно независимым от конкретных классов. Теперь мы можем добавлять в программу новые виды работников и компаний, не внося изменений в основной код базового класса компаний.

Кстати, вы только что увидели пример одного из паттернов, а именно *Фабричного метода*. Мы ещё вернёмся к нему в дальнейшем.

Предпочитайте композицию наследованию

Наследование – это самый простой и быстрый способ повторного использования кода между классами. У вас есть два класса с дублирующимся кодом. Создайте для них общий базовый класс и перенесите в него общее поведение. Что может быть проще?

Но у наследования есть и проблемы, которые становятся очевидными только тогда, когда программа уже обросла классами, и изменить ситуацию довольно тяжело. Вот некоторые их возможных проблем с наследованием.

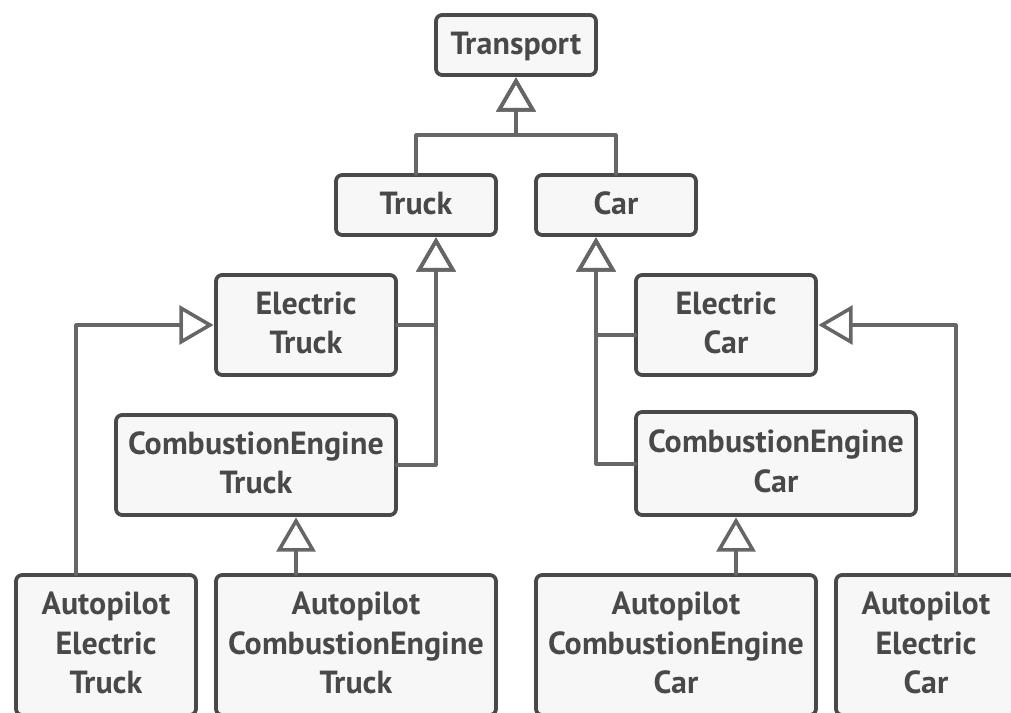
- Подкласс **не может отказаться от интерфейса или реализации** своего родителя. Вы должны будете реализовать все абстрактные методы родителя, даже если они не нужны для конкретного подкласса.
- Переопределяя методы родителя, вы должны **заботиться о том, чтобы не сломать базовое поведение** суперкласса. Это важно, ведь подкласс может быть использован в любом коде, работающим с суперклассом.
- Наследование **нарушает инкапсуляцию суперкласса**, так как подклассам доступны детали родителя. Суперклассы могут сами стать зависимыми от подклассов, например, если программист вынесет в суперкласс какие-то общие детали подклассов, чтобы облегчить дальнейшее наследование.
- Подклассы **слишком тесно связаны** с родительским классом. Любое изменение в родителе может сломать поведение в подклассах.
- Повторное использование кода через наследование может привести к **разрастанию иерархии классов**.

У наследования есть альтернатива, называемая *композицией*. Если наследование можно выразить словом «является» (автомобиль является транспортом), то композицию – словом «содержит» (автомобиль содержит двигатель).

Этот принцип распространяется и на агрегацию – более свободный вид композиции, когда два объекта являются равноправными, и ни один из них не управляет жизненным циклом другого. Оцените разницу: автомобиль *содержит* водителя, но тот может выйти и пересесть в другой автомобиль или вообще пойти пешком *самостоятельно*.

Пример

Предположим, вам нужно смоделировать модельный ряд автопроизводителя. У вас есть легковые автомобили и грузовики. Причём они бывают с электрическим двигателем и с двигателем на бензине. К тому же они отличаются режимами навигации: есть модели с ручным управлением и автопилотом.



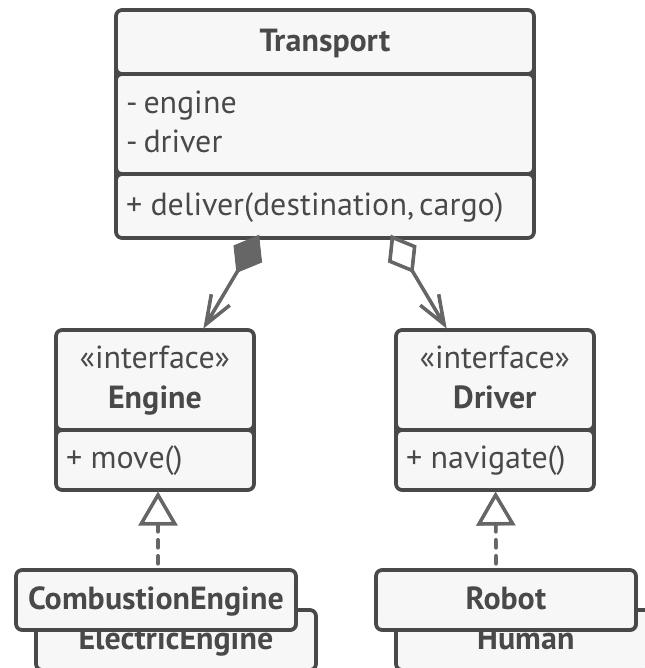
НАСЛЕДОВАНИЕ: развитие классов в нескольких плоскостях (*тип груза × тип двигателя × тип навигации*) приводит к комбинаторному взрыву.

Как видите, каждый такой параметр приводит к умножению количества классов. Кроме того, возникает проблема дублирования кода, так как подклассы не могут наследовать нескольких родителей одновременно.

Решить проблему можно с помощью композиции. Вместо того, чтобы объекты сами реализовывали то или иное поведение, они могут делегировать его другим объектам.

Композиция даёт вам и другое преимущество. К примеру, теперь вы можете заменить тип

двигателя автомобиля прямо во время выполнения программы, подставив в объект транспорта другой объект двигателя.



КОМПОЗИЦИЯ: различные виды функциональности выделены в собственные иерархии классов.

Такая структура свойственна паттерну *Стратегия*, о котором мы тоже поговорим в этой книге.

Принципы SOLID

Рассмотрим ещё пять принципов проектирования, которые известны как SOLID. Эти принципы были впервые изложены Робертом Мартином в книге *Agile Software Development, Principles, Patterns, and Practices*³.

Достичь такой лаконичности в названии удалось, используя небольшую хитрость. Дело в том, что термин SOLID – это аббревиатура, за каждой буквой которой стоит отдельный принцип проектирования.

Главная цель этих принципов – повысить гибкость вашей архитектуры, уменьшить связанность между её компонентами и облегчить повторное использование кода.

Но, как и всё в этой жизни, соблюдение этих принципов имеет свою цену. Здесь это в основном выражается в усложнении кода программы. В реальной жизни, пожалуй, нет такого кода, в котором бы соблюдались все эти принципы сразу. Поэтому помните о балансе и не воспринимайте всё изложенное как догму.

S Принцип единой ответственности **ingle Responsibility Principle**

У класса должен быть только один мотив для изменения.

Стремитесь к тому, чтобы каждый класс отвечал только за одну часть функциональности программы, причём она должна быть полностью инкапсулирована в этот класс (читай, скрыта внутри класса).

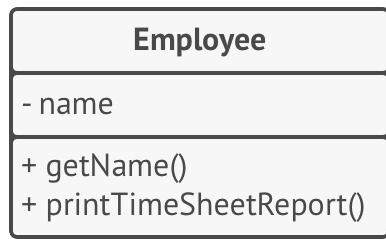
Принцип единственной ответственности предназначен для борьбы со сложностью. Когда в вашем приложении всего 200 строк, то дизайн как таковой вообще не нужен. Достаточно аккуратно написать 5-7 методов, и всё будет хорошо. Проблемы возникают, когда система растёт и увеличивается в масштабах. Когда класс разрастается, он просто перестаёт помещаться в голове. Навигация затрудняется, на глаза попадаются ненужные детали, связанные с другим аспектом, в результате количество понятий начинает превышать мозговой стек, и вы начинаете терять контроль над кодом.

Если класс делает слишком много вещей сразу, вам приходится изменять его каждый раз, когда одна из этих вещей изменяется. При этом есть риск сломать остальные части класса, которые вы даже не планировали трогать.

Хорошо иметь возможность сосредоточиться на сложных аспектах системы по отдельности. Но если вам становится сложно это делать, применяйте принцип единственной ответственности, разделяя ваши классы на части.

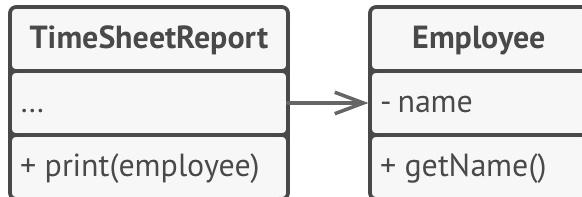
Пример

Класс `Employee` имеет сразу несколько причин для изменения. Первая связана с основной задачей класса – управлением данными сотрудника. Но есть и вторая: изменения, связанные с форматированием отчёта для печати, будут затрагивать класс сотрудников.



ДО: класс сотрудника содержит разнородные поведения.

Проблему можно решить, выделив операцию печати в отдельный класс.



ПОСЛЕ: лишнее поведение переехало в собственный класс.

O

Принцип открытости/закрытости **open/closed Principle**

Расширяйте классы, но не изменяйте их первоначальный код.

Стремитесь к тому, чтобы классы были открыты для расширения, но закрыты для изменения. Главная идея этого принципа в том, чтобы не ломать существующий код при внесении изменений в программу.

Класс можно назвать открытым, если он доступен для расширения. Например, у вас есть возможность расширить набор его операций или добавить к нему новые поля, создав собственный подкласс.

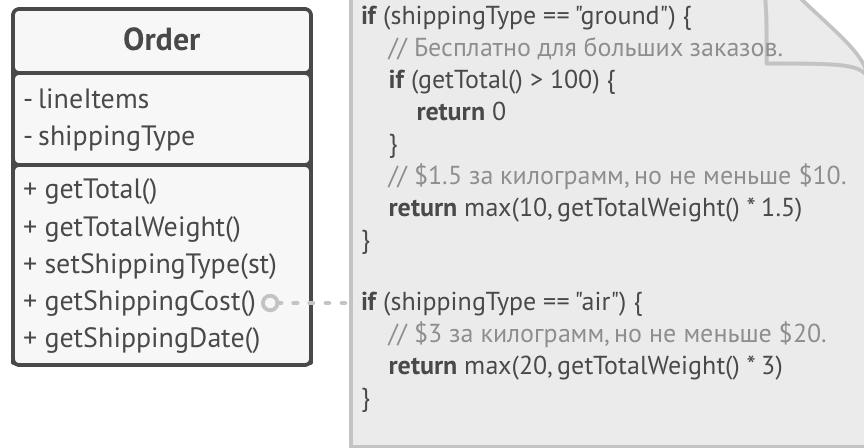
В то же время, класс можно назвать закрытым (а лучше сказать, **законченным**), если он готов для использования другими классами. Это означает, что интерфейс класса уже окончательно определён и не будет изменяться в будущем.

Если класс уже был написан, одобрен, протестирован, возможно, внесён в библиотеку и включён в проект, после этого пытаться модифицировать его содержимое нежелательно. Вместо этого вы можете создать подкласс и расширить в нём базовое поведение, не изменяя код родительского класса напрямую.

Но не стоит следовать этому принципу буквально для каждого изменения. Если вам нужно исправить ошибку в исходном классе, просто возьмите и сделайте это. Нет смысла решать проблему родителя в дочернем классе.

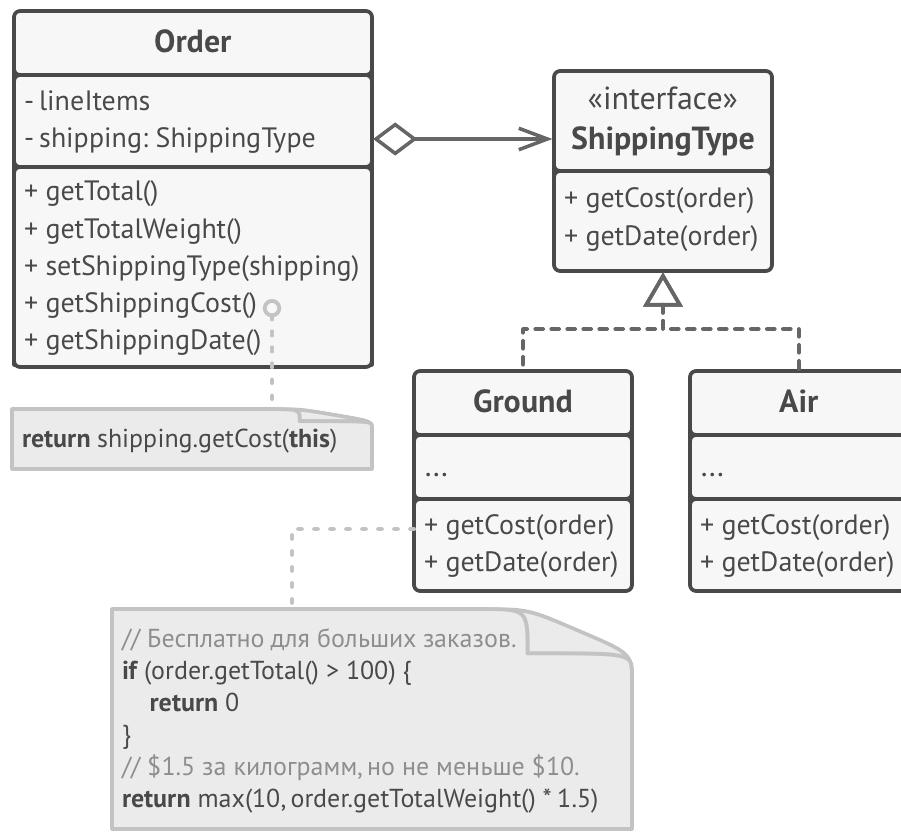
Пример

Класс заказов имеет метод расчёта стоимости доставки, причём способы доставки «зашиты» непосредственно в сам метод. Если вам нужно будет добавить новый способ доставки, то придётся трогать весь класс `Order`.



ДО: код класса заказа нужно будет изменять при добавлении нового способа доставки.

Проблему можно решить, если применить паттерн *Стратегия*. Для этого нужно выделить способы доставки в собственные классы с общим интерфейсом.



ПОСЛЕ: новые способы доставки можно добавлять, не трогая класс заказов.

Теперь при добавлении нового способа доставки нужно будет реализовать новый класс интерфейса доставки, не трогая класс заказов. Объект способа доставки в класс заказа будет подавать клиентский код, который раньше устанавливал способ доставки простой строкой.

Бонус этого решения в том, что расчёт времени и даты доставки тоже можно поместить в новые классы, повинуясь принципу единственной ответственности.

Liskov Substitution Principle

Подклассы должны дополнять, а не замещать поведение базового класса.

Стремитесь создавать подклассы таким образом, чтобы их объекты можно было бы подставлять вместо объектов базового класса, не ломая при этом функциональности клиентского кода.

Принцип подстановки – это ряд проверок, которые помогают предсказать, останется ли подкласс совместим с остальным кодом программы, который до этого успешно работал, используя объекты базового класса. Это особенно важно при разработке библиотек и фреймворков, когда ваши классы используются другими людьми, и вы не можете повлиять на чужой клиентский код, даже если бы хотели.

В отличие от других принципов, которые определены очень свободно и имеют массу трактовок, принцип подстановки имеет ряд формальных требований к подклассам, а точнее, к переопределённым в них методам.

- **Типы параметров метода подкласса должны совпадать или быть более абстрактными, чем типы параметров базового метода.** Довольно запутанно? Рассмотрим, как это работает на примере.

- Базовый класс содержит метод `feed(Cat c)`, который умеет кормить домашних котов. Клиентский код это знает и всегда передаёт в метод кота.
- **Хорошо:** Вы создали подкласс и переопределили метод кормёжки так, чтобы накормить любое животное: `feed(Animal c)`. Если подставить этот подкласс в клиентский код, то ничего страшного не произойдёт. Клиентский код подаст в метод кота, но метод умеет кормить всех животных, поэтому накормит и кота.
- **Плохо:** Вы создали другой подкласс, в котором метод умеет кормить только бенгальскую породу котов (подкласс котов): `feed(BengalCat t)`. Что будет с

клиентским кодом? Он всё так же подаст в метод обычного кота. Но метод умеет кормить только бенгалов, поэтому не сможет отработать, сломав клиентский код.

- **Тип возвращаемого значения метода подкласса должен совпадать или быть подтипом возвращаемого значения базового метода.** Здесь всё то же, что и в предыдущем пункте, но наоборот.
 - Базовый метод: `buyCat(): Cat`. Клиентский код ожидает на выходе любого домашнего кота.
 - **Хорошо:** Метод подкласса: `buyCat(): BengalCat`. Клиентский код получит бенгальского кота, который является домашним котом, поэтому всё будет хорошо.
 - **Плохо:** Метод подкласса: `buyCat(): Animal`. Клиентский код сломается, так как это непонятное животное (возможно, крокодил) не поместится в ящике-переноске для кота.

Ещё один анти-пример из мира языков с динамической типизацией: базовый метод возвращает строку, а переопределённый метод – число.

- **Метод не должен выбрасывать исключения, которые не свойственны базовому методу.** Типы исключений в переопределённом методе должны совпадать или быть подтипами исключений, которые выбрасывает базовый метод. Блоки `try-catch` в клиентском коде нацелены на конкретные типы исключений, выбрасываемые базовым методом. Поэтому неожиданное исключение, выброшенное подклассом, может проскочить сквозь обработчики клиентского кода и обрушить программу.

В большинстве современных языков программирования, особенно строго типизированных (Java, C# и другие), перечисленные ограничения встроены прямо в компилятор. Поэтому вы попросту не сможете собрать программу, нарушив их.

- **Метод не должен ужесточать _пред_условия.** Например, базовый метод работает с параметром типа `int`. Если подкласс требует, чтобы значение этого параметра к тому же было больше нуля, то это ужесточает предусловия. Клиентский код, который до

этого отлично работал, подавая в метод негативные числа, теперь сломается при работе с объектом подкласса.

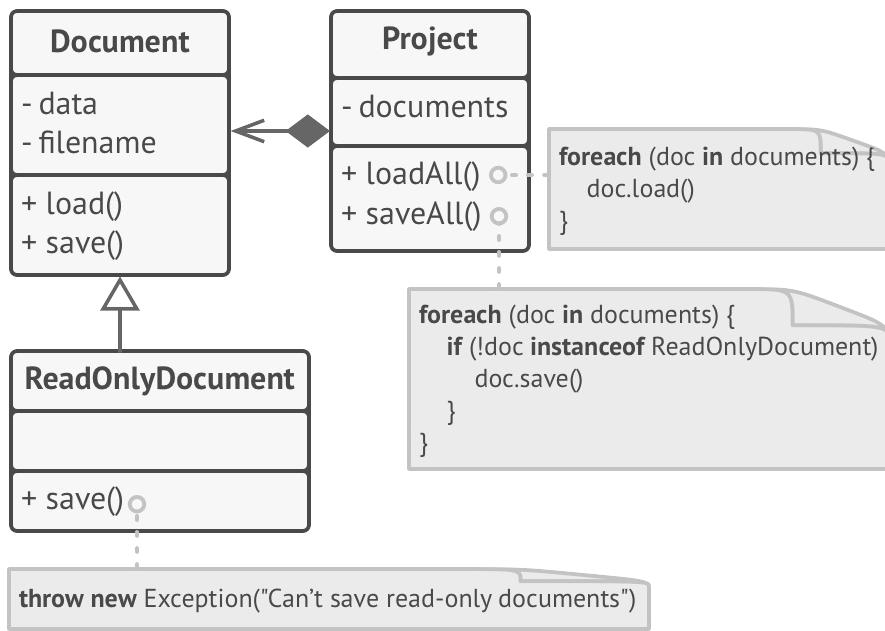
- **Метод не должен ослаблять _пост_условия.** Например, базовый метод требует, чтобы по завершению метода все подключения к базе данных были закрыты, а подкласс оставляет эти подключения открытыми, чтобы потом повторно использовать. Но клиентский код базового класса ничего об этом не знает. Он может завершить программу сразу после вызова метода, оставив запущенные процессы-призраки в системе.
- **Инварианты класса должны остаться без изменений.** Инвариант – это набор условий, при которых объект имеет смысл. Например, инвариант кота – это наличие четырёх лап, хвоста, способность мурчать и прочее. Инвариант может быть описан не только явным контрактом или проверками в методах класса, но и косвенно, например, юнит-тестами или клиентским кодом.

Этот пункт проще всего нарушить при наследовании, так как вы можете попросту не подозревать о каком-то из условий инварианта сложного класса. Идеальным в этом отношении был бы подкласс, который только вводит новые методы и поля, не прикасаясь к полям базового класса.

- **Подкласс не должен изменять значения приватных полей базового класса.** Этот пункт звучит странно, но в некоторых языках доступ к приватным полям можно получить через механизм рефлексии. В некоторых других языках (Python, JavaScript) и вовсе нет жёсткой защиты приватных полей.

Пример

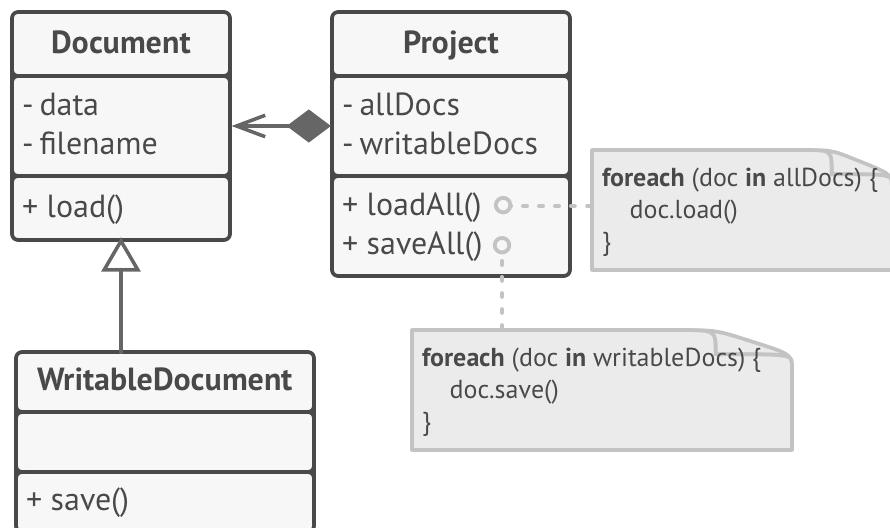
Чтобы закрыть тему принципа подстановки, давайте рассмотрим пример неудачной иерархии классов документов.



ДО: подкласс «обнуляет» работу базового метода.

Метод сохранения в подклассе `ReadOnlyDocuments` выбросит исключение, если кто-то попытается вызвать его метод сохранения. Базовый метод не имеет такого ограничения. Из-за этого клиентский код вынужден проверять тип документа при сохранении всех документов.

При этом нарушается ещё и принцип открытости/закрытости, так как клиентский код начинает зависеть от конкретного класса, который нельзя заменить на другой, не внося изменений в клиентский код.



ПОСЛЕ: подкласс расширяет базовый класс новым поведением.

Проблему можно решить, перепроектировав иерархию классов. Базовый класс сможет только открывать документы, но не сохранять их. Подкласс, который теперь будет называться `WritableDocument`, расширит поведение родителя, позволив сохранять документ.

Принцип разделения интерфейса

Interface Segregation Principle

Клиенты не должны зависеть от методов, которые они не используют.

Стремитесь к тому, чтобы интерфейсы были достаточно узкими, чтобы классам не приходилось реализовывать избыточное поведение.

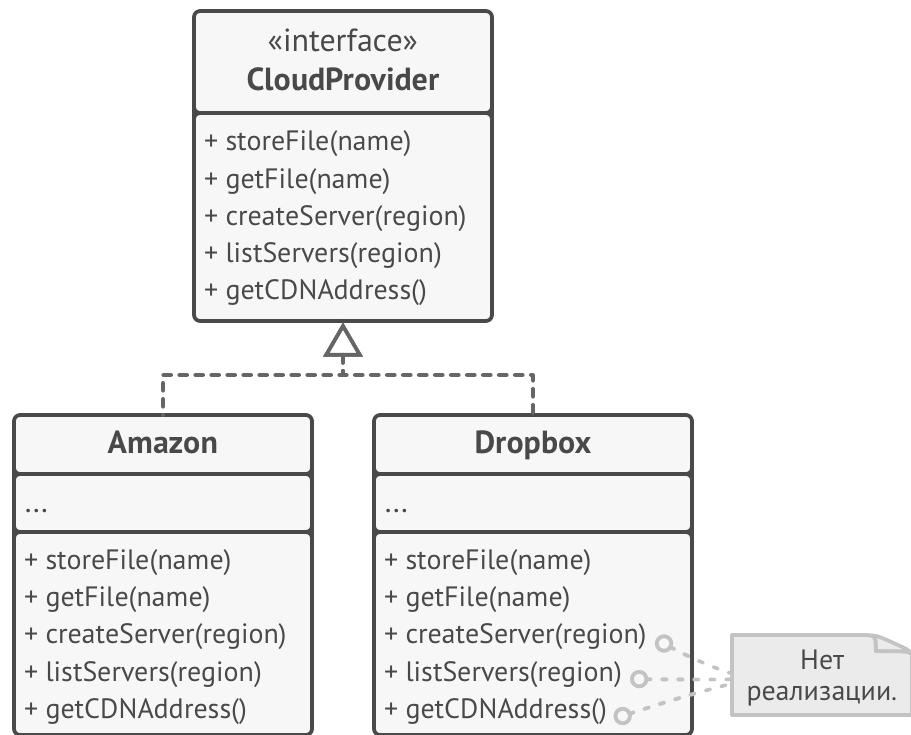
Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

Наследование позволяет классу иметь только один суперкласс, но не ограничивает количество интерфейсов, которые он может реализовать. Большинство объектных языков программирования позволяют классам реализовывать сразу несколько интерфейсов, поэтому нет необходимости заталкивать в ваш интерфейс больше поведений, чем он того требует. Вы всегда можете присвоить классу сразу несколько интерфейсов поменьше.

Пример

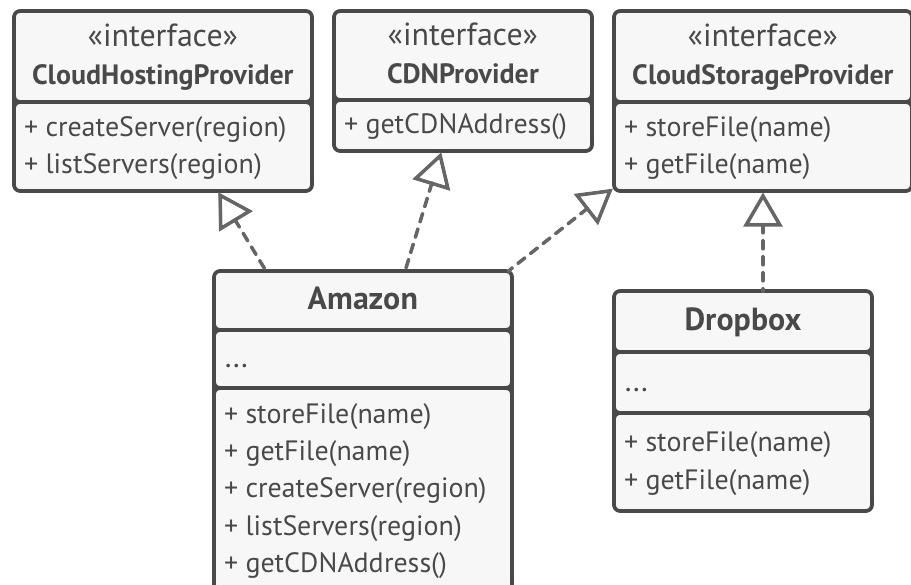
Представьте библиотеку для работы с облачными провайдерами. В первой версии она поддерживала только Amazon, имеющий полный набор облачных услуг. Исходя из них и проектировался интерфейс будущих классов.

Но позже стало ясно, что получившийся интерфейс облачного провайдера слишком широк, так как есть другие провайдеры, реализующие только часть из всех возможных сервисов.



ДО: не все клиенты могут реализовать операции интерфейса.

Чтобы не плодить классы с пустой реализацией, раздутый интерфейс можно разбить на части. Классы, которые были способны реализовать все операции старого интерфейса, могут реализовать сразу несколько новых частичных интерфейсов.



ПОСЛЕ: раздутый интерфейс разбит на части.

D

Принцип инверсии зависимостей

Dependency Inversion Principle

Классы верхних уровней не должны зависеть от классов нижних уровней. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Обычно при проектировании программ можно выделить два уровня классов.

- **Классы нижнего уровня** реализуют базовые операции вроде работы с диском, передачи данных по сети, подключения к базе данных и прочее.
- **Классы высокого уровня** содержат сложную бизнес-логику программы, которая опирается на классы низкого уровня для осуществления более простых операций.

Зачастую вы сперва проектируете классы нижнего уровня, а только потом берётесь за верхний уровень. При таком подходе классы бизнес-логики становятся зависимыми от более примитивных низкоуровневых классов. Каждое изменение в низкоуровневом классе может затронуть классы бизнес-логики, которые его используют.

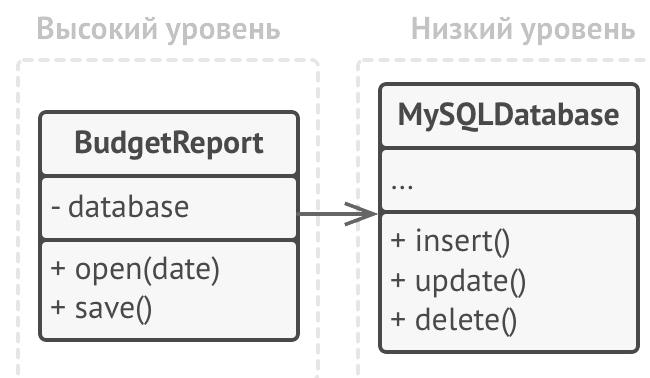
Принцип инверсии зависимостей предлагает изменить направление, в котором происходит проектирование.

1. Для начала вам нужно описать интерфейс низкоуровневых операций, которые нужны классу бизнес-логики.
2. Это позволит вам убрать зависимость класса бизнес-логики от конкретного низкоуровневого класса, заменив её «мягкой» зависимостью от интерфейса.
3. Низкоуровневый класс, в свою очередь, станет зависимым от интерфейса, определённого бизнес-логикой.

Принцип инверсии зависимостей часто идёт в ногу с принципом открытости/закрытости: вы сможете расширять низкоуровневые классы и использовать их вместе с классами бизнес-логики, не изменяя код последних.

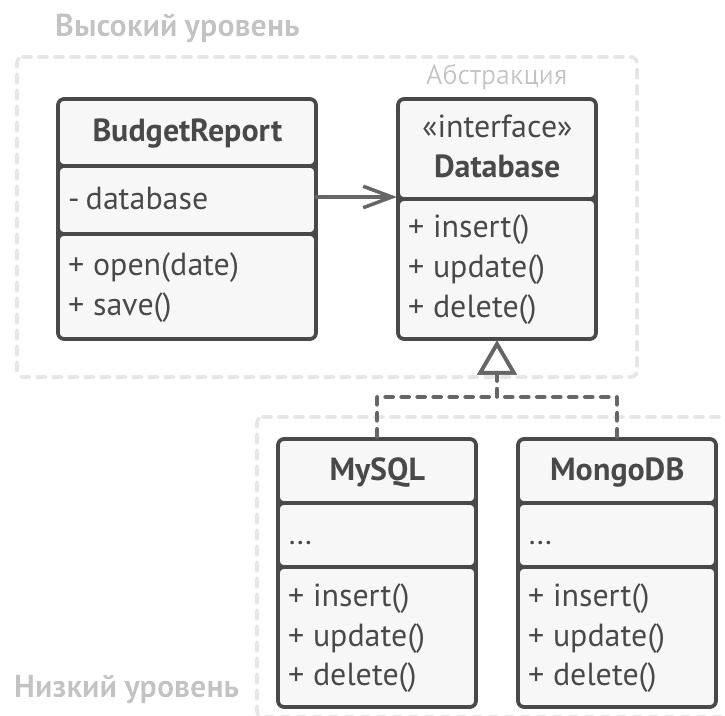
Пример

В этом примере высокоуровневый класс формирования бюджетных отчётов напрямую использует класс базы данных для загрузки и сохранения своей информации.



ДО: высокоуровневый класс зависит от низкоуровневого.

Вы можете исправить проблему, создав высокоуровневый интерфейс для загрузки/сохранения данных и привязав к нему класс отчётов. Низкоуровневые классы должны реализовать этот интерфейс, чтобы их объекты можно было использовать внутри объекта отчётов.



ПОСЛЕ: низкоуровневые классы зависят от высокоуровневой абстракции.

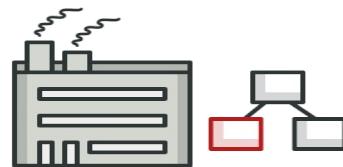
Таким образом, меняется направление зависимости. Если раньше высокий уровень зависел от низкого, то сейчас всё наоборот – низкоуровневые классы зависят от

высокоуровневого интерфейса.

КАТАЛОГ ПАТТЕРНОВ

Порождающие паттерны

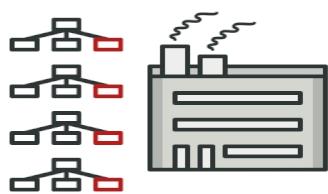
Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.



Фабричный Метод

Factory Method

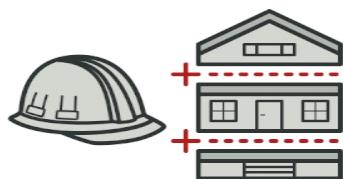
Определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.



Абстрактная Фабрика

Abstract Factory

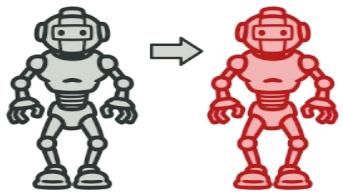
Позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



Строитель

Builder

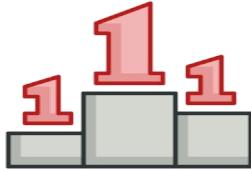
Позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.



Прототип

Prototype

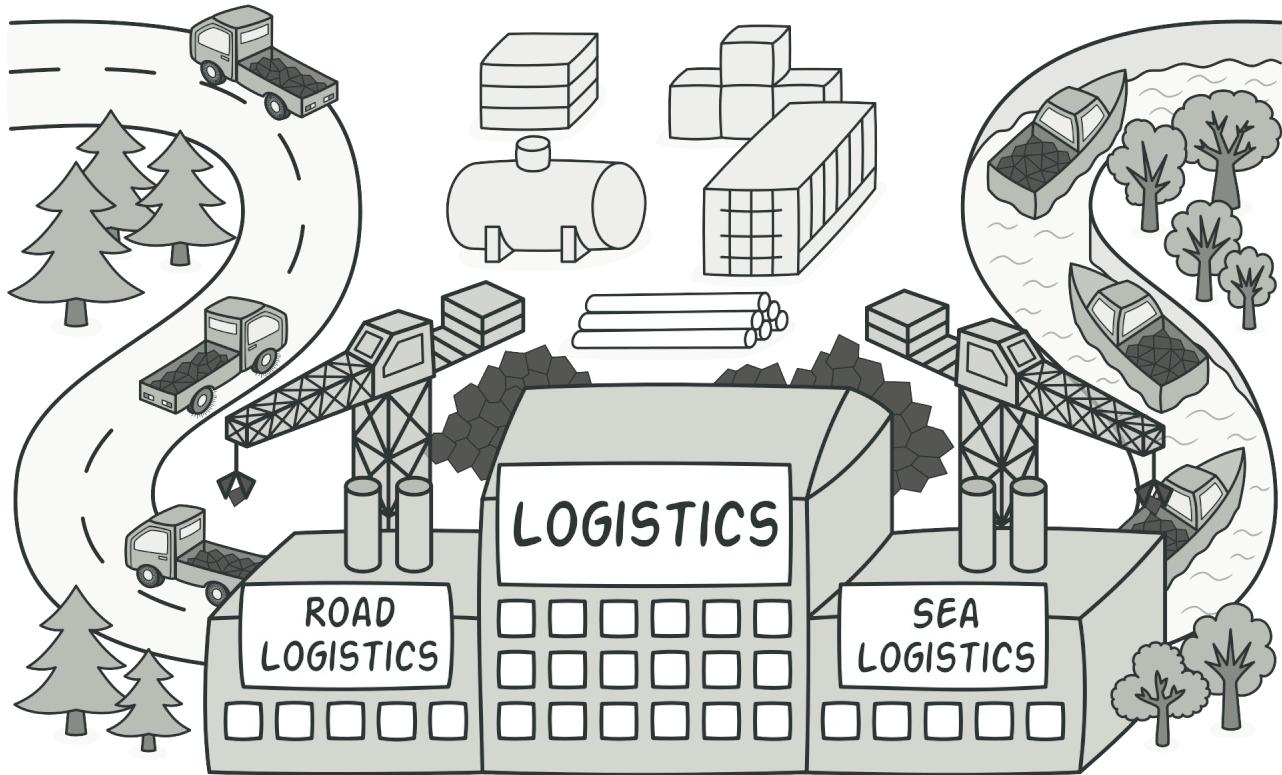
Позволяет копировать объекты, не вдаваясь в подробности их реализации.



Одиночка

Singleton

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



ФАБРИЧНЫЙ МЕТОД

Также известен как: Виртуальный конструктор, *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`.

В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.



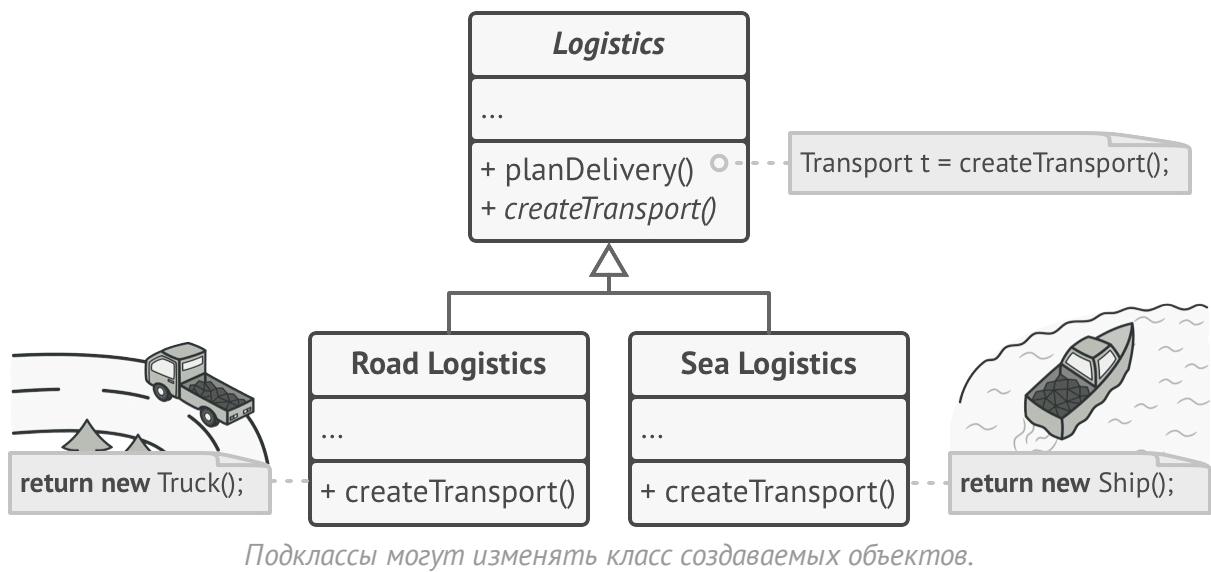
Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.

Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам `Грузовиков`. Чтобы добавить в программу классы морских `Судов`, понадобится перелопатить 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.

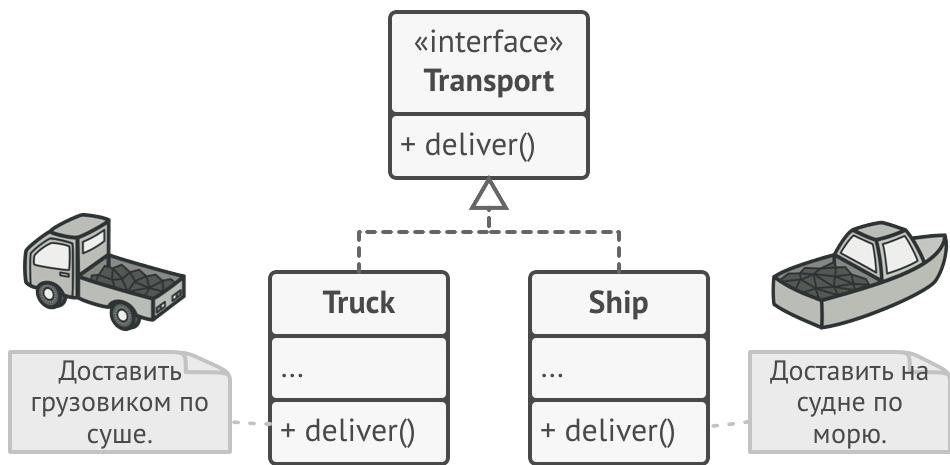
Решение

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.



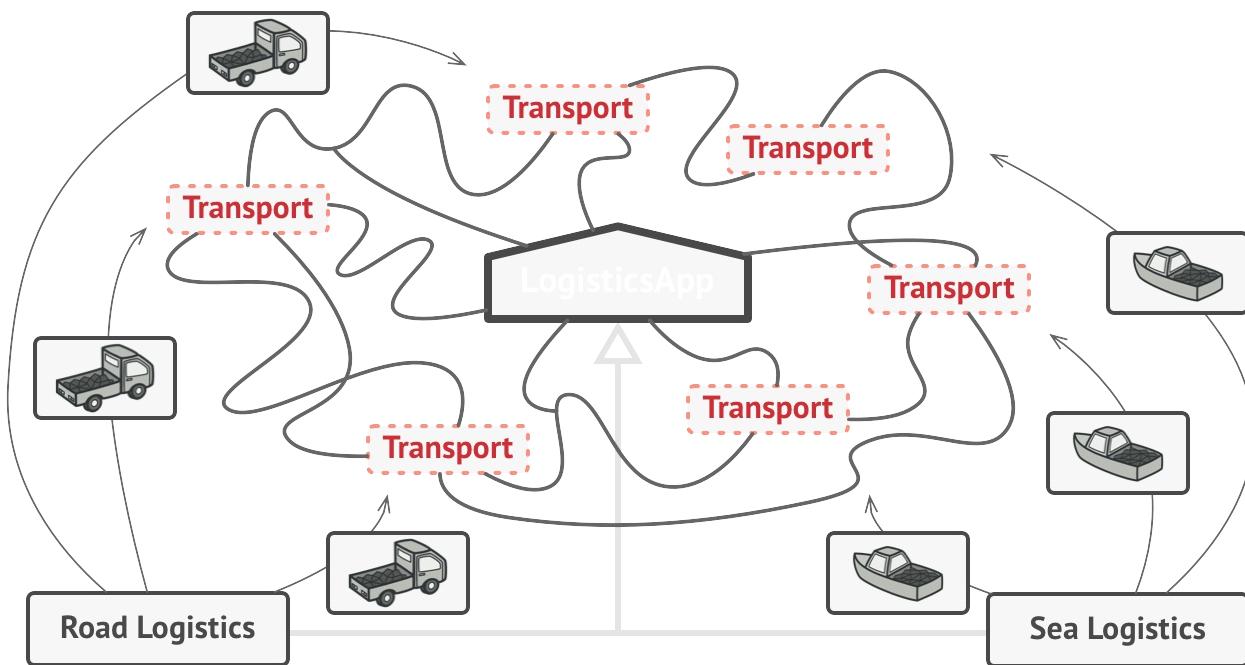
На первый взгляд, это может показаться бессмысленным: мы просто переместили вызов конструктора из одного конца программы в другой. Но теперь вы сможете переопределить фабричный метод в подклассе, чтобы изменить тип создаваемого продукта.

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



Все объекты-продукты должны иметь общий интерфейс.

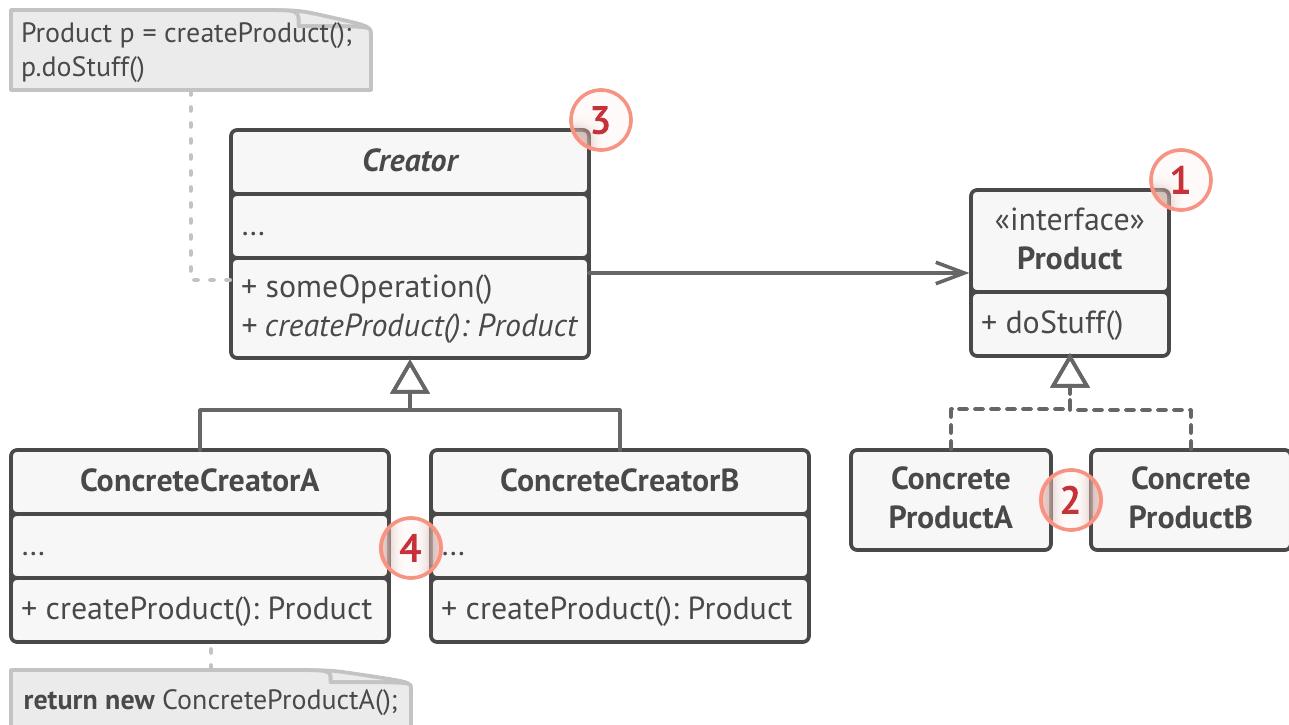
Например, классы Грузовик и Судно реализуют интерфейс Транспорт с методом доставить. Каждый из этих классов реализует метод по-своему: грузовики везут грузы по земле, а судна – по морю. Фабричный метод в классе ДорожнойЛогистики вернёт объект-грузовик, а класс МорскойЛогистики – объект-судно.



Пока все продукты реализуют общий интерфейс, их объекты можно взаимозаменять в клиентском коде.

Для клиента фабричного метода нет разницы между этими объектами, так как он будет трактовать их как некий абстрактный Транспорт . Для него будет важно, чтобы объект имел метод доставить , а как конкретно он работает – не важно.

Структура



1. **Продукт** определяет общий интерфейс объектов, которые может произвести создатель и его подклассы.
2. **Конкретные продукты** содержат код различных продуктов. Продукты будут отличаться реализацией, но интерфейс у них будет общий.
3. **Создатель** объявляет фабричный метод, который должен возвращать новые объекты продуктов. Важно, чтобы тип результата совпадал с общим интерфейсом продуктов.

Зачастую фабричный метод объявляют абстрактным, чтобы заставить все подклассы реализовать его по-своему. Но он может возвращать и некий стандартный продукт.

Несмотря на название, важно понимать, что создание продуктов **не является** единственной функцией создателя. Обычно он содержит и другой полезный код работы с продуктом. Аналогия: большая софтверная компания может иметь центр подготовки программистов, но основная задача компании — создавать программные продукты, а не готовить программистов.

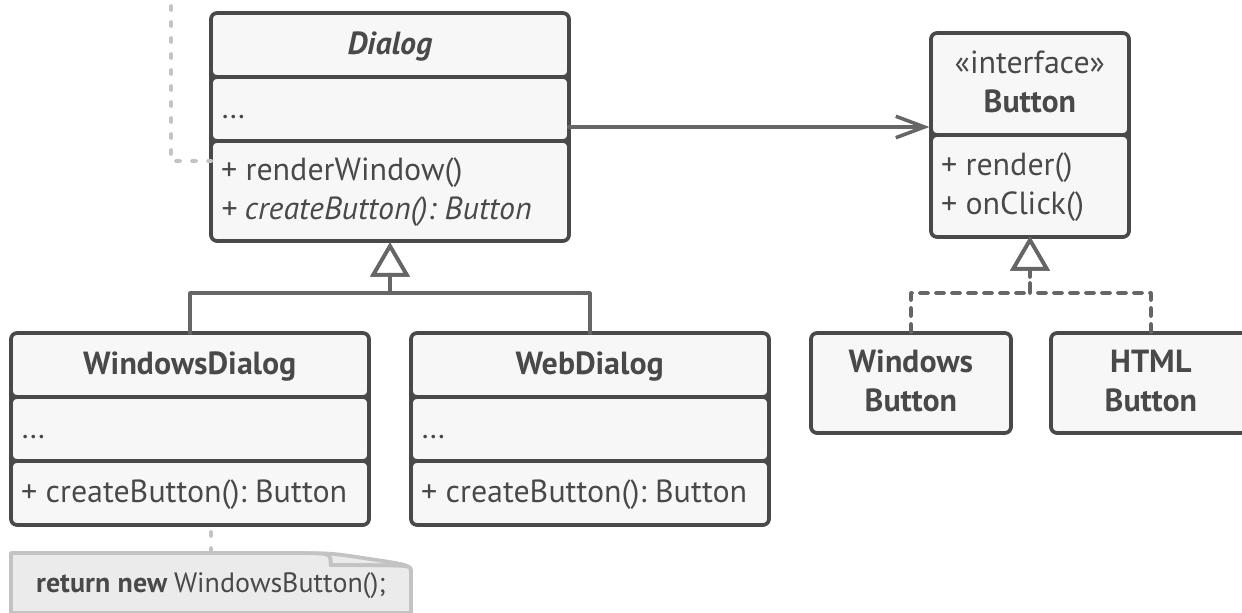
4. **Конкретные создатели** по-своему реализуют фабричный метод, производя те или иные конкретные продукты.

Фабричный метод не обязан всё время создавать новые объекты. Его можно переписать так, чтобы возвращать существующие объекты из какого-то хранилища или кэша.

Псевдокод

В этом примере **Фабричный метод** помогает создавать кросс-платформенные элементы интерфейса, не привязывая основной код программы к конкретным классам элементов.

```
Button okButton = createButton();
okButton.onClick(closeDialog);
okButton.render();
```



Пример кросс-платформенного диалога.

Фабричный метод объявлен в классе диалогов. Его подклассы относятся к различным операционным системам. Благодаря фабричному методу, вам не нужно переписывать логику диалогов под каждую систему. Подклассы могут наследовать почти весь код из базового диалога, изменяя типы кнопок и других элементов, из которых базовый код строит окна графического пользовательского интерфейса.

Базовый класс диалогов работает с кнопками через их общий программный интерфейс. Поэтому, какую вариацию кнопок бы ни вернул фабричный метод, диалог останется рабочим. Базовый класс не зависит от конкретных классов кнопок, оставляя подклассам решение о том, какой тип кнопок создавать.

Такой подход можно применить и для создания других элементов интерфейса. Хотя каждый новый тип элементов будет приближать вас к [Абстрактной фабрике](#).

```
1 // Паттерн Фабричный метод применим тогда, когда в программе
2 // есть иерархия классов продуктов.
3 interface Button is
4     method render()
5     method onClick(f)
6
```

```
7 class WindowsButton implements Button is
8     method render(a, b) is
9         // Отрисовать кнопку в стиле Windows.
10    method onClick(f) is
11        // Навесить на кнопку обработчик событий Windows.
12
13 class HTMLButton implements Button is
14    method render(a, b) is
15        // Вернуть HTML-код кнопки.
16    method onClick(f) is
17        // Навесить на кнопку обработчик события браузера.
18
19
20 // Базовый класс фабрики. Заметьте, что "фабрика" – это всего
21 // лишь дополнительная роль для класса. Скорее всего, он уже
22 // имеет какую-то бизнес-логику, в которой требуется создание
23 // разнообразных продуктов.
24 class Dialog is
25     method renderWindow() is
26         // Чтобы использовать фабричный метод, вы должны
27         // убедиться в том, что эта бизнес-логика не зависит от
28         // конкретных классов продуктов. Button – это общий
29         // интерфейс кнопок, поэтому все хорошо.
30     Button okButton = createButton()
31     okButton.onClick(closeDialog)
32     okButton.render()
33
34 // Мы выносим весь код создания продуктов в особый метод,
35 // который называют "фабричным".
36 abstract method createButton()
37
38
39 // Конкретные фабрики переопределяют фабричный метод и
40 // возвращают из него собственные продукты.
41 class WindowsDialog extends Dialog is
42     method createButton() is
43         return new WindowsButton()
44
45 class WebDialog extends Dialog is
46     method createButton() is
47         return new HTMLButton()
```

```
48
49
50 class ClientApplication is
51   field dialog: Dialog
52
53   // Приложение создаёт определённую фабрику в зависимости от
54   // конфигурации или окружения.
55 method initialize() is
56   config = readApplicationConfigFile()
57
58   if (config.OS == "Windows") then
59     dialog = new WindowsDialog()
60   else if (config.OS == "Web") then
61     dialog = new WebDialog()
62   else
63     throw new Exception("Error! Unknown operating system.")
64
65   // Если весь остальной клиентский код работает с фабриками и
66   // продуктами только через общий интерфейс, то для него
67   // будет не важно, какая фабрика была создана изначально.
68 method main() is
69   dialog.initialize()
70   dialog.render()
```

Применимость

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.

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.

Например, вы используете готовый UI-фреймворк для своего приложения. Но вот беда – требуется иметь круглые кнопки, вместо стандартных прямоугольных. Вы создаёте класс `RoundButton`. Но как сказать главному классу фреймворка `UIFramework`, чтобы он теперь создавал круглые кнопки, вместо стандартных?

Для этого вы создаёте подкласс `UIWithRoundButtons` из базового класса фреймворка, переопределяете в нём метод создания кнопки (а-ля `createButton`) и вписываете туда создание своего класса кнопок. Затем используете `UIWithRoundButtons` вместо стандартного `UIFramework`.

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 вам нужно совершить, чтобы повторно использовать существующие объекты:

1. Сначала вам следует создать общее хранилище, чтобы хранить в нём все создаваемые объекты.

2. При запросе нового объекта нужно будет заглянуть в хранилище и проверить, есть ли там неиспользуемый объект.
3. А затем вернуть его клиентскому коду.
4. 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

1. Приведите все создаваемые продукты к общему интерфейсу.
2. В классе, который производит продукты, создайте пустой фабричный метод. В качестве возвращаемого типа укажите общий интерфейс продукта.
3. Затем пройдитесь по коду класса и найдите все участки, 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. Но не волнуйтесь, мы вот-вот исправим это.

4. Для каждого типа продуктов заведите подкласс и переопределите в нём фабричный метод. Переместите туда код создания соответствующего продукта из суперкласса.
5. 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.

6. 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.

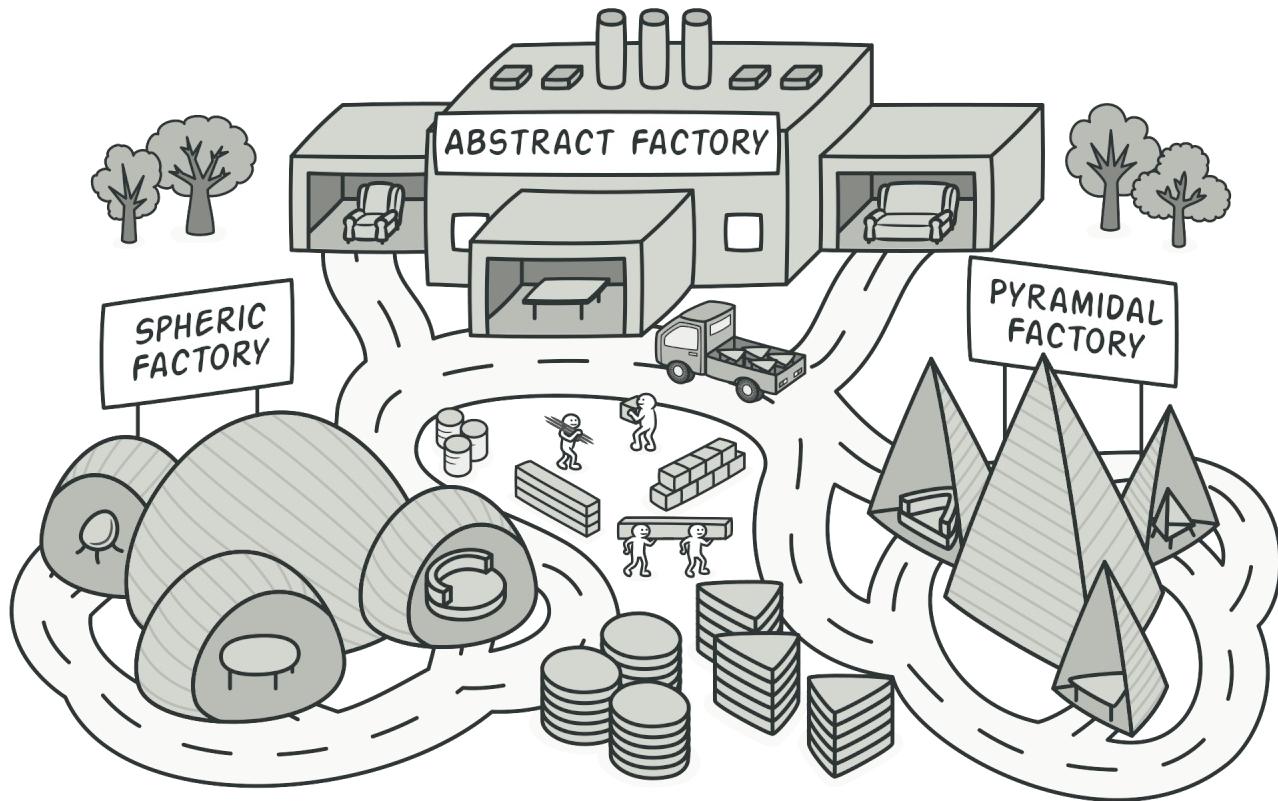
Упрощает добавление новых продуктов в программу.

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.



АБСТРАКТНАЯ ФАБРИКА

Также известен как: *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:

1. Семейство зависимых продуктов. Скажем, Кресло + Диван + Столик .
2. Несколько вариаций этого семейства. Например, продукты Кресло , Диван И Столик представлены в трёх разных стилях: Ар-деко , Викторианском И Модерне .



Семейства продуктов и их вариации.

Вам 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.

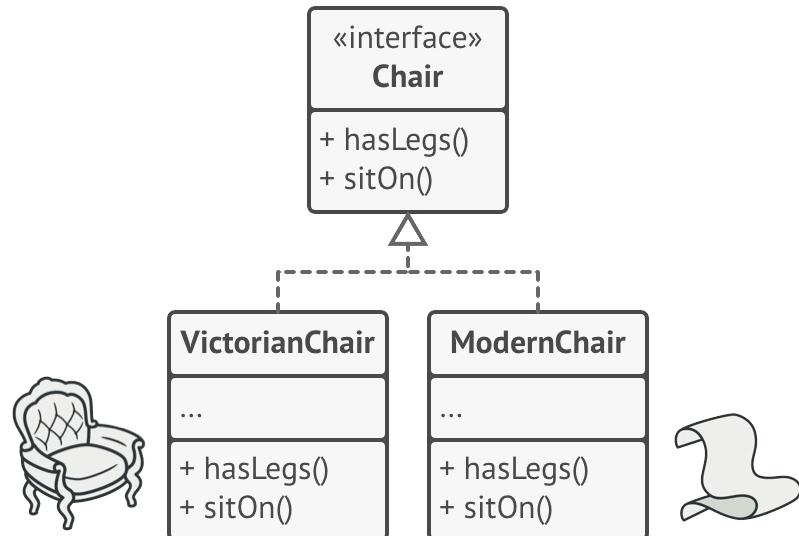


Клиенты расстраиваются, если получают несочетающиеся продукты.

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.

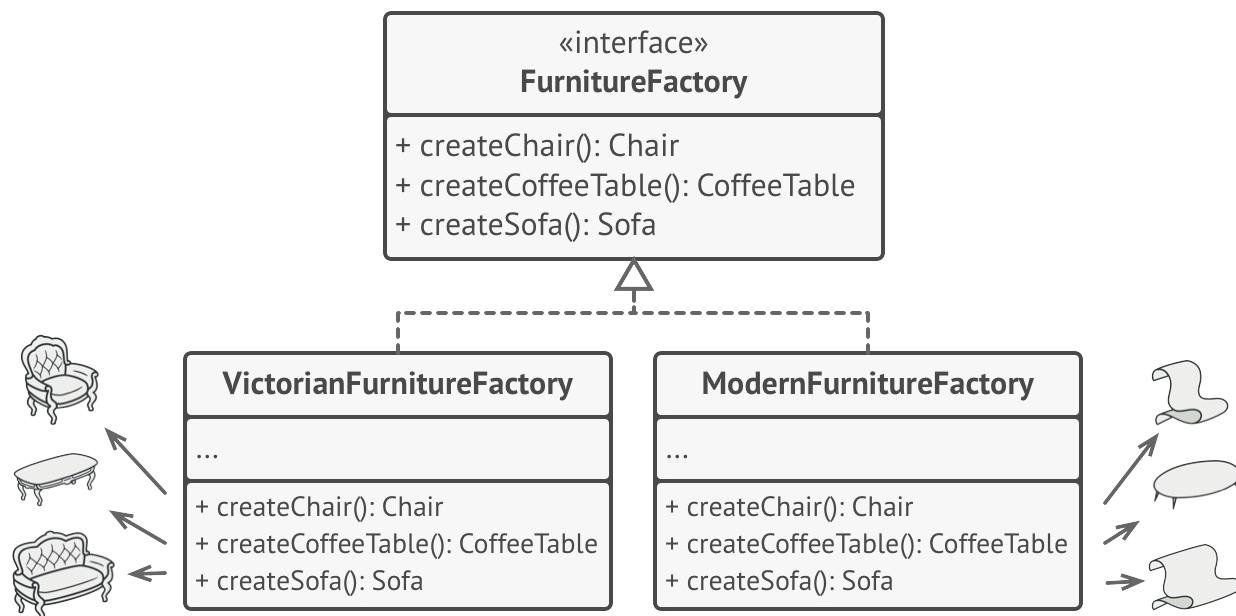
Решение

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.

Далее вы создаёте *абстрактную фабрику* – общий интерфейс, который содержит методы создания всех продуктов семейства (например, `создатьКресло` , `создатьДиван` И `создатьСтолик`). Эти операции должны возвращать **абстрактные** типы продуктов, представленные интерфейсами, которые мы выделили ранее – Кресла , Диваны И Столики .



Конкретные фабрики соответствуют определённой вариации семейства продуктов.

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 ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

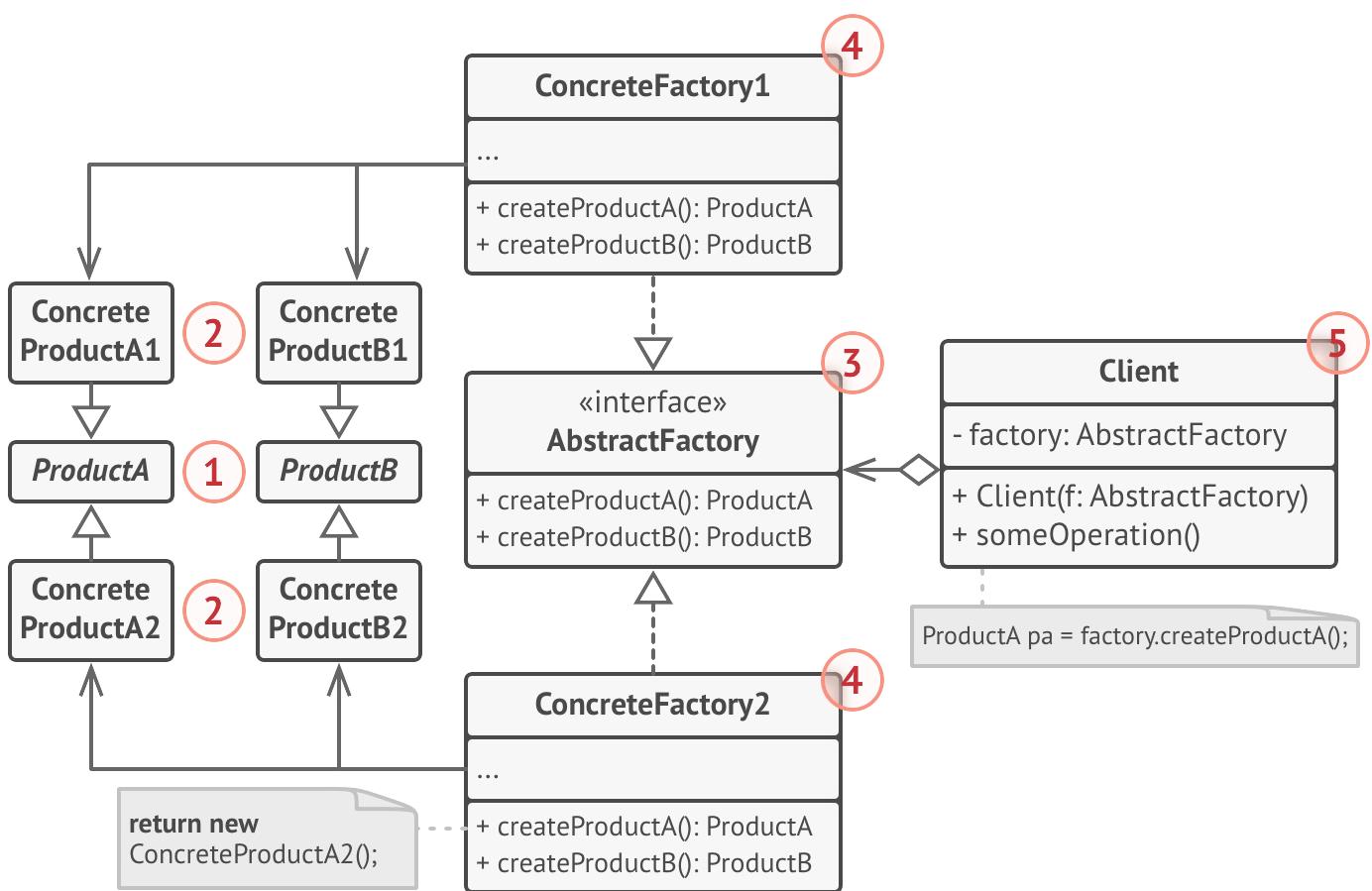


Для клиентского кода должно быть безразлично, с какой фабрикой работать.

Например, клиентский код просит фабрику сделать стул. Он не знает, какого типа была эта фабрика. Он не знает, получит викторианский или модерновый стул. Для него важно, чтобы на стуле можно было сидеть и чтобы этот стул отлично смотрелся с диваном той же фабрики.

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.

Структура

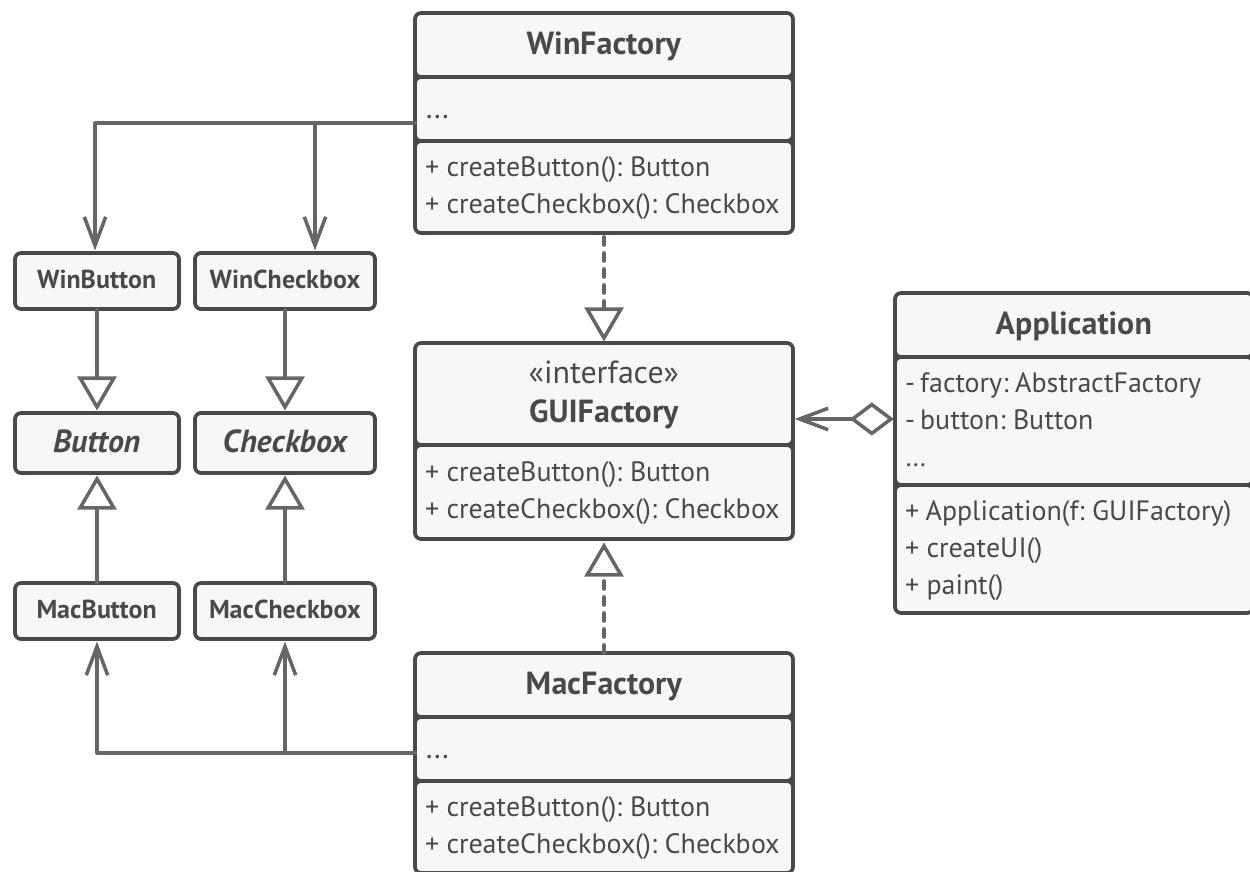


- 1. Абстрактные продукты** объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.
- 2. Конкретные продукты** – большой набор классов, которые относятся к различным абстрактным продуктам (кресло/столик), но имеют одни и те же вариации (Викторианский/Модерн).
- 3. Абстрактная фабрика** объявляет методы создания различных абстрактных продуктов (кресло/столик).
- 4. Конкретные фабрики** относятся каждая к своей вариации продуктов (Викторианский/Модерн) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.
- Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам.

продуктов. Клиент сможет работать с любыми вариациями продуктов через абстрактные интерфейсы.

Псевдокод

В этом примере **Абстрактная фабрика** создаёт кросс-платформенные элементы интерфейса и следит за тем, чтобы они соответствовали выбранной операционной системе.



Пример кросс-платформенного графического интерфейса пользователя.

Кросс-платформенная программа может показывать одни и те же элементы интерфейса, выглядящие чуточку по-другому в различных операционных системах. В такой программе важно, чтобы все создаваемые элементы всегда соответствовали текущей операционной системе. Вы бы не хотели, чтобы программа, запущенная на Windows, вдруг начала показывать чекбоксы в стиле macOS.

Абстрактная фабрика объявляет список создающих методов, которые клиентский код может использовать для получения тех или иных разновидностей элементов интерфейса. Конкретные фабрики относятся к различным операционным системам и создают элементы, совместимые с этой системой.

В самом начале программа определяет, какая из фабрик соответствует текущей операционке. Затем создаёт эту фабрику и отдаёт её клиентскому коду. В дальнейшем клиент будет работать только с этой фабрикой, чтобы исключить несовместимость возвращаемых продуктов.

Клиентский код не зависит от конкретных классов фабрик и элементов интерфейса. Он общается с ними через абстрактные интерфейсы. Благодаря этому клиент может работать с любой разновидностью фабрик и элементов интерфейса.

Чтобы добавить в программу новую вариацию элементов (например, для поддержки Linux), вам не нужно трогать клиентский код. Достаточно создать ещё одну фабрику, производящую эти элементы.

```
1 // Этот паттерн предполагает, что у вас есть несколько семейств
2 // продуктов, находящихся в отдельных иерархиях классов
3 // (Button/Checkbox). Продукты одного семейства должны иметь
4 // общий интерфейс.
5 interface Button is
6     method paint()
7
8 // Семейства продуктов имеют те же вариации (macOS/Windows).
9 class WinButton implements Button is
10    method paint() is
11        // Отрисовать кнопку в стиле Windows.
12
13 class MacButton implements Button is
14    method paint() is
15        // Отрисовать кнопку в стиле macOS.
16
17
18 interface Checkbox is
19     method paint()
```

```
20
21 class WinCheckbox implements Checkbox is
22     method paint() is
23         // Отрисовать чекбокс в стиле Windows.
24
25 class MacCheckbox implements Checkbox is
26     method paint() is
27         // Отрисовать чекбокс в стиле macOS.
28
29
30 // Абстрактная фабрика знает обо всех абстрактных типах
31 // продуктов.
32 interface GUIFactory is
33     method createButton():Button
34     method createCheckbox():Checkbox
35
36
37 // Каждая конкретная фабрика знает и создаёт только продукты
38 // своей вариации.
39 class WinFactory implements GUIFactory is
40     method createButton():Button is
41         return new WinButton()
42     method createCheckbox():Checkbox is
43         return new WinCheckbox()
44
45 // Несмотря на то, что фабрики оперируют конкретными классами,
46 // их методы возвращают абстрактные типы продуктов. Благодаря
47 // этому фабрики можно взаимозаменять, не изменяя клиентский
48 // код.
49 class MacFactory implements GUIFactory is
50     method createButton():Button is
51         return new MacButton()
52     method createCheckbox():Checkbox is
53         return new MacCheckbox()
54
55
56 // Для кода, использующего фабрику, не важно, с какой конкретно
57 // фабрикой он работает. Все получатели продуктов работают с
58 // ними через общие интерфейсы.
59 class Application is
60     private field button: Button
```

```
61 constructor Application(factory: GUIFactory) is
62     this.factory = factory
63 method createUI()
64     this.button = factory.createButton()
65 method paint()
66     button.paint()
67
68
69 // Приложение выбирает тип конкретной фабрики и создаёт её
70 // динамически, исходя из конфигурации или окружения.
71 class ApplicationConfigurator is
72     method main() is
73         config = readApplicationConfigFile()
74
75         if (config.OS == "Windows") then
76             factory = new WinFactory()
77         else if (config.OS == "Web") then
78             factory = new MacFactory()
79         else
80             throw new Exception("Error! Unknown operating system.")
81
82     Application app = new Application(factory)
```

Применимость

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.

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

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

Преимущества и недостатки

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.

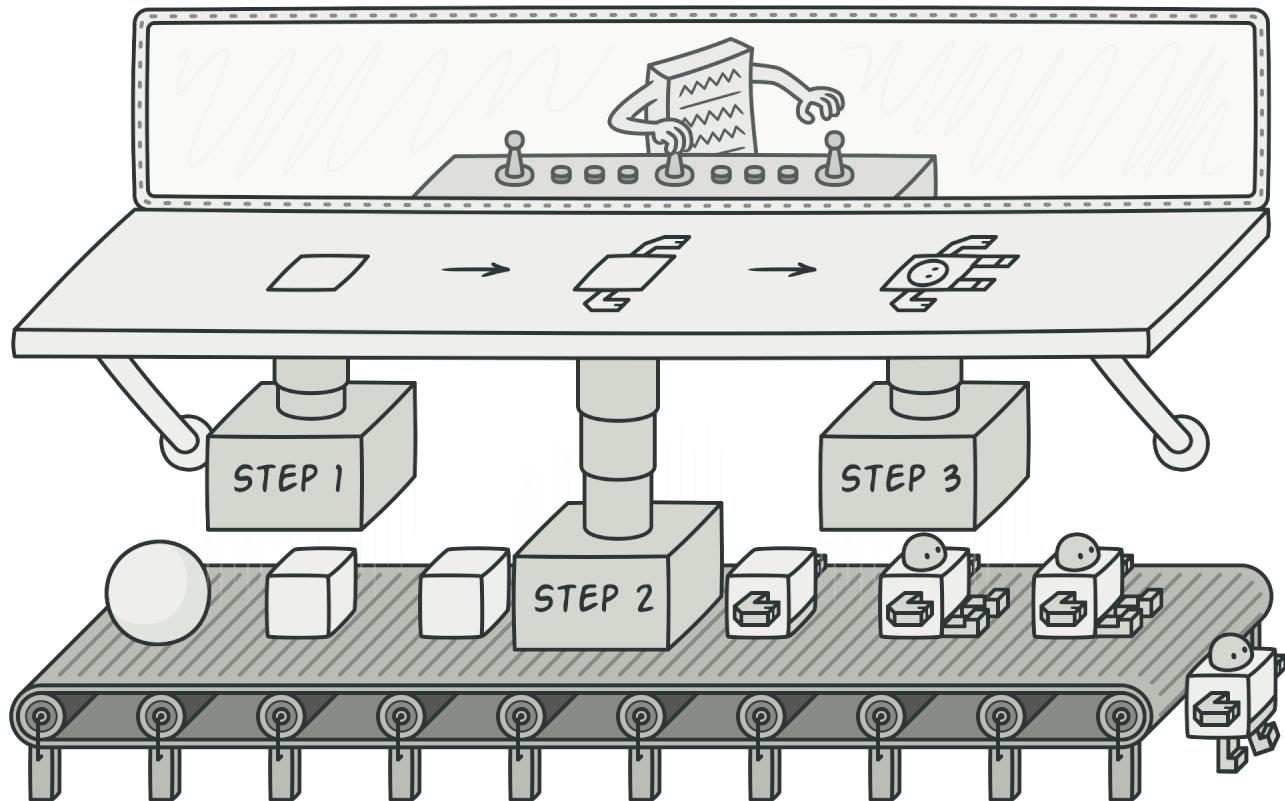
Реализует принцип открытости/закрытости.

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**.



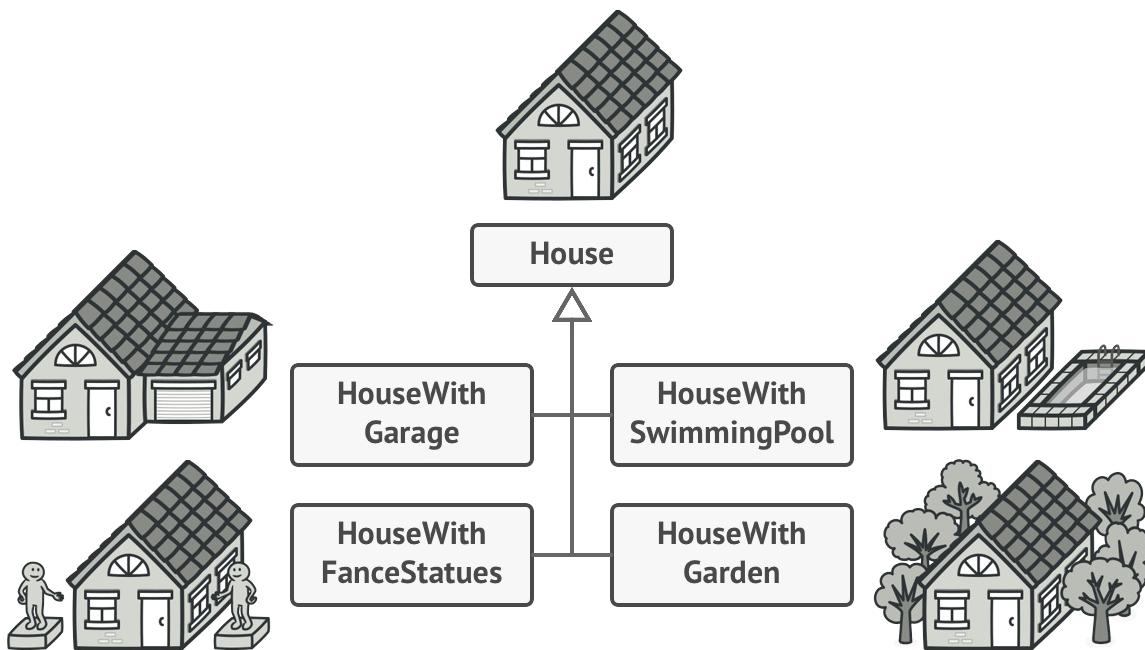
СТРОИТЕЛЬ

Также известен как: *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.

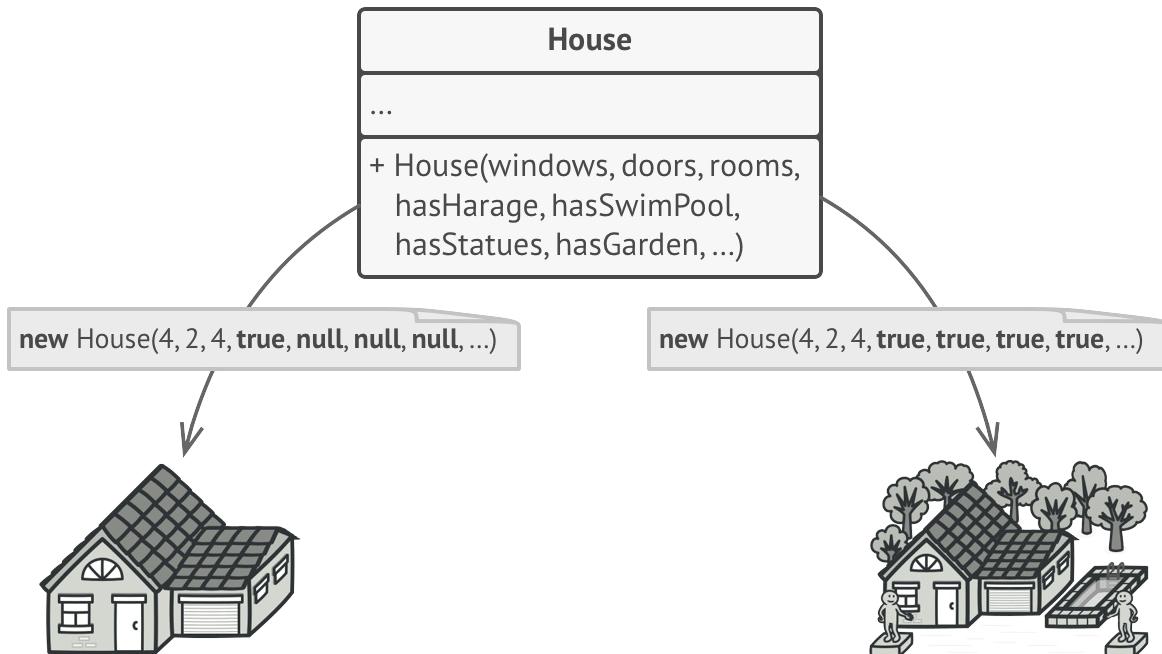


Создав кучу подклассов для всех конфигураций объектов, вы можете излишне усложнить программу.

Например, давайте подумаем о том, как создать объект `Дом`. Чтобы построить стандартный дом, нужно поставить 4 стены, установить двери, вставить пару окон и положить крышу. Но что, если вы хотите дом побольше да посветлее, имеющий сад, бассейн и прочее добро?

Самое простое решение – расширить класс `Дом`, создав подклассы для всех комбинаций параметров дома. Проблема такого подхода – это громадное количество классов, которые вам придётся создать. Каждый новый параметр, вроде цвета обоев или материала кровли, заставит вас создавать всё больше и больше классов для перечисления всех возможных вариантов.

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.

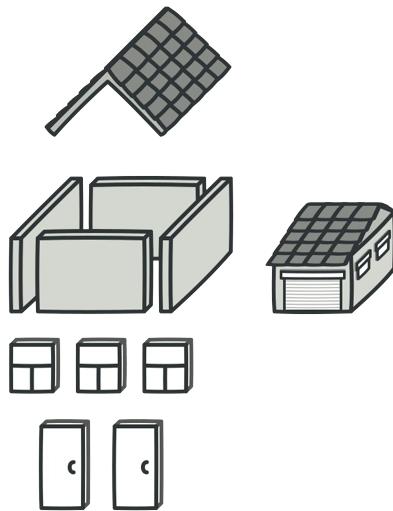
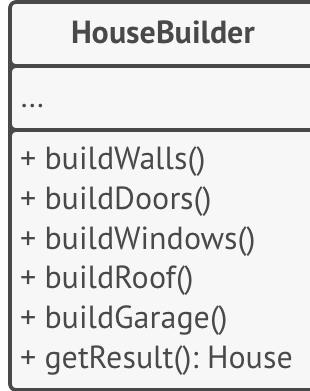


Конструктор со множеством параметров имеет свой недостаток: не все параметры нужны большую часть времени.

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*.

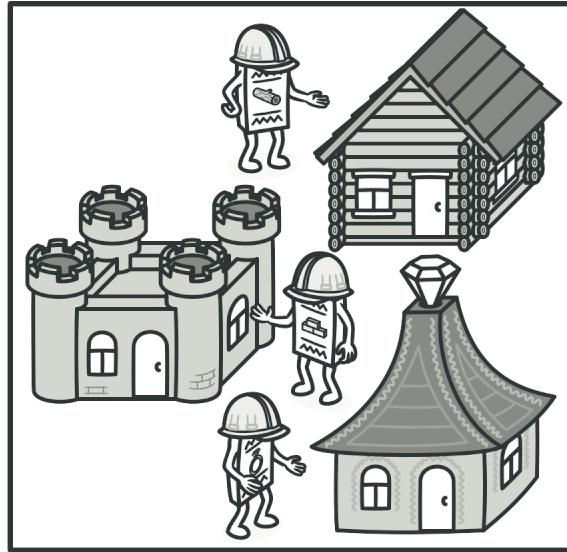


Строитель позволяет создавать сложные объекты пошагово. Промежуточный результат защищён от стороннего вмешательства.

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.



Разные строители выполняют одну и ту же задачу по-разному.

Например, один строитель делает стены из дерева и стекла, другой из камня и железа, третий из золота и бриллиантов. Вызвав одни и те же шаги строительства, в первом случае вы получите обычный жилой дом, во втором – маленькую крепость, а в третьем – роскошное жилище. Замечу, что код, который вызывает шаги строительства, должен работать со строителями через общий интерфейс, чтобы их можно было свободно взаимозаменять.

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.

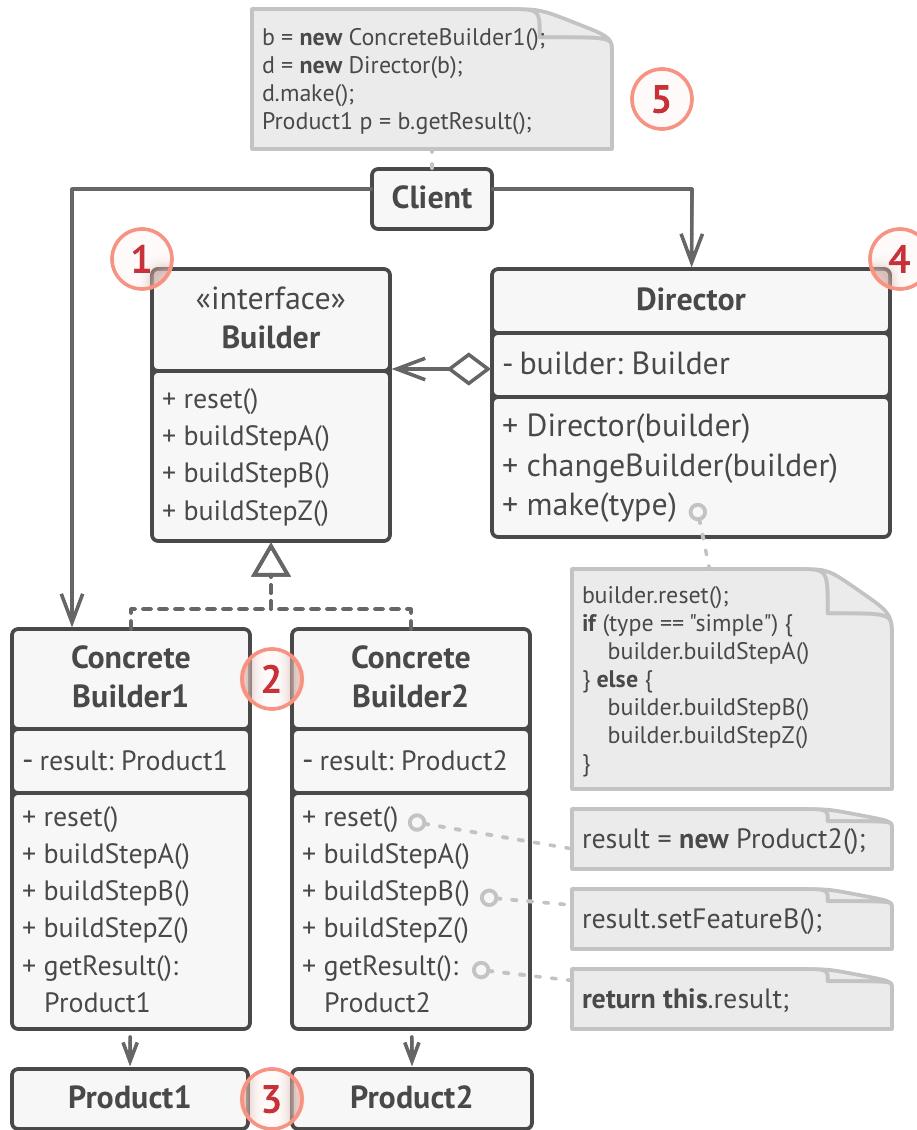


Директор знает, какие шаги должен выполнить объект-строитель, чтобы произвести продукт.

Отдельный класс *директора* не является строго обязательным. Вы можете вызывать методы строителя и напрямую из клиентского кода. Тем не менее, 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

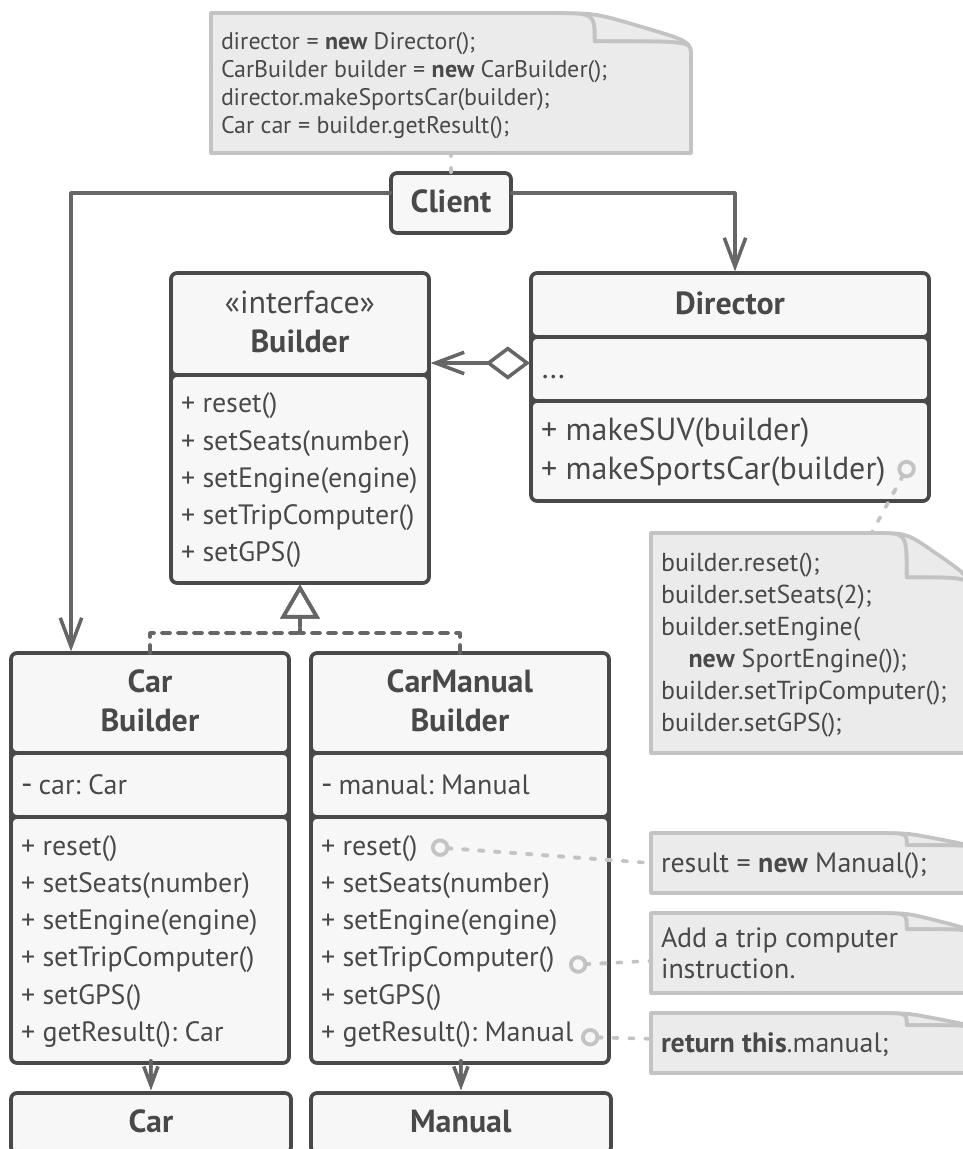


1. **Интерфейс строителя** объявляет шаги конструирования продуктов, общие для всех видов строителей.
2. **Конкретные строители** реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.
3. **Продукт** – создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.
4. **Директор** определяет порядок вызова строительных шагов для производства той или иной конфигурации продуктов.
5. Обычно **Клиент** подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Но возможен и другой вариант,

когда клиент передаёт строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

Псевдокод

В этом примере **Строитель** используется для пошагового конструирования автомобилей, а также технических руководств к ним.



Пример пошагового конструирования автомобилей и инструкций к ним.

Автомобиль – это сложный объект, который может быть сконфигурирован сотней разных способов. Вместо того, чтобы настраивать автомобиль через конструктор, мы вынесем его

сборку в отдельный класс-строитель, предусмотрев методы для конфигурации всех частей автомобиля.

Клиент может собирать автомобили, работая со строителем напрямую. Но, с другой стороны, он может поручить это дело директору. Это объект, который знает, какие шаги строителя нужно вызвать, чтобы получить несколько самых популярных конфигураций автомобилей.

Но к каждому автомобилю нужно ещё и руководство, совпадающее с его конфигурацией. Для этого мы создадим ещё один класс строителя, который вместо конструирования автомобиля, будет печатать страницы руководства к той детали, которую мы встраиваем в продукт. Теперь, пропустив оба типа строителей через одни и те же шаги, мы получим автомобиль и подходящее к нему руководство пользователя.

Очевидно, что бумажное руководство и железный автомобиль – это две разных вещи, не имеющих ничего общего. По этой причине мы должны получать результат напрямую от строителей, а не от директора. Иначе нам пришлось бы жёстко привязать директора к конкретным классам автомобилей и руководств.

```
1 // Строитель может создавать различные продукты, используя один
2 // и тот же процесс строительства.
3 class Car is
4     // Автомобили могут отличаться комплектацией: типом
5     // двигателя, количеством сидений, могут иметь или не иметь
6     // GPS и систему навигации и т. д. Кроме того, автомобили
7     // могут быть городскими, спортивными или внедорожниками.
8
9 class Manual is
10    // Руководство пользователя для данной конфигурации
11    // автомобиля.
12
13
14 // Интерфейс строителя объявляет все возможные этапы и шаги
15 // конфигурации продукта.
16 interface Builder is
17     method reset()
18     method setSeats(...)
```

```
19 method setEngine(...)  
20 method setTripComputer(...)  
21 method setGPS(...)  
22  
23 // Все конкретные строители реализуют общий интерфейс по-своему.  
24 class CarBuilder implements Builder is  
25     private field car:Car  
26     method reset()  
27         // Поместить новый объект Car в поле "car".  
28     method setSeats(...) is  
29         // Установить указанное количество сидений.  
30     method setEngine(...) is  
31         // Установить поданный двигатель.  
32     method setTripComputer(...) is  
33         // Установить поданную систему навигации.  
34     method setGPS(...) is  
35         // Установить или снять GPS.  
36     method getResult(): Car is  
37         // Вернуть текущий объект автомобиля.  
38  
39 // В отличие от других порождающих паттернов, где продукты  
40 // должны быть частью одной иерархии классов или следовать  
41 // общему интерфейсу, строители могут создавать совершенно  
42 // разные продукты, которые не имеют общего предка.  
43 class CarManualBuilder implements Builder is  
44     private field manual:Manual  
45     method reset()  
46         // Поместить новый объект Manual в поле "manual".  
47     method setSeats(...) is  
48         // Описать, сколько мест в машине.  
49     method setEngine(...) is  
50         // Добавить в руководство описание двигателя.  
51     method setTripComputer(...) is  
52         // Добавить в руководство описание системы навигации.  
53     method setGPS(...) is  
54         // Добавить в инструкцию инструкцию GPS.  
55     method getResult(): Manual is  
56         // Вернуть текущий объект руководства.  
57  
58 // Директор знает, в какой последовательности нужно заставлять
```

```
60 // работать строителя, чтобы получить ту или иную версию
61 // продукта. Заметьте, что директор работает со строителем через
62 // общий интерфейс, благодаря чему он не знает тип продукта,
63 // который изготавливает строитель.
64 class Director is
65     method constructSportsCar(builder: Builder) is
66         builder.reset()
67         builder.setSeats(2)
68         builder.setEngine(new SportEngine())
69         builder.setTripComputer(true)
70         builder.setGPS(true)
71
72
73 // Директор получает объект конкретного строителя от клиента
74 // (приложения). Приложение само знает, какого строителя нужно
75 // использовать, чтобы получить определённый продукт.
76 class Application is
77     method makeCar is
78         director = new Director()
79
80         CarBuilder builder = new CarBuilder()
81         director.constructSportsCar(builder)
82         Car car = builder.getResult()
83
84         CarManualBuilder builder = new CarManualBuilder()
85         director.constructSportsCar(builder)
86
87 // Готовый продукт возвращает строитель, так как
88 // директор чаще всего не знает и не зависит от
89 // конкретных классов строителей и продуктов.
90         Manual manual = builder.getResult()
```

Применимость

Когда вы 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.

```
1 class Pizza {  
2     Pizza(int size) { ... }  
3     Pizza(int size, boolean cheese) { ... }  
4     Pizza(int size, boolean cheese, boolean pepperoni) { ... }  
5     // ...
```

Такого монстра можно создать только в языках, имеющих механизм перегрузки методов, например, C# или Java.

Паттерн Строитель позволяет собирать объекты пошагово, вызывая только те шаги, которые вам нужны. А значит, больше не нужно пытаться «запихнуть» в конструктор все возможные опции продукта.

Когда ваш 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.

Когда вам 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.

Заметьте, что 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

1. Убедитесь в том, что создание разных представлений объекта можно свести к общим шагам.
2. Опишите эти шаги в общем интерфейсе строителей.
3. Для каждого из представлений объекта-продукта создайте по одному классу-строителю и реализуйте их методы строительства.

Не забудьте про метод получения результата. Обычно конкретные строители определяют собственные методы получения результата строительства. Вы не можете описать эти методы в интерфейсе строителей, поскольку продукты не обязательно должны иметь общий базовый класс или интерфейс. Но вы всегда сможете добавить метод получения результата в общий интерфейс, если ваши строители производят однородные продукты с общим предком.

4. Подумайте о создании класса директора. Его методы будут создавать различные конфигурации продуктов, вызывая разные шаги одного и того же строителя.
5. 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.
6. 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

Позволяет создавать продукты пошагово.

Позволяет использовать один и тот же код для создания различных продуктов.

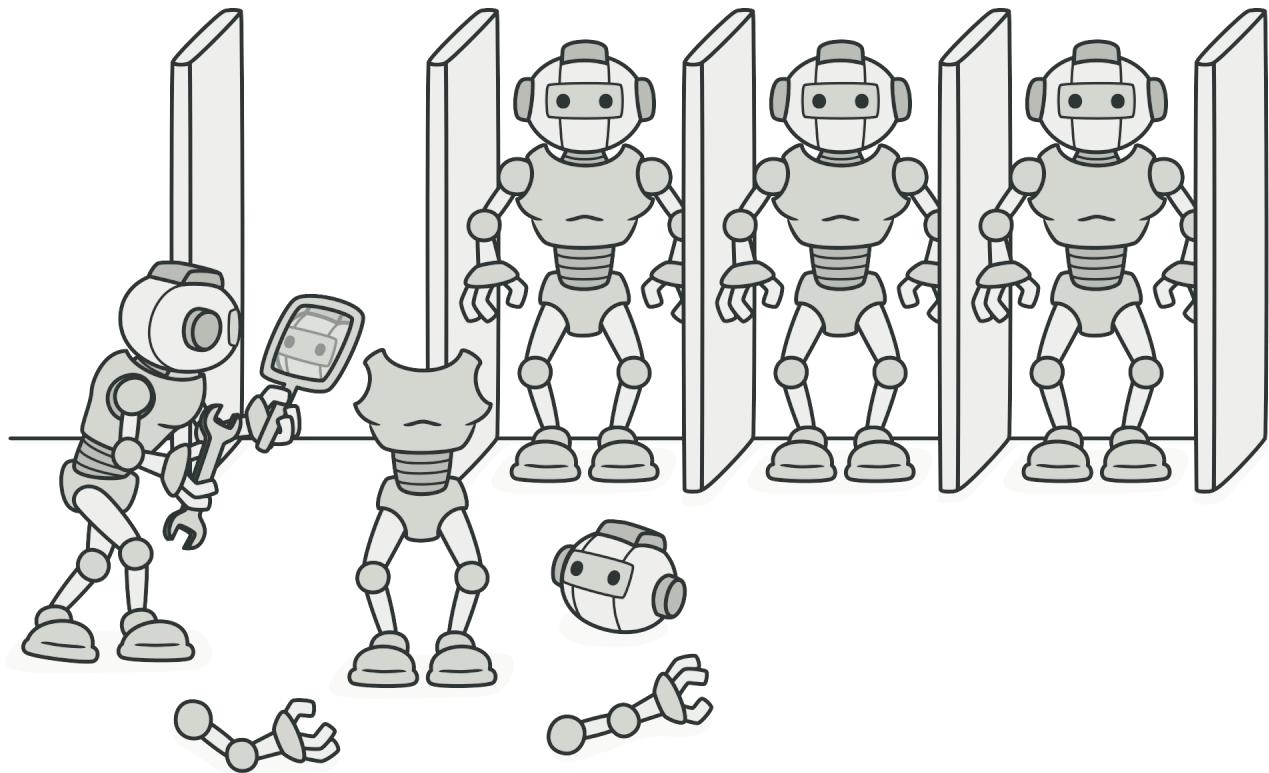
Изолирует сложный код сборки продукта от его основной бизнес-логики.

Усложняет код программы из-за введения дополнительных классов.

Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя 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 строитель – реализации.
- Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.



ПРОТОТИП

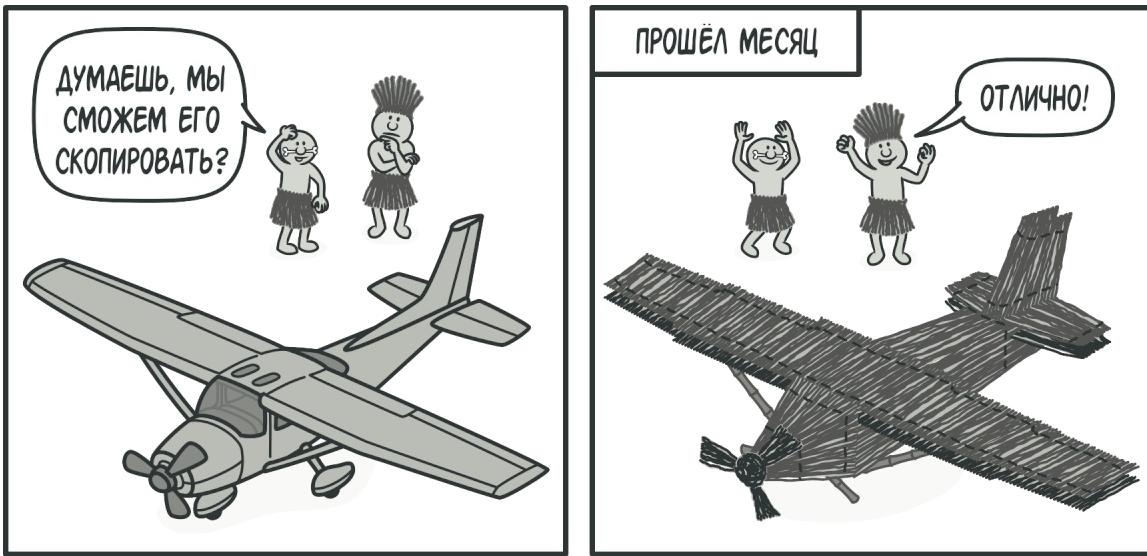
Также известен как: Клон, Prototype

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

Проблема

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

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



Копирование «извне» не всегда возможно в реальности.

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

Решение

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

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

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

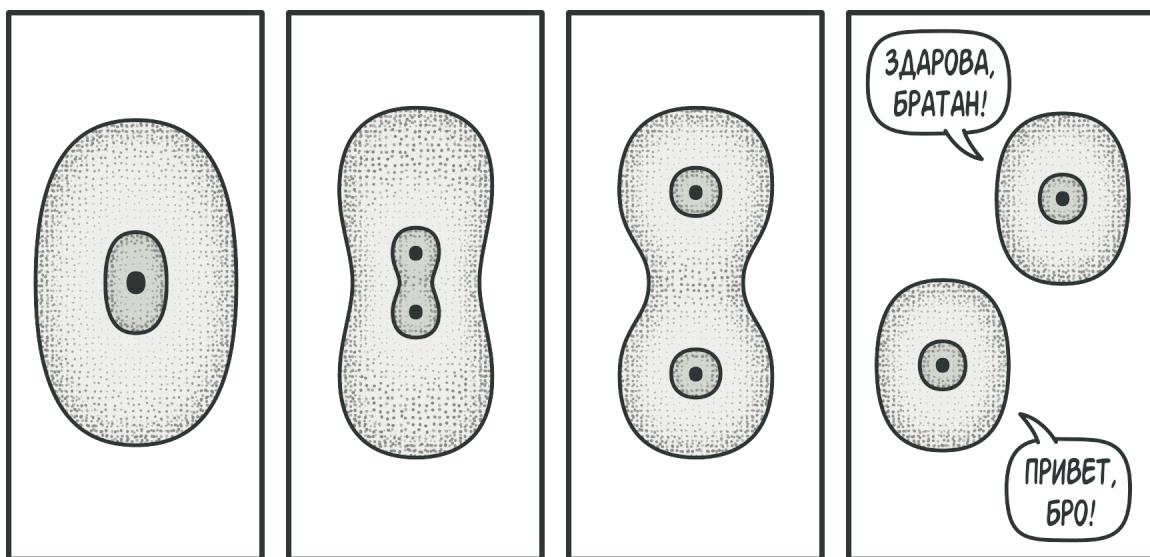


Предварительно заготовленные прототипы могут стать заменой подклассам.

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.

Аналогия из жизни

В промышленном производстве прототипы создаются перед основной партией продуктов для проведения всевозможных испытаний. При этом прототип не участвует в последующем производстве, отыгравая пассивную роль.

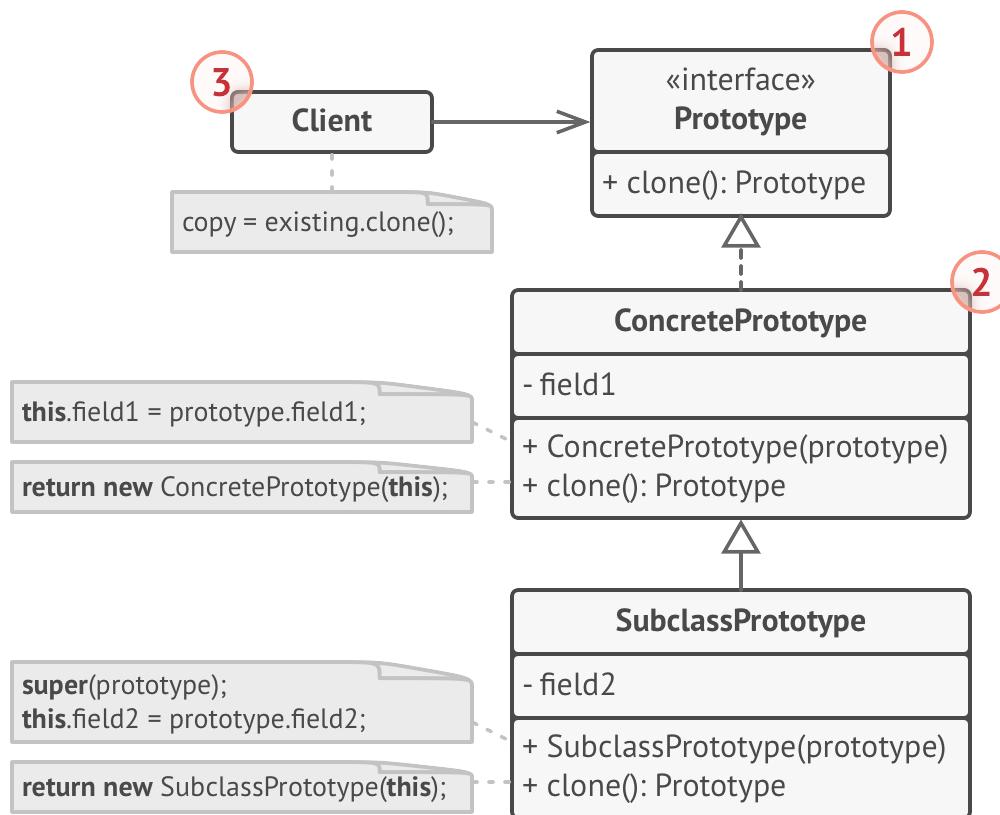


Пример деления клетки.

Прототип на производстве не делает копию самого себя, поэтому более 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.

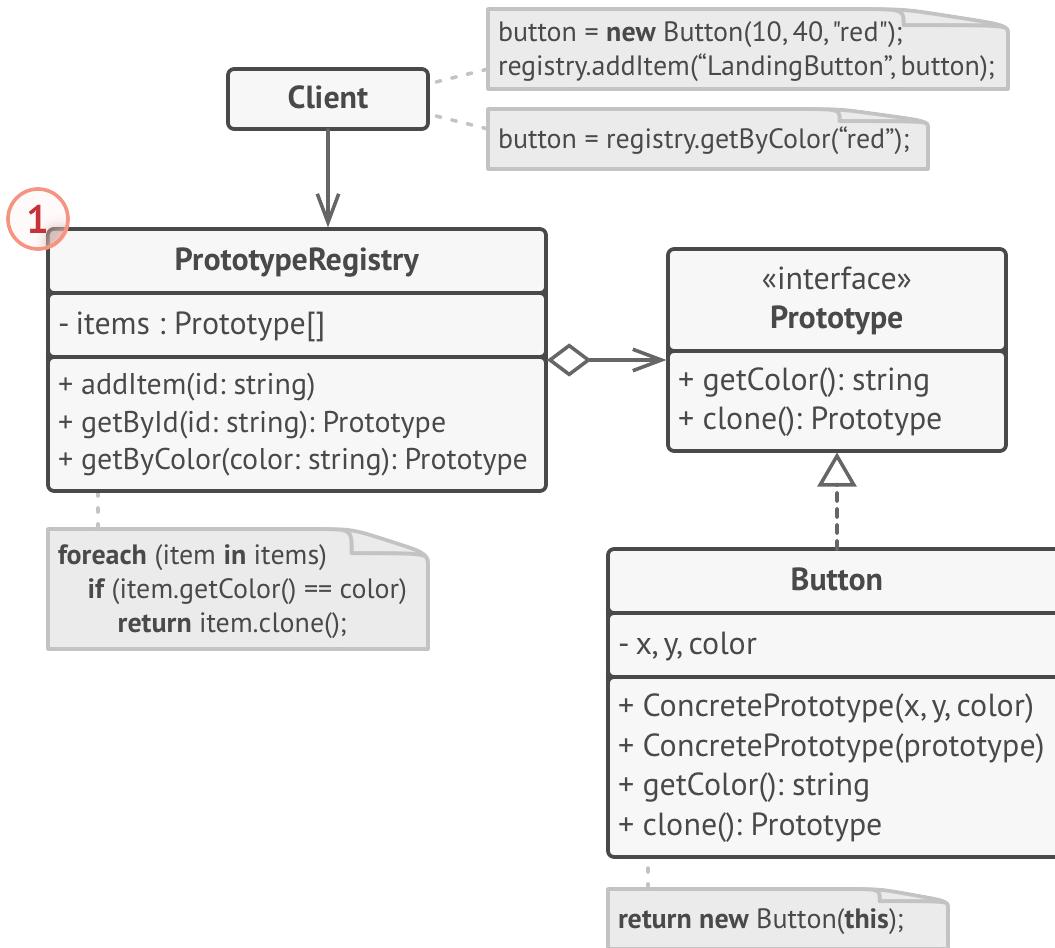
Структура

Базовая реализация



1. **Интерфейс прототипов** описывает операции клонирования. В большинстве случаев – это единственный метод `clone`.
2. **Конкретный прототип** реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.
3. **Клиент** создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

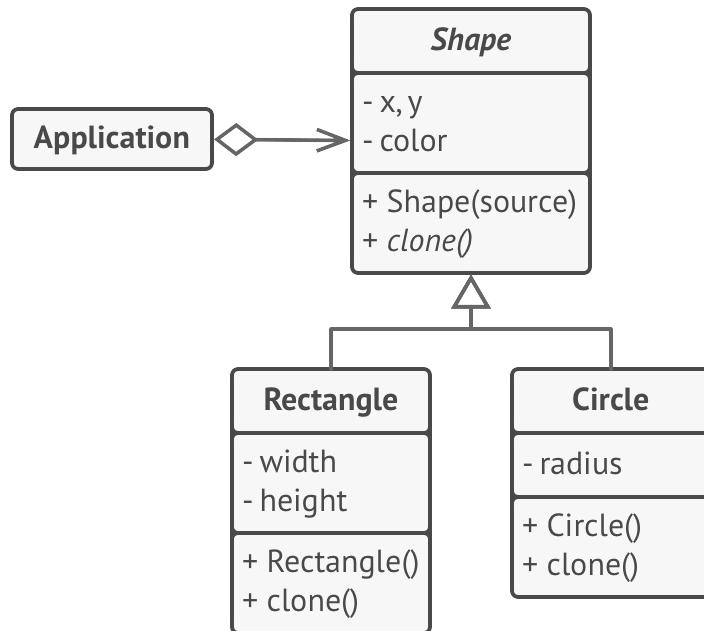
Реализация с общим хранилищем прототипов



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

Псевдокод

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



Пример клонирования иерархии геометрических фигур.

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

```

1 // Базовый прототип.
2 abstract class Shape is
3     field X: int
4     field Y: int
5     field color: string
6
7     // Копирование всех полей объекта происходит в конструкторе.
8     constructor Shape(source: Shape) is
9         if (source != null) then
10             this.X = source.X
11             this.Y = source.Y
12             this.color = source.color
13
14     // Результатом операции клонирования всегда будет объект из
15     // иерархии классов Shape.
16     abstract method clone(): Shape
17
18
19 // Конкретный прототип. Метод клонирования создаёт новый объект
20 // текущего класса, передавая в его конструктор ссылку на

```

```
21 // собственный объект. Благодаря этому операция клонирования
22 // получается атомарной – пока не выполнится конструктор, нового
23 // объекта ещё не существует. Но как только конструктор завершит
24 // работу, мы получим полностью готовый объект-клон, а не пустой
25 // объект, который нужно ещё заполнить.
26 class Rectangle extends Shape is
27     field width: int
28     field height: int
29
30     constructor Rectangle(source: Rectangle) is
31         // Вызов родительского конструктора нужен, чтобы
32         // скопировать потенциальные приватные поля, объявленные
33         // в родительском классе.
34         super(source)
35         if (source != null) then
36             this.width = source.width
37             this.height = source.height
38
39     method clone(): Shape is
40         return new Rectangle(this)
41
42
43 class Circle extends Shape is
44     field radius: int
45
46     constructor Circle(source: Circle) is
47         super(source)
48         if (source != null) then
49             this.radius = source.radius
50
51     method clone(): Shape is
52         return new Circle(this)
53
54
55 // Где-то в клиентском коде.
56 class Application is
57     field shapes: array of Shape
58
59     constructor Application() is
60         Circle circle = new Circle()
61         circle.X = 10
```

```
62     circle.Y = 20
63     circle.radius = 15
64     shapes.add(circle)
65
66     Circle anotherCircle = circle.clone()
67     shapes.add(anotherCircle)
68     // anotherCircle будет содержать точную копию circle.
69
70     Rectangle rectangle = new Rectangle()
71     rectangle.width = 10
72     rectangle.height = 20
73     shapes.add(rectangle)
74
75 method businessLogic() is
76     // Не очевидный плюс Прототипа в том, что вы можете
77     // клонировать набор объектов, не зная их конкретных
78     // классов.
79     Array shapesCopy = new Array of Shapes.
80
81     // Например, мы не знаем, какие конкретно объекты
82     // находятся внутри массива shapes, так как он объявлен
83     // с типом Shape. Но благодаря полиморфизму, мы можем
84     // клонировать все объекты «вслепую». Будет выполнен
85     // метод clone того класса, которым является этот
86     // объект.
87     foreach (s in shapes) do
88         shapesCopy.add(s.clone())
89
90     // Переменная shapesCopy будет содержать точные копии
91     // элементов массива shapes.
```

Применимость

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.

Когда вы 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

1. Создайте интерфейс прототипов с единственным методом `clone`. Если у вас уже есть иерархия продуктов, метод клонирования можно объявить непосредственно в каждом из её классов.
2. 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.

3. 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.
4. 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

Позволяет клонировать объекты, не привязываясь к их конкретным классам.

Меньше повторяющегося кода инициализации объектов.

Ускоряет создание объектов.

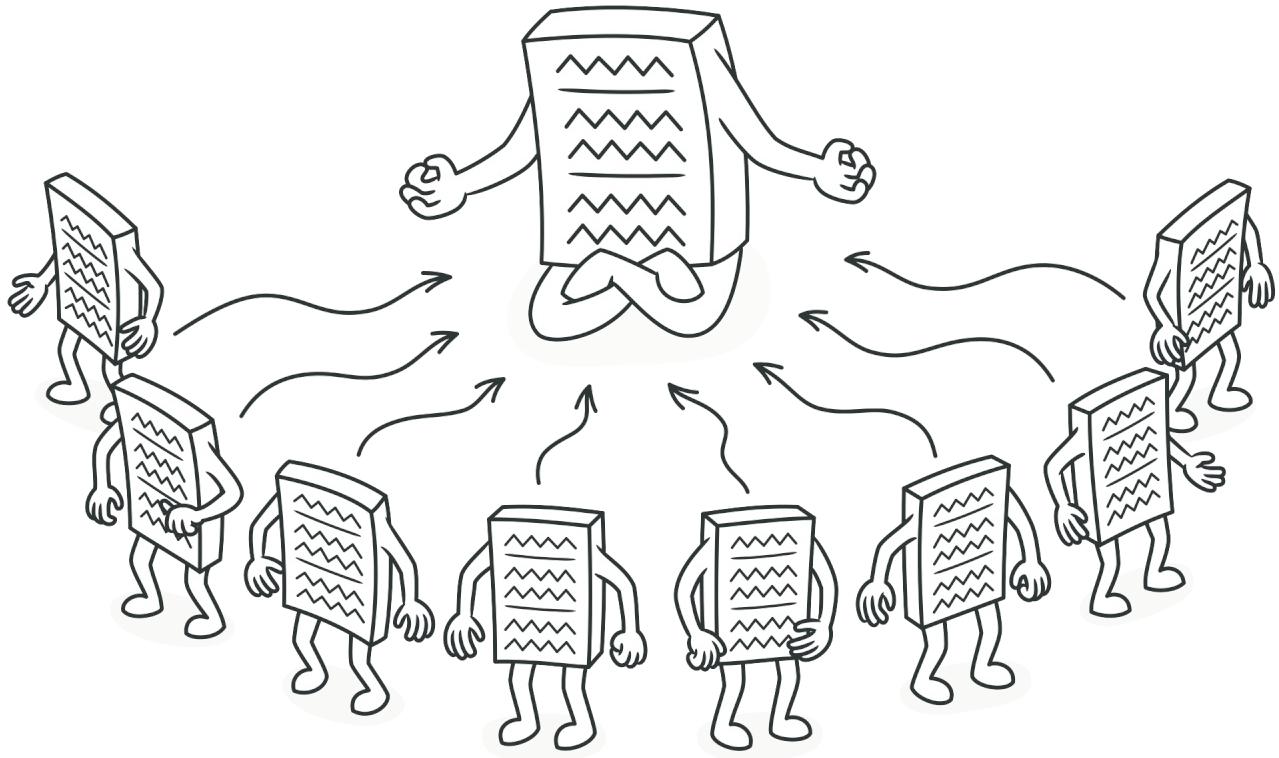
Альтернатива созданию подклассов для конструирования сложных объектов.

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 **Команду** 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**.



ОДИНОЧКА

Также известен как: *Singleton*

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

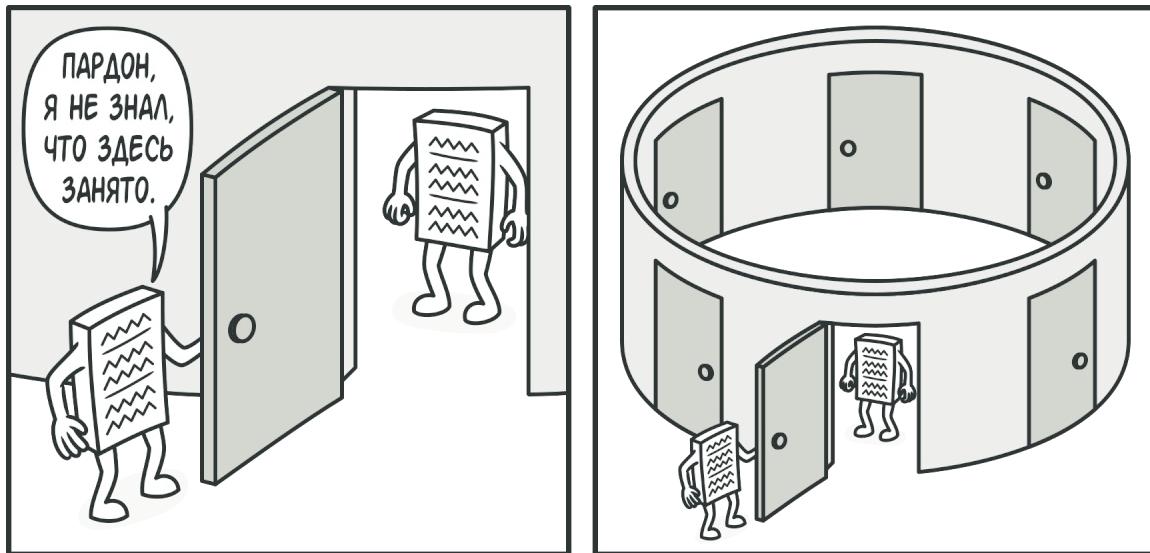
Проблема

Одиночка решает сразу две проблемы, нарушая *принцип единственной ответственности* класса.

- Гарантирует наличие единственного экземпляра класса.** Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

Представьте, что вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса **всегда** возвращает новый объект.



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

2. Предоставляет глобальную точку доступа. Это не просто глобальная переменная, через которую можно достучаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Но есть и другой нюанс. Неплохо бы хранить в одном месте и код, который решает проблему №1, а также иметь к нему простой и доступный интерфейс.

Интересно, что в наше время паттерн стал настолько известен, что теперь люди называют «одиночками» даже те классы, которые решают лишь одну из проблем, перечисленных выше.

Решение

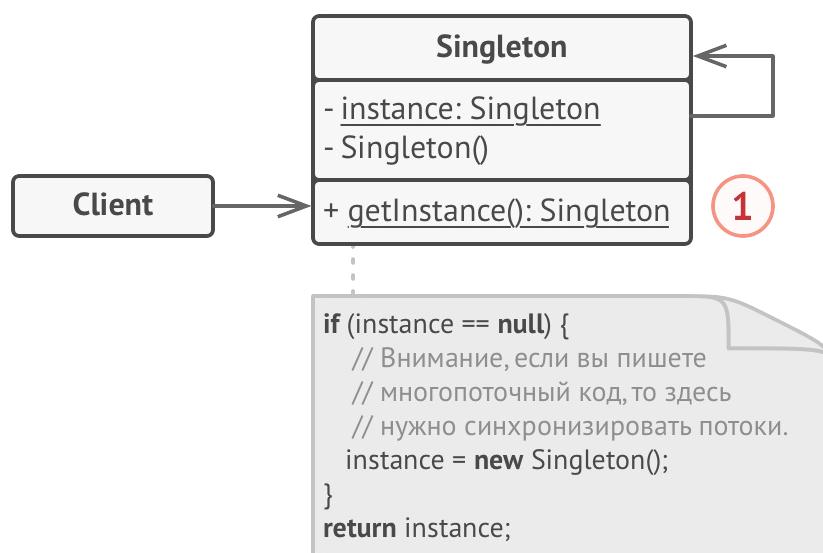
Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

Аналогия из жизни

Правительство государства – хороший пример одиночки. В государстве может быть только одно официальное правительство. вне зависимости от того, кто конкретно заседает в правительстве, оно имеет глобальную точку доступа «Правительство страны N».

Структура



1. **Одиночка** определяет статический метод `getInstance`, который возвращает единственный экземпляр своего класса.

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода `getInstance` должен стать единственным способом получить объект этого класса.

Псевдокод

В этом примере роль **Одиночки** отыгрывает класс подключения к базе данных.

Этот класс не имеет публичного конструктора, поэтому единственный способ получить его объект — это вызвать метод `getInstance`. Этот метод сохранит первый созданный объект и будет возвращать его при всех последующих вызовах.

```
1 class Database is
2     private field instance: Database
3
4     // Метод получения одиночки.
5     static method getInstance() is
6         if (this.instance == null) then
7             acquireThreadLock() and then
8                 // На всякий случай ещё раз проверим, не был ли
9                 // объект создан другим потоком, пока текущий
10                // ждал освобождения блокировки.
11         if (this.instance == null) then
12             this.instance = new Database()
13
14         return this.instance
15
16     private constructor Database() is
17         // Здесь может жить код инициализации подключения к
18         // серверу баз данных.
19         // ...
20
21     public method query(sql) is
22         // Все запросы к базе данных будут проходить через этот
23         // метод. Поэтому имеет смысл поместить сюда какую-то
24         // логику кеширования.
25         // ...
26
27 class Application is
28     method main() is
29         Database foo = Database.getInstance()
30         foo.query("SELECT ...")
31         // ...
32         Database bar = Database.getInstance()
33         bar.query("SELECT ...")
34         // Переменная "bar" содержит тот же объект, что и
```

Применимость

Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).

Одиночка скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.

Когда вам хочется иметь больше контроля над глобальными переменными.

В отличие от глобальных переменных, Одиночка гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.

Тем не менее, в любой момент вы можете расширить это ограничение и позволить любое количество объектов-одиночек, поменяв код в одном месте (метод `getInstance`).

Шаги реализации

1. Добавьте в класс приватное статическое поле, которое будет содержать одиночный объект.
2. Объявите статический создающий метод, который будет использоваться для получения одиночки.
3. Добавьте «ленивую инициализацию» (создание объекта при первом вызове метода) в создающий метод одиночки.

4. Сделайте конструктор класса приватным.

5. В клиентском коде замените вызовы конструктора одиночка вызовами его создающего метода.

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.

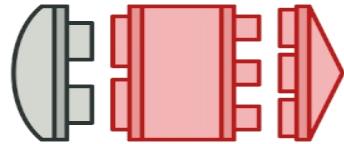
Требует постоянного создания Mock-объектов при юнит-тестировании.

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.

Структурные паттерны

Эти паттерны отвечают за построение удобных в поддержке иерархий классов.



Адаптер

Adapter

Позволяет объектам с несовместимыми интерфейсами работать вместе.



Мост

Bridge

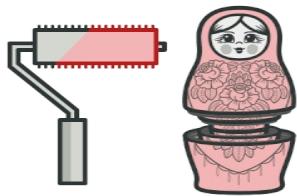
Разделяет один или несколько классов на две отдельные иерархии – абстракцию и реализацию, позволяя изменять их независимо друг от друга.



Компоновщик

Composite

Позволяет сгруппировать объекты в древовидную структуру, а затем работать с ними так, как будто это единичный объект.



Декоратор

Decorator

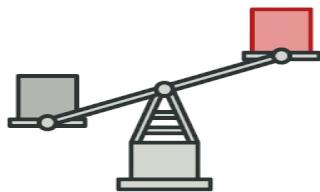
Позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».



Фасад

Facade

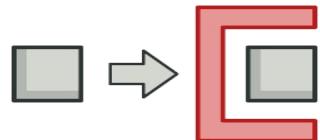
Предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.



Легковес

Flyweight

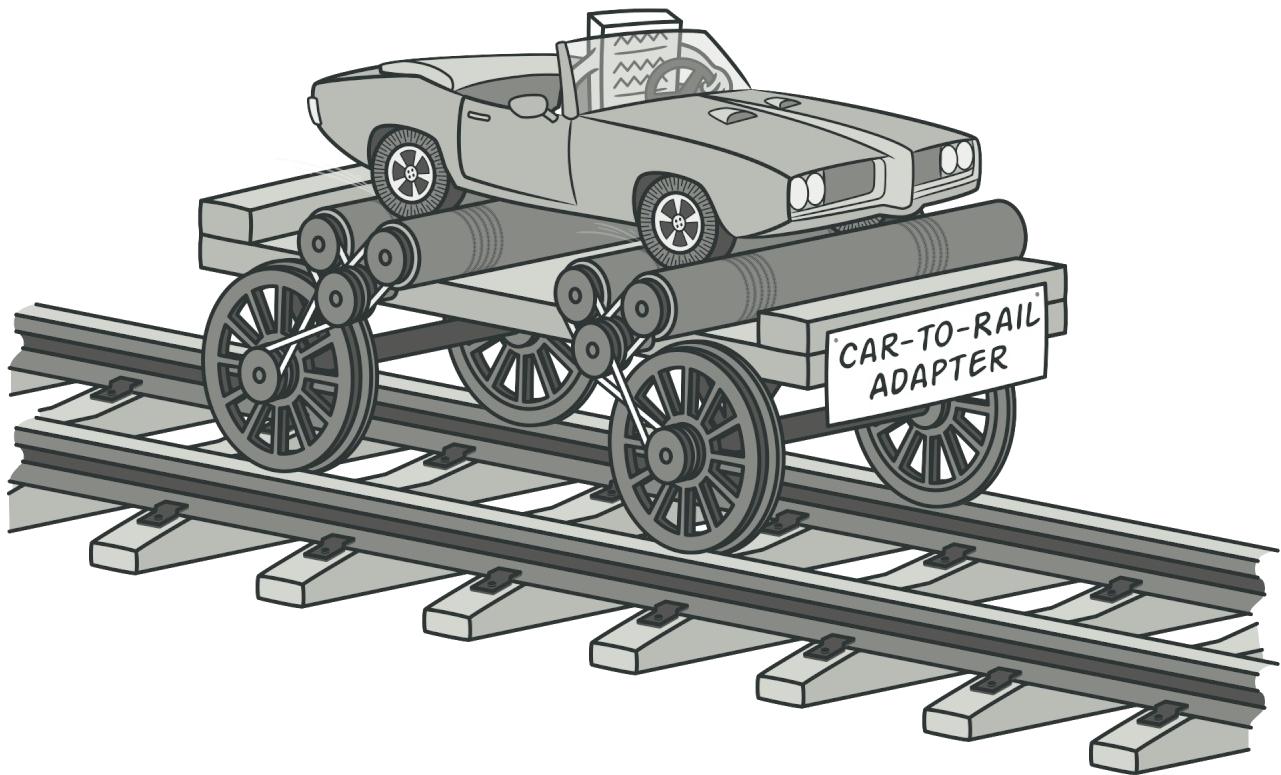
Позволяет вместить большее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.



Заместитель

Proxy

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.



АДАПТЕР

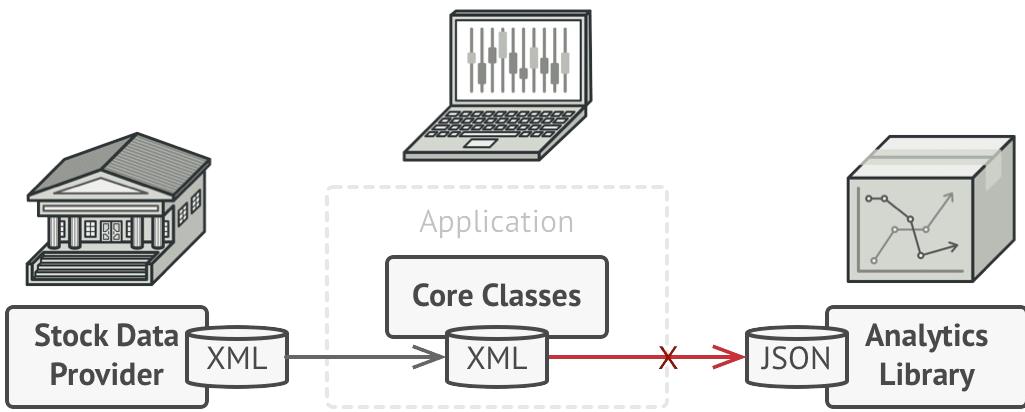
Также известен как: *Обёртка, Adapter*

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.



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

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.

Решение

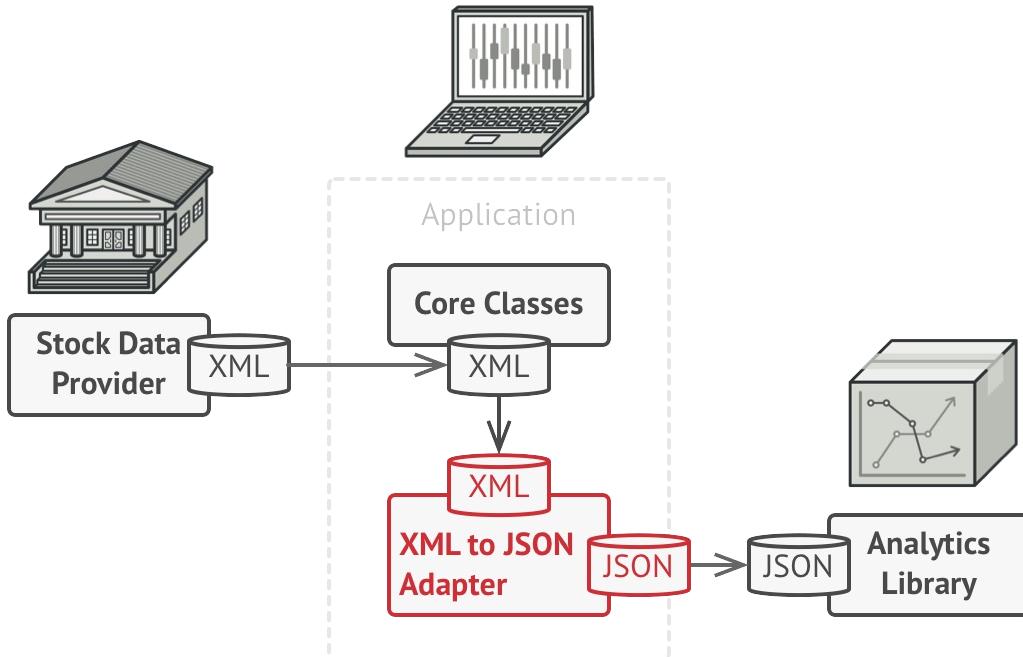
Вы можете создать *адаптер*. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер обворачивает 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:

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

Иногда возможно создать даже *двухсторонний адаптер*, который работал бы в обе стороны.



Программа может работать со сторонней библиотекой через адаптер.

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.

Аналогия из жизни



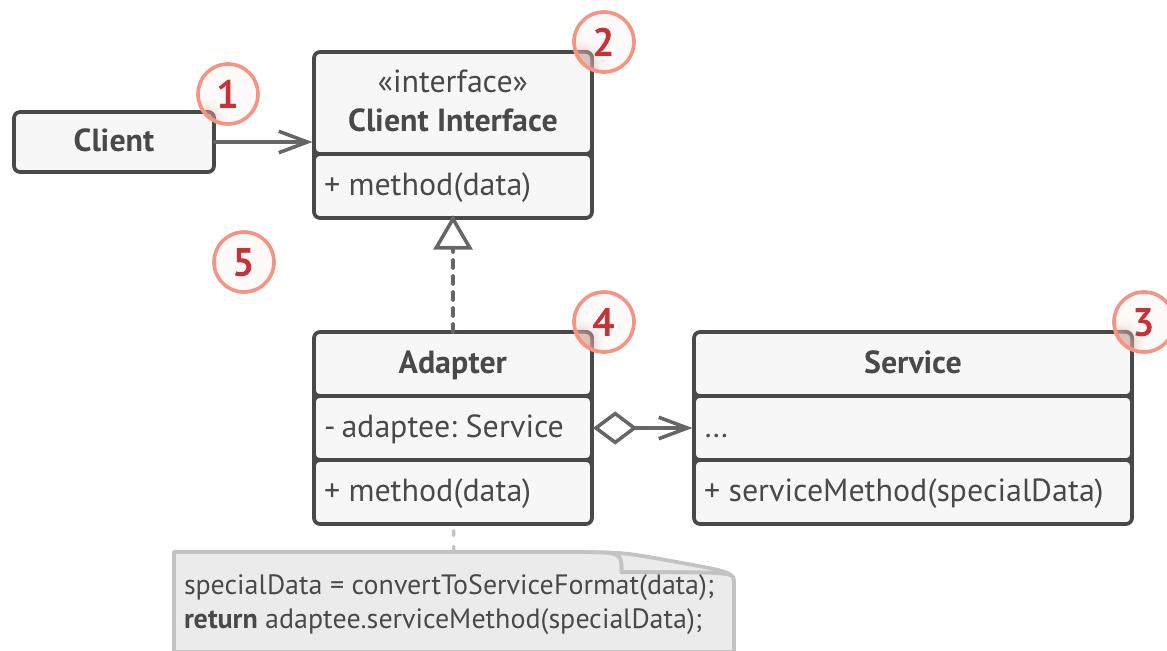
Содержимое чемоданов до и после поездки за границу.

Когда вы в первый раз летите за границу, вас может ждать сюрприз при попытке зарядить ноутбук. Стандарты розеток в разных странах отличаются. Ваша европейская зарядка будет бесполезна в США без специального адаптера, позволяющего подключиться к розетке другого типа.

Структура

Адаптер объектов

Эта реализация использует композицию: объект адаптера «обворачивает», то есть содержит ссылку на служебный объект. Такой подход работает во всех языках программирования.

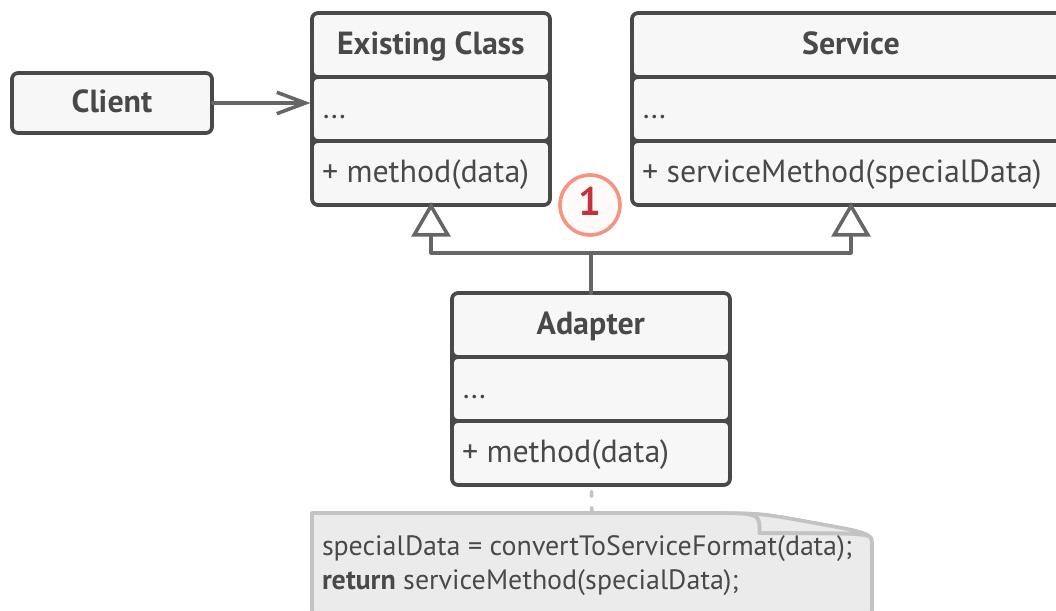


1. **Клиент** – это класс, который содержит существующую бизнес-логику программы.
2. **Клиентский интерфейс** описывает протокол, через который клиент может работать с другими классами.
3. **Сервис** – это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.

4. **Адаптер** – это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.
5. Работая с адаптером через интерфейс, клиент не привязывается к конкретному классу адаптера. Благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки.

Адаптер классов

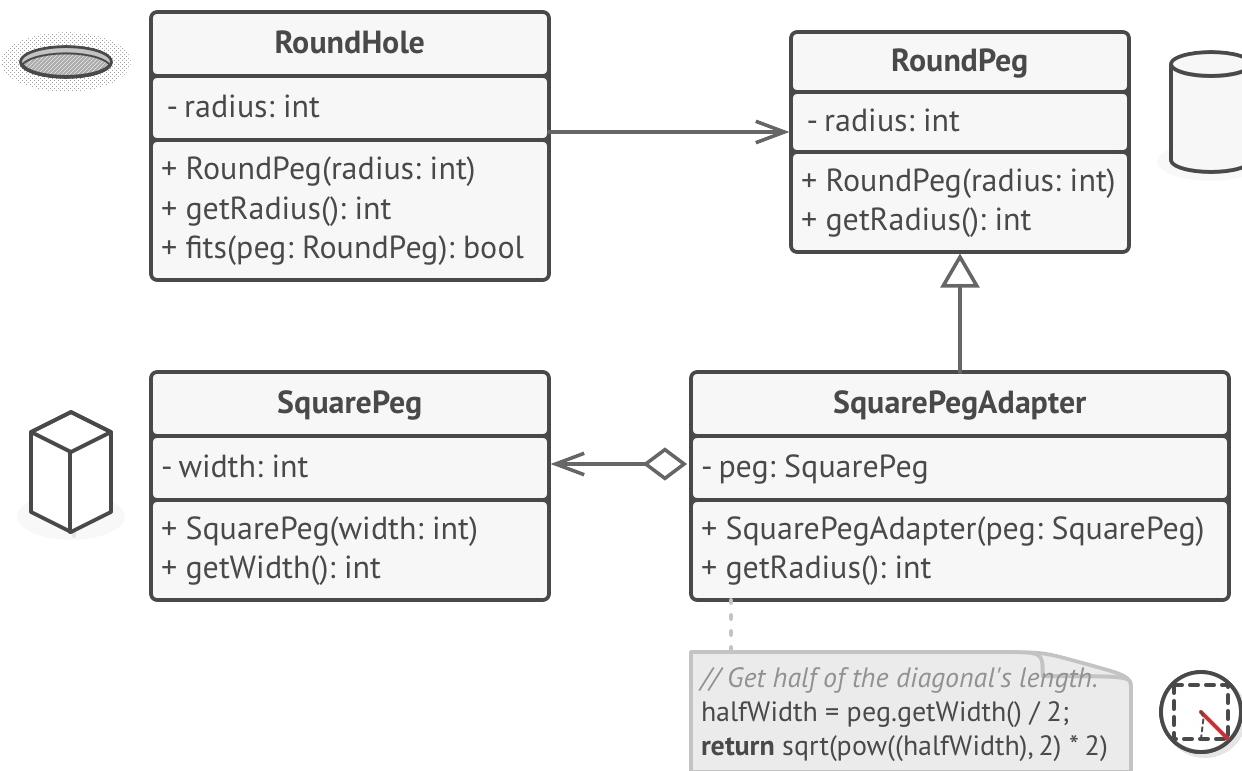
Эта реализация базируется на наследовании: адаптер наследует оба интерфейса одновременно. Такой подход возможен только в языках, поддерживающих множественное наследование, например, C++.



1. **Адаптер классов** не нуждается во вложенном объекте, так как он может одновременно наследовать и часть существующего класса, и часть сервиса.

Псевдокод

В этом шуточном примере **Адаптер** преобразует один интерфейс в другой, позволяя совместить квадратные колышки и круглые отверстия.



Пример адаптации квадратных колышков и круглых отверстий.

Адаптер вычисляет наименьший радиус окружности, в которую можно вписать квадратный колышек, и представляет его как круглый колышек с этим радиусом.

```
1 // Классы с совместимыми интерфейсами: КруглоеОтверстие и  
2 // КруглыйКолышек.  
3 class RoundHole is  
4     constructor RoundHole(radius) { ... }  
5  
6     method getRadius is  
7         // Вернуть радиус отверстия.  
8  
9     method fits(peg: RoundPeg) is  
10        return this.getRadius() >= peg.radius()  
11  
12 class RoundPeg is  
13     constructor RoundPeg(radius) { ... }  
14  
15     method getRadius() is
```

```
16 // Вернуть радиус круглого колышка.
17
18
19 // Устаревший, несовместимый класс: КвадратныйКолышек.
20 class SquarePeg is
21     constructor SquarePeg(width) { ... }
22
23     method getWidth() is
24         // Вернуть ширину квадратного колышка.
25
26
27 // Адаптер позволяет использовать квадратные колышки и круглые
28 // отверстия вместе.
29 class SquarePegAdapter extends RoundPeg is
30     private field peg: SquarePeg
31
32     constructor SquarePegAdapter(peg: SquarePeg) is
33         this.peg = peg
34
35     method getRadius() is
36         // Вычислить половину диагонали квадратного колышка по
37         // теореме Пифагора.
38         return Math.sqrt(2 * Math.pow(peg.getWidth(), 2)) / 2
39
40
41 // Где–то в клиентском коде.
42 hole = new RoundHole(5)
43 rpeg = new RoundPeg(5)
44 hole.fits(rpeg) // TRUE
45
46 small_sqpeg = new SquarePeg(2)
47 large_sqpeg = new SquarePeg(5)
48 hole.fits(small_sqpeg) // Ошибка компиляции, несовместимые типы
49
50 small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
51 large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
52 hole.fits(small_sqpeg_adapter) // TRUE
53 hole.fits(large_sqpeg_adapter) // FALSE
```

Применимость

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.

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

Вы могли бы 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 **Посетитель**.

Шаги реализации

1. Убедитесь, что у вас есть два класса с неудобными 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.

2. Опишите клиентский интерфейс, через который классы приложения смогли бы использовать класс сервиса.

3. 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.

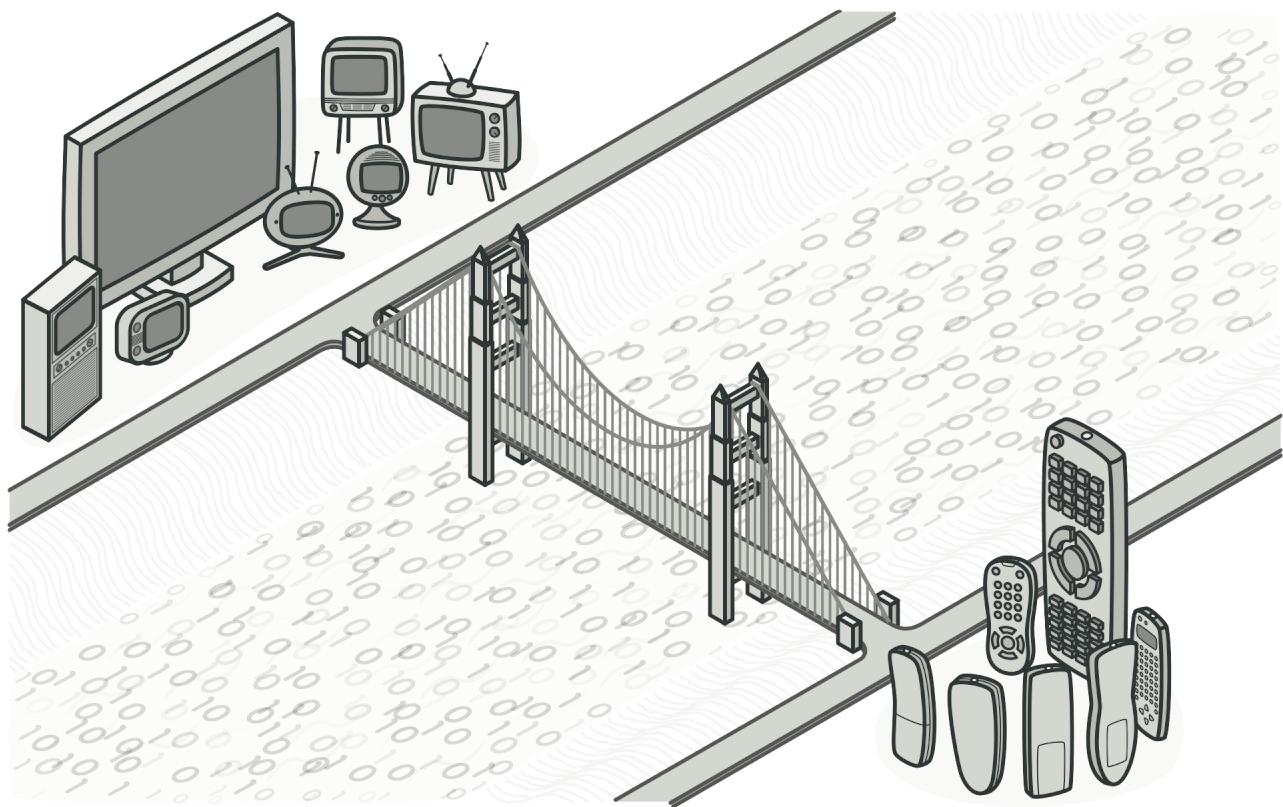
Усложняет код программы из-за введения дополнительных классов.

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 рекурсивную вложенность, чего не скажешь об **Адаптере**.
- **Адаптер** предоставляет 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.



МОСТ

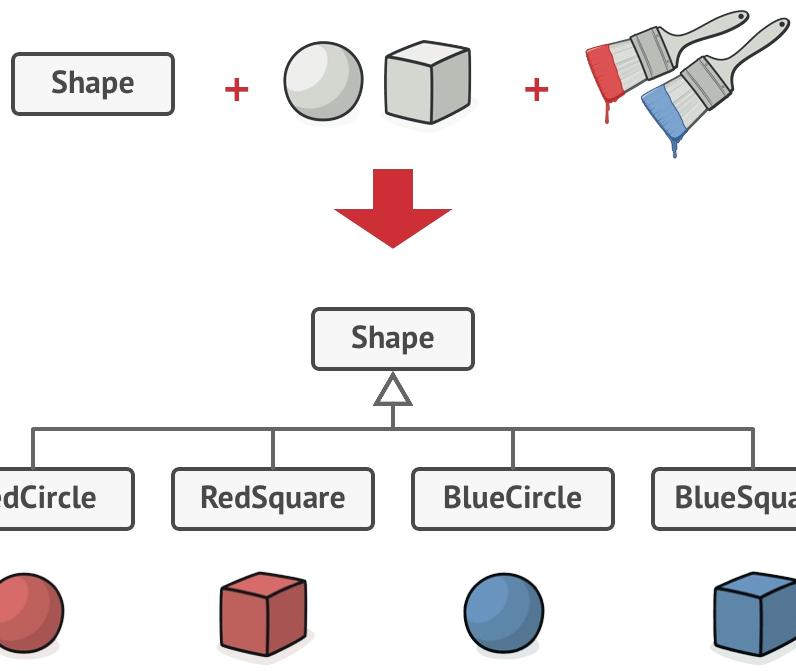
Также известен как: *Bridge*

Мост – это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии – абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Проблема

Абстракция? Реализация?! Звучит пугающе! Чтобы понять, о чём идёт речь, давайте разберём очень простой пример.

У вас есть класс геометрических **Фигур**, который имеет подклассы **Круг** И **Квадрат**. Вы хотите расширить иерархию фигур по цвету, то есть иметь **Красные** И **Синие** фигуры. Но чтобы всё это объединить, вам придётся создать 4 комбинации подклассов, вроде **СиниеКруги** И **КрасныеКвадраты**.



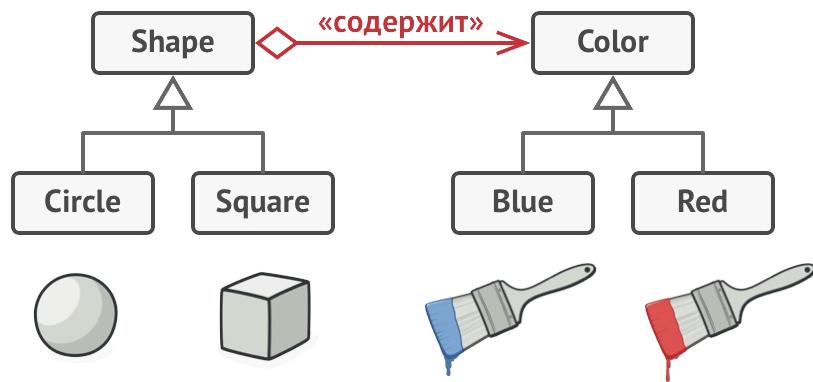
Количество подклассов растёт в геометрической прогрессии.

При добавлении новых видов фигур и цветов количество комбинаций будет расти в геометрической прогрессии. Например, чтобы ввести в программу фигуры треугольников, придётся создать сразу два новых подкласса треугольников под каждый цвет. После этого новый цвет потребует создания уже трёх классов для всех видов фигур. Чем дальше, тем хуже.

Решение

Корень проблемы заключается в том, что мы пытаемся расширить классы фигур сразу в двух независимых плоскостях – по виду и по цвету. Именно это приводит к разрастанию дерева классов.

Паттерн Мост предлагает заменить наследование делегированием. Для этого нужно выделить одну из таких «плоскостей» в отдельную иерархию и ссылаться на объект этой иерархии, вместо хранения его состояния и поведения внутри одного класса.



Размножение подклассов можно остановить, разбив классы на несколько иерархий.

Таким образом, мы можем сделать **Цвет** отдельным классом с подклассами **Красный** и **Синий**. Класс **Фигур** получит ссылку на объект **Цвета** и сможет делегировать ему работу, если потребуется. Такая связь и станет мостом между **Фигурами** и **Цветом**. При добавлении новых классов цветов не потребуется трогать классы фигур и наоборот.

Абстракция и Реализация

Эти термины были введены в книге GoF⁵ при описании Моста. На мой взгляд, они выглядят слишком академичными, делая описание паттерна сложнее, чем он есть на самом деле. Помня о примере с фигурами и цветами, давайте все же разберёмся, что имели в виду авторы паттерна.

Итак, *абстракция* (или *интерфейс*) – это образный слой управления чем-либо. Он не делает работу самостоятельно, а делегирует её слою *реализации* (иногда называемому *платформой*).

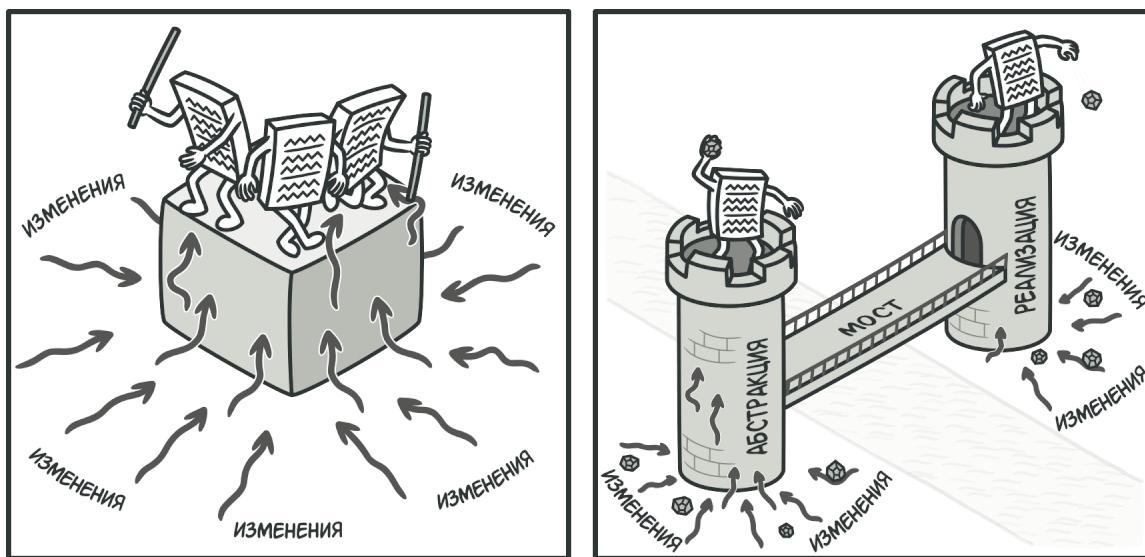
Только не путайте эти термины с *интерфейсами* или *абстрактными классами* из вашего языка программирования, это не одно и то же.

Если говорить о реальных программах, то абстракцией может выступать графический интерфейс программы (GUI), а реализацией – низкоуровневый код операционной системы (API), к которому графический интерфейс обращается по реакции на действия пользователя.

Вы можете развивать программу в двух разных направлениях:

- иметь несколько видов GUI (например, для простых пользователей и администраторов);
- поддерживать много видов API (например, работать под Windows, Linux и MacOS).

Такая программа может выглядеть как один большой клубок кода, в котором намешаны условные операторы слоёв GUI и API.

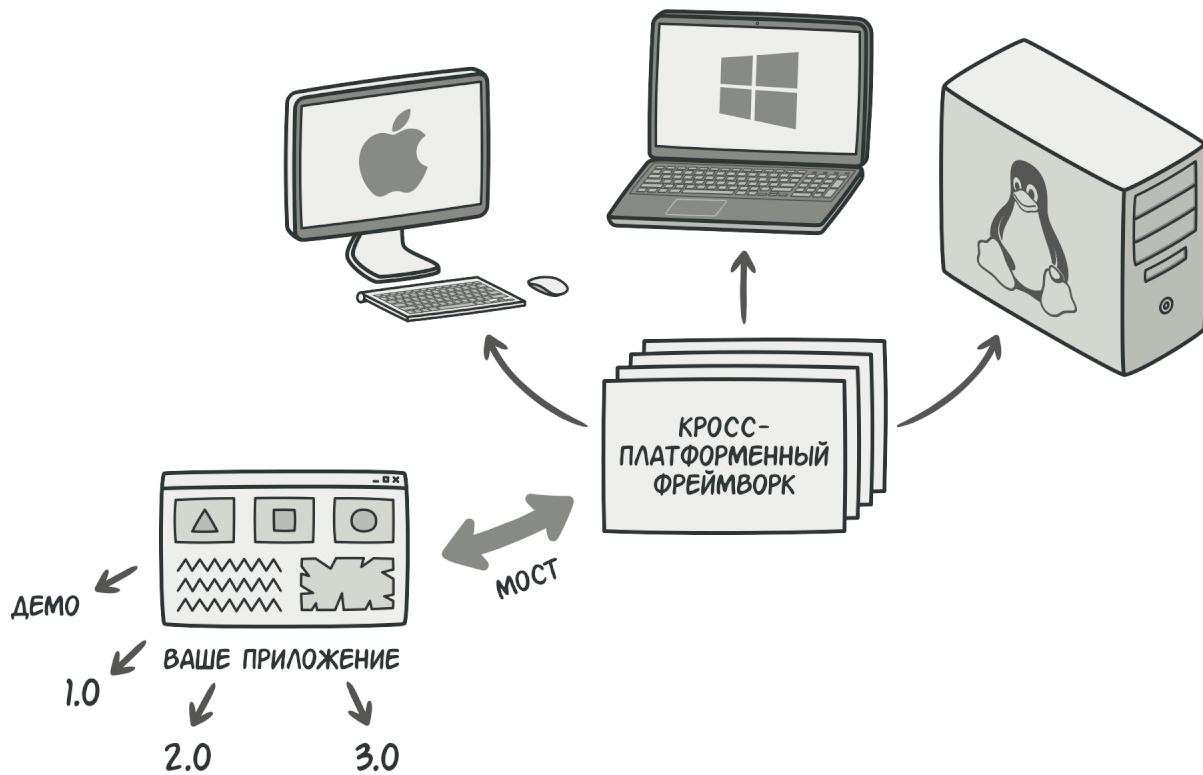


Когда изменения «осаждают» проект, вам легче отбиваться, если разделить монолитный код на части.

Вы можете попытаться структурировать этот хаос, создав для каждой вариации интерфейса-платформы свои подклассы. Но такой подход приведёт к росту классов комбинаций, и с каждой новой платформой их будет всё больше.

Мы можем решить эту проблему, применив Мост. Паттерн предлагает распутать этот код, разделив его на две части:

- Абстракцию: слой графического интерфейса приложения.
- Реализацию: слой взаимодействия с операционной системой.

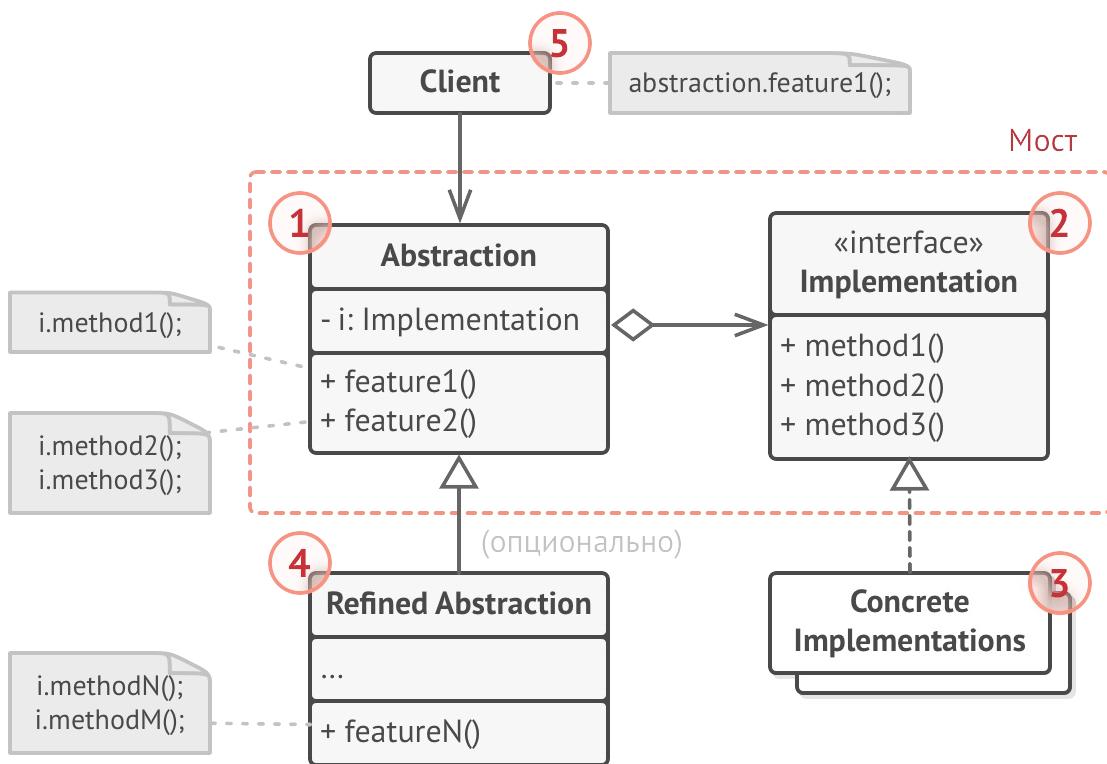


Один из вариантов кросс-платформенной архитектуры.

Абстракция будет делегировать работу одному из объектов реализаций. Причём, реализации можно будет взаимозаменять, но только при условии, что все они будут следовать общему интерфейсу.

Таким образом, вы сможете изменять графический интерфейс приложения, не трогая низкоуровневый код работы с операционной системой. И наоборот, вы сможете добавлять поддержку новых операционных систем, создавая подклассы реализации, без необходимости менять классы графического интерфейса.

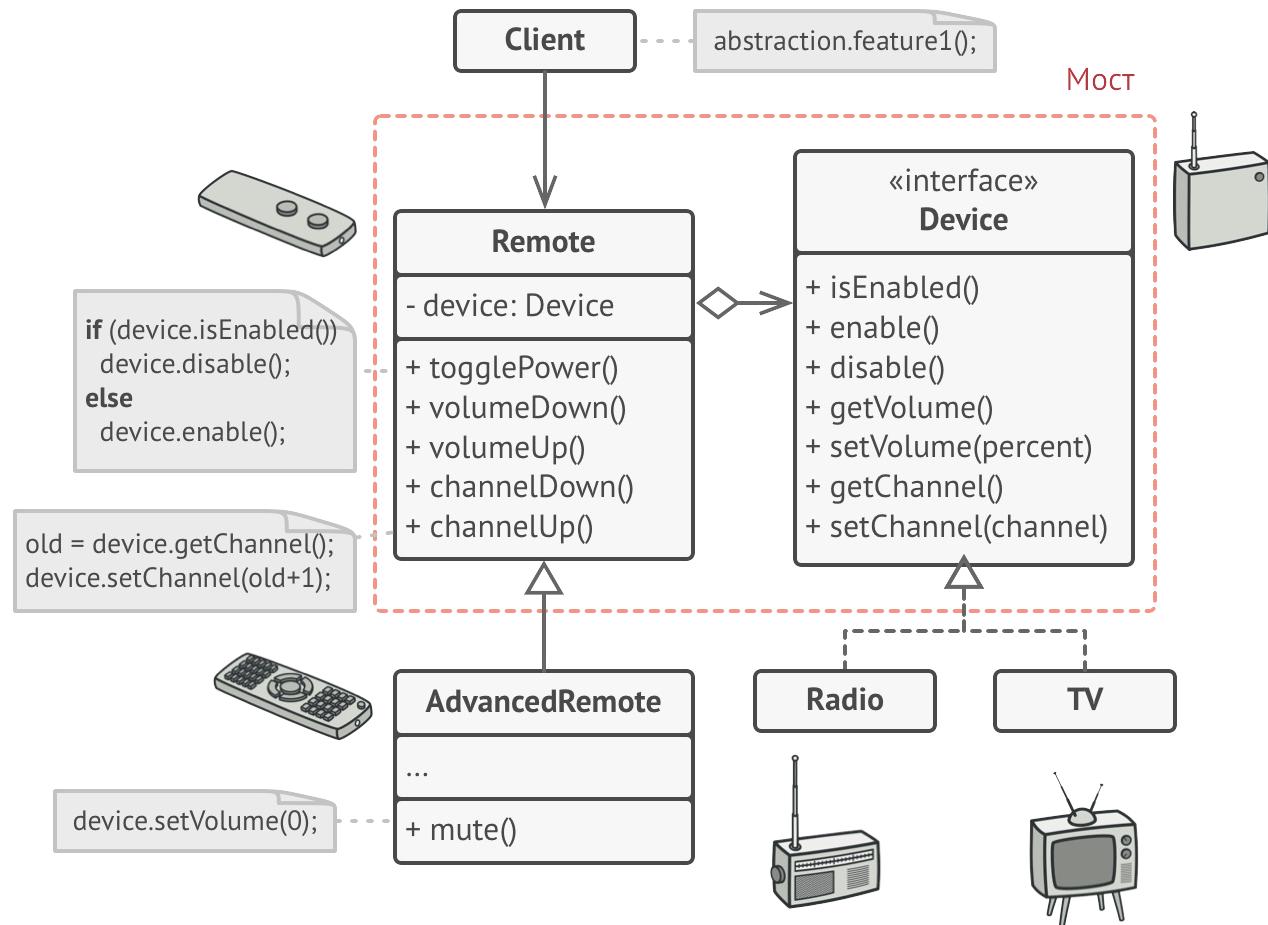
Структура



1. **Абстракция** содержит управляющую логику. Код абстракции делегирует реальную работу связанному объекту реализации.
2. **Реализация** задаёт общий интерфейс для всех реализаций. Все методы, которые здесь описаны, будут доступны из класса абстракции и его подклассов.
- Интерфейсы абстракции и реализации могут как совпадать, так и быть совершенно разными. Но обычно в реализации живут базовые операции, на которых строятся сложные операции абстракции.
3. **Конкретные Реализации** содержат платформо-зависимый код.
4. **Расширенные Абстракции** содержат различные вариации управляющей логики. Как и родитель, работает с реализациями только через общий интерфейс реализации.
5. **Клиент** работает только с объектами абстракции. Не считая начального связывания абстракции с одной из реализаций, клиентский код не имеет прямого доступа к объектам реализации.

Псевдокод

В этом примере **Мост** разделяет монолитный код приборов и пультов на две части: приборы (выступают реализацией) и пульты управления ими (выступают абстракцией).



Пример разделения двух иерархий классов – приборов и пультов управления.

Класс пульта имеет ссылку на объект прибора, которым он управляет. Пульты работают с приборами через общий интерфейс. Это даёт возможность связать пульты с различными приборами.

Сами пульты можно развивать независимо от приборов. Для этого достаточно создать новый подкласс абстракции. Вы можете создать как простой пульт с двумя кнопками, так и более сложный пульт с тач-интерфейсом.

Клиентскому коду остаётся выбрать версию абстракции и реализации, с которым он хочет работать, и связать их между собой.

```
1 // Класс пультов имеет ссылку на устройство, которым управляет.
2 // Методы этого класса делегируют работу методам связанного
3 // устройства.
4 class Remote is
5     protected field device: Device
6     constructor Remote(device: Device) is
7         this.device = device
8     method togglePower() is
9         if (device.isEnabled()) then
10             device.disable()
11         else
12             device.enable()
13     method volumeDown() is
14         device.setVolume(device.getVolume() - 10)
15     method volumeUp() is
16         device.setVolume(device.getVolume() + 10)
17     method channelDown() is
18         device.setChannel(device.getChannel() - 1)
19     method channelUp() is
20         device.setChannel(device.getChannel() + 1)
21
22
23 // Вы можете расширять класс пультов, не трогая код устройств.
24 class AdvancedRemote extends Remote is
25     method mute() is
26         device.setVolume(0)
27
28
29 // Все устройства имеют общий интерфейс. Поэтому с ними может
30 // работать любой пульт.
31 interface Device is
32     method isEnabled()
33     method enable()
34     method disable()
35     method getVolume()
36     method setVolume(percent)
37     method getChannel()
38     method setChannel(channel)
39
40
```

```
41 // Но каждое устройство имеет особую реализацию.  
42 class Tv implements Device is  
43     // ...  
44  
45 class Radio implements Device is  
46     // ...  
47  
48  
49 // Где-то в клиентском коде.  
50 tv = new Tv()  
51 remote = new Remote(tv)  
52 remote.power()  
53  
54 radio = new Radio()  
55 remote = new AdvancedRemote(radio)
```

Применимость

Когда вы хотите разделить монолитный класс, который содержит несколько различных реализаций какой-то функциональности (например, если класс может работать с разными системами баз данных).

Чем больше класс, тем тяжелее разобраться в его коде, и тем больше это затягивает разработку. Кроме того, изменения, вносимые в одну из реализаций, приводят к редактированию всего класса, что может привести к внесению случайных ошибок в код.

Мост позволяет разделить монолитный класс на несколько отдельных иерархий. После этого вы можете менять их код независимо друг от друга. Это упрощает работу над кодом и уменьшает вероятность внесения ошибок.

Когда класс нужно расширять в двух независимых плоскостях.

Мост предлагает выделить одну из таких плоскостей в отдельную иерархию классов, храня ссылку на один из её объектов в первоначальном классе.

Когда вы хотите, чтобы реализацию можно было бы изменять во время выполнения программы.

Мост позволяет заменять реализацию даже во время выполнения программы, так как конкретная реализация не «вшита» в класс абстракции.

Кстати, из-за этого пункта Мост часто путают со Стратегией. Обратите внимание, что у Моста этот пункт стоит на последнем месте по значимости, поскольку его главная задача – структурная.

Шаги реализации

1. Определите, существует ли в ваших классах два непересекающихся измерения. Это может быть функциональность/платформа, предметная-область/инфраструктура, фронт-энд/бэк-энд или интерфейс/реализация.
2. Продумайте, какие операции будут нужны клиентам, и опишите их в базовом классе *абстракции*.
3. Определите поведения, доступные на всех платформах, и выделите из них ту часть, которая нужна абстракции. На основании этого опишите общий интерфейс *реализации*.
4. Для каждой платформы создайте свой класс конкретной реализации. Все они должны следовать общему интерфейсу, который мы выделили перед этим.
5. Добавьте в класс абстракции ссылку на объект реализации. Реализуйте методы абстракции, делегируя основную работу связанному объекту реализации.
6. Если у вас есть несколько вариаций абстракции, создайте для каждой из них свой подкласс.
7. Клиент должен подать объект реализации в конструктор абстракции, чтобы связать их воедино. После этого он может свободно использовать объект абстракции, забыв о

реализации.

Преимущества и недостатки

Позволяет строить платформо-независимые программы.

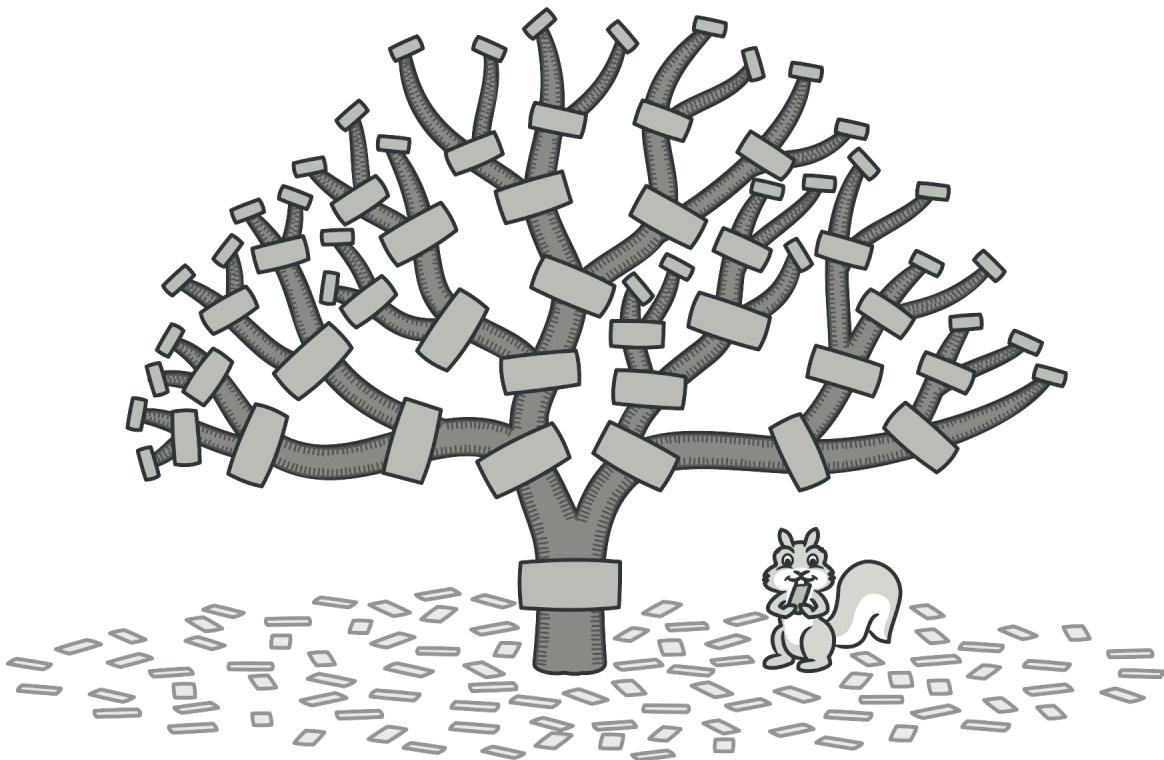
Скрывает лишние или опасные детали реализации от клиентского кода.

Реализует *принцип открытости/закрытости*.

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.
- **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.
- **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.
- Паттерн **Строитель** может быть построен в виде **Моста**: **директор** будет играть роль абстракции, а **строитель** – реализации.



КОМПОНОВЩИК

Также известен как: *Дерево, Composite*

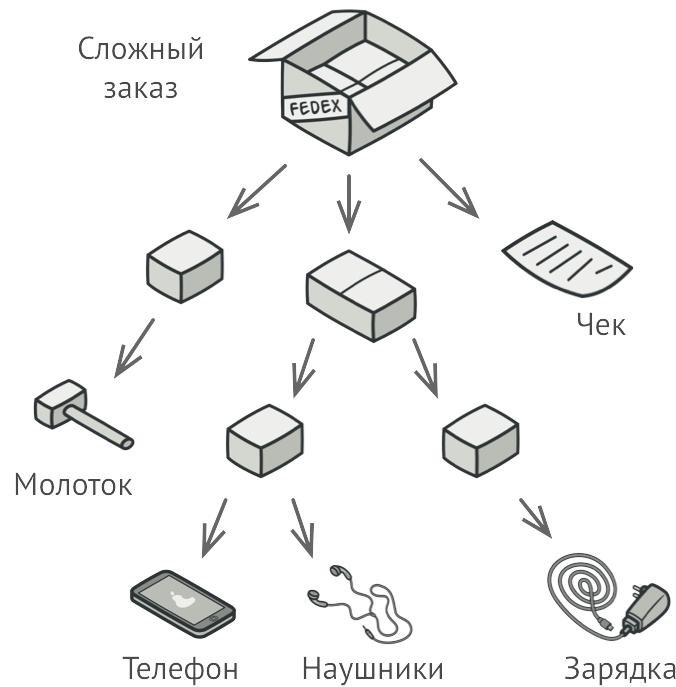
Компоновщик – это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

Проблема

Паттерн Компоновщик имеет смысл только тогда, когда основная модель вашей программы может быть структурирована в виде дерева.

Например, есть два объекта: **Продукт** и **Коробка**. **Коробка** может содержать несколько **Продуктов** и других **Коробок** поменьше. Те, в свою очередь, тоже содержат либо **Продукты**, либо **Коробки** и так далее.

Теперь предположим, ваши Продукты И Коробки могут быть частью заказов. Каждый заказ может содержать как простые Продукты без упаковки, так и составные Коробки . Ваша задача состоит в том, чтобы узнать цену всего заказа.



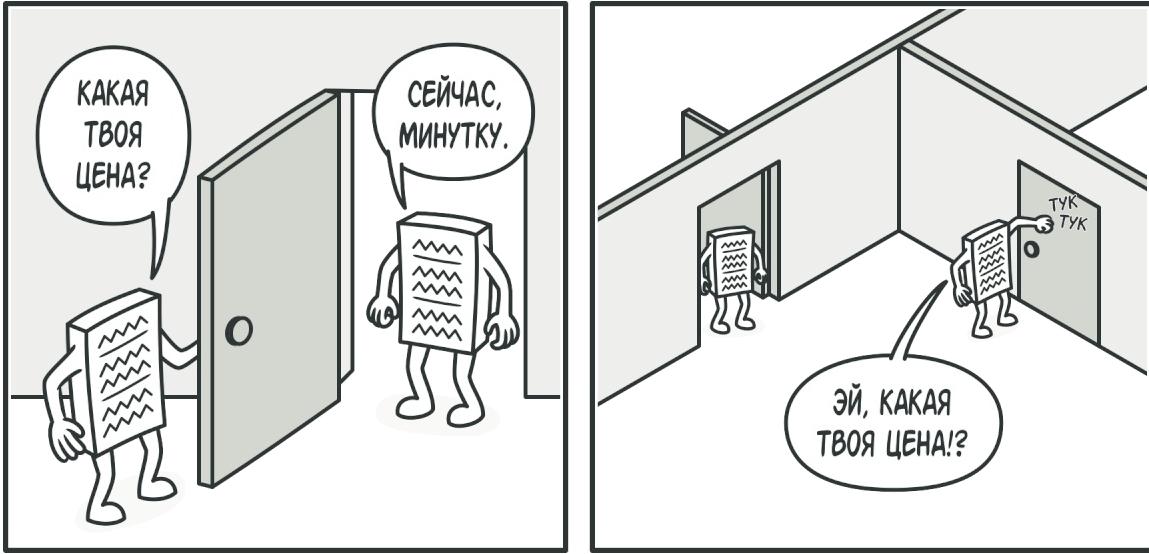
Заказ может состоять из различных продуктов, упакованных в собственные коробки.

Если решать задачу в лоб, то вам потребуется открыть все коробки заказа, перебрать все продукты и посчитать их суммарную стоимость. Но это слишком хлопотно, так как типы коробок и их содержимое могут быть вам неизвестны. Кроме того, наперёд неизвестно и количество уровней вложенности коробок, поэтому перебрать коробки простым циклом не выйдет.

Решение

Компоновщик предлагает рассматривать Продукт И Коробку через единый интерфейс с общим методом получения стоимости.

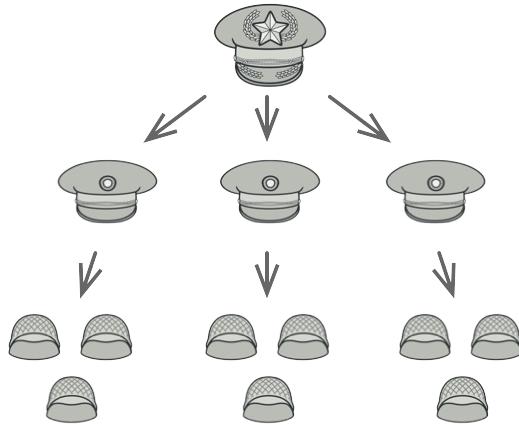
Продукт просто вернёт свою цену. Коробка спросит цену каждого предмета внутри себя и вернёт сумму результатов. Если одним из внутренних предметов окажется коробка поменьше, она тоже будет перебирать своё содержимое, и так далее, пока не будут посчитаны все составные части.



Компоновщик рекурсивно запускает действие по всем элементам дерева – от корня к листьям.

Для вас, клиента, главное, что теперь не нужно ничего знать о структуре заказов. Вы вызываете метод получения цены, он возвращает цифру, а вы не тонете в горах картона и скотча.

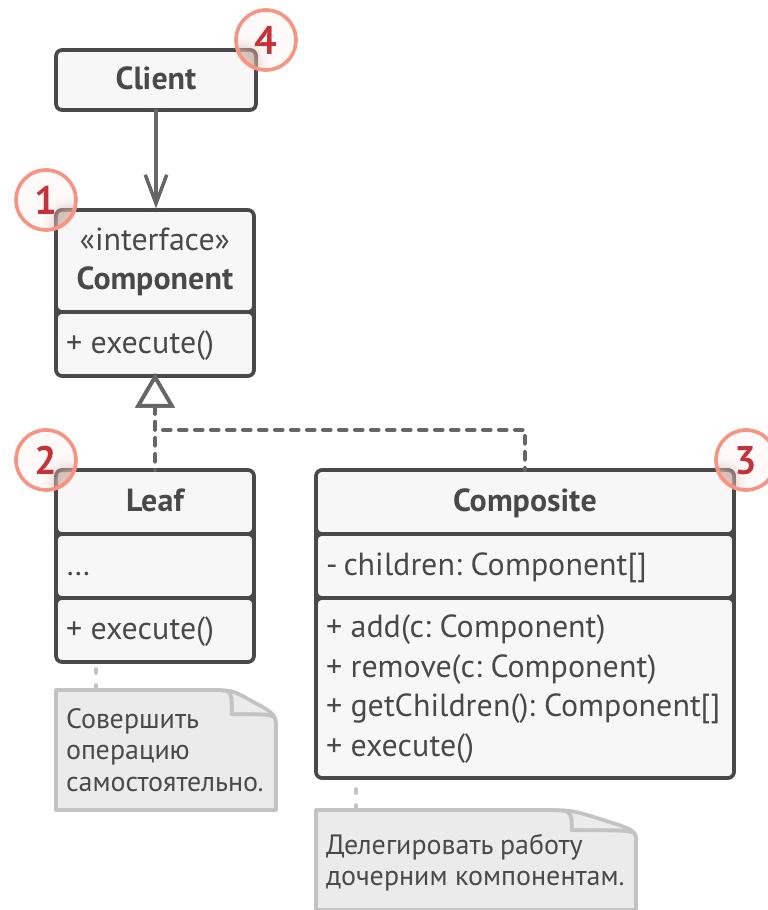
Аналогия из жизни



Пример армейской структуры.

Армии большинства государств могут быть представлены в виде перевёрнутых деревьев. На нижнем уровне у вас есть солдаты, затем взводы, затем полки, а затем целые армии. Приказы отдаются сверху и спускаются вниз по структуре командования, пока не доходят до конкретного солдата.

Структура



1. **Компонент** определяет общий интерфейс для простых и составных компонентов дерева.

2. **Лист** – это простой компонент дерева, не имеющий ответвлений.

Из-за того, что им некому больше передавать выполнение, классы листьев будут содержать большую часть полезного кода.

3. **Контейнер** (или *композит*) – это составной компонент дерева. Он содержит набор дочерних компонентов, но ничего не знает об их типах. Это могут быть как простые компоненты-листья, так и другие компоненты-контейнеры. Но это не является проблемой, если все дочерние компоненты следуют единому интерфейсу.

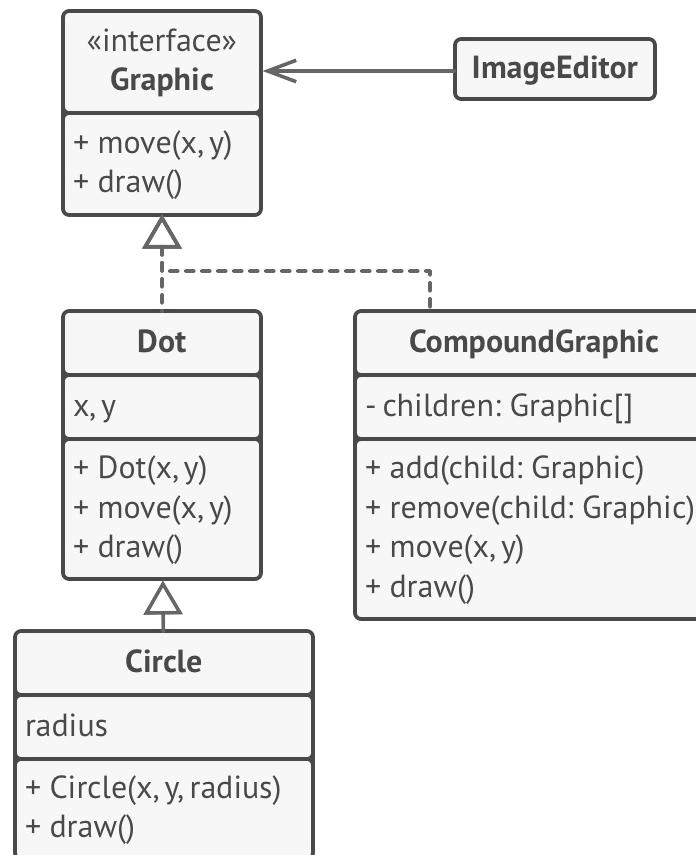
Методы контейнера переадресуют основную работу своим дочерним компонентам, хотя и могут добавлять что-то своё к результату.

4. Клиент работает с деревом через общий интерфейс компонентов.

Благодаря этому, клиенту не важно, что перед ним находится – простой или составной компонент дерева.

Псевдокод

В этом примере **Компоновщик** помогает реализовать вложенные геометрические фигуры.



Пример редактора геометрических фигур.

Класс `CompoundGraphic` может содержать любое количество подфигур, включая такие же контейнеры, как он сам. Контейнер реализует те же методы, что и простые фигуры. Но, вместо непосредственного действия, он передаёт вызовы всем вложенным компонентам, используя рекурсию. Затем он как бы «суммирует» результаты всех вложенных фигур.

Клиентский код работает со всеми фигурами через общий интерфейс фигур и не знает, что перед ним – простая фигура или составная. Это позволяет клиентскому коду работать

с деревьями объектов любой сложности, не привязываясь к конкретным классам объектов, формирующих дерево.

```
1 // Общий интерфейс компонентов.
2 interface Graphic is
3     method move(x, y)
4     method draw()
5
6 // Простой компонент.
7 class Dot implements Graphic is
8     field x, y
9
10 constructor Dot(x, y) { ... }
11
12 method move(x, y) is
13     this.x += x, this.y += y
14
15 method draw() is
16     // Нарисовать точку в координате X, Y.
17
18 // Компоненты могут расширять другие компоненты.
19 class Circle extends Dot is
20     field radius
21
22 constructor Circle(x, y, radius) { ... }
23
24 method draw() is
25     // Нарисовать окружность в координате X, Y и радиусом R.
26
27 // Контейнер содержит операции добавления/удаления дочерних
28 // компонентов. Все стандартные операции интерфейса компонентов
29 // он делегирует каждому из дочерних компонентов.
30 class CompoundGraphic implements Graphic is
31     field children: array of Graphic
32
33 method add(child: Graphic) is
34     // Добавить компонент в список дочерних.
35
36 method remove(child: Graphic) is
37     // Убрать компонент из списка дочерних.
```

```

38     method move(x, y) is
39         foreach (child in children) do
40             child.move(x, y)
41
42
43     method draw() is
44         // 1. Для каждого дочернего компонента:
45         //      – Отрисовать компонент.
46         //      – Определить координаты максимальной границы.
47         // 2. Нарисовать пунктирную границу вокруг всей области.
48
49
50 // Приложение работает единообразно как с единичными
51 // компонентами, так и с целыми группами компонентов.
52 class ImageEditor is
53     method load() is
54         all = new CompoundGraphic()
55         all.add(new Dot(1, 2))
56         all.add(new Circle(5, 3, 10))
57         // ...
58
59 // Группировка выбранных компонентов в один сложный
60 // компонент.
61 method groupSelected(components: array of Graphic) is
62     group = new CompoundGraphic()
63     group.add(components)
64     all.remove(components)
65     all.add(group)
66     // Все компоненты будут отрисованы.
67     all.draw()

```

Применимость

Когда вам нужно представить древовидную структуру объектов.

Паттерн Компоновщик предлагает хранить в составных объектах ссылки на другие простые или составные объекты. Те, в свою очередь, тоже могут хранить свои вложенные

объекты и так далее. В итоге вы можете строить сложную древовидную структуру данных, используя всего две основные разновидности объектов.

Когда клиенты должны единообразно трактовать простые и составные объекты.

Благодаря тому, что простые и составные объекты реализуют общий интерфейс, клиенту безразлично, с каким именно объектом ему предстоит работать.

Шаги реализации

1. Убедитесь, что вашу бизнес-логику можно представить как древовидную структуру. Попытайтесь разбить её на простые компоненты и контейнеры. Помните, что контейнеры могут содержать как простые компоненты, так и другие вложенные контейнеры.
2. Создайте общий интерфейс компонентов, который объединит операции контейнеров и простых компонентов дерева. Интерфейс будет удачным, если вы сможете использовать его, чтобы взаимозаменять простые и составные компоненты без потери смысла.
3. Создайте класс компонентов-листьев, не имеющих дальнейших ответвлений. Имейте в виду, что программа может содержать несколько таких классов.
4. Создайте класс компонентов-контейнеров и добавьте в него массив для хранения ссылок на вложенные компоненты. Этот массив должен быть способен содержать как простые, так и составные компоненты, поэтому убедитесь, что он объявлен с типом интерфейса компонентов.

Реализуйте в контейнере методы интерфейса компонентов, помня о том, что контейнеры должны делегировать основную работу своим дочерним компонентам.

5. Добавьте операции добавления и удаления дочерних компонентов в класс контейнеров.

Имейте в виду, что методы добавления/удаления дочерних компонентов можно поместить и в интерфейс компонентов. Да, это нарушит *принцип разделения интерфейса*, так как реализации методов будут пустыми в компонентах-листьях. Но зато все компоненты дерева станут действительно одинаковыми для клиента.

Преимущества и недостатки

Упрощает архитектуру клиента при работе со сложным деревом компонентов.

Облегчает добавление новых видов компонентов.

Создаёт слишком общий дизайн классов.

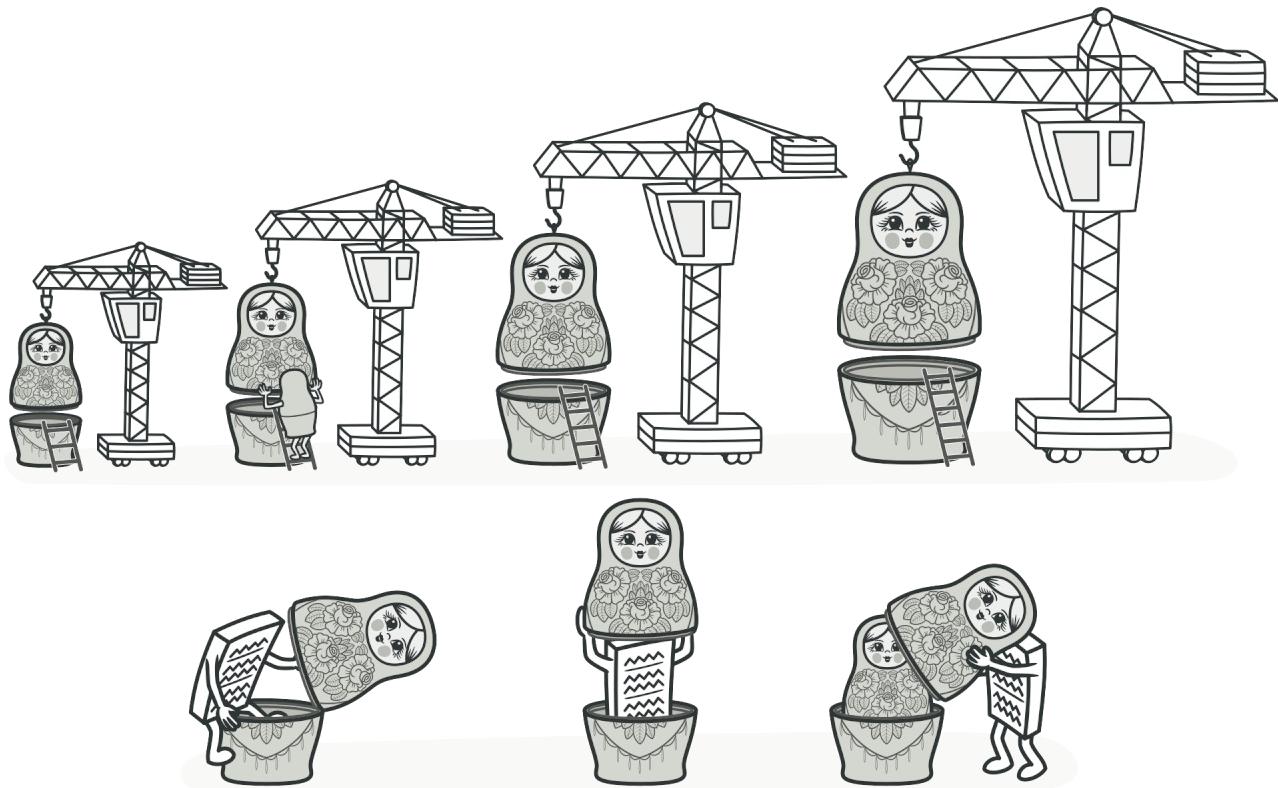
Отношения с другими паттернами

- **Строитель** позволяет пошагово сооружать дерево **Компоновщика**.
- **Цепочку обязанностей** часто используют вместе с **Компоновщиком**. В этом случае запрос передаётся от дочерних компонентов к их родителям.
- Вы можете обходить дерево **Компоновщика**, используя **Итератор**.
- Вы можете выполнить какое-то действие над всем деревом **Компоновщика** при помощи **Посетителя**.
- **Компоновщик** часто совмещают с **Легковесом**, чтобы реализовать общие ветви дерева и сэкономить при этом память.
- **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.

Декоратор оборачивает только один объект, а узел *Компоновщика* может иметь много детей. *Декоратор* добавляет вложенному объекту новую функциональность, а *Компоновщик* не добавляет ничего нового, но «суммирует» результаты всех своих детей.

Но они могут и сотрудничать: *Компоновщик* может использовать *Декоратор*, чтобы переопределить функции отдельных частей дерева компонентов.

- 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.



ДЕКОРАТОР

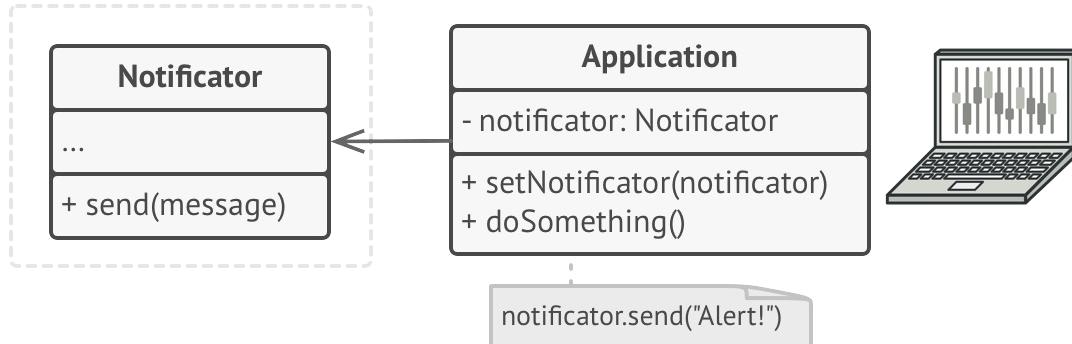
Также известен как: *Обёртка, Decorator*

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

Проблема

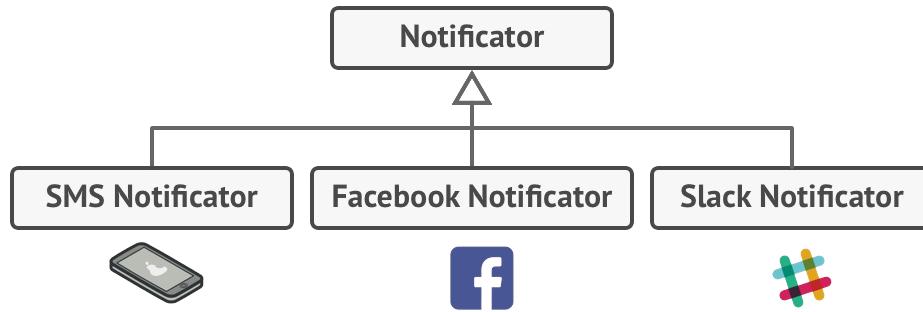
Вы работаете над библиотекой оповещений, которую можно подключать к разнообразным программам, чтобы получать уведомления о важных событиях.

Основой 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.



Сторонние программы используют главный класс оповещений.

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.

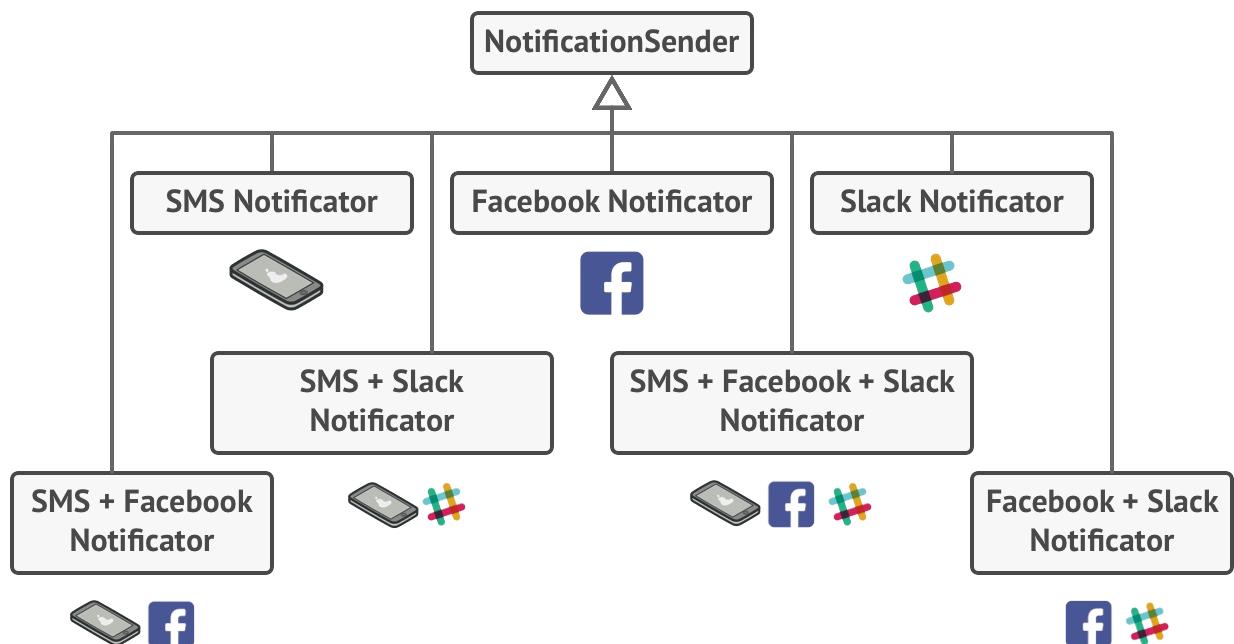


Каждый тип оповещения живёт в собственном подклассе.

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



Комбинаторный взрыв подклассов при совмещении типов оповещений.

Итак, нужен какой-то 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 вам приходится создавать множество подклассов-комбинаций для получения совмешённого поведения.

Одним из способов обойти эти проблемы является механизм *композиции*. Это когда один объект *содержит* ссылку на другой и делегирует ему работу, вместо того чтобы самому *наследовать* его поведение. Как раз на этом принципе построен паттерн Декоратор.

Наследование

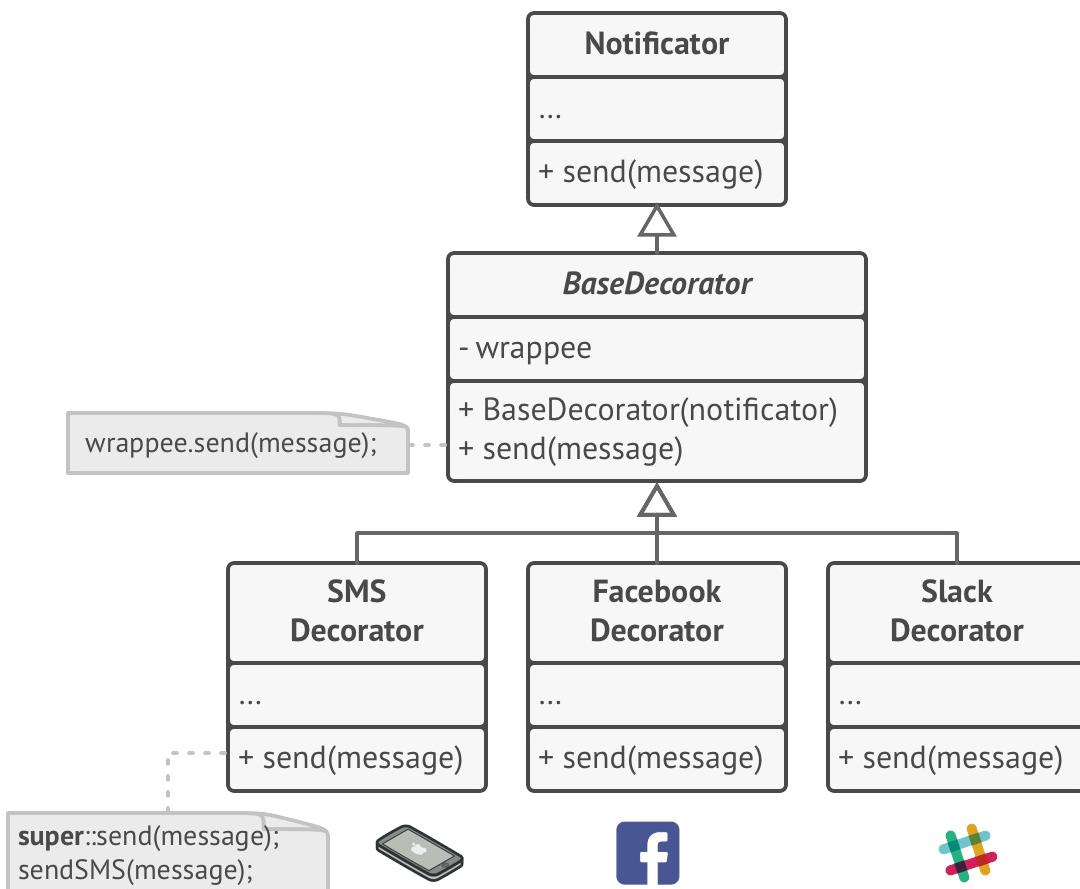
Композиция



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.

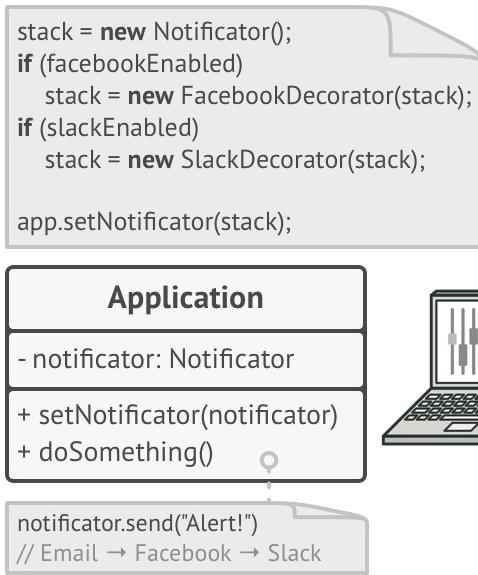
Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать – чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно – результат будет иметь объединённое поведение всех обёрток сразу.

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



Расширенные способы оповещения становятся декораторами.

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



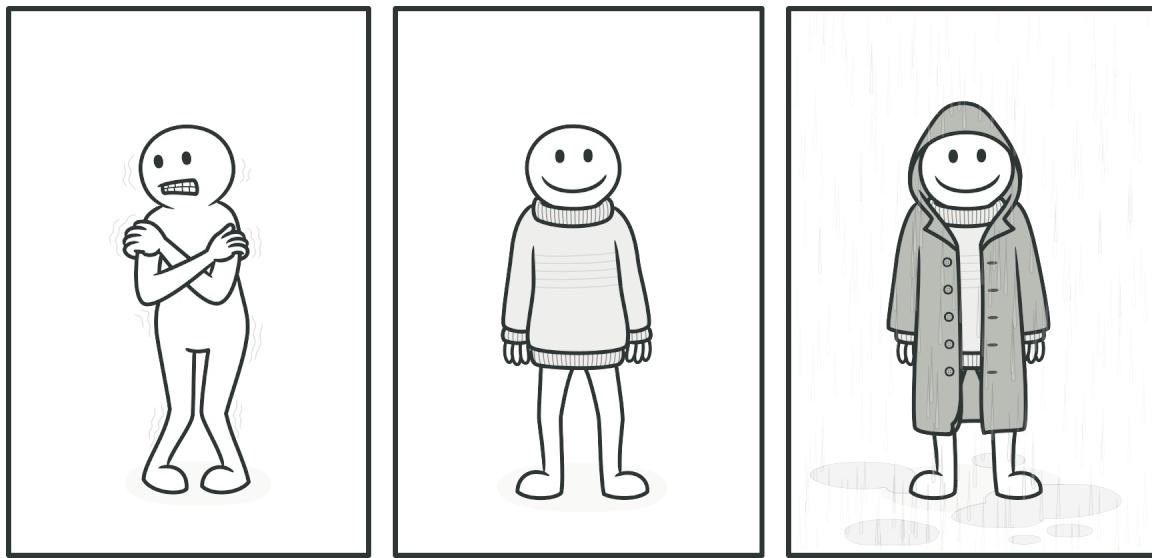
Программа может составлять составные объекты из декораторов.

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 «дообернуть» объект любыми другими обёртками, когда ему захочется.

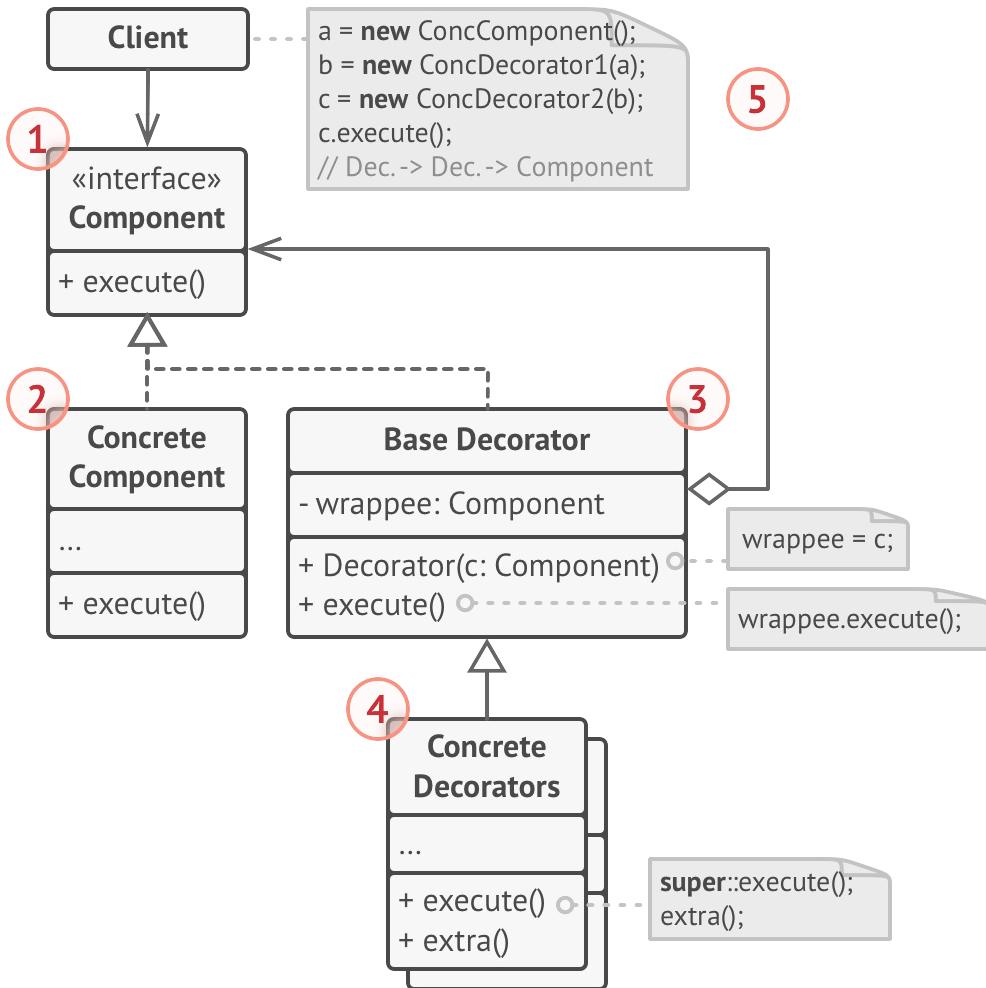
Аналогия из жизни



Одежду можно надевать слоями, получая комбинированный эффект.

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.

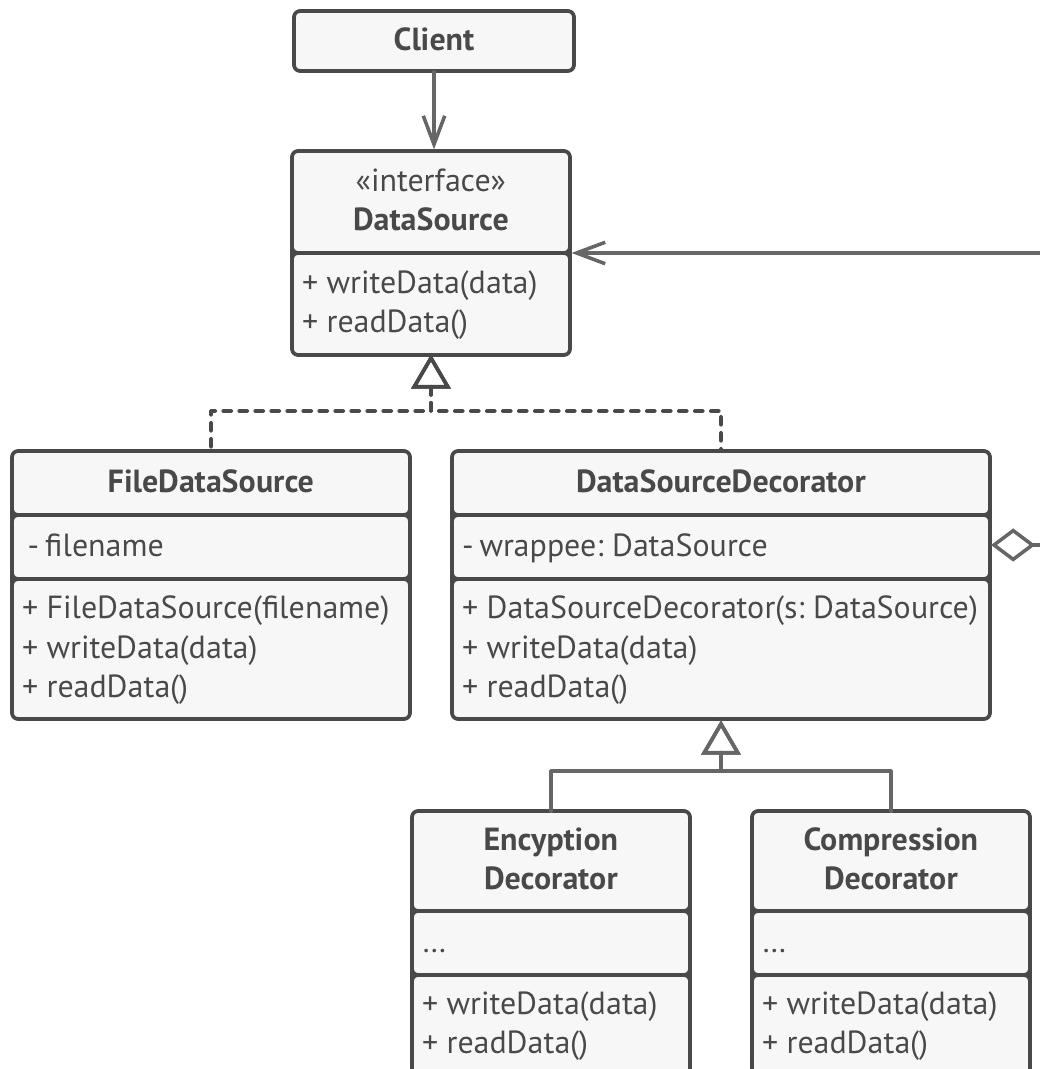
Структура



- Компонент** задаёт общий интерфейс обёрток и обываемых объектов.
- Конкретный Компонент** определяет класс обываемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы.
- Базовый Декоратор** хранит ссылку на вложенный объект-компонент. Им может быть как конкретный компонент, так и один из конкретных декораторов. Базовый декоратор делегирует все свои операции вложенному объекту. Дополнительное поведение будет жить в конкретных декораторах.
- Конкретные Декораторы** – это различные вариации декораторов, которые содержат добавочное поведение. Оно выполняется до или после вызова аналогичного поведения обёрнутого объекта.
- Клиент** может обрабатывать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

Псевдокод

В этом примере **Декоратор** защищает финансовые данные дополнительными уровнями безопасности прозрачно для кода, который их использует.



Пример шифрования и компрессии данных с помощью обёрток.

Приложение оборачивает класс данных в шифрующую и сжимающую обёртки, которые при чтении выдают оригинальные данные, а при записи – зашифрованные и сжатые.

Декораторы, как и сам класс данных, имеют общий интерфейс. Поэтому клиентскому коду не важно, с чем работать – с «чистым» объектом данных или с «обёрнутым».

```
1 // Общий интерфейс компонентов.
2 interface Datasource is
3     method writeData(data)
```

```
4 method readData():data
5
6 // Один из конкретных компонентов реализует базовую
7 // функциональность.
8 class FileDataSource implements DataSource is
9     constructor FileDataSource(filename) { ... }
10
11    method writeData(data) is
12        // Записать данные в файл.
13
14    method readData():data is
15        // Прочитать данные из файла.
16
17 // Родитель всех декораторов содержит код обёртывания.
18 class DataSourceDecorator implements DataSource is
19     protected field wrappee: DataSource
20
21     constructor DataSourceDecorator(source: DataSource) is
22         wrappee = source
23
24     method writeData(data) is
25         wrappee.writeData(data)
26
27     method readData():data is
28         return wrappee.readData()
29
30 // Конкретные декораторы добавляют что-то своё к базовому
31 // поведению обёрнутого компонента.
32 class EncryptionDecorator extends DataSourceDecorator is
33     method writeData(data) is
34         // 1. Зашифровать поданные данные.
35         // 2. Передать зашифрованные данные в метод writeData
36         // обёрнутого объекта (wrappee).
37
38     method readData():data is
39         // 1. Получить данные из метода readData обёрнутого
40         // объекта (wrappee).
41         // 2. Расшифровать их, если они зашифрованы.
42         // 3. Вернуть результат.
43
44 // Декорировать можно не только базовые компоненты, но и уже
```

```
45 // обёрнутые объекты.
46 class CompressionDecorator extends DataSourceDecorator is
47     method writeData(data) is
48         // 1. Запаковать поданные данные.
49         // 2. Передать запакованные данные в метод writeData
50         // обёрнутого объекта (wrappee).
51
52     method readData():data is
53         // 1. Получить данные из метода readData обёрнутого
54         // объекта (wrappee).
55         // 2. Распаковать их, если они запакованы.
56         // 3. Вернуть результат.
57
58
59 // Вариант 1. Простой пример сборки и использования декораторов.
60 class Application is
61     method dumbUsageExample() is
62         source = new FileDataSource("somefile.dat")
63         source.writeData(salaryRecords)
64         // В файл были записаны чистые данные.
65
66         source = new CompressionDecorator(source)
67         source.writeData(salaryRecords)
68         // В файл были записаны сжатые данные.
69
70         source = new EncryptionDecorator(source)
71         // Сейчас в source находится связка из трёх объектов:
72         // Encryption > Compression > FileDataSource
73
74         source.writeData(salaryRecords)
75         // В файл были записаны сжатые и зашифрованные данные.
76
77
78 // Вариант 2. Клиентский код, использующий внешний источник
79 // данных. Класс SalaryManager ничего не знает о том, как именно
80 // будут считаны и записаны данные. Он получает уже готовый
81 // источник данных.
82 class SalaryManager is
83     field source: DataSource
84
85     constructor SalaryManager(source: DataSource) { ... }
```

```
86
87     method load() is
88         return source.readData()
89
90     method save() is
91         source.writeData(salaryRecords)
92     // ...Остальные полезные методы...
93
94
95 // Приложение может по-разному собирать декорируемые объекты, в
96 // зависимости от условий использования.
97 class ApplicationConfigurator is
98     method configurationExample() is
99         source = new FileDataSource("salary.dat")
100        if (enabledEncryption)
101            source = new EncryptionDecorator(source)
102        if (enabledCompression)
103            source = new CompressionDecorator(source)
104
105        logger = new SalaryManager(source)
106        salary = logger.load()
107    // ...
```

Применимость

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.

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

1. Убедитесь, что в вашей задаче есть один основной компонент и несколько опциональных дополнений или надстроек над ним.
2. Создайте интерфейс компонента, который описывал бы общие методы как для основного компонента, так и для его дополнений.
3. Создайте класс конкретного компонента и поместите в него основную бизнес-логику.
4. Создайте базовый класс декораторов. Он должен иметь поле для хранения ссылки на вложенный объект-компонент. Все методы базового декоратора должны делегировать действие вложенному объекту.
5. И конкретный компонент, и базовый декоратор должны следовать одному и тому же интерфейсу компонента.
6. 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.
7. 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.

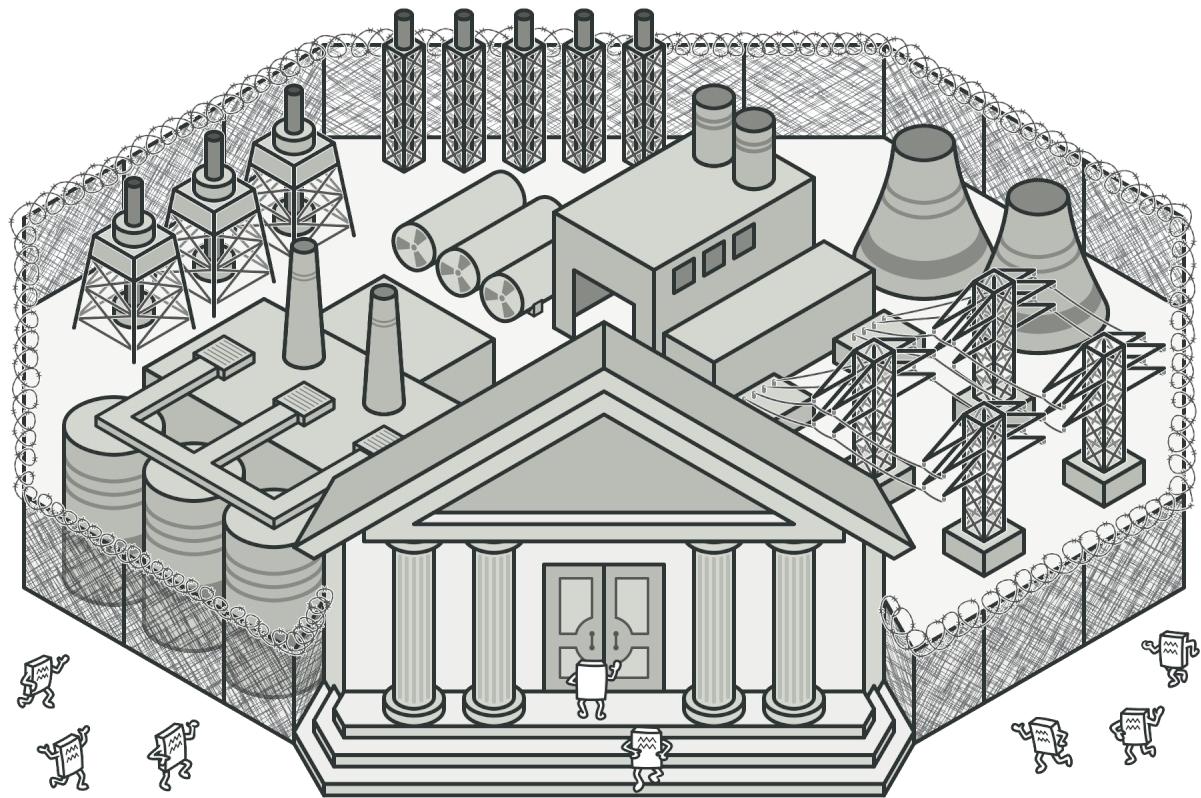
Обработчики в *Цепочке обязанностей* могут выполнять произвольные действия, независимые друг от друга, а также в любой момент прерывать дальнейшую передачу по цепочке. С другой стороны *Декораторы* расширяют какое-то определённое действие, не ломая интерфейс базовой операции и не прерывая выполнение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

- **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.

Декоратор обворачивает только один объект, а узел *Компоновщика* может иметь много детей. *Декоратор* добавляет вложенному объекту новую функциональность, а *Компоновщик* не добавляет ничего нового, но «суммирует» результаты всех своих детей.

Но они могут и сотрудничать: *Компоновщик* может использовать *Декоратор*, чтобы переопределить функции отдельных частей дерева компонентов.

- 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 композиции и делегируют работу другим объектам. Паттерны отличаются тем, что **Заместитель** сам управляет жизнью сервисного объекта, а обёртывание **Декораторов** контролируется клиентом.



ФАСАД

Также известен как: *Facade*

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

Проблема

Вашему коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка. Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее.

В результате бизнес-логика ваших классов тесно переплетается с деталями реализации сторонних классов. Такой код довольно сложно понимать и поддерживать.

Решение

Фасад – это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.

Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.

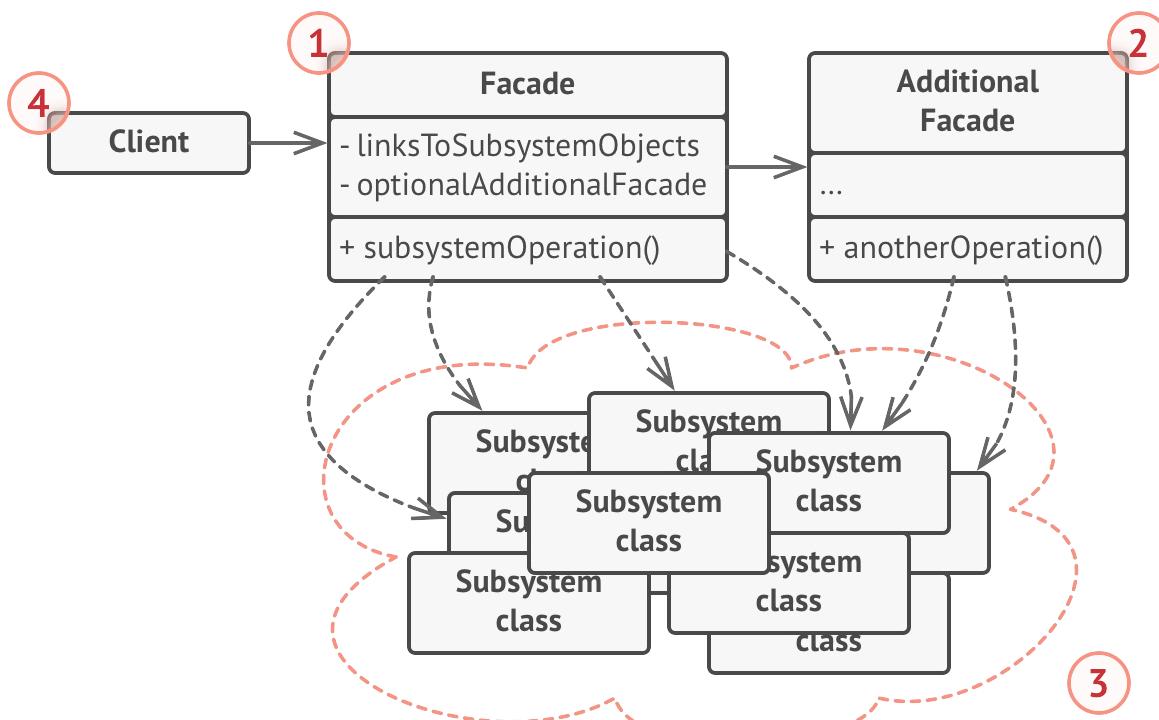
К примеру, программа, заливающая видео котиков в социальные сети, может использовать профессиональную библиотеку сжатия видео. Но все, что нужно клиентскому коду этой программы – простой метод `encode(filename, format)`. Создав класс с таким методом, вы реализуете свой первый фасад.

Аналогия из жизни



Когда вы звоните в магазин и делаете заказ по телефону, сотрудник службы поддержки является вашим фасадом ко всем службам и отделам магазина. Он предоставляет вам упрощённый интерфейс к системе создания заказа, платёжной системе и отделу доставки.

Структура



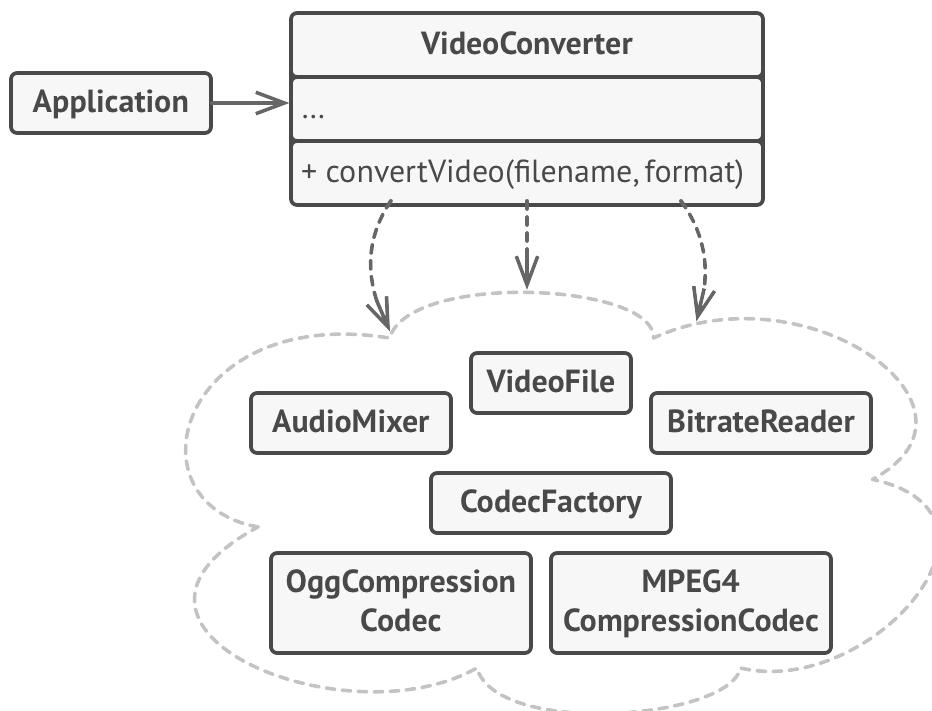
- Фасад** предоставляет быстрый доступ к определённой функциональности подсистемы. Он «знает», каким классам нужно переадресовать запрос, и какие данные для этого нужны.
- Дополнительный фасад** можно ввести, чтобы не «захламлять» единственный фасад разнородной функциональностью. Он может использоваться как клиентом, так и другими фасадами.
- Сложная подсистема** состоит из множества разнообразных классов. Для того, чтобы заставить их что-то делать, нужно знать подробности устройства подсистемы, порядок инициализации объектов и так далее.

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

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

Псевдокод

В этом примере **Фасад** упрощает работу со сложным фреймворком видеоконвертации.



Пример изоляции множества зависимостей в одном фасаде.

Вместо непосредственной работы с дюжиной классов, фасад предоставляет коду приложения единственный метод для конвертации видео, который сам заботится о том, чтобы правильно сконфигурировать нужные объекты фреймворка и получить требуемый результат.

```

1 // Классы сложного стороннего фреймворка конвертации видео. Мы
2 // не контролируем этот код, поэтому не можем его упростить.
3
4 class VideoFile
5 // ...
6
7 class OggCompressionCodec
8 // ...
9
10 class MPEG4CompressionCodec
11 // ...
12
13 class CodecFactory
14 // ...
15
16 class BitrateReader
17 // ...

```

```

18
19 class AudioMixer
20 // ...
21
22
23 // Вместо этого мы создаём Фасад – простой интерфейс для работы
24 // со сложным фреймворком. Фасад не имеет всей функциональности
25 // фреймворка, но зато скрывает его сложность от клиентов.
26 class VideoConverter is
27     method convert(filename, format):File is
28         file = new VideoFile(filename)
29         sourceCodec = new CodecFactory.extract(file)
30         if (format == "mp4")
31             distinationCodec = new MPEG4CompressionCodec()
32         else
33             distinationCodec = new OggCompressionCodec()
34         buffer = BitrateReader.read(filename, sourceCodec)
35         result = BitrateReader.convert(buffer, distinationCodec)
36         result = (new AudioMixer()).fix(result)
37         return new File(result)
38
39 // Приложение не зависит от сложного фреймворка конвертации
40 // видео. Кстати, если вы вдруг решите сменить фреймворк, вам
41 // нужно будет переписать только класс фасада.
42 class Application is
43     method main() is
44         convertor = new VideoConverter()
45         mp4 = convertor.convert("youtubevideo.ogg", "mp4")
46         mp4.save()

```

Применимость

Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме.

Часто подсистемы усложняются по мере развития программы. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать, настраивая её каждый раз под конкретные

нужды, но вместе с тем, применять подсистему без настройки становится труднее. Фасад предлагает определённый вид системы по умолчанию, устраивающий большинство клиентов.

Когда вы хотите разложить подсистему на отдельные слои.

Используйте фасады для определения точек входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Например, возьмём ту же сложную систему видеоконвертации. Вы хотите разбить её на слои работы с аудио и видео. Для каждой из этих частей можно попытаться создать фасад и заставить классы аудио и видео обработки общаться друг с другом через эти фасады, а не напрямую.

Шаги реализации

1. Определите, можно ли создать более простой интерфейс, чем тот, который предоставляет сложная подсистема. Вы на правильном пути, если этот интерфейс избавит клиента от необходимости знать о подробностях подсистемы.
2. Создайте класс фасада, реализующий этот интерфейс. Он должен переадресовывать вызовы клиента нужным объектам подсистемы. Фасад должен будет позаботиться о том, чтобы правильно инициализировать объекты подсистемы.
3. Вы получите максимум пользы, если клиент будет работать только с фасадом. В этом случае изменения в подсистеме будут затрагивать только код фасада, а клиентский код останется рабочим.
4. Если ответственность фасада начинает размываться, подумайте о введении дополнительных фасадов.

Преимущества и недостатки

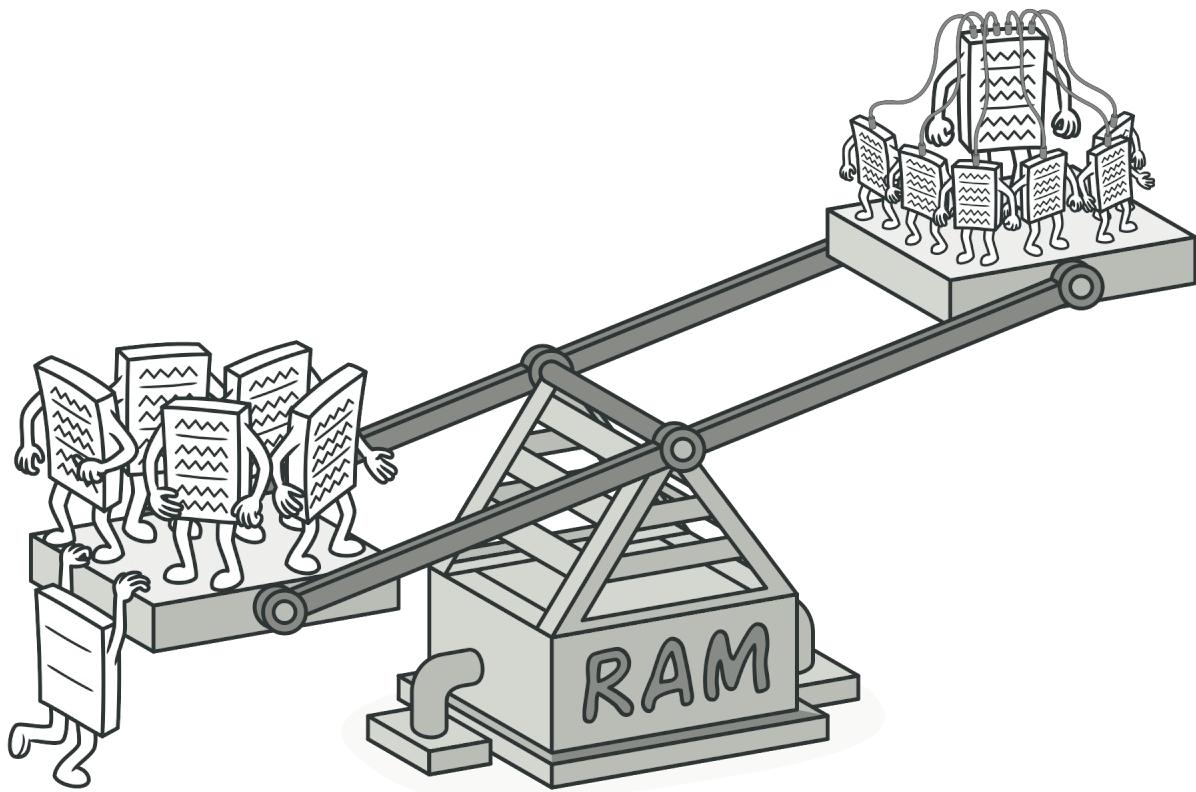
Изолирует клиентов от компонентов сложной подсистемы.

Фасад рискует стать **божественным объектом**, привязанным ко всем классам программы.

Отношения с другими паттернами

- **Фасад** создаёт 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.
- **Абстрактная фабрика** может быть использована вместо **Фасада** для того, чтобы скрыть платформо-зависимые классы.
- **Легковес** показывает, как создавать много мелких объектов, а **Фасад** показывает, как создать один объект, который отображает целую подсистему.
- **Посредник** и **Фасад** похожи тем, что пытаются организовать работу множества существующих классов.
 - **Фасад** создаёт упрощённый интерфейс к подсистеме, не внося в неё никакой добавочной функциональности. Сама подсистема не знает о существовании **Фасада**. Классы подсистемы общаются друг с другом напрямую.
 - **Посредник** централизует общение между компонентами системы. Компоненты системы знают только о существовании **Посредника**, у них нет прямого доступа к другим компонентам.
- **Фасад** можно сделать **Одиночкой**, так как обычно нужен только один объект-фасад.
- **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.



ЛЕГКОВЕС

Также известен как: *Приспособленец, Кэш, Flyweight*

Легковес – это структурный паттерн проектирования, который позволяет вместить большее количество объектов в отведенную оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

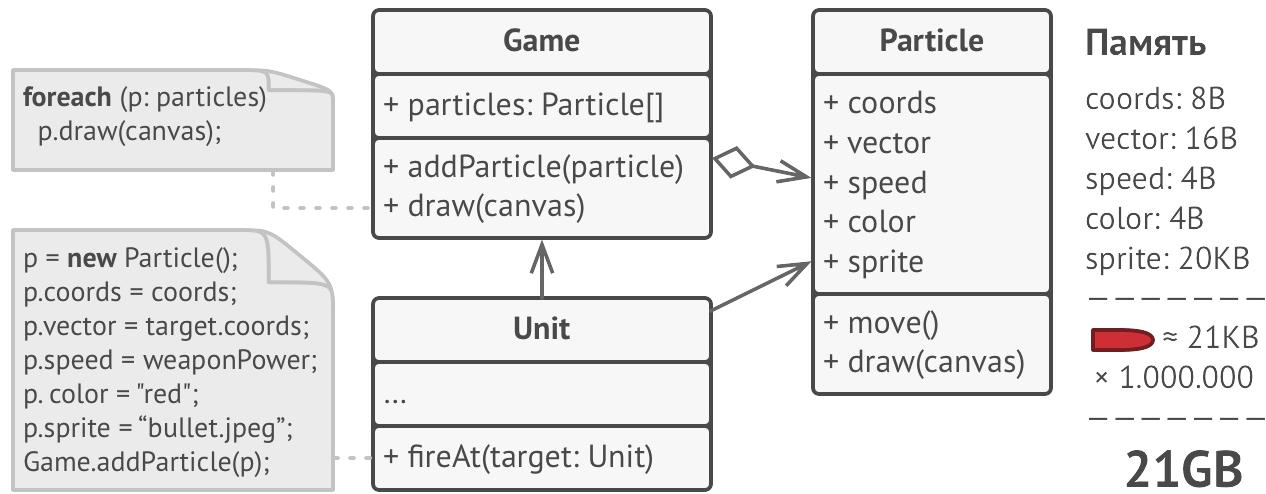
Проблема

На досуге вы решили написать небольшую игру, в которой игроки перемещаются по карте и стреляют друг в друга. Фишкой игры должна была стать реалистичная система частиц. Пули, снаряды, осколки от взрывов – всё это должно красиво летать и радовать взгляд.

Игра отлично работала на вашем мощном компьютере. Однако ваш друг сообщил, что игра начинает тормозить и вылетает через несколько минут после запуска. Покопавшись в логах, вы обнаружили, что игра вылетает из-за недостатка оперативной памяти. У вашего

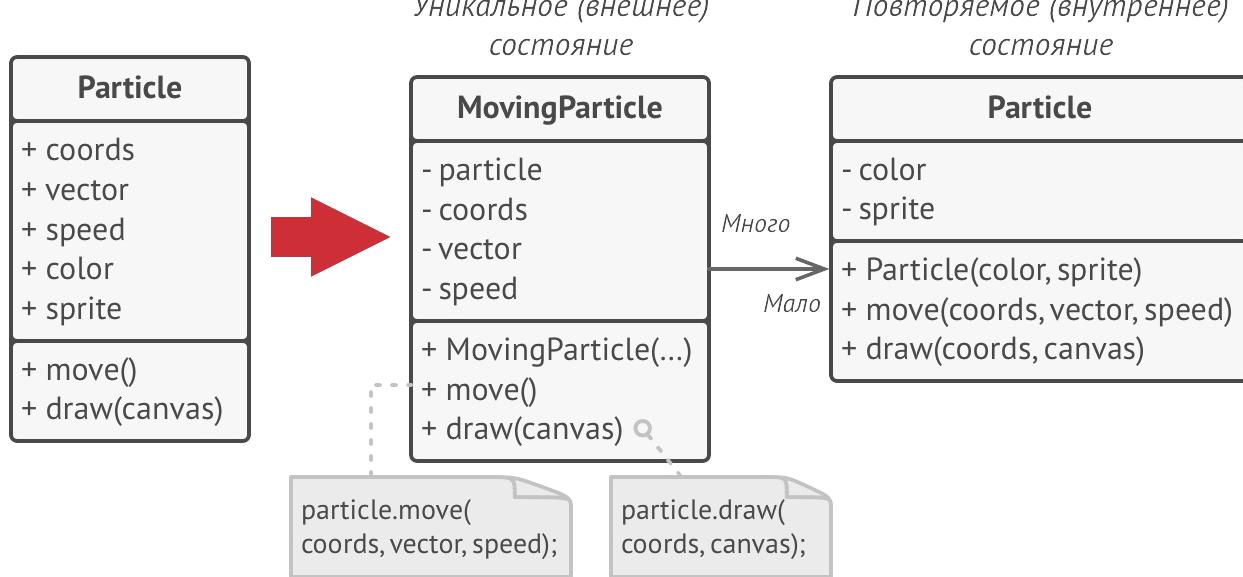
друга компьютер значительно менее «прокачанный», поэтому проблема у него и проявляется так быстро.

И действительно, каждая частица представлена собственным объектом, имеющим множество данных. В определённый момент, когда побоище на экране достигает кульминации, новые объекты частиц уже не вмещаются в оперативную память компьютера, и программа вылетает.



Решение

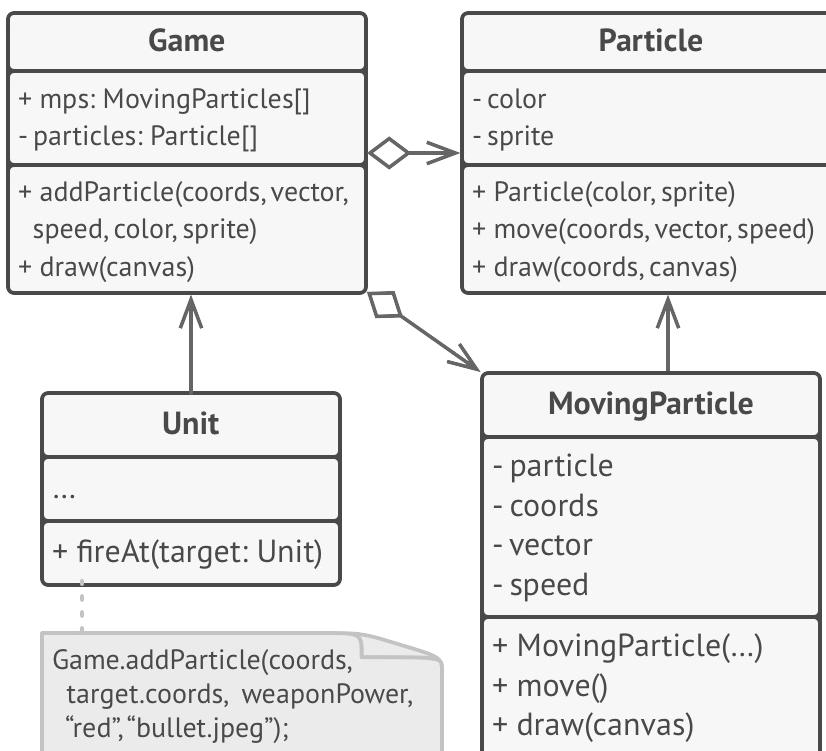
Если внимательно посмотреть на класс частиц, то можно заметить, что цвет и спрайт занимают больше всего памяти. Более того, они хранятся в каждом объекте, хотя фактически их значения одинаковы для большинства частиц.



Остальное состояние объектов – координаты, вектор движения и скорость – отличаются для всех частиц. Таким образом, эти поля можно рассматривать как контекст, в котором частица используется. А цвет и спрайт – это данные, не изменяющиеся во времени.

Неизменяемые данные объекта принято называть «внутренним состоянием». Все остальные данные – это «внешнее состояние».

Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Но главное – понадобится гораздо меньше объектов, ведь теперь они будут отличаться только внутренним состоянием, а оно имеет не так много вариаций.



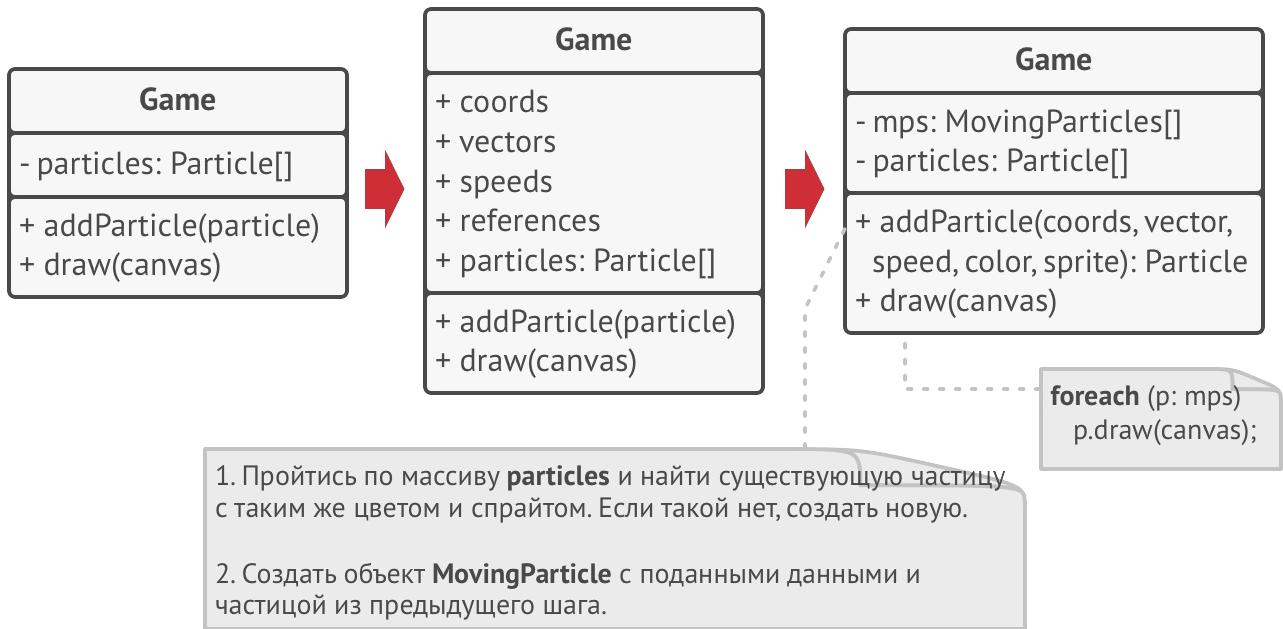
Память	coords: 8B vector: 16B color: 4B sprite: 20KB	speed: 4B particle: 4B	× 1 × 1.000.000
	----- ≈ 21KB	----- ≈ 32B	32МВ

В нашем примере с частицами достаточно будет оставить всего три объекта с отличающимися спрайтами и цветом – для пуль, снарядов и осколков. Несложно догадаться, что такие облегчённые объекты называют *легковёсами*⁶. □

Хранилище внешнего состояния

Но куда переедет внешнее состояние? Ведь кто-то должен его хранить. Чаще всего, его перемещают в контейнер, который управлял объектами до применения паттерна.

В нашем случае это был главный объект игры. Вы могли бы добавить в его класс поля-массивы для хранения координат, векторов и скоростей частиц. Кроме этого, понадобится ещё один массив для хранения ссылок на объекты-легковесы, соответствующие той или иной частице.



Но более элегантным решением было бы создать дополнительный класс-контекст, который бы связывал внешнее состояние с тем или иным легковесом. Это позволит обойтись только одним полем-массивом в классе контейнера.

«Но погодите-ка, нам потребуется столько же этих объектов, сколько было в самом начале!», – скажете вы и будете правы! Но дело в том, что объекты-контексты занимают намного меньше места, чем первоначальные. Ведь самые тяжёлые поля остались в легковесах (простите за каламбур), и сейчас мы будем ссылаться на эти объекты из контекстов, вместо того, чтобы повторно хранить дублирующееся состояние.

Неизменяемость Легковесов

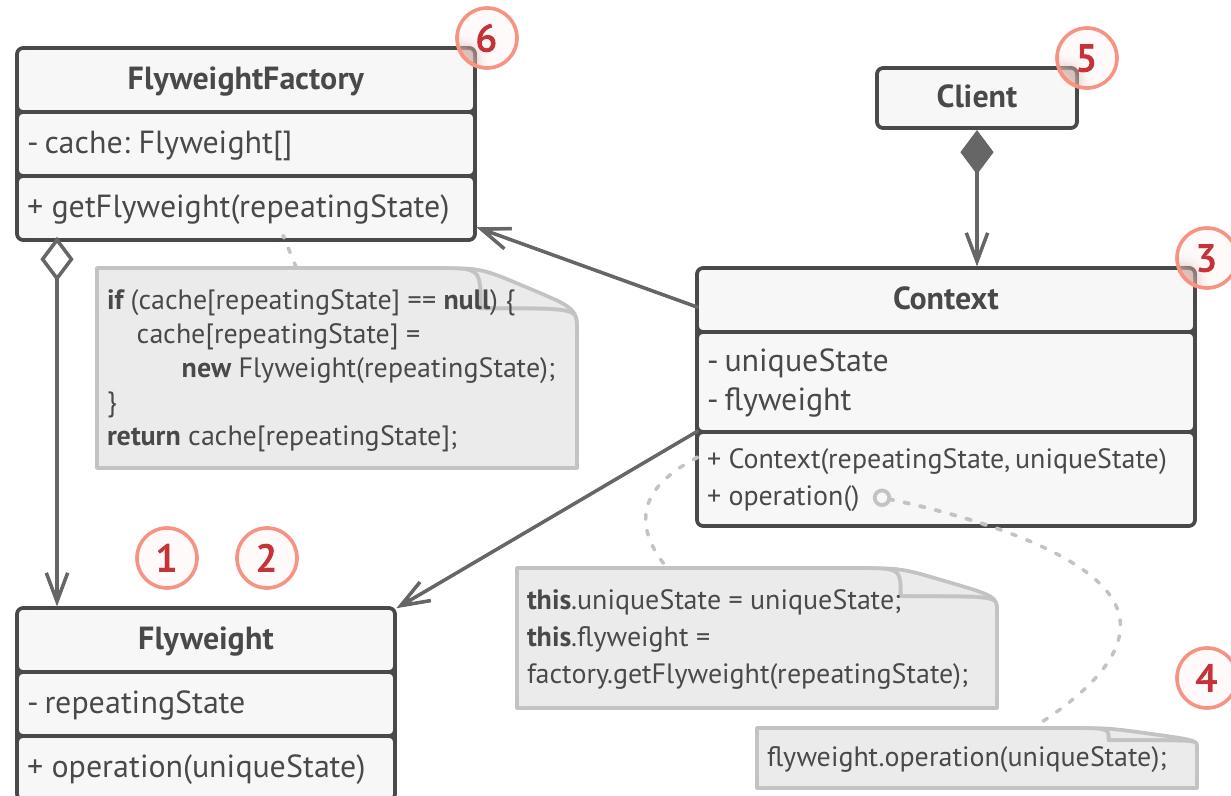
Так как объекты легковесов будут использованы в разных контекстах, вы должны быть уверены в том, что их состояние невозможно изменить после создания. Всё внутреннее состояние легковес должен получать через параметры конструктора. Он не должен иметь сеттеров и публичных полей.

Фабрика Легковесов

Для удобства работы с легковесами и контекстами можно создать фабричный метод, принимающий в параметрах всё внутреннее (а иногда и внешнее) состояние желаемого объекта.

Главная польза от этого метода в том, чтобы искать уже созданные легковесы с таким же внутренним состоянием, что и требуемое. Если легковес находится, его можно повторно использовать. Если нет – просто создаём новый. Обычно этот метод добавляют в контейнер легковесов либо создают отдельный класс-фабрику. Его даже можно сделать статическим и поместить в класс легковесов.

Структура

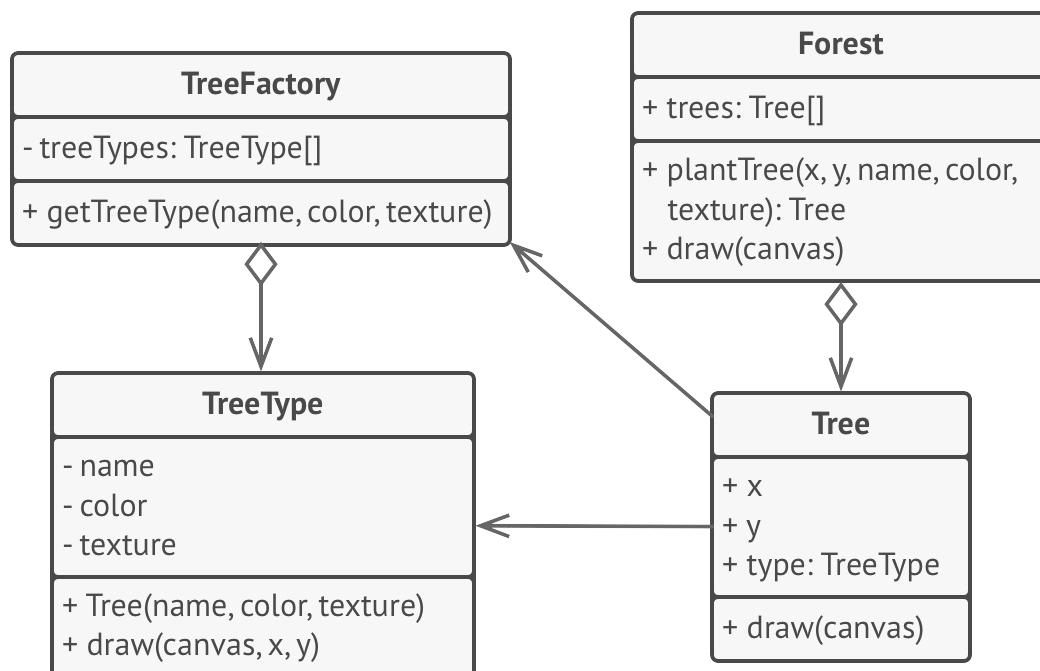


1. Вы всегда должны помнить о том, что Легковес применяется в программе, имеющей громадное количество одинаковых объектов. Этих объектов должно быть так много, чтобы они не помещались в доступную оперативную память без ухищрений. Паттерн разделяет данные этих объектов на две части – легковесы и контексты.
2. **Легковес** содержит состояние, которое повторялось во множестве первоначальных объектов. Один и тот же легковес можно использовать в связке со множеством контекстов. Состояние, которое хранится здесь, называется *внутренним*, а то, которое он получает извне – *внешним*.

3. **Контекст** содержит «внешнюю» часть состояния, уникальную для каждого объекта. Контекст связан с одним из объектов-легковесов, хранящих оставшееся состояние.
4. Поведение оригинального объекта чаще всего оставляют в Легковесе, передавая значения контекста через параметры методов. Тем не менее, поведение можно поместить и в контекст, используя легковес как объект данных.
5. **Клиент** вычисляет или хранит контекст, то есть внешнее состояние легковесов. Для клиента легковесы выглядят как шаблонные объекты, которые можно настроить во время использования, передав контекст через параметры.
6. **Фабрика легковесов** управляет созданием и повторным использованием легковесов. Фабрика получает запросы, в которых указано желаемое состояние легковеса. Если легковес с таким состоянием уже создан, фабрика сразу его возвращает, а если нет – создаёт новый объект.

Псевдокод

В этом примере **Легковес** помогает сэкономить оперативную память при отрисовке на экране миллионов объектов-деревьев.



Легковес выделяет повторяющуюся часть состояния из основного класса `Tree` и помещает его в дополнительный класс `TreeType`.

Теперь, вместо хранения повторяющихся данных во всех объектах, отдельные деревья будут ссылаться на несколько общих объектов, хранящих эти данные. Клиент работает с деревьями через фабрику деревьев, которая скрывает от него сложность кеширования общих данных деревьев.

Таким образом, программа будет использовать намного меньше оперативной памяти, что позволит отрисовать больше деревьев на экране на том же железе.

```
1 // Этот класс-легковес содержит часть полей, которые описывают
2 // деревья. Эти поля не уникальны для каждого дерева, в отличие,
3 // например, от координат: несколько деревьев могут иметь ту же
4 // текстуру.
5 //
6 // Поэтому мы переносим повторяющиеся данные в один-единственный
7 // объект и ссылаемся на него из множества отдельных деревьев.
8 class TreeType is
9     field name
10    field color
11    field texture
12    constructor TreeType(name, color, texture) { ... }
13    method draw(canvas, x, y) is
14        // 1. Создать картинку данного типа, цвета и текстуры.
15        // 2. Нарисовать картинку на холсте в позиции X, Y.
16
17 // Фабрика легковесов решает, когда нужно создать новый
18 // легковес, а когда можно обойтись существующим.
19 class TreeFactory is
20    static field treeTypes: collection of tree types
21    static method getTreeType(name, color, texture) is
22        type = treeTypes.find(name, color, texture)
23        if (type == null)
24            type = new TreeType(name, color, texture)
25            treeTypes.add(type)
26    return type
27
```

```

28 // Контекстный объект, из которого мы выделили легковес
29 // TreeType. В программе могут быть тысячи объектов Tree, так
30 // как накладные расходы на их хранение совсем небольшие – в
31 // памяти нужно держать всего три целых числа (две координаты и
32 // ссылка).
33 class Tree is
34   field x,y
35   field type: TreeType
36   constructor Tree(x, y, type) { ... }
37   method draw(canvas) is
38     type.draw(canvas, this.x, this.y)
39
40 // Классы Tree и Forest являются клиентами Легковеса. При
41 // желании их можно слить в один класс, если вам не нужно
42 // расширять класс деревьев далее.
43 class Forest is
44   field trees: collection of Trees
45
46   method plantTree(x, y, name, color, texture) is
47     type = TreeFactory.getType(name, color, texture)
48     tree = new Tree(x, y, type)
49     trees.add(tree)
50
51   method draw(canvas) is
52     foreach (tree in trees) do
53       tree.draw(canvas)

```

Применимость

Когда не хватает оперативной памяти для поддержки всех нужных объектов.

Эффективность паттерна **Легковес** во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены все перечисленные условия:

- в приложении используется большое число объектов;
- из-за этого высоки расходы оперативной памяти;

- большую часть состояния объектов можно вынести за пределы их классов;
- большие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено.

Шаги реализации

1. Разделите поля класса, который станет легковесом, на две части:
 - внутреннее состояние: значения этих полей одинаковы для большого числа объектов;
 - внешнее состояние (контекст): значения полей уникальны для каждого объекта.
2. Оставьте поля внутреннего состояния в классе, но убедитесь, что их значения неизменяемы. Эти поля должны инициализироваться только через конструктор.
3. Превратите поля внешнего состояния в параметры методов, где эти поля использовались. Затем удалите поля из класса.
4. Создайте фабрику, которая будет кешировать и повторно отдавать уже созданные объекты. Клиент должен запрашивать из этой фабрики легковеса с определённым внутренним состоянием, а не создавать его напрямую.
5. Клиент должен хранить или вычислять значения внешнего состояния (контекст) и передавать его в методы объекта легковеса.

Преимущества и недостатки

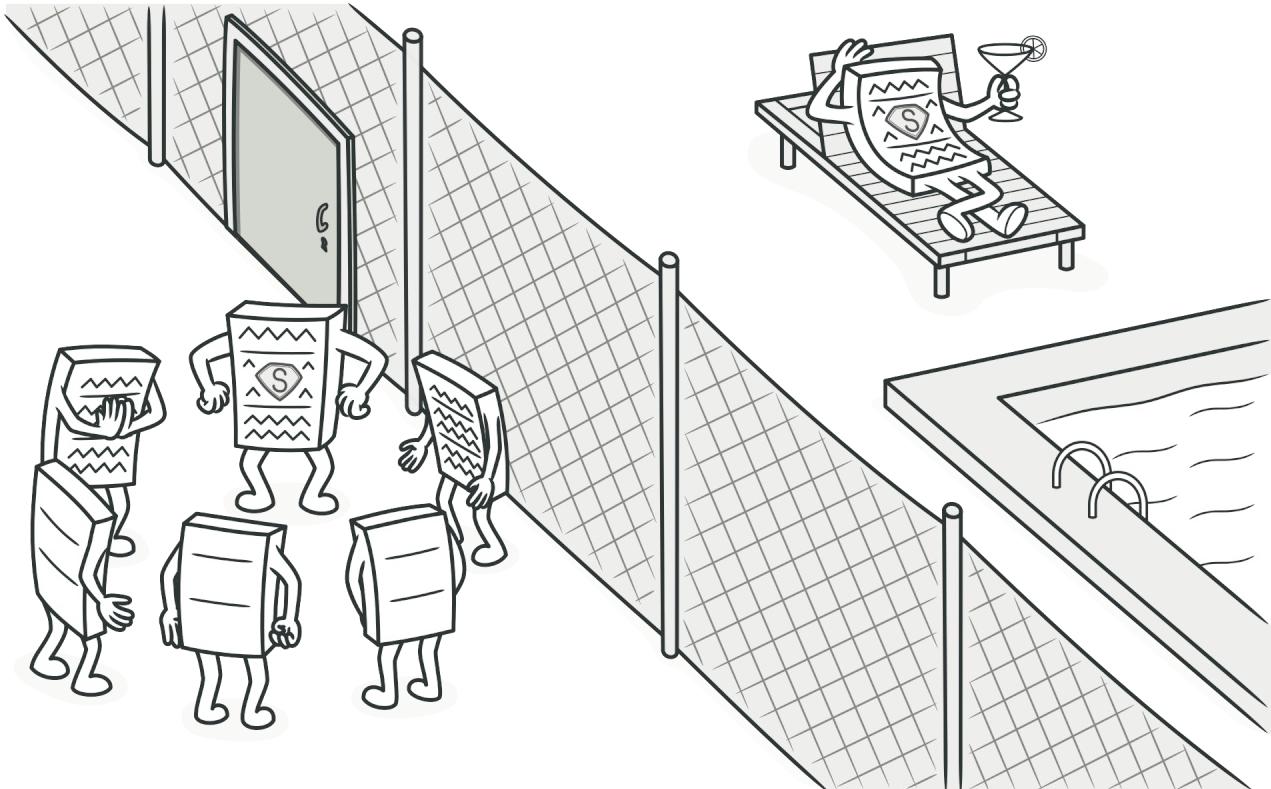
Экономит оперативную память.

Расходует процессорное время на поиск/вычисление контекста.

Усложняет код программы из-за введения множества дополнительных классов.

Отношения с другими паттернами

- **Компоновщик** часто совмещают с **Легковесом**, чтобы реализовать общие ветки дерева и сэкономить при этом память.
- **Легковес** показывает, как создавать много мелких объектов, а **Фасад** показывает, как создать один объект, который отображает целую подсистему.
- 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.



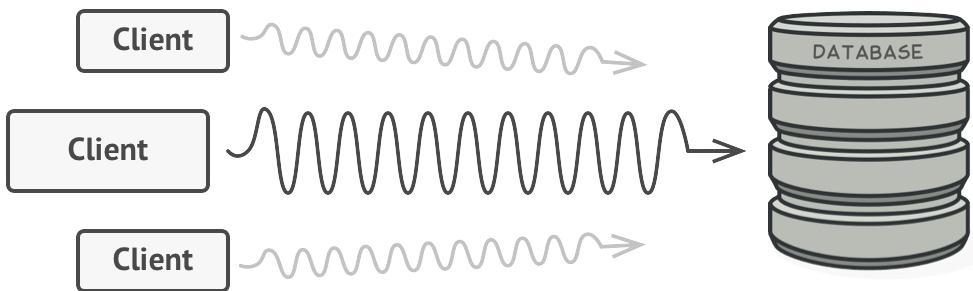
ЗАМЕСТИТЕЛЬ

Также известен как: *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.

Проблема

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



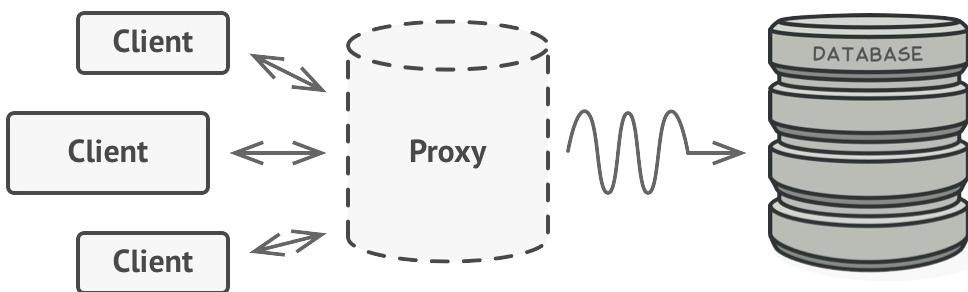
Запросы к базе данных могут быть очень медленными.

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.

Решение

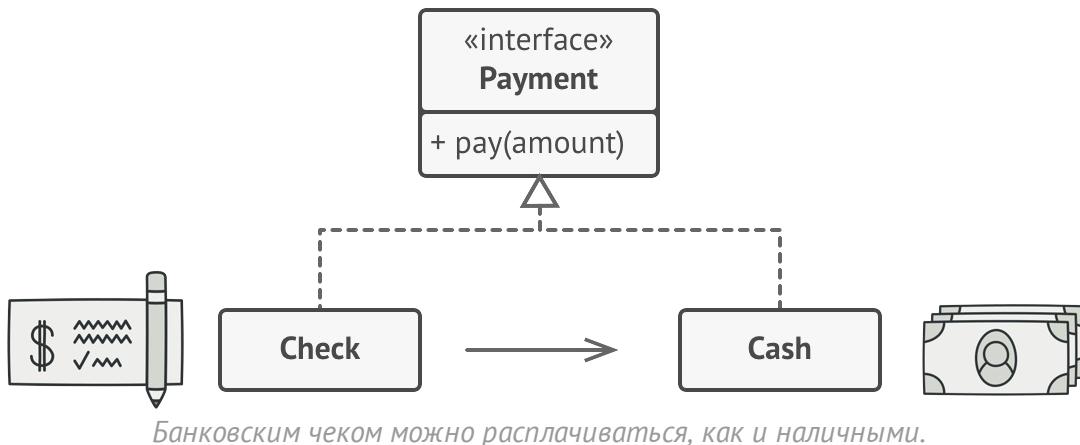
Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.



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

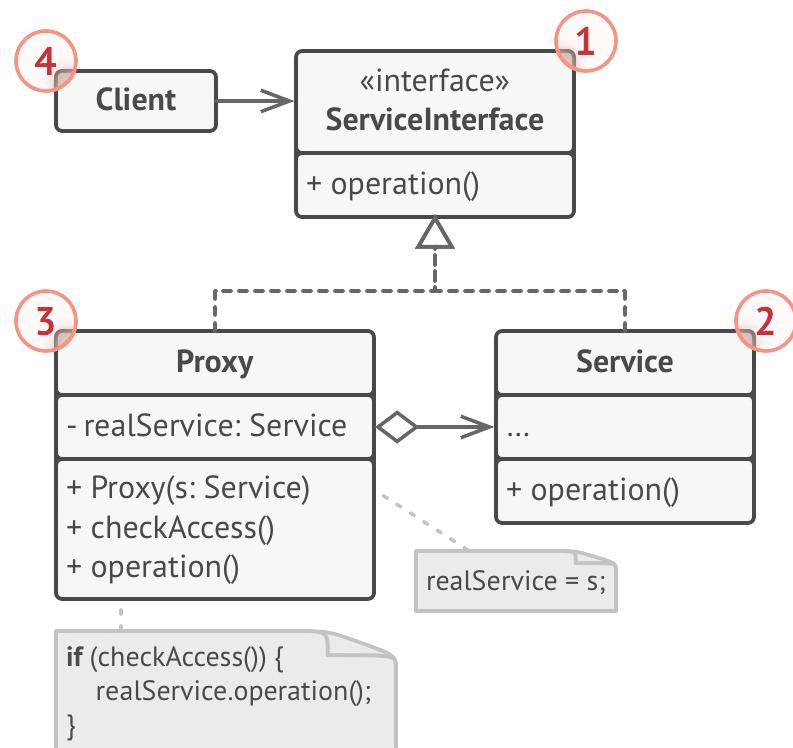
Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одноковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.

Аналогия из жизни



Банковский чек – это заместитель пачки наличных. И чек, 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 может превратить чек в зелёные бумажки, обратившись в банк.

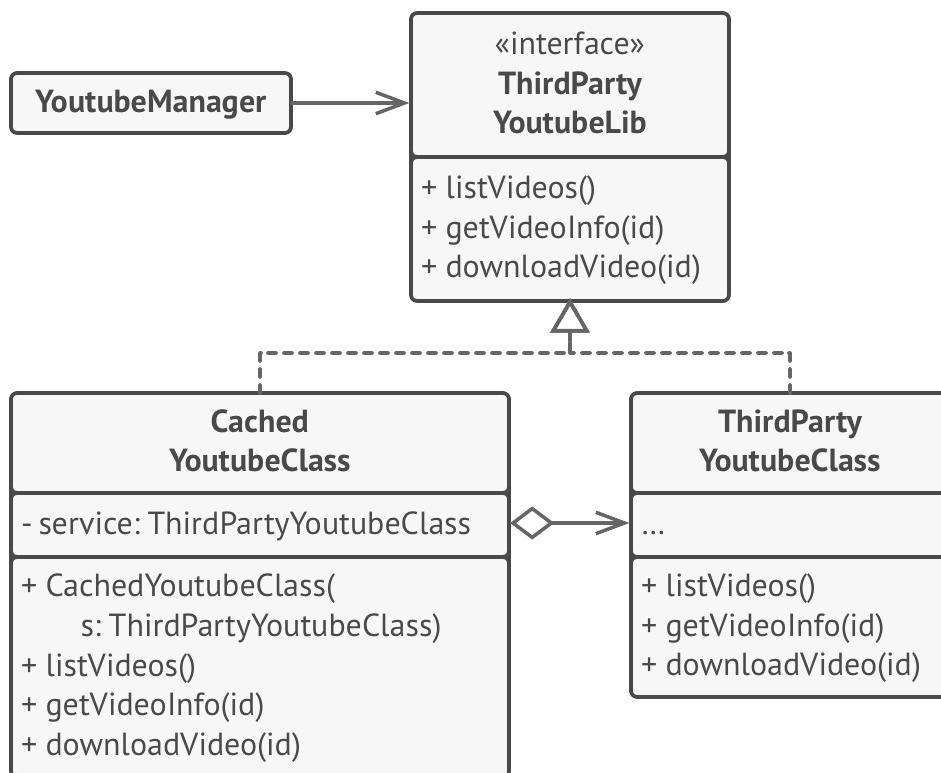
Структура



1. **Интерфейс сервиса** определяет общий интерфейс для сервиса и заместителя. Благодаря этому, объект заместителя можно использовать там, где ожидается объект сервиса.
2. **Сервис** содержит полезную бизнес-логику.
3. **Заместитель** хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису.
Заместитель может сам отвечать за создание и удаление объекта сервиса.
4. **Клиент** работает с объектами через интерфейс сервиса. Благодаря этому, его можно «одурачить», подменив объект сервиса объектом заместителя.

Псевдокод

В этом примере **Заместитель** помогает добавить в программу механизм ленивой инициализации и кеширования результатов работы библиотеки интеграции с Youtube.



Пример кеширования результатов работы реального сервиса с помощью заместителя.

Оригинальный объект начинал загрузку по сети, даже если пользователь запрашивал одно и то же видео. Заместитель же загружает видео только один раз, используя для этого служебный объект, но в остальных случаях возвращает закешированный файл.

```

1 // Интерфейс удалённого сервиса.
2 interface ThirdPartyYoutubeLib is
3     method listVideos()
4     method getVideoInfo(id)
5     method downloadVideo(id)
6
7 // Конкретная реализация сервиса. Методы этого класса
8 // запрашивают у Youtube различную информацию. Скорость запроса
9 // зависит не только от качества интернет-канала пользователя,
10 // но и от состояния самого Youtube. Значит, чем больше будет
11 // вызовов к сервису, тем менее отзывчивой станет программа.
12 class ThirdPartyYoutubeClass is
13     method listVideos() is
14         // Получить список видеороликов с помощью API Youtube.
15
16     method getVideoInfo(id) is
17         // Получить детальную информацию о каком-то видеоролике.
18

```

```
19 method downloadVideo(id) is
20     // Скачать видео с Youtube.
21
22 // С другой стороны, можно кешировать запросы к Youtube и не
23 // повторять их какое-то время, пока кеш не устареет. Но внести
24 // этот код напрямую в сервисный класс нельзя, так как он
25 // находится в сторонней библиотеке. Поэтому мы поместим логику
26 // кеширования в отдельный класс-обёртку. Он будет делегировать
27 // запросы к сервисному объекту, только если нужно
28 // непосредственно выслать запрос.
29 class CachedYoutubeClass implements ThirdPartyYoutubeLib is
30     private field service: ThirdPartyYoutubeClass
31     private field listCache, videoCache
32     field needReset
33
34 constructor CachedYoutubeClass(service: ThirdPartyYoutubeLib) is
35     this.service = service
36
37 method listVideos() is
38     if (listCache == null || needReset)
39         listCache = service.listVideos()
40     return listCache
41
42 method getVideoInfo(id) is
43     if (videoCache == null || needReset)
44         videoCache = service.getVideoInfo(id)
45     return videoCache
46
47 method downloadVideo(id) is
48     if (!downloadExists(id) || needReset)
49         service.downloadVideo(id)
50
51 // Класс GUI, который использует сервисный объект. Вместо
52 // реального сервиса, мы подсунем ему объект-заместитель. Клиент
53 // ничего не заметит, так как заместитель имеет тот же
54 // интерфейс, что и сервис.
55 class YoutubeManager is
56     protected field service: ThirdPartyYoutubeLib
57
58 constructor YoutubeManager(service: ThirdPartyYoutubeLib) is
59     this.service = service
```

```
60
61     method renderVideoPage() is
62         info = service.getVideoInfo()
63         // Отобразить страницу видеоролика.
64
65     method renderListPanel() is
66         list = service.listVideos()
67         // Отобразить список превьюшек видеороликов.
68
69     method reactOnUserInput() is
70         renderVideoPage()
71         renderListPanel()
72
73 // Конфигурационная часть приложения создаёт и передаёт клиентам
74 // объект заместителя.
75 class Application is
76     method init() is
77         youtubeService = new ThirdPartyYoutubeClass()
78         youtubeProxy = new CachedYoutubeClass(youtubeService)
79         manager = new YoutubeManager(youtubeProxy)
80         manager.reactOnUserInput()
```

Применимость

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.

Кэширование объектов («умная» ссылка). Когда нужно кэшировать 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 здоро́во экономить ресурсы, особенно если речь идёт о больших прожорливых сервисах.

Шаги реализации

1. Определите интерфейс, который бы сделал заместитель и оригинальный объект взаимозаменяемыми.
2. 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.

3. Реализуйте методы заместителя в зависимости от его предназначения. В большинстве случаев, проделав какую-то полезную работу, методы заместителя должны передать запрос сервисному объекту.
4. 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.
5. Подумайте, не реализовать ли вам ленивую инициализацию сервисного объекта при первом обращении клиента к методам заместителя.

Преимущества и недостатки

Позволяет контролировать сервисный объект незаметно для клиента.

Может работать, даже если сервисный объект ещё не создан.

Может контролировать жизненный цикл служебного объекта.

Усложняет код программы из-за введения дополнительных классов.

Увеличивает время отклика от сервиса.

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 композиции и делегируют работу другим объектам. Паттерны отличаются тем, что Заместитель сам управляет жизнью сервисного объекта, а обёртывание Декораторов контролируется клиентом.

Поведенческие паттерны

Эти паттерны решают задачи эффективного и безопасного взаимодействия между объектами программы.



Цепочка обязанностей

Chain of Responsibility

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



Команда

Command

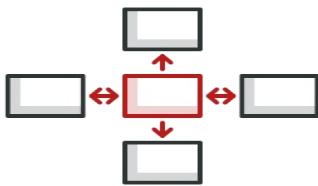
Превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.



Итератор

Iterator

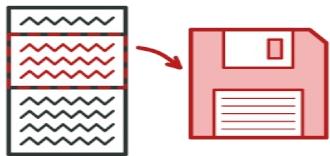
Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.



Посредник

Mediator

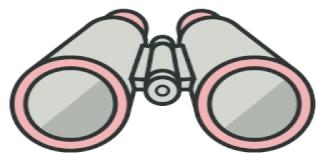
Позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.



Снимок

Memento

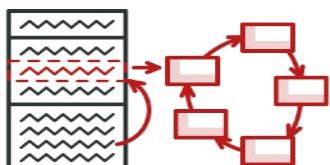
Позволяет делать снимки состояния объектов, не раскрывая подробностей их реализации. Затем снимки можно использовать, чтобы восстановить прошлое состояние объектов.



Наблюдатель

Observer

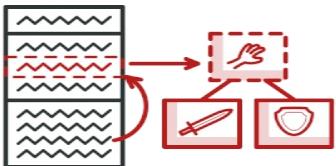
Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



Состояние

State

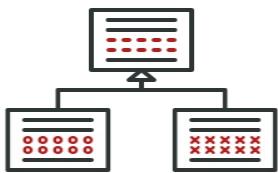
Позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



Стратегия

Strategy

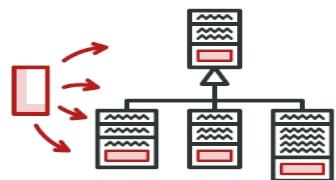
Определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.



Шаблонный метод

Template method

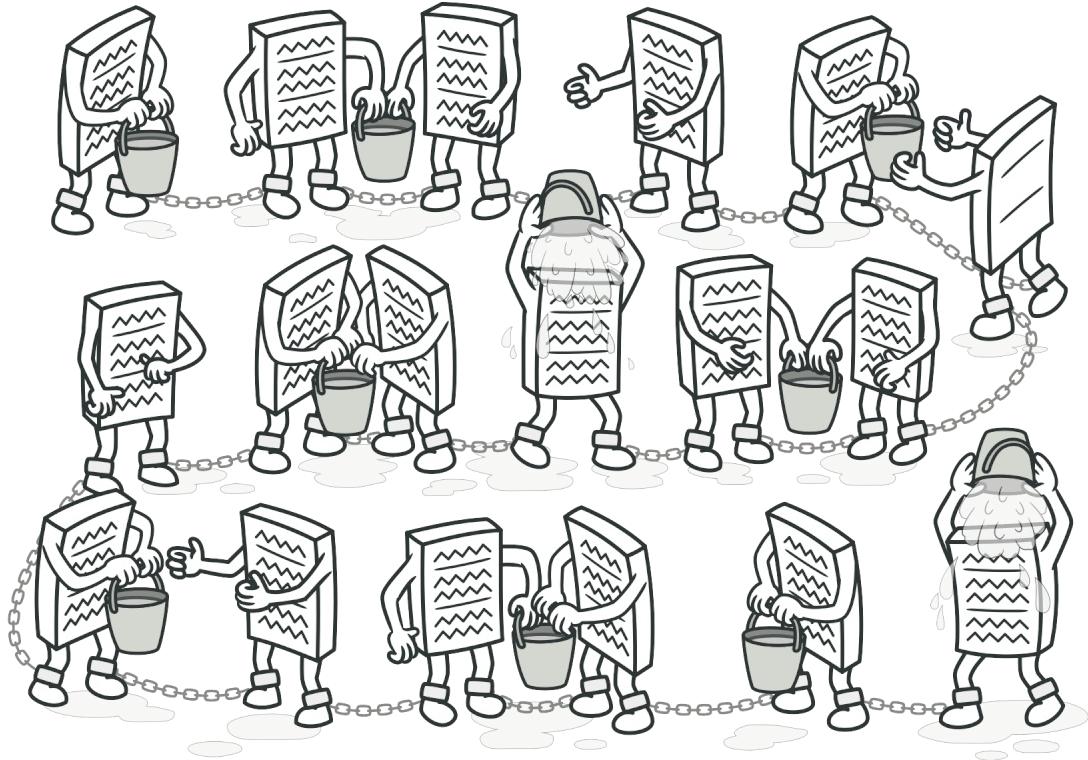
Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



Посетитель

Visitor

Позволяет создавать новые операции, не меняя классы объектов, над которыми эти операции могут выполняться.



ЦЕПОЧКА ОБЯЗАННОСТЕЙ

Также известен как: *CoR, Chain of Command, Chain of Responsibility*

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

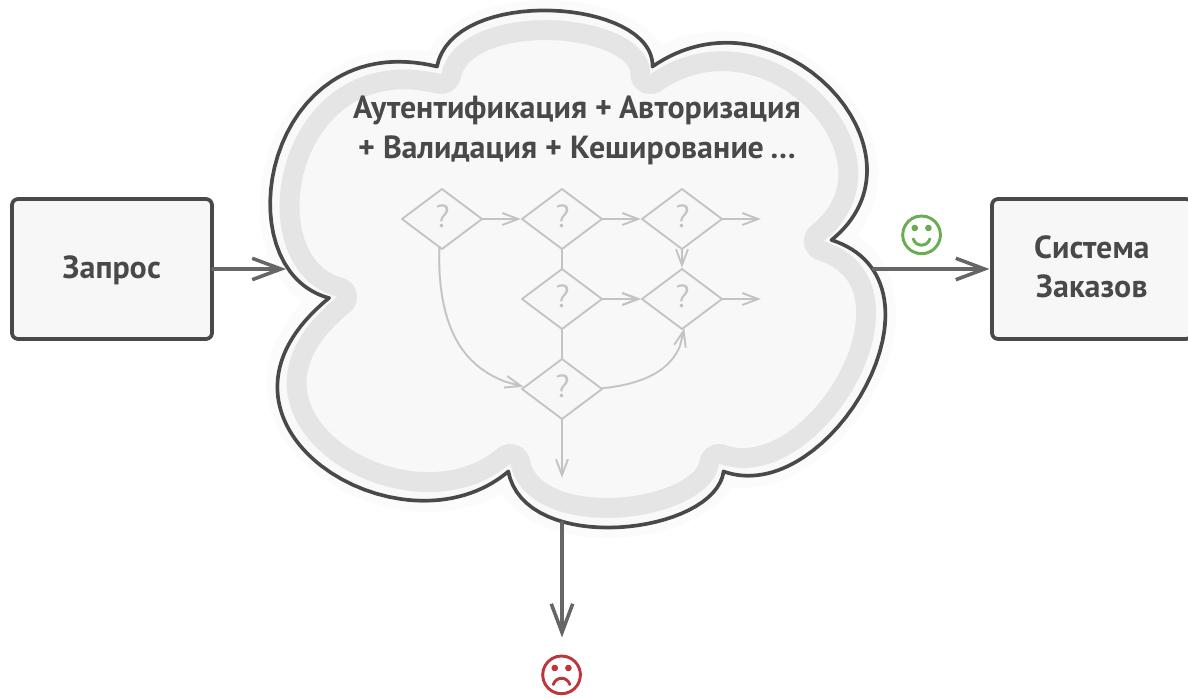
и пароль. Но если такая попытка не удалась, то проверять расширенные права доступа попросту не имеет смысла.



Запрос проходит ряд проверок перед доступом в систему заказов.

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.



Со временем код проверок становится всё более запутанным.

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

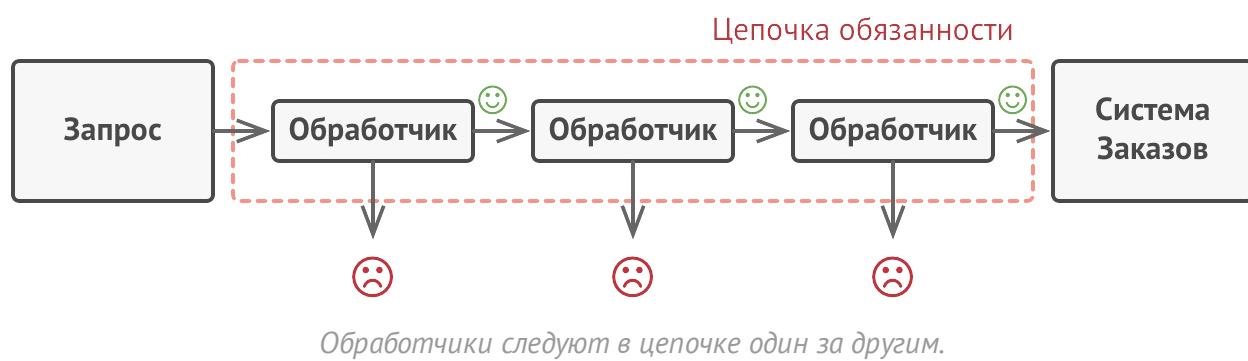
Как и многие другие поведенческие паттерны, Цепочка обязанностей базируется на том, чтобы превратить отдельные поведения в объекты. В нашем случае каждая проверка переедет в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод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

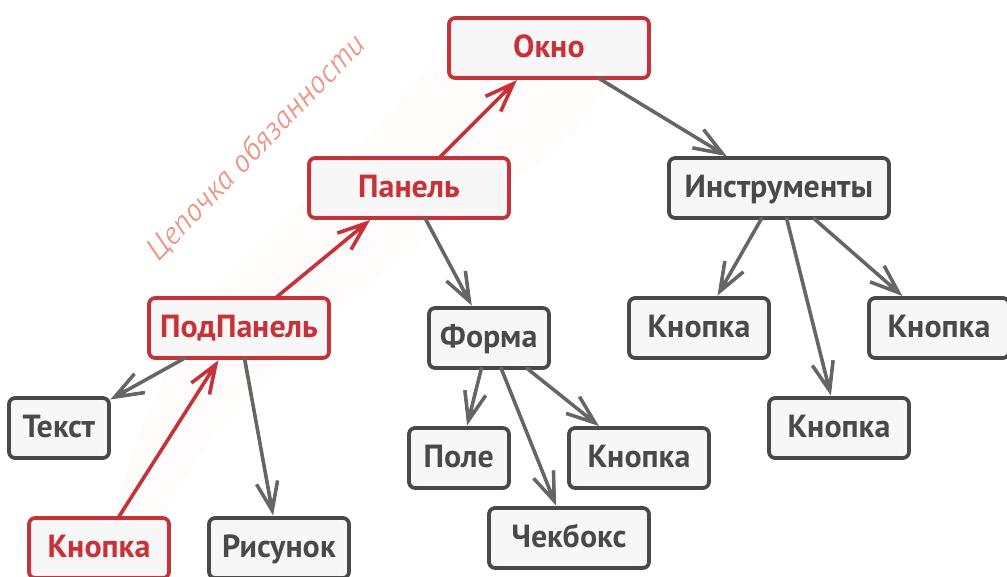
И последний штрих. Обработчик не обязательно должен передавать запрос дальше, причём эта особенность может быть использована по-разному.

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.



Цепочку можно выделить даже из дерева объектов.

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.

Аналогия из жизни



Пример общения с поддержкой.

Вы купили новую видеокарту. Она автоматически определилась и заработала под Windows, но в вашей любимой Ubuntu «завести» её не удалось. Со слабой надеждой вы звоните в

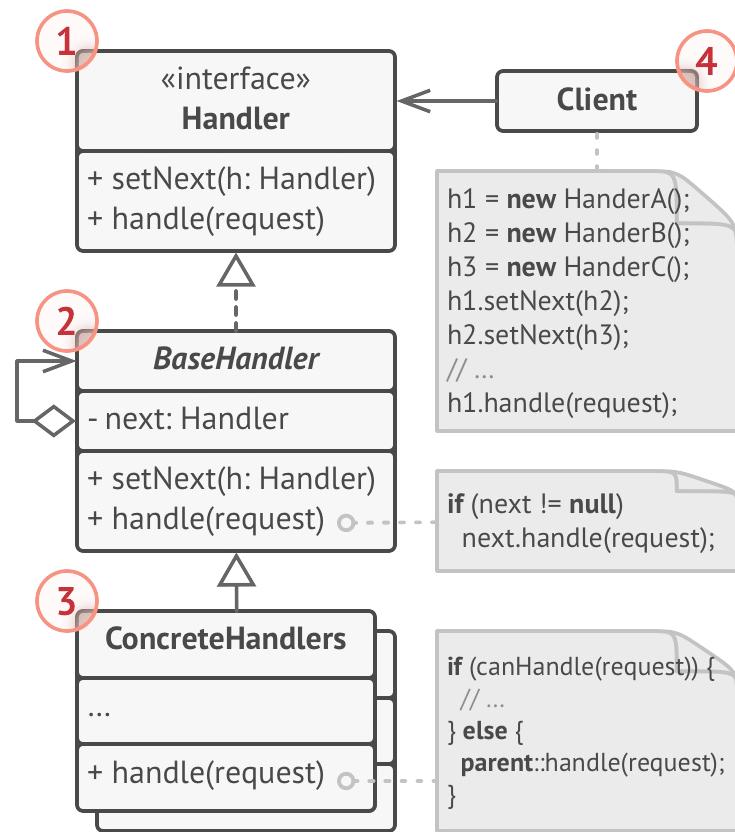
службу поддержки.

Первым вы слышите голос автоответчика, предлагающий выбор из десятка стандартных решений. Ни один из вариантов не подходит, и робот соединяет вас с живым оператором.

Увы, но рядовой оператор поддержки умеет общаться только заученными фразами и давать шаблонные ответы. После очередного предложения «выключить и включить компьютер» вы просите связать вас с настоящими инженерами.

Оператор перебрасывает звонок дежурному инженеру, изнывающему от скуки в своей каморке. Уж он-то знает, как вам помочь! Инженер рассказывает вам, где скачать подходящие драйвера и как настроить их под Ubuntu. Запрос удовлетворён. Вы кладёте трубку.

Структура



1. **Обработчик** определяет общий для всех конкретных обработчиков интерфейс. Обычно достаточно описать единственный метод обработки запросов, но иногда здесь может

быть объявлен и метод выставления следующего обработчика.

2. **Базовый обработчик** – опциональный класс, который позволяет избавиться от дублирования одного и того же кода во всех конкретных обработчиках.

Обычно этот класс имеет поле для хранения ссылки на следующий обработчик в цепочке. Клиент связывает обработчики в цепь, подавая ссылку на следующий обработчик через конструктор или сеттер поля. Также здесь можно реализовать базовый метод обработки, который бы просто перенаправлял запрос следующему обработчику, проверив его наличие.

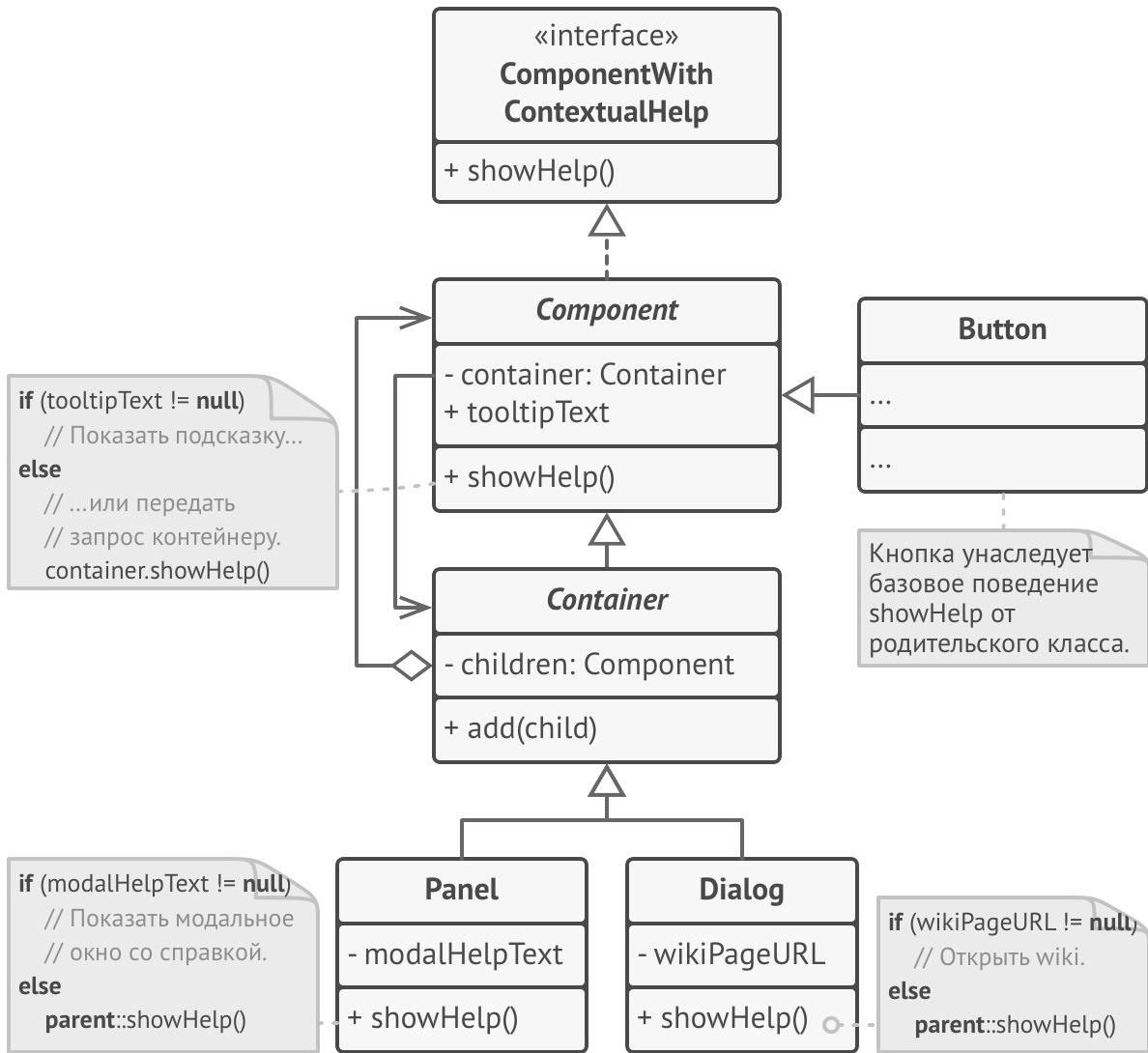
3. **Конкретные обработчики** содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту.

В большинстве случаев обработчики могут работать сами по себе и быть неизменяемыми, получив все нужные детали через параметры конструктора.

4. **Клиент** может либо сформировать цепочку обработчиков единожды, либо перестраивать её динамически, в зависимости от логики программы. Клиент может отправлять запросы любому из объектов цепочки, не обязательно первому из них.

Псевдокод

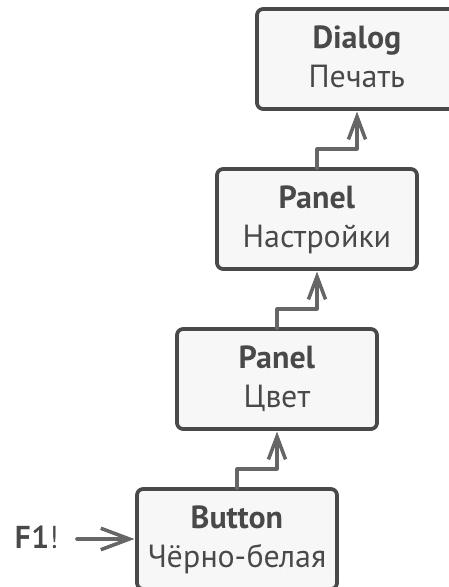
В этом примере **Цепочка обязанностей** отвечает за показ контекстной помощи для активных элементов пользовательского интерфейса.



Графический интерфейс построен с помощью компоновщика, где у каждого элемента есть ссылка на свой элемент-контейнер. Цепочку можно выстроить, пройдясь по всем контейнерам, в которые вложен элемент.

Графический интерфейс приложения обычно структурирован в виде дерева. Класс **Диалог**, отображающий всё окно приложения – это корень дерева. Диалог содержит **Панели**, которые, в свою очередь, могут содержать либо другие вложенные панели, либо простые элементы, вроде **Кнопок**.

Простые элементы могут показывать небольшие подсказки, если для них указан текст помощи. Но есть и более сложные компоненты, для которых этот способ демонстрации помощи слишком прост. Они определяют собственный способ отображения контекстной помощи.



Пример вызова контекстной помощи в цепочке объектов UI.

Когда пользователь наводит указатель мыши на элемент и жмёт клавишу `F1`, приложение шлёт этому элементу запрос на показ помощи. Если он не содержит никакой справочной информации, запрос путешествует далее по списку контейнера элемента, пока не находится тот, который способен отобразить помощь.

```

1 // Интерфейс обработчиков.
2 interface ComponentWithContextualHelp is
3     method showHelp()
4
5
6 // Базовый класс простых компонентов.
7 abstract class Component implements ComponentWithContextualHelp is
8     field tooltipText: string
9
10 // Контейнер, содержащий компонент, служит в качестве
11 // следующего звена цепочки.
12 protected field container: Container
13
14 // Базовое поведение компонента заключается в том, чтобы
15 // показать всплывающую подсказку, если для неё задан текст.
16 // В обратном случае – перенаправить запрос своему
17 // контейнеру, если тот существует.
18 method showHelp() is
19     if (tooltipText != null)
20         // Показать подсказку.

```

```
21     else
22         container.showHelp()
23
24
25 // Контейнеры могут включать в себя как простые компоненты, так
26 // и другие контейнеры. Здесь формируются связи цепочки. Класс
27 // контейнера унаследует метод showHelp от своего родителя –
28 // базового компонента.
29 abstract class Container extends Component is
30     protected field children: array of Component
31
32     method add(child) is
33         children.add(child)
34         child.container = this
35
36
37 // Большинство примитивных компонентов устроит базовое поведение
38 // показа помощи через подсказку, которое они унаследуют из
39 // класса Component.
40 class Button extends Component is
41     // ...
42
43 // Но сложные компоненты могут переопределять метод показа
44 // помощи по–своему. Но и в этом случае они всегда могут
45 // вернуться к базовой реализации, вызвав метод родителя.
46 class Panel extends Container is
47     field modalHelpText: string
48
49     method showHelp() is
50         if (modalHelpText != null)
51             // Показать модальное окно с помощью.
52         else
53             super.showHelp()
54
55 // ...то же, что и выше...
56 class Dialog extends Container is
57     field wikiPageURL: string
58
59     method showHelp() is
60         if (wikiPageURL != null)
61             // Открыть страницу Wiki в браузере.
```

```
62     else
63         super.showHelp()
64
65
66 // Клиентский код.
67 class Application is
68     // Каждое приложение конфигурирует цепочку по-своему.
69     method createUI() is
70         dialog = new Dialog("Budget Reports")
71         dialog.wikiPage = "http://..."
72         panel = new Panel(0, 0, 400, 800)
73         panel.modalHelpText = "This panel does..."
74         ok = new Button(250, 760, 50, 20, "OK")
75         ok.tooltipText = "This is a OK button that..."
76         cancel = new Button(320, 760, 50, 20, "Cancel")
77         // ...
78         panel.add(ok)
79         panel.add(cancel)
80         dialog.add(panel)
81
82 // Представьте, что здесь произойдёт.
83 method onF1KeyPress() is
84     component = this.getComponentAtMouseCoords()
85     component.showHelp()
```

Применимость

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.

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.

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

1. 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.

2. 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.

3. 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?

4. 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.
5. Клиент может посылать 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.
6. 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

Уменьшает зависимость между клиентом и обработчиками.

Реализует *принцип единственной обязанности*.

Реализует *принцип открытости/закрытости*.

Запрос может остаться никем не обработанным.

Отношения с другими паттернами

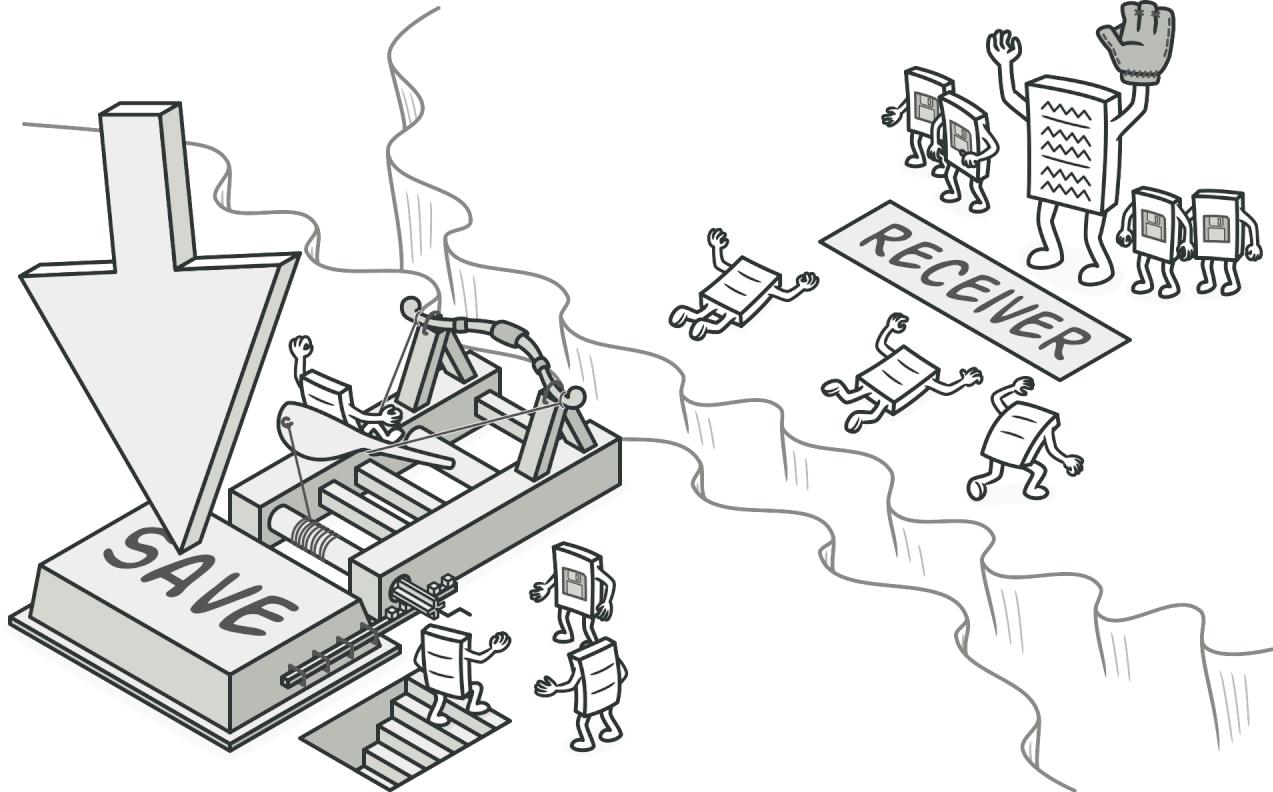
- **Цепочка обязанностей, Команда, Посредник и Наблюдатель** показывают различные способы работы отправителей запросов с их получателями:

- 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.

- 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.



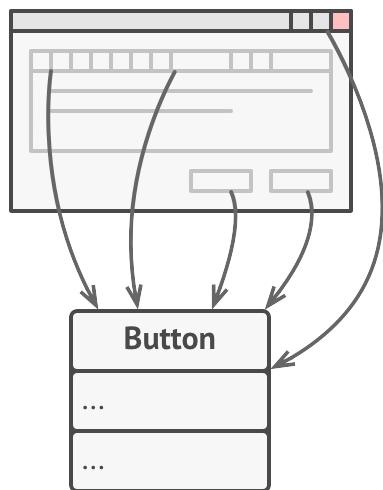
КОМАНДА

Также известен как: *Действие, Транзакция, Action, Command*

Команда – это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

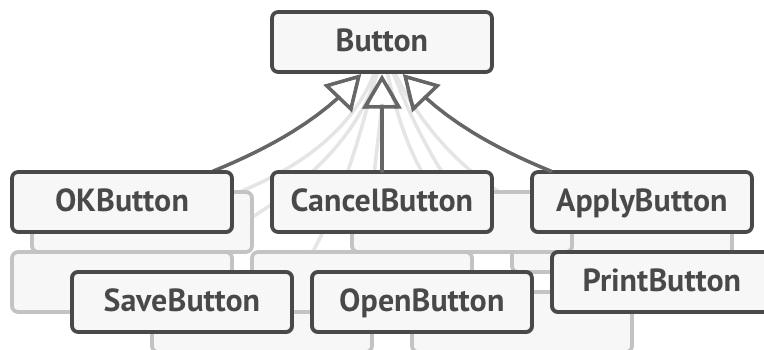
Проблема

Представьте, что вы работаете над программой текстового редактора. Дело как раз подошло к разработке панели управления. Вы создали класс красивых Кнопок и хотите использовать его для всех кнопок приложения, начиная от панели управления, заканчивая простыми кнопками в диалогах.



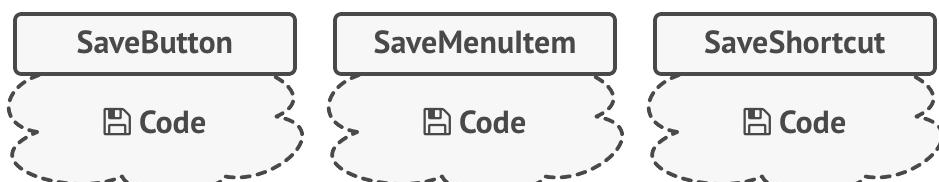
Все кнопки приложения унаследованы от одного класса.

Все эти кнопки, хоть и выглядят схоже, но делают разные вещи. Поэтому возникает вопрос: куда поместить код обработчиков кликов по этим кнопкам? Самым простым решением было бы создать подклассы для каждой кнопки и переопределить в них метод действия под разные задачи.



Множество подклассов кнопок.

Но скоро стало понятно, что такой подход никуда не годится. Во-первых, получается очень много подклассов. Во-вторых, код кнопок, относящийся к графическому интерфейсу, начинает зависеть от классов бизнес-логики, которая довольно часто меняется.



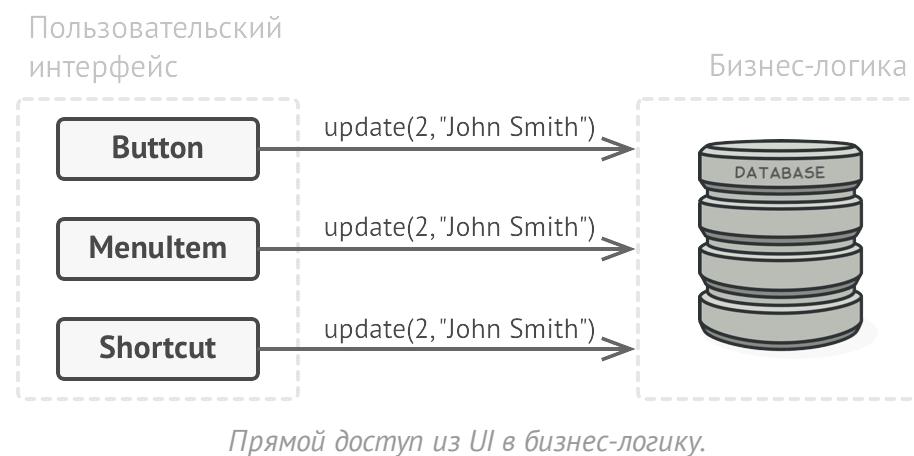
Несколько классов дублируют одну и ту же функциональность.

Но самое обидное ещё впереди. Ведь некоторые операции, например, «сохранить», можно вызывать из нескольких мест: нажав кнопку на панели управления, вызвав контекстное меню или просто нажав клавиши `Ctrl+S`. Когда в программе были только кнопки, код сохранения имелся только в подклассе `SaveButton`. Но теперь его придётся продублировать ещё в два класса.

Решение

Хорошие программы обычно структурированы в виде слоёв. Самый распространённый пример – слои пользовательского интерфейса и бизнес-логики. Первый всего лишь рисует красивую картинку для пользователя. Но когда нужно сделать что-то важное, интерфейс «просит» слой бизнес-логики заняться этим.

В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры.

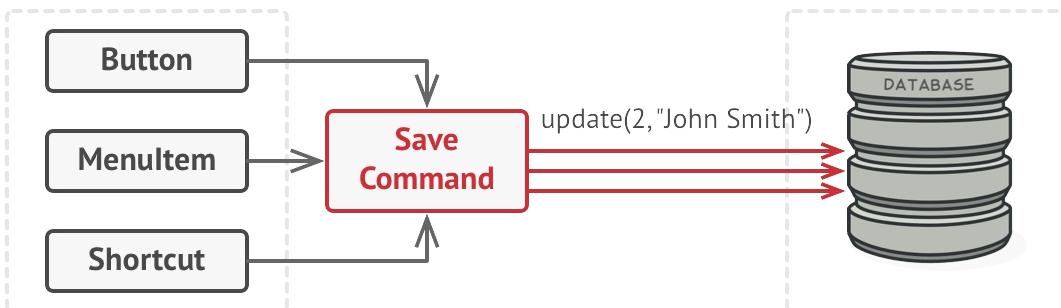


Паттерн Команда предлагает больше не отправлять такие вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют **командами**.

К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та – позаботится обо всём остальном.

Пользовательский
интерфейс

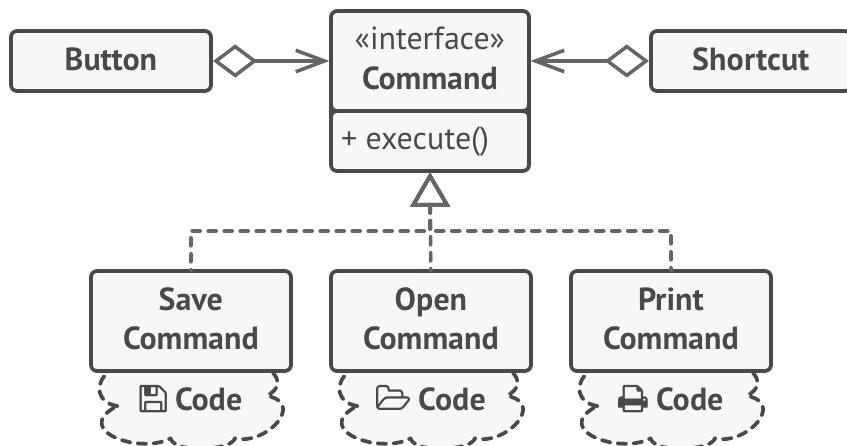
Бизнес-логика



Доступ из UI в бизнес-логику через команду.

Классы команд можно объединить под общим интерфейсом с единственным методом запуска. После этого одни и те же отправители смогут работать с различными командами, не привязываясь к их классам. Даже больше: команды можно будет взаимозаменять на лету, изменяя итоговое поведение отправителей.

Параметры, с которыми должен быть вызван метод объекта получателя, можно загодя сохранить в полях объекта-команды. Благодаря этому, объекты, отправляющие запросы, могут не беспокоиться о том, чтобы собрать необходимые для получателя данные. Более того, они теперь вообще не знают, кто будет получателем запроса. Вся эта информация скрыта внутри команды.



Классы UI делегируют работу командам.

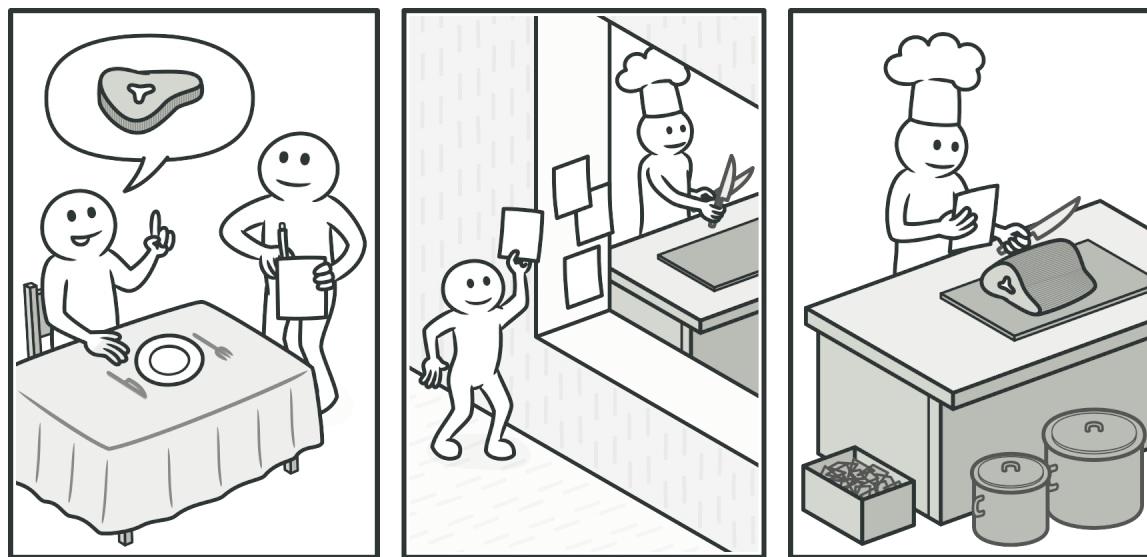
После применения Команды в нашем примере с текстовым редактором вам больше не потребуется создавать уйму подклассов кнопок под разные действия. Будет достаточно единственного класса с полем для хранения объекта команды.

Используя общий интерфейс команд, объекты кнопок будут ссылаться на объекты команд различных типов. При нажатии кнопки будут делегировать работу связанным командам, а команды – перенаправлять вызовы тем или иным объектам бизнес-логики.

Так же можно поступить и с контекстным меню, и с горячими клавишами. Они будут привязаны к тем же объектам команд, что и кнопки, избавляя классы от дублирования.

Таким образом, команды станут гибкой прослойкой между пользовательским интерфейсом и бизнес-логикой. И это лишь малая доля пользы, которую может принести паттерн Команда!

Аналогия из жизни



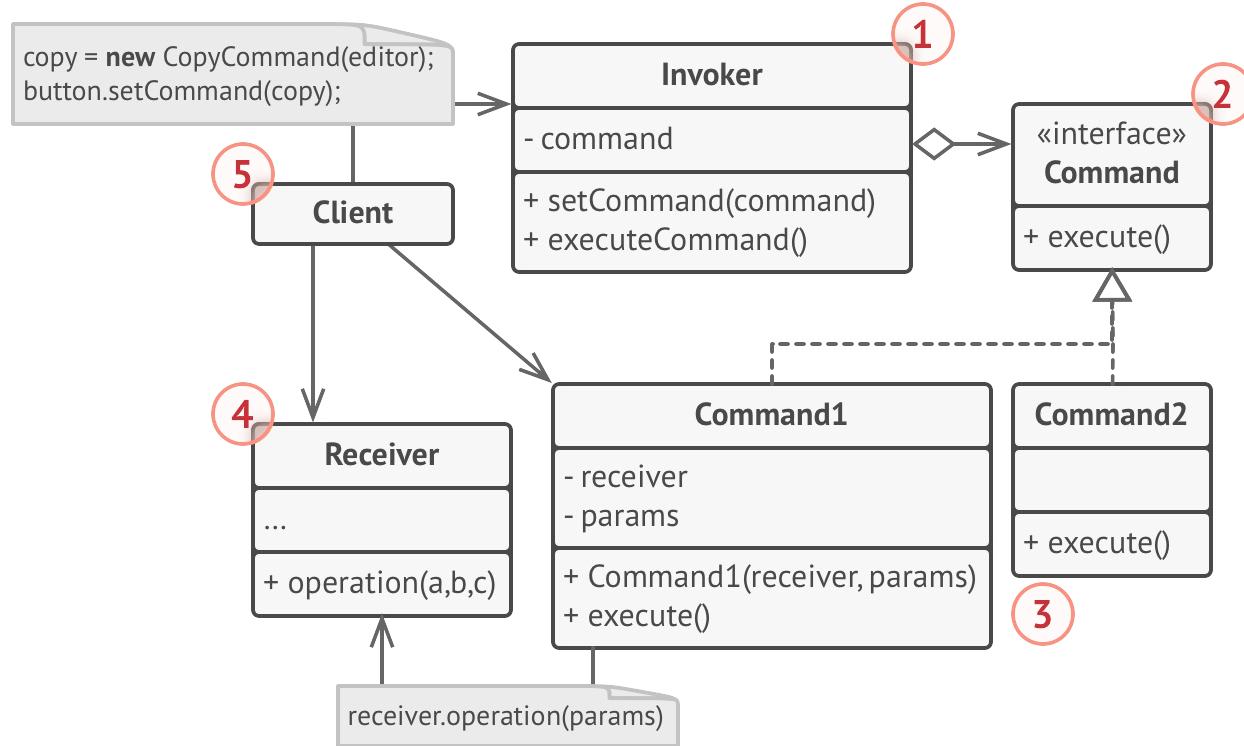
Пример заказа в ресторане.

Вы заходите в ресторан и садитесь у окна. К вам подходит вежливый официант и принимает заказ, записывая все пожелания в блокнот. Откланявшись, он уходит на кухню, где вырывает лист из блокнота и клеит на стену. Далее лист оказывается в руках повара, который читает содержание заказа и готовит заказанные блюда.

В этом примере вы являетесь *отправителем*, официант с блокнотом – *командой*, а повар – *получателем*. Как и в паттерне, вы не соприкасаетесь напрямую с поваром. Вместо этого вы отправляете заказ с официантом, который самостоятельно «настраивает» повара на

работу. С другой стороны, повар не знает, кто конкретно послал ему заказ. Но это ему безразлично, так как вся необходимая информация есть в листе заказа.

Структура



1. **Отправитель** хранит ссылку на объект команды и обращается к нему, когда нужно выполнить какое-то действие. Отправитель работает с командами только через их общий интерфейс. Он не знает, какую конкретно команду использует, так как получает готовый объект команды от клиента.
2. **Команда** описывает общий для всех конкретных команд интерфейс. Обычно здесь описан всего один метод для запуска команды.
3. **Конкретные команды** реализуют различные запросы, следуя общему интерфейсу команд. Обычно команда не делает всю работу самостоятельно, а лишь передаёт вызов получателю, которым является один из объектов бизнес-логики.

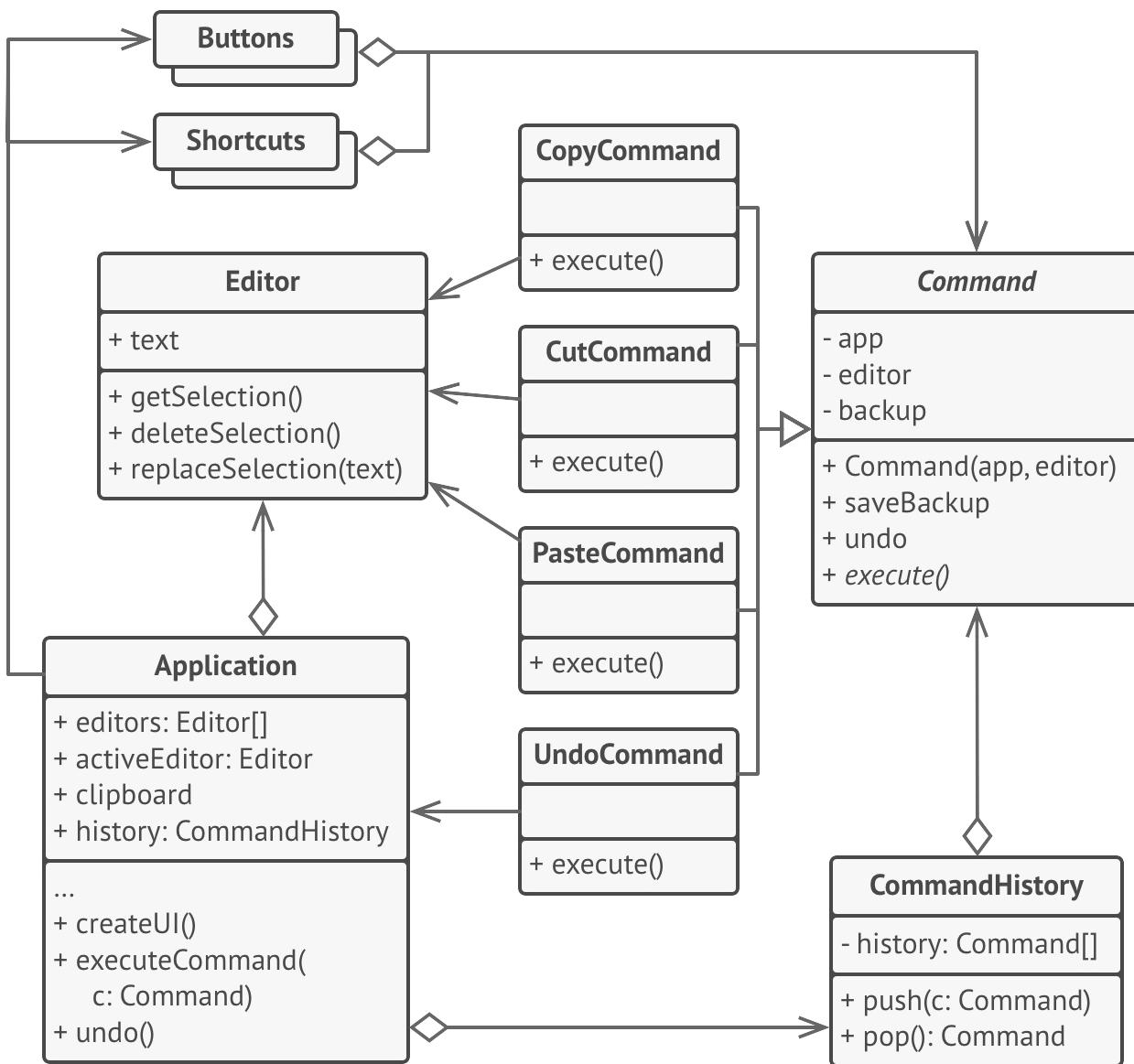
Параметры, с которыми команда обращается к получателю, следует хранить в виде полей. В большинстве случаев объекты команд можно сделать неизменяемыми,

передавая в них все необходимые параметры только через конструктор.

4. **Получатель** содержит бизнес-логику программы. В этой роли может выступать практически любой объект. Обычно команды перенаправляют вызовы получателям. Но иногда, чтобы упростить программу, вы можете избавиться от получателей, «слив» их код в классы команд.
5. **Клиент** создаёт объекты конкретных команд, передавая в них все необходимые параметры, среди которых могут быть и ссылки на объекты получателей. После этого клиент связывает объекты отправителей с созданными командами.

Псевдокод

В этом примере паттерн **Команда** служит для ведения истории выполненных операций, позволяя отменять их, если потребуется.



Пример реализации отмены в текстовом редакторе.

Команды, которые меняют состояние редактора (например, команда вставки текста из буфера обмена), сохраняют копию состояния редактора перед выполнением действия. Копии выполненных команд помещаются в историю команд, откуда они могут быть доставлены, если нужно будет сделать отмену операции.

Классы элементов интерфейса, истории команд и прочие не зависят от конкретных классов команд, так как работают с ними через общий интерфейс. Это позволяет добавлять в приложение новые команды, не изменяя существующий код.

```

1 // Абстрактная команда задаёт общий интерфейс для конкретных
2 // классов команд и содержит базовое поведение отмены операции.
3 abstract class Command is
4     protected field app: Application

```

```
5 protected field editor: Editor
6 protected field backup: text
7
8 constructor Command(app: Application, editor: Editor) is
9     this.app = app
10    this.editor = editor
11
12 // Сохраняем состояние редактора.
13 method saveBackup() is
14     backup = editor.text
15
16 // Восстанавливаем состояние редактора.
17 method undo() is
18     editor.text = backup
19
20 // Главный метод команды остаётся абстрактным, чтобы каждая
21 // конкретная команда определила его по-своему. Метод должен
22 // возвратить true или false в зависимости о того, изменила
23 // ли команда состояние редактора, а значит, нужно ли её
24 // сохранить в историю.
25 abstract method execute()
26
27
28 // Конкретные команды.
29 class CopyCommand extends Command is
30     // Команда копирования не записывается в историю, так как
31     // она не меняет состояние редактора.
32     method execute() is
33         app.clipboard = editor.getSelection()
34         return false
35
36 class CutCommand extends Command is
37     // Команды, меняющие состояние редактора, сохраняют
38     // состояние редактора перед своим действием и сигнализируют
39     // об изменении, возвращая true.
40     method execute() is
41         saveBackup()
42         app.clipboard = editor.getSelection()
43         editor.deleteSelection()
44         return true
45
```

```
46 class PasteCommand extends Command is
47     method execute() is
48         saveBackup()
49         editor.replaceSelection(app.clipboard)
50         return true
51
52 // Отмена – это тоже команда.
53 class UndoCommand extends Command is
54     method execute() is
55         app.undo()
56         return false
57
58
59 // Глобальная история команд – это стек.
60 class CommandHistory is
61     private field history: array of Command
62
63     // Последний зашедший...
64     method push(c: Command) is
65         // Добавить команду в конец массива-истории.
66
67     // ...выходит первым.
68     method pop():Command is
69         // Достать последнюю команду из массива-истории.
70
71
72 // Класс редактора содержит непосредственные операции над
73 // текстом. Он отыгрывает роль получателя – команды делегируют
74 // ему свои действия.
75 class Editor is
76     field text: string
77
78     method getSelection() is
79         // Вернуть выбранный текст.
80
81     method deleteSelection() is
82         // Удалить выбранный текст.
83
84     method replaceSelection(text) is
85         // Вставить текст из буфера обмена в текущей позиции.
86
```

```
87
88 // Класс приложения настраивает объекты для совместной работы.
89 // Он выступает в роли отправителя – создаёт команды, чтобы
90 // выполнить какие-то действия.
91 class Application is
92     field clipboard: string
93     field editors: array of Editors
94     field activeEditor: Editor
95     field history: CommandHistory
96
97     // Код, привязывающий команды к элементам интерфейса, может
98     // выглядеть примерно так.
99 method createUI() is
100    // ...
101    copy = function() {executeCommand(
102        new CopyCommand(this, activeEditor)) }
103    copyButton.setCommand(copy)
104    shortcuts.onKeyPress("Ctrl+C", copy)
105
106    cut = function() { executeCommand(
107        new CutCommand(this, activeEditor)) }
108    cutButton.setCommand(cut)
109    shortcuts.onKeyPress("Ctrl+X", cut)
110
111    paste = function() { executeCommand(
112        new PasteCommand(this, activeEditor)) }
113    pasteButton.setCommand(paste)
114    shortcuts.onKeyPress("Ctrl+V", paste)
115
116    undo = function() { executeCommand(
117        new UndoCommand(this, activeEditor)) }
118    undoButton.setCommand(undo)
119    shortcuts.onKeyPress("Ctrl+Z", undo)
120
121    // Запускаем команду и проверяем, надо ли добавить её в
122    // историю.
123 method executeCommand(command) is
124    if (command.execute())
125        history.push(command)
126
127    // Берём последнюю команду из истории и заставляем её все
```

```
128 // отменить. Мы не знаем конкретный тип команды, но это и не  
129 // важно, так как каждая команда знает, как отменить своё  
130 // действие.  
131 method undo() is  
132     command = history.pop()  
133     if (command != null)  
134         command.undo()
```

Применимость

Когда вы хотите параметризовать объекты выполняемым действием.

Команда превращает операции в объекты. А объекты можно передавать, хранить и взаимозаменять внутри других объектов.

Скажем, вы разрабатываете библиотеку графического меню и хотите, чтобы пользователи могли использовать меню в разных приложениях, не меняя каждый раз код ваших классов. Применив паттерн, пользователям не придётся изменять классы меню, вместо этого они будут конфигурировать объекты меню различными командами.

Когда вы хотите ставить операции в очередь, выполнять их по расписанию или передавать по сети.

Как и любые другие объекты, команды можно сериализовать, то есть превратить в строку, чтобы потом сохранить в файл или базу данных. Затем в любой удобный момент её можно достать обратно, снова превратить в объект команды и выполнить. Таким же образом команды можно передавать по сети, логировать или выполнять на удалённом сервере.

Когда вам нужна операция отмены.

Главная вещь, которая вам нужна, чтобы иметь возможность отмены операций, – это хранение истории. Среди многих способов, которыми можно это сделать, паттерн Команда является, пожалуй, самым популярным.

История команд выглядит как стек, в который попадают все выполненные объекты команд. Каждая команда перед выполнением операции сохраняет текущее состояние объекта, с которым она будет работать. После выполнения операции копия команды попадает в стек истории, все ещё неся в себе сохранённое состояние объекта. Если потребуется отмена, программа возьмёт последнюю команду из истории и возобновит сохранённое в ней состояние.

Этот способ имеет две особенности. Во-первых, точное состояние объектов не так-то просто сохранить, ведь часть его может быть приватным. Но с этим может помочь справиться паттерн **Снимок**.

Во-вторых, копии состояния могут занимать довольно много оперативной памяти. Поэтому иногда можно прибегнуть к альтернативной реализации, когда вместо восстановления старого состояния команда выполняет обратное действие. Недостаток этого способа в сложности (а иногда и невозможности) реализации обратного действия.

Шаги реализации

1. Создайте общий интерфейс команд и определите в нём метод запуска.
2. Один за другим создайте классы конкретных команд. В каждом классе должно быть поле для хранения ссылки на один или несколько объектов-получателей, которым команда будет перенаправлять основную работу.

Кроме этого, команда должна иметь поля для хранения параметров, которые нужны при вызове методов получателя. Значения всех этих полей команда должна получать через конструктор.

И, наконец, реализуйте основной метод команды, вызывая в нём те или иные методы получателя.

3. Добавьте в классы отправителей поля для хранения команд. Обычно объекты-отправители принимают готовые объекты команд извне – через конструктор либо

через сеттер поля команды.

4. Измените основной код отправителей так, чтобы они делегировали выполнение действия команде.

5. Порядок инициализации объектов должен выглядеть так:

- Создаём объекты получателей.
- Создаём объекты команд, связав их с получателями.
- Создаём объекты отправителей, связав их с командами.

Преимущества и недостатки

Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.

Позволяет реализовать простую отмену и повтор операций.

Позволяет реализовать отложенный запуск операций.

Позволяет собирать сложные команды из простых.

Реализует *принцип открытости/закрытости*.

Усложняет код программы из-за введения множества дополнительных классов.

Отношения с другими паттернами

- **Цепочка обязанностей, Команда, Посредник и Наблюдатель** показывают различные способы работы отправителей запросов с их получателями:

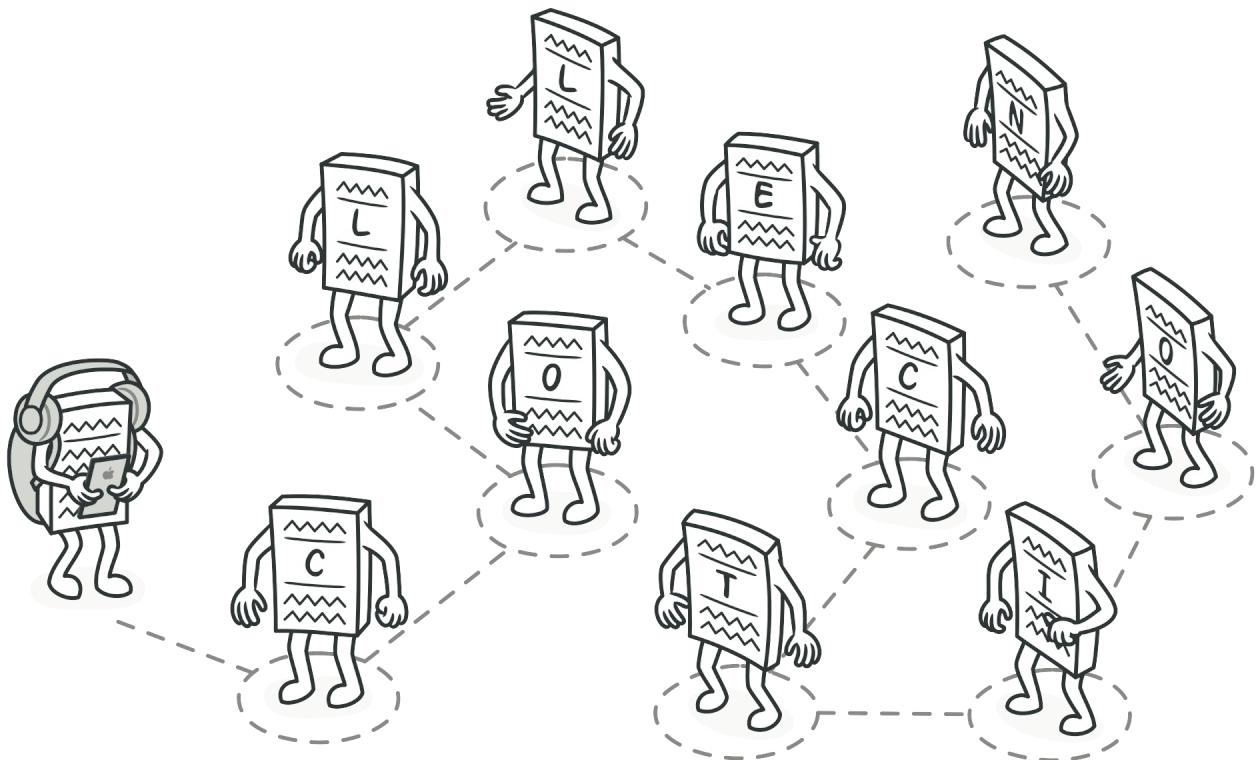
- *Цепочка обязанностей* передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.

- *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.
- 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.

- **Команду** и **Снимок** можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.
- **Команда** и **Стратегия** похожи по духу, но отличаются масштабом и применением:
 - *Команду* используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.
 - С другой стороны, *Стратегия* описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.
- Если **Команду** нужно копировать перед вставкой в историю выполненных команд, вам может помочь **Прототип**.

- **Посетитель** можно рассматривать как расширенный аналог **Команды**, который способен работать сразу с несколькими видами получателей.



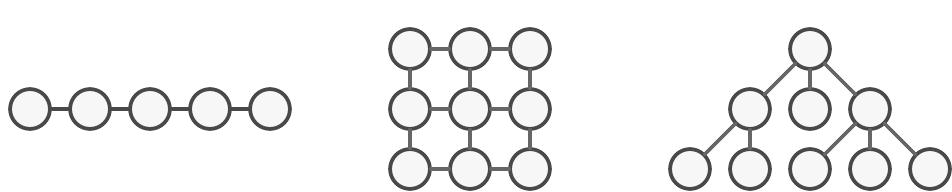
ИТЕРАТОР

Также известен как: *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.

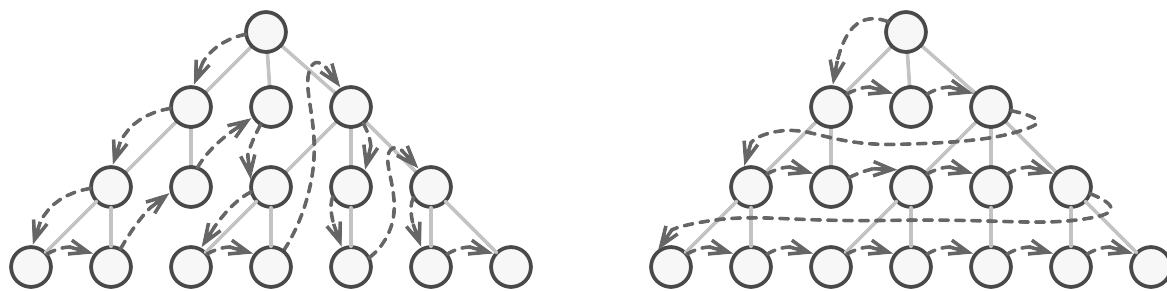


Разные типы коллекций.

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 деревя в глубину, но завтра потребуется возможность перемещаться по дереву в ширину. А на следующей неделе и того хуже – понадобится обход коллекции в случайном порядке.

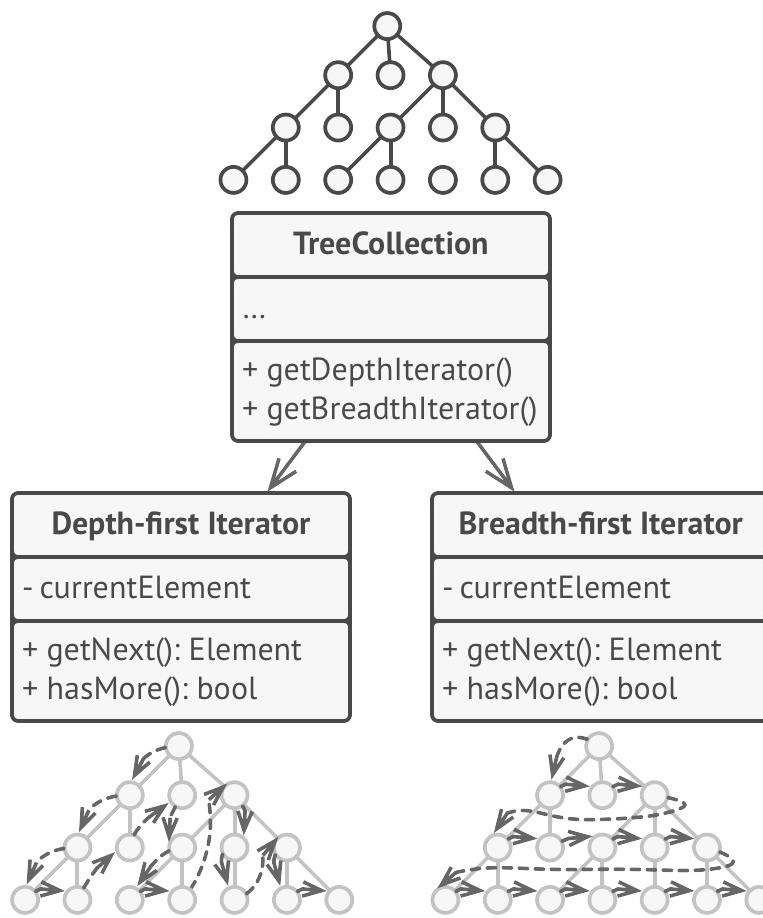


Одну и ту же коллекцию можно обходить разными способами.

Добавляя всё новые алгоритмы в код коллекции, вы понемногу размываете её основную функцию, 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.

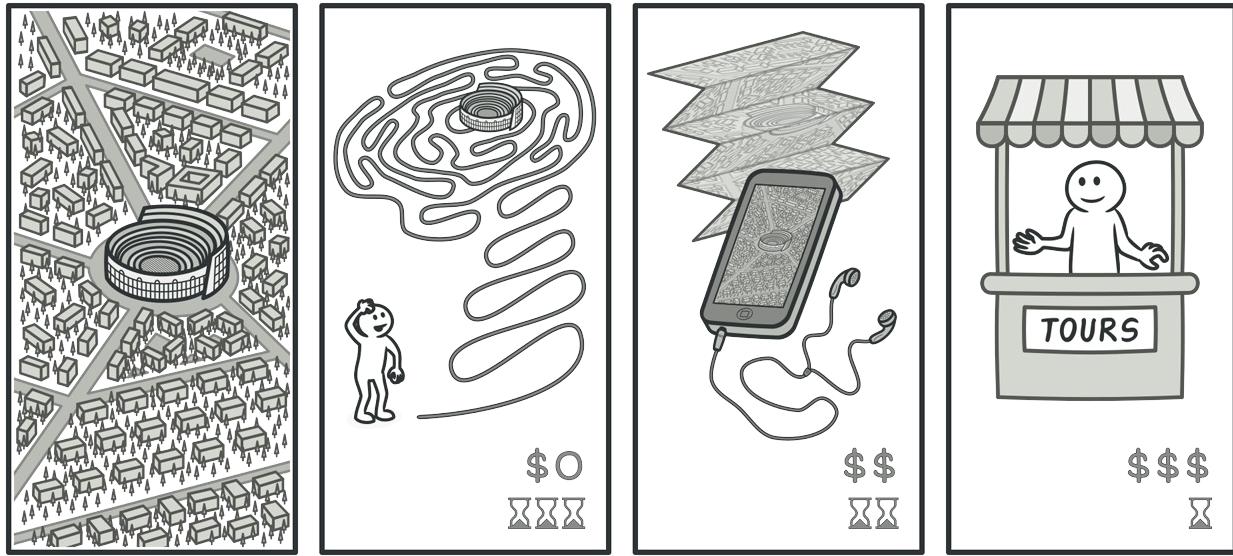


Итераторы содержат код обхода коллекции. Одну коллекцию могут обходить сразу несколько итераторов.

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.

Аналогия из жизни



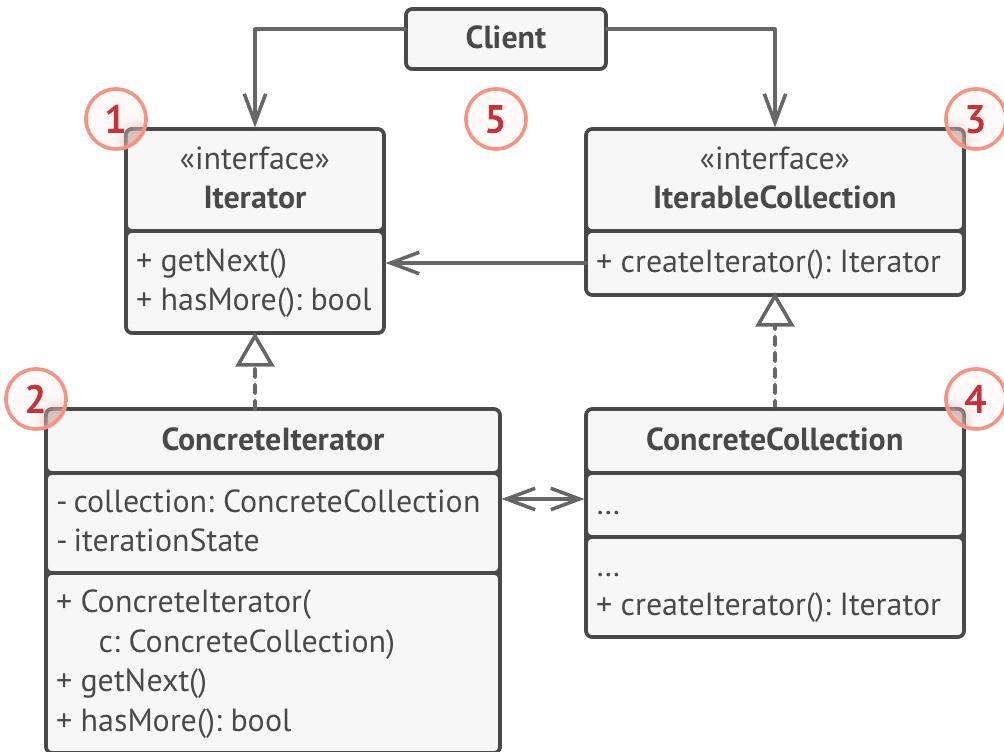
Варианты прогулок по Риму.

Вы планируете полететь в Рим и обойти все достопримечательности за пару дней. Но приехав, вы можете долго петлять узкими улочками, пытаясь найти Колизей.

Если у вас ограниченный бюджет – не беда. Вы можете воспользоваться виртуальным гидом, скачанным на телефон, который позволит отфильтровать только интересные вам точки. А можете плюнуть и нанять локального гида, который хоть и обойдётся в копеечку, но знает город как свои пять пальцев, и сможет посвятить вас во все городские легенды.

Таким образом, 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.

Структура

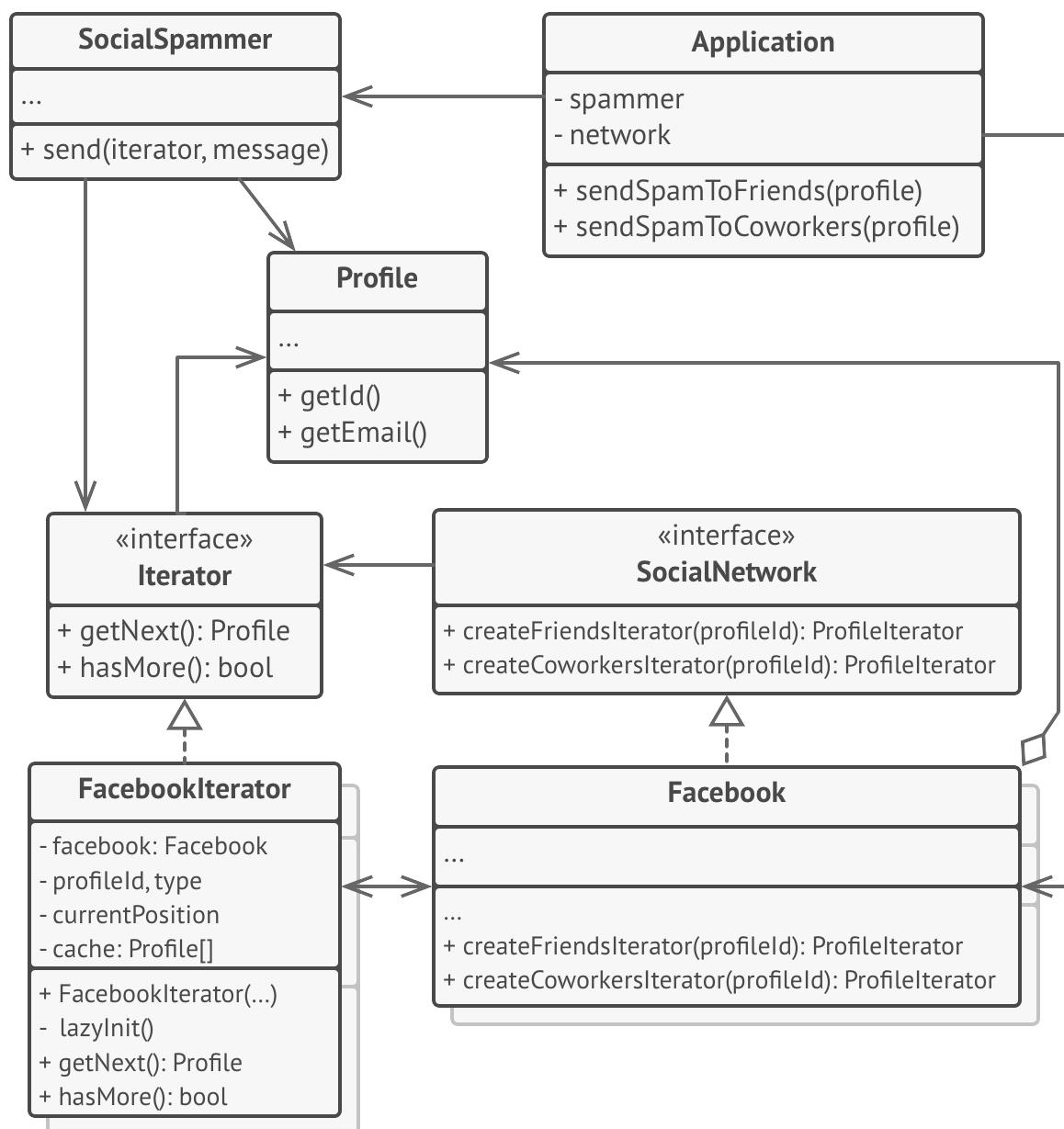


- Итератор** описывает интерфейс для доступа и обхода элементов коллекции.
- Конкретный итератор** реализует алгоритм обхода какой-то конкретной коллекции. Объект итератора должен сам отслеживать текущую позицию при обходе коллекции, чтобы отдельные итераторы могли обходить одну и ту же коллекцию независимо.
- Коллекция** описывает интерфейс получения итератора из коллекции. Как мы уже говорили, коллекции не всегда являются списком. Это может быть и база данных, и удалённое API, и даже дерево **Компоновщика**. Поэтому сама **коллекция** может создавать итераторы, так как она знает, какие именно итераторы способны с ней работать.
- Конкретная коллекция** возвращает новый экземпляр определённого конкретного итератора, связав его с текущим объектом коллекции. Обратите внимание, что сигнатура метода возвращает интерфейс итератора. Это позволяет клиенту не зависеть от конкретных классов итераторов.
- Клиент** работает со всеми объектами через интерфейсы коллекции и итератора. Так клиентский код не зависит от конкретных классов, что позволяет применять различные итераторы, не изменяя существующий код программы.

В общем случае клиенты не создают объекты итераторов, а получают их из коллекций. Тем не менее, если клиенту требуется специальный итератор, он всегда может создать его самостоятельно.

Псевдокод

В этом примере паттерн **Итератор** используется для реализации обхода нестандартной коллекции, которая инкапсулирует доступ к социальному графу Facebook. Коллекция предоставляет несколько итераторов, которые могут по-разному обходить профили людей.



Пример обхода социальных профилей через итератор.

Так, итератор друзей перебирает всех друзей профиля, а итератор коллег – фильтрует друзей по принадлежности к компании профиля. Все итераторы реализуют общий интерфейс, который позволяет клиентам работать с профилями, не вникая в детали работы с социальной сетью (например, в авторизацию, отправку REST-запросов и т. д.)

Кроме того, Итератор избавляет код от привязки к конкретным классам коллекций. Это позволяет добавить поддержку другого вида коллекций (например, LinkedIn), не меняя клиентский код, который работает с итераторами и коллекциями.

```
1 // Общий интерфейс коллекций должен определить фабричный метод
2 // для производства итератора. Можно определить сразу несколько
3 // методов, чтобы дать пользователям различные варианты обхода
4 // одной и той же коллекции.
5 interface SocialNetwork is
6     method createFriendsIterator(profileId): ProfileIterator
7     method createCoworkersIterator(profileId): ProfileIterator
8
9
10 // Конкретная коллекция знает, объекты каких итераторов нужно
11 // создавать.
12 class Facebook implements SocialNetwork is
13     // ...Основной код коллекции...
14
15     // Код получения нужного итератора.
16     method createFriendsIterator(profileId) is
17         return new FacebookIterator(this, profileId, "friends")
18     method createCoworkersIterator(profileId) is
19         return new FacebookIterator(this, profileId, "coworkers")
20
21
22 // Общий интерфейс итераторов.
23 interface ProfileIterator is
24     method getNext(): Profile
25     method hasMore(): bool
26
27
28 // Конкретный итератор.
29 class FacebookIterator implements ProfileIterator is
30     // Итератору нужна ссылка на коллекцию, которую он обходит.
```

```
31 private field facebook: Facebook
32 private field profileId, type: string
33
34 // Но каждый итератор обходит коллекцию, независимо от
35 // остальных, поэтому он содержит информацию о текущей
36 // позиции обхода.
37 private field currentPosition
38 private field cache: array of Profile
39
40 constructor FacebookIterator(facebook, profileId, type) is
41     this.facebook = network
42     this.profileId = profileId
43     this.type = type
44
45 private method lazyInit() is
46     if (cache == null)
47         cache = facebook.sendSophisticatedSocialGraphRequest(profileId, type)
48
49 // Итератор реализует методы базового интерфейса по-своему.
50 method getNext() is
51     if (hasMore())
52         currentPosition++
53         return cache[currentPosition]
54
55 method hasMore() is
56     lazyInit()
57     return cache.length < currentPosition
58
59
60 // Вот ещё полезная тактика: мы можем передавать объект
61 // итератора вместо коллекции в клиентские классы. При таком
62 // подходе клиентский код не будет иметь доступа к коллекциям, а
63 // значит, его не будут волновать подробности их реализаций. Ему
64 // будет доступен только общий интерфейс итераторов.
65 class SocialSpammer is
66     method send(iterator: ProfileIterator, message: string) is
67         while (iterator.hasNext())
68             profile = iterator.getNext()
69             System.sendEmail(profile.getEmail(), message)
70
71
```

```
72 // Класс приложение конфигурирует классы, как захочет.
73 class Application is
74     field network: SocialNetwork
75     field spammer: SocialSpammer
76
77     method config() is
78         if working with Facebook
79             this.network = new Facebook()
80         if working with LinkedIn
81             this.network = new LinkedIn()
82         this.spammer = new SocialSpammer()
83
84     method sendSpamToFriends(profile) is
85         iterator = network.createFriendsIterator(profile.getId())
86         spammer.send(iterator, "Very important message")
87
88     method sendSpamToCoworkers(profile) is
89         iterator = network.createCoworkersIterator(profile.getId())
90         spammer.send(iterator, "Very important message")
```

Применимость

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.

Когда вам 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.

Когда вам хочется иметь единый интерфейс обхода различных структур данных.

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

Шаги реализации

1. 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.
2. Создайте интерфейс коллекции и опишите в нём метод получения итератора. Важно, чтобы сигнатура метода возвращала общий интерфейс итераторов, а не один из конкретных итераторов.
3. 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.
4. 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.
5. В клиентском коде и в классах коллекций не должно остаться кода обхода элементов. Клиент должен получать новый итератор из объекта коллекции каждый раз, когда ему нужно перебрать её элементы.

Преимущества и недостатки

Упрощает классы хранения данных.

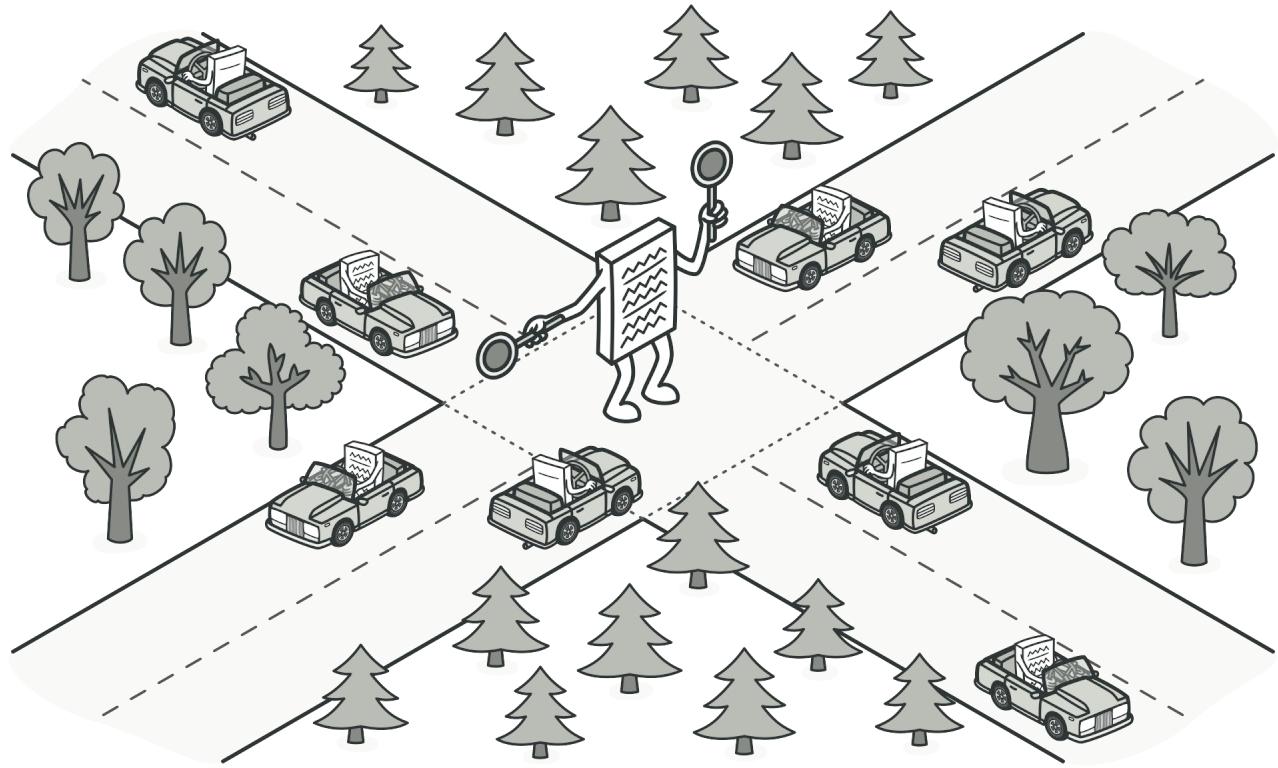
Позволяет реализовать различные способы обхода структуры данных.

Позволяет одновременно перемещаться по структуре данных в разные стороны.

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.



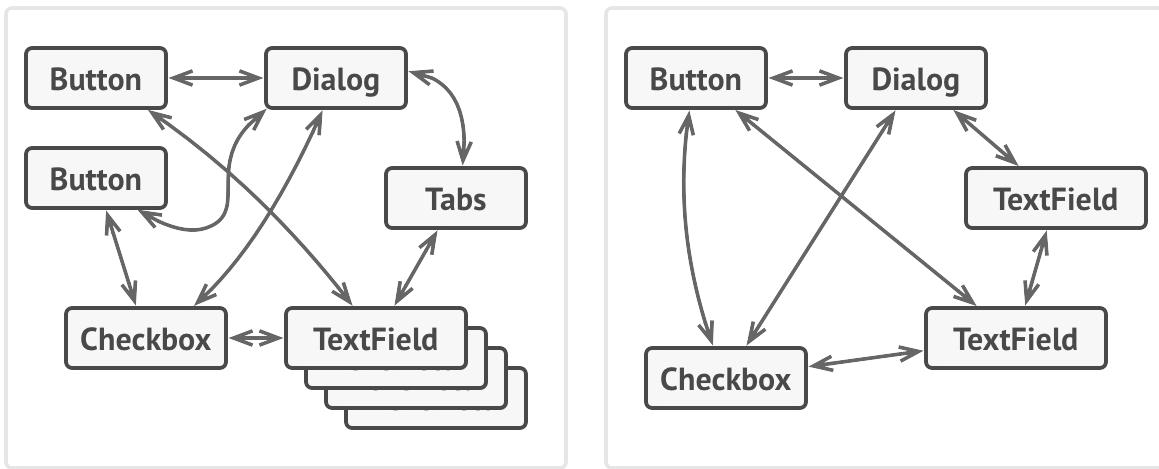
ПОСРЕДНИК

Также известен как: *Intermediary, Controller, Mediator*

Посредник – это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Проблема

Предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления – текстовых полей, чекбоксов, кнопок.



Беспорядочные связи между элементами пользовательского интерфейса.

Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть собака» открывает скрытое поле для ввода имени домашнего любимца, а клик по кнопке отправки запускает проверку значений всех полей формы.



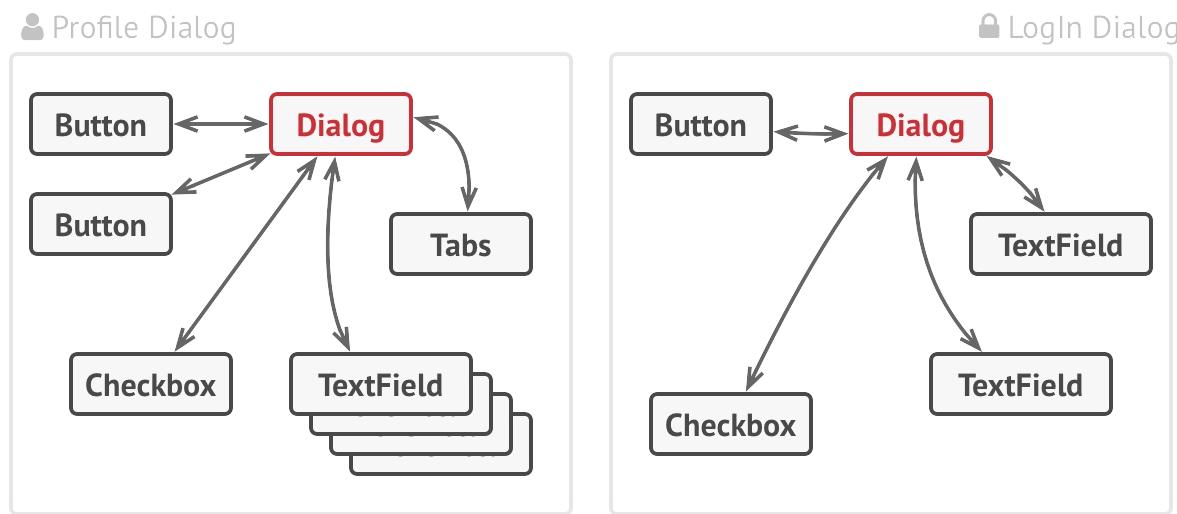
Код элементов нужно трогать при изменении каждого диалога.

Прописав эту логику прямо в коде элементов управления, вы поставите крест на их повторном использовании в других местах приложения. Они станут слишком тесно связанными с элементами диалога редактирования профиля, которые не нужны в других контекстах. Поэтому вы сможете использовать либо все элементы сразу, либо ни одного.

Решение

Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.

В нашем примере посредником мог бы стать диалог. Скорее всего, класс диалога и так знает, из каких элементов состоит, поэтому никаких новых связей добавлять в него не придётся.



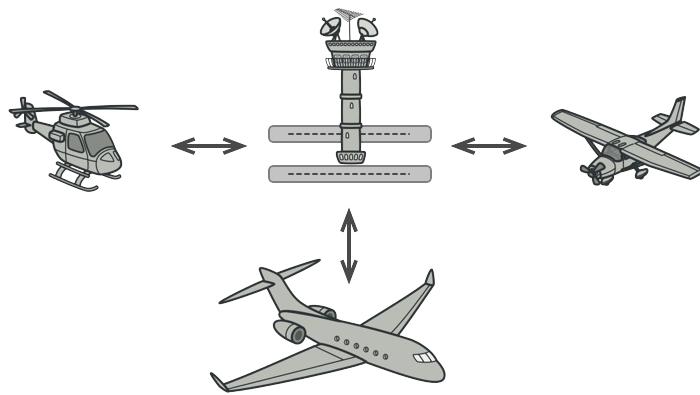
Элементы интерфейса общаются через посредника.

Основные изменения произойдут внутри отдельных элементов диалога. Если раньше при получении клика от пользователя объект кнопки сам проверял значения полей диалога, то теперь его единственной обязанностью будет сообщить диалогу о том, что произошёл клик. Получив извещение, диалог выполнит все необходимые проверки полей. Таким образом, вместо нескольких зависимостей от остальных элементов кнопка получит только одну — от самого диалога.

Чтобы сделать код ещё более гибким, можно выделить общий интерфейс для всех посредников, то есть диалогов программы. Наша кнопка станет зависимой не от конкретного диалога создания пользователя, а от абстрактного, что позволит использовать её и в других диалогах.

Таким образом, посредник скрывает в себе все сложные связи и зависимости между классами отдельных компонентов программы. А чем меньше связей имеют классы, тем проще их изменять, расширять и повторно использовать.

Аналогия из жизни

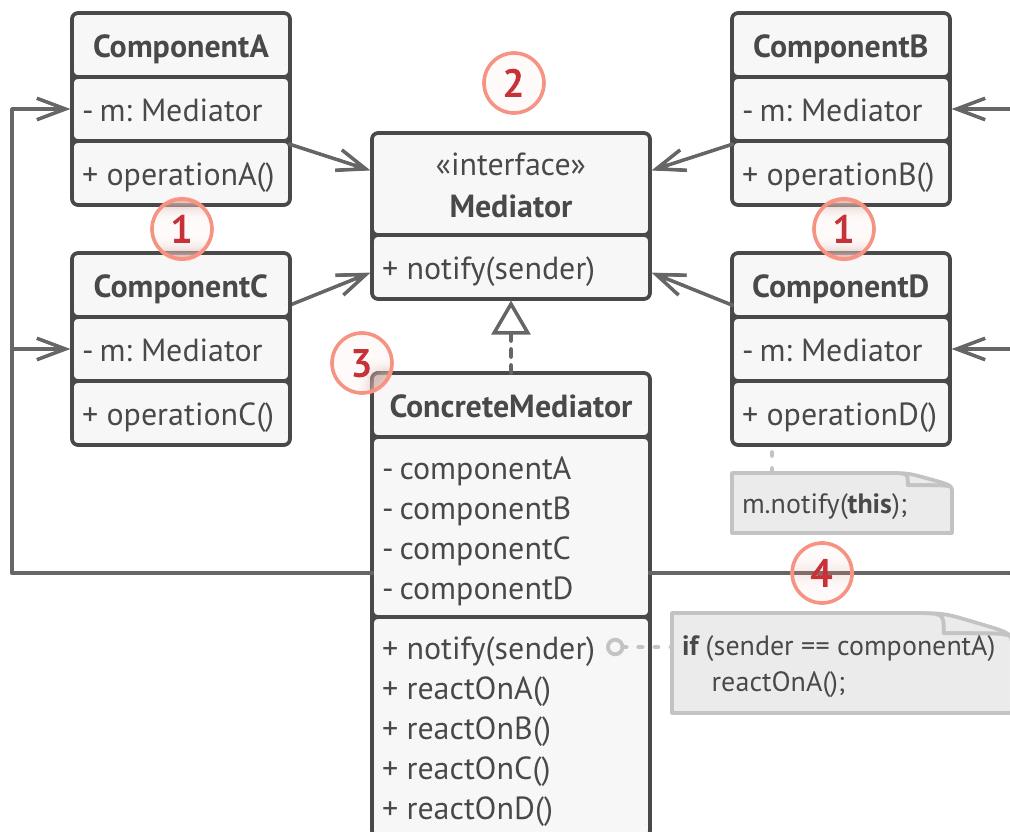


Пилоты самолётов общаются не напрямую, а через диспетчера.

Пилоты садящихся или улетающих самолётов не общаются напрямую с другими пилотами. Вместо этого они связываются с диспетчером, который координирует действия нескольких самолётов одновременно. Без диспетчера пилотам приходилось бы все время быть начеку и следить за всеми окружающими самолётами самостоятельно, а это приводило бы к частым катастрофам в небе.

Важно понимать, что диспетчер не нужен во время всего полёта. Он задействован только в зоне аэропорта, когда нужно координировать взаимодействие многих самолётов.

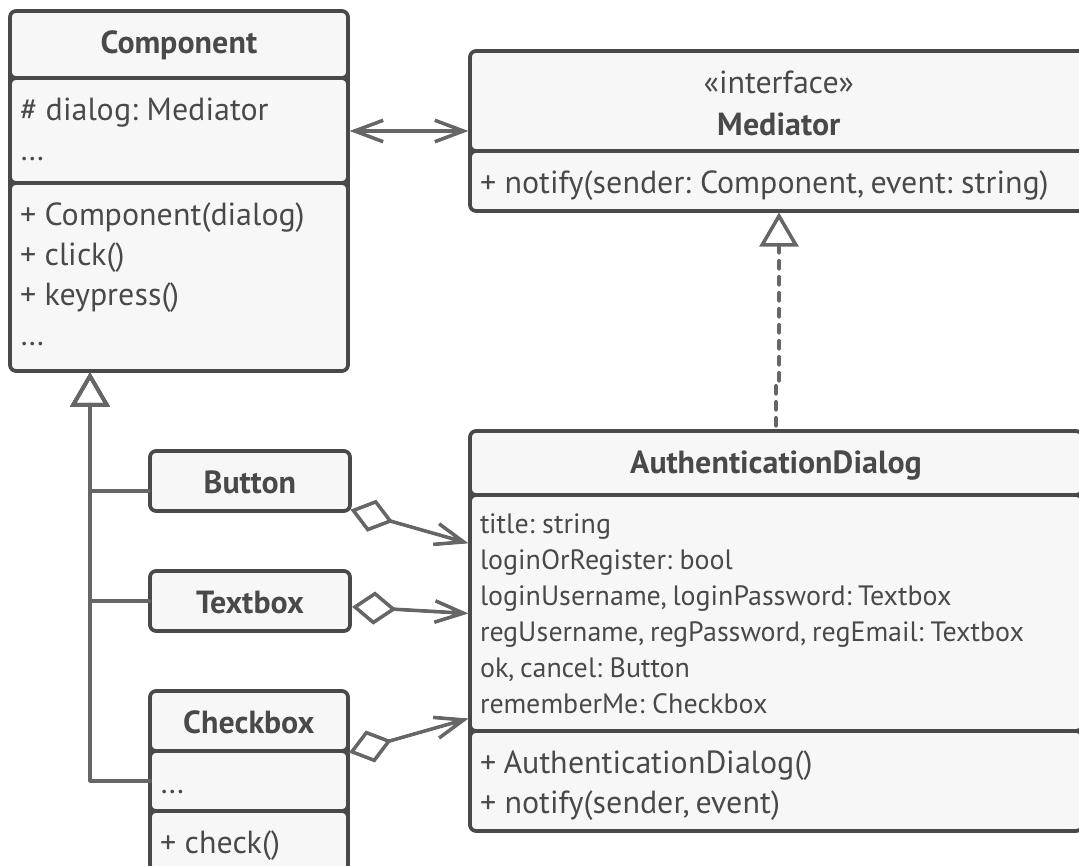
Структура



- Компоненты** – это разнородные объекты, содержащие бизнес-логику программы. Каждый компонент хранит ссылку на объект посредника, но работает с ним только через абстрактный интерфейс посредников. Благодаря этому, компоненты можно повторно использовать в другой программе, связав их с посредником другого типа.
- Посредник** определяет интерфейс для обмена информацией с компонентами. Обычно хватает одного метода, чтобы оповещать посредника о событиях, произошедших в компонентах. В параметрах этого метода можно передавать детали события: ссылку на компонент, в котором оно произошло, и любые другие данные.
- Конкретный посредник** содержит код взаимодействия нескольких компонентов между собой. Зачастую этот объект не только хранит ссылки на все свои компоненты, но и сам их создаёт, управляя дальнейшим жизненным циклом.
- Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, он должен оповестить своего посредника, а тот сам решит – касается ли событие других компонентов, и стоит ли их оповещать. При этом компонент-отправитель не знает, кто обработает его запрос, а компонент-получатель не знает, кто его прислал.

Псевдокод

В этом примере **Посредник** помогает избавиться от зависимостей между классами различных элементов пользовательского интерфейса: кнопками, чекбоксами и надписями.



Пример структурирования классов UI-диалогов.

По реакции на действия пользователей элементы не взаимодействуют напрямую, а всего лишь уведомляют посредника о том, что они изменились.

Посредник в виде диалога авторизации знает, как конкретные элементы должны взаимодействовать. Поэтому при получении уведомлений он может перенаправить вызов тому или иному элементу.

```
1 // Общий интерфейс посредников.
2 interface Mediator is
3     method notify(sender: Component, event: string)
4
5
6 // Конкретный посредник. Все связи между конкретными
```

```
7 // компонентами переехали в код посредника. Он получает
8 // извещения от своих компонентов и знает, как на них
9 // реагировать.
10 class AuthenticationDialog implements Mediator is
11     private field title: string
12     private field loginOrRegisterChkBx: Checkbox
13     private field loginUsername, loginPassword: Textbox
14     private field registrationUsername, registrationPassword
15     private field registrationEmail: Textbox
16     private field okBtn, cancelBtn: Button
17
18     constructor AuthenticationDialog() is
19         // Здесь нужно создать объекты всех компонентов, подав
20         // текущий объект-посредник в их конструктор.
21
22     // Когда что-то случается с компонентом, он шлёт посреднику
23     // оповещение. После получения извещения посредник может
24     // либо сделать что-то самостоятельно, либо перенаправить
25     // запрос другому компоненту.
26     method notify(sender, event) is
27         if (sender == loginOrRegisterChkBx and event == "check")
28             if (loginOrRegisterChkBx.checked)
29                 title = "Log in"
30                 // 1. Показать компоненты формы входа.
31                 // 2. Скрыть компоненты формы регистрации.
32             else
33                 title = "Register"
34                 // 1. Показать компоненты формы регистрации.
35                 // 2. Скрыть компоненты формы входа.
36
37         if (sender == okBtn && event == "click")
38             if (loginOrRegister.checked)
39                 // Попробовать найти пользователя с данными из
40                 // формы логина.
41                 if (!found)
42                     // Показать ошибку над формой логина.
43             else
44                 // 1. Создать пользовательский аккаунт с данными
45                 // из формы регистрации.
46                 // 2. Авторизировать этого пользователя.
47                 // ...
```

```
48
49
50 // Классы компонентов общаются с посредниками через их общий
51 // интерфейс. Благодаря этому одни и те же компоненты можно
52 // использовать в разных посредниках.
53 class Component is
54     field dialog: Mediator
55
56     constructor Component(dialog) is
57         this.dialog = dialog
58
59     method click() is
60         dialog.notify(this, "click")
61
62     method keypress() is
63         dialog.notify(this, "keypress")
64
65 // Конкретные компоненты не связаны между собой напрямую. У них
66 // есть только один канал общения – через отправку уведомлений
67 // посреднику.
68 class Button extends Component is
69     // ...
70
71 class Textbox extends Component is
72     // ...
73
74 class Checkbox extends Component is
75     method check() is
76         dialog.notify(this, "check")
77     // ...
```

Применимость

Когда вам сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими классами.

Посредник позволяет поместить все эти связи в один класс, после чего вам будет легче их отрефакторить, сделать более понятными и гибкими.

Когда вы не можете повторно использовать класс, поскольку он зависит от уймы других классов.

После применения паттерна компоненты теряют прежние связи с другими компонентами, а всё их общение происходит косвенно, через объект-посредник.

Когда вам приходится создавать множество подклассов компонентов, чтобы использовать одни и те же компоненты в разных контекстах.

Если раньше изменение отношений в одном компоненте могли повлечь за собой лавину изменений во всех остальных компонентах, то теперь вам достаточно создать подкласс посредника и поменять в нём связи между компонентами.

Шаги реализации

1. Найдите группу тесно переплетённых классов, отвязав которые друг от друга, можно получить некоторую пользу. Например, чтобы повторно использовать их код в другой программе.
2. Создайте общий интерфейс посредников и опишите в нём методы для взаимодействия с компонентами. В простейшем случае достаточно одного метода для получения оповещений от компонентов.

Этот интерфейс необходим, если вы хотите повторно использовать классы компонентов для других задач. В этом случае всё, что нужно сделать – это создать новый класс конкретного посредника.

3. Реализуйте этот интерфейс в классе конкретного посредника. Поместите в него поля, которые будут содержать ссылки на все объекты компонентов.
4. Вы можете пойти дальше и переместить код создания компонентов в класс конкретного посредника, превратив его в фабрику.

5. Компоненты тоже должны иметь ссылку на объект посредника. Связь между ними удобнее всего установить, подавая посредника в параметры конструктора компонентов.
6. Измените код компонентов так, чтобы они вызывали метод оповещения посредника, вместо методов других компонентов. С противоположной стороны, посредник должен вызывать методы нужного компонента, когда получает оповещение от компонента.

Преимущества и недостатки

Устраняет зависимости между компонентами, позволяя повторно их использовать.

Упрощает взаимодействие между компонентами.

Централизует управление в одном месте.

Посредник может **сильно раздуться**.

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.

- Посредник и Фасад похожи тем, что пытаются организовать работу множества существующих классов.
 - *Фасад* создаёт упрощённый интерфейс к подсистеме, не внося в неё никакой добавочной функциональности. Сама подсистема не знает о существовании *Фасада*. Классы подсистемы общаются друг с другом напрямую.
 - *Посредник* централизует общение между компонентами системы. Компоненты системы знают только о существовании *Посредника*, у них нет прямого доступа к другим компонентам.
- Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

Цель *Посредника* – убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель *Наблюдателя* – обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

Довольно популярна реализация *Посредника* при помощи *Наблюдателя*. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.

Но *Посредник* имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать *Наблюдателя*, но всё же останется *Посредником*.

Напротив, в случае реализации посредника с помощью *Наблюдателя* представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети *Наблюдателей*, не имея центрального объекта-*Посредника*.



СНИМОК

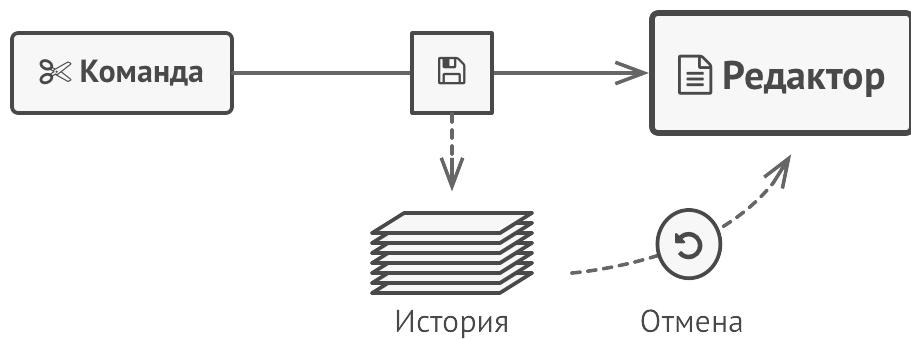
Также известен как: *Хранитель, Memento*

Снимок – это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

Проблема

Предположим, что вы пишете программу текстового редактора. Помимо обычного редактирования, ваш редактор позволяет менять форматирование текста, вставлять картинки и прочее.

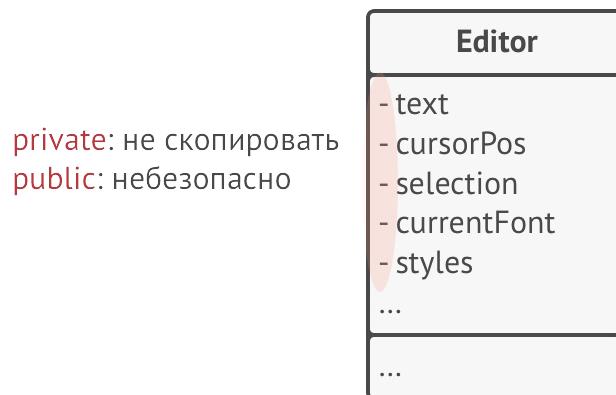
В какой-то момент вы решили сделать все эти действия отменяемыми. Для этого вам нужно сохранять текущее состояние редактора перед тем, как выполнить любое действие. Если потом пользователь решит отменить своё действие, вы достанете копию состояния из истории и восстановите старое состояние редактора.



Перед выполнением команды вы можете сохранить копию состояния редактора, чтобы потом иметь возможность отменить операцию.

Чтобы сделать копию состояния объекта, достаточно скопировать значение его полей. Таким образом, если вы сделали класс редактора достаточно открытым, то любой другой класс сможет заглянуть внутрь, чтобы скопировать его состояние.

Казалось бы, что ещё нужно? Ведь теперь любая операция сможет сделать резервную копию редактора перед своим действием. Но такой наивный подход обеспечит вам уйму проблем в будущем. Ведь если вы решите провести рефакторинг – убрать или добавить парочку полей в класс редактора – то придётся менять код всех классов, которые могли копировать состояние редактора.



Как команде создать снимок состояния редактора, если все его поля приватные?

Но это ещё не все. Давайте теперь рассмотрим сами копии состояния редактора. Из чего состоит состояние редактора? Даже самый примитивный редактор должен иметь несколько полей для хранения текущего текста, позиции курсора и прокрутки экрана. Чтобы сделать копию состояния, вам нужно записать значения всех этих полей в некий «контейнер».

Скорее всего, вам понадобится хранить массу таких контейнеров в качестве истории операций, поэтому удобнее всего сделать их объектами одного класса. Этот класс должен иметь много полей, но практически никаких методов. Чтобы другие объекты могли записывать и читать из него данные, вам придётся сделать его поля публичными. Но это приведёт к той же проблеме, что и с открытым классом редактора. Другие классы станут зависимыми от любых изменений в классе контейнера, который подвержен тем же изменениям, что и класс редактора.

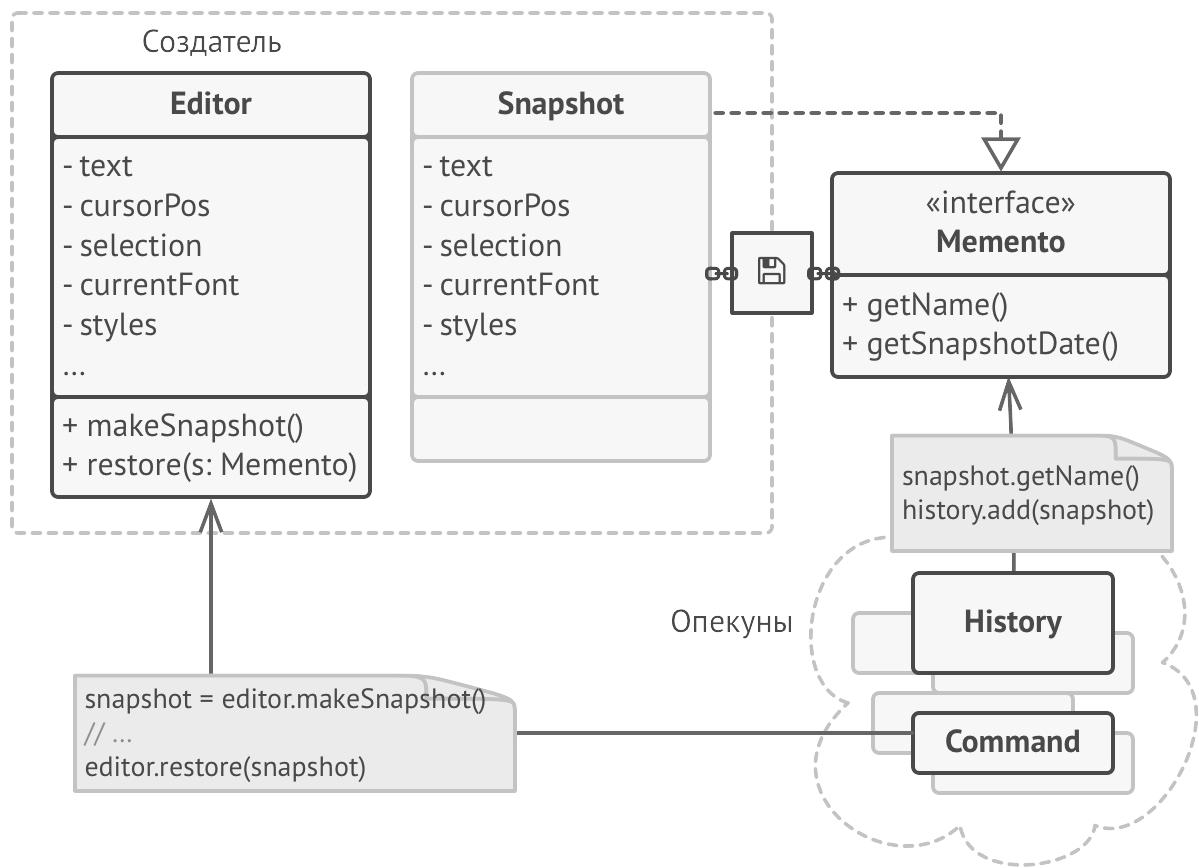
Получается, нам придётся либо открыть классы для всех желающих, испытывая массу хлопот с поддержкой кода, либо оставить классы закрытыми, отказавшись от идеи отмены операций. Нет ли какого-то другого пути?

Решение

Все проблемы, описанные выше, возникают из-за нарушения инкапсуляции. Это когда одни объекты пытаются сделать работу за других, влезая в их приватную зону, чтобы собрать необходимые для операции данные.

Паттерн Снимок поручает создание копии состояния объекта самому объекту, который этим состоянием владеет. Вместо того, чтобы делать снимок «извне», наш редактор сам сделает копию своих полей, ведь ему доступны все поля, даже приватные.

Паттерн предлагает держать копию состояния в специальном объекте-снимке с ограниченным интерфейсом, позволяющим, например, узнать дату изготовления или название снимка. Но, с другой стороны, снимок должен быть открыт для своего *создателя*, позволяя прочесть и восстановить его внутреннее состояние.



Снимок полностью открыт для создателя, но лишь частично открыт для опекунов.

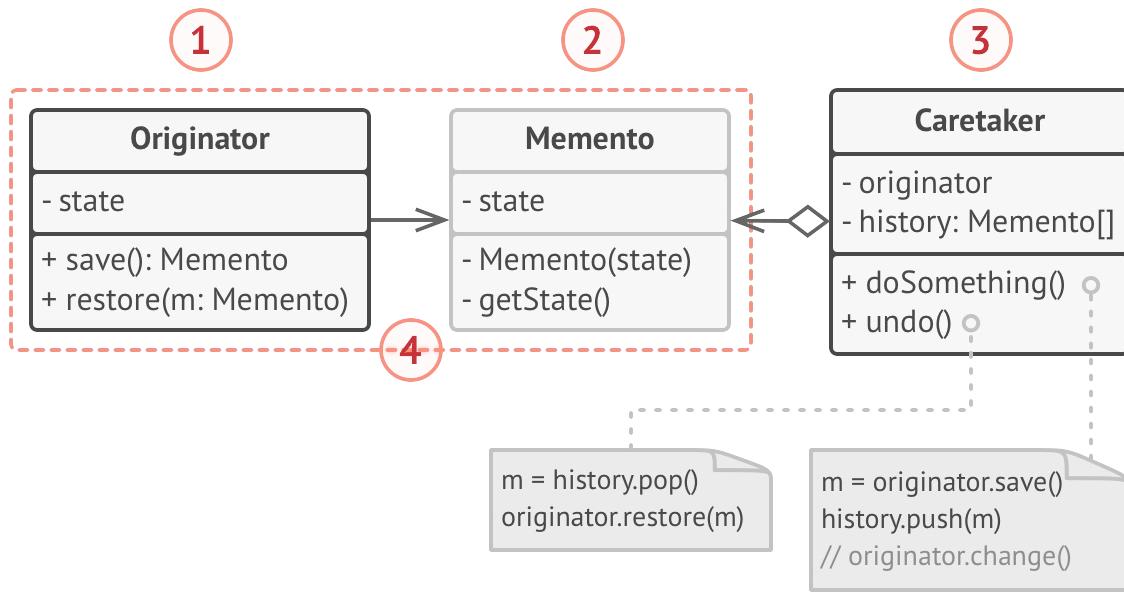
Такая схема позволяет создателям производить снимки и отдавать их для хранения другим объектам, называемым *опекунами*. Опекунам будет доступен только ограниченный интерфейс снимка, поэтому они никак не смогут повлиять на «внутренности» самого снимка. В нужный момент опекун может попросить создателя восстановить своё состояние, передав ему соответствующий снимок.

В примере с редактором вы можете сделать опекуном отдельный класс, который будет хранить список выполненных операций. Ограниченный интерфейс снимков позволит демонстрировать пользователю красивый список с названиями и датами выполненных операций. А когда пользователь решит откатить операцию, класс истории возьмёт последний снимок из стека и отправит его объекту редактор для восстановления.

Структура

Классическая реализация на вложенных классах

Классическая реализация паттерна полагается на механизм вложенных классов, который доступен лишь в некоторых языках программирования (C++, C#, Java).



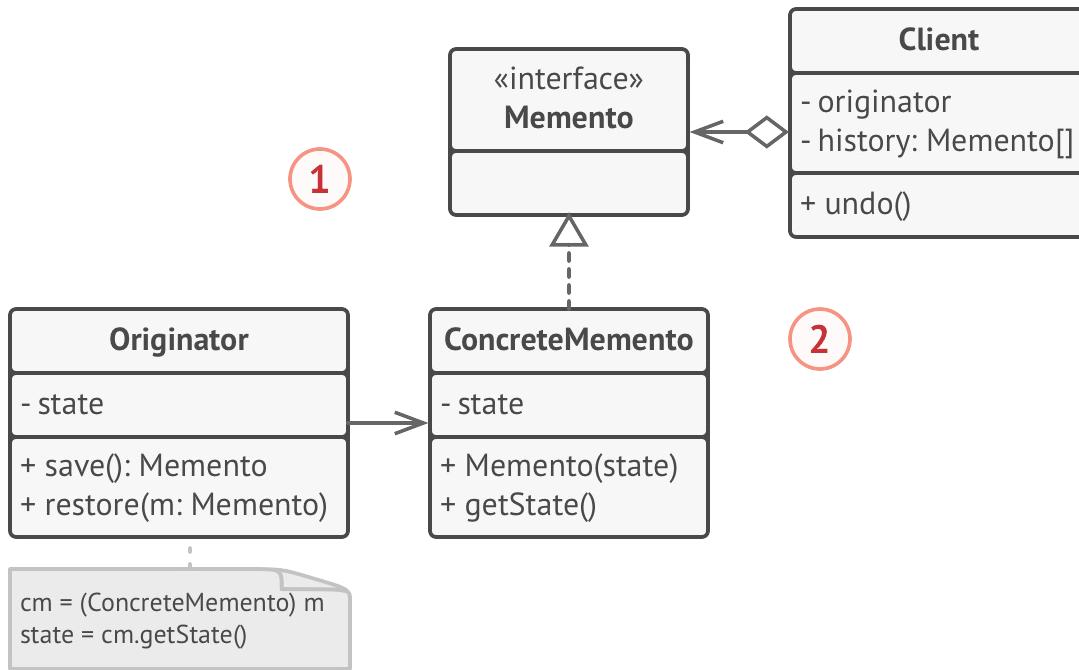
1. **Создатель** может производить снимки своего состояния, а также воспроизводить прошлое состояние, если подать в него готовый снимок.
2. **Снимок** – это простой объект данных, содержащий состояние создателя. Надёжнее всего сделать объекты снимков неизменяемыми, передавая в них состояние только через конструктор.
3. **Опекун** должен знать, когда делать снимок создателя и когда его нужно восстанавливать.

Опекун может хранить историю прошлых состояний создателя в виде стека из снимков. Когда понадобится отменить выполненную операцию, он возьмёт «верхний» снимок из стека и передаст его создателю для восстановления.

4. В данной реализации снимок – это внутренний класс по отношению к классу создателя. Именно поэтому он имеет полный доступ к полям и методам создателя, даже приватным. С другой стороны, опекун не имеет доступа ни к состоянию, ни к методам снимков и может всего лишь хранить ссылки на эти объекты.

Реализация с пустым промежуточным интерфейсом

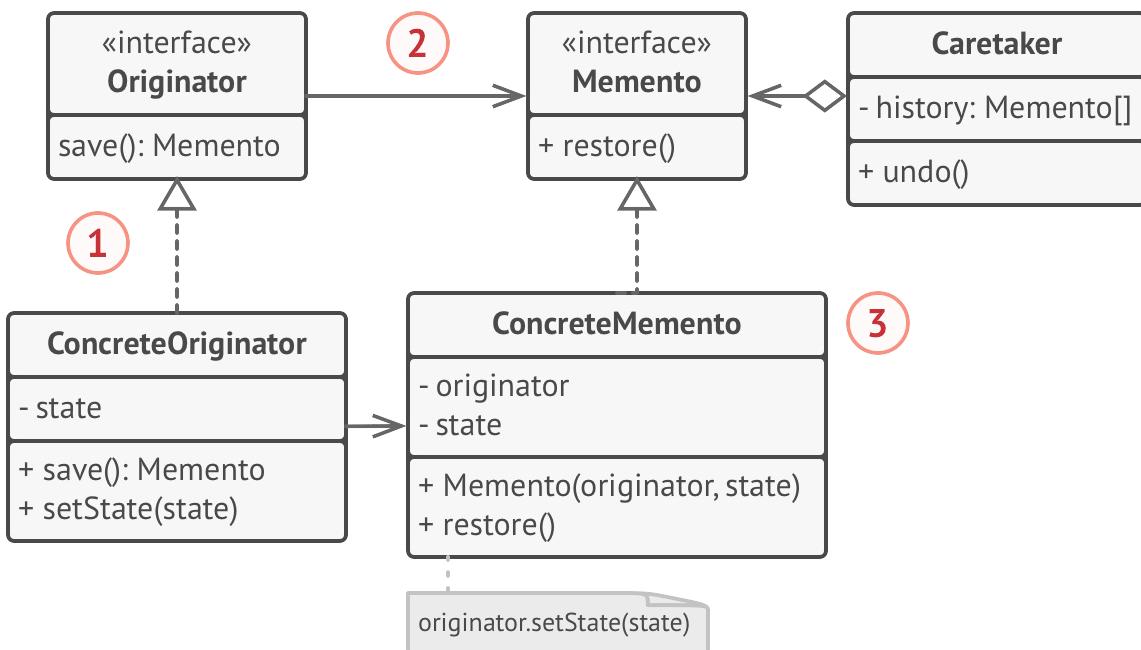
Подходит для языков, не имеющих механизма вложенных классов (например, PHP).



1. В этой реализации создатель работает напрямую с конкретным классом снимка, а опекун – только с его ограниченным интерфейсом.
2. Благодаря этому достигается тот же эффект, что и в классической реализации. Создатель имеет полный доступ к снимку, а опекун – нет.

Снимки с повышенной защитой

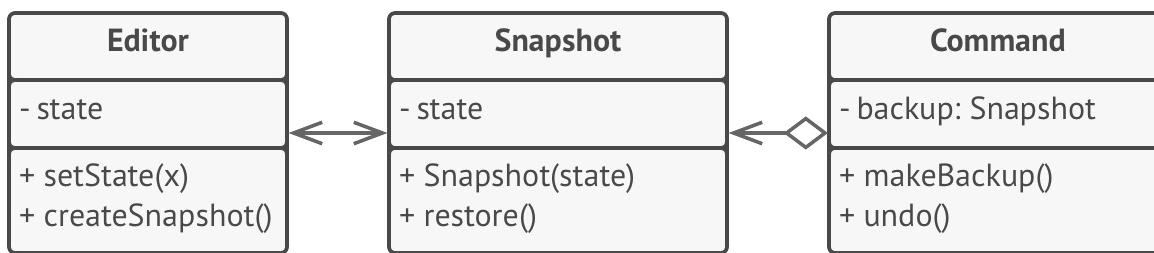
Когда нужно полностью исключить возможность доступа к состоянию создателей и снимков.



1. Эта реализация разрешает иметь несколько видов создателей и снимков. Каждому классу создателей соответствует свой класс снимков. Ни создатели, ни снимки не позволяют другим объектам прочесть своё состояние.
2. Здесь опекун ещё более жёстко ограничен в доступе к состоянию создателей и снимков. Но, с другой стороны, опекун становится независим от создателей, поскольку метод восстановления теперь находится в самих снимках.
3. Снимки теперь связаны с теми создателями, из которых они сделаны. Они по-прежнему получают состояние через конструктор. Благодаря близкой связи между классами, снимки знают, как восстановить состояние своих создателей.

Псевдокод

В этом примере паттерн **Снимок** используется совместно с паттерном **Команда** и позволяет хранить резервные копии сложного состояния текстового редактора и восстанавливать его, если потребуется.



Пример сохранения снимков состояния текстового редактора.

Объекты команд выступают в роли опекунов и запрашивают снимки у редактора перед тем, как выполнить своё действие. Если потребуется отменить операцию, команда сможет восстановить состояние редактора, используя сохранённый снимок.

При этом снимок не имеет публичных полей, поэтому другие объекты не имеют доступа к его внутренним данным. Снимки связаны с определённым редактором, который их создал. Они же и восстанавливают состояние своего редактора. Это позволяет программе иметь одновременно несколько объектов редакторов, например, разбитых по разным вкладкам программы.

```

1 // Класс создателя должен иметь специальный метод, который
2 // сохраняет состояние создателя в новом объекте-снимке.
3 class Editor is
4     private field text, curX, curY, selectionWidth
5
6     method setText(text) is
7         this.text = text
8
9     method setCursor(x, y) is
10        this.curX = curX
11        this.curY = curY
12
13    method setSelectionWidth(width) is
14        this.selectionWidth = width
15
16    method createSnapshot(): Snapshot is
17        // Снимок – неизменяемый объект, поэтому Создатель
18        // передаёт все своё состояние через параметры
19        // конструктора.
20        return new Snapshot(this, text, curX, curY, selectionWidth)
21

```

```
22 // Снимок хранит прошлое состояние редактора.
23 class Snapshot is
24     private field editor: Editor
25     private field text, curX, curY, selectionWidth
26
27     constructor Snapshot(editor, text, curX, curY, selectionWidth) is
28         this.editor = editor
29         this.text = text
30         this.curX = curX
31         this.curY = curY
32         this.selectionWidth = selectionWidth
33
34     // В нужный момент владелец снимка может восстановить
35     // состояние редактора.
36     method restore() is
37         editor.setText(text)
38         editor.setCursor(curX, curY)
39         editor.setSelectionWidth(selectionWidth)
40
41 // Опекуном может выступать класс команд (см. паттерн Команда).
42 // В этом случае команда сохраняет снимок состояния объекта-
43 // получателя, перед тем как передать ему своё действие. А в
44 // случае отмены команда вернёт объект в прежнее состояние.
45 class Command is
46     private field backup: Snapshot
47
48     method makeBackup() is
49         backup = editor.createSnapshot()
50
51     method undo() is
52         if (backup != null)
53             backup.restore()
54     // ...
```

Применимость

Когда вам нужно сохранять мгновенные снимки состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии.

Паттерн Снимок позволяет создавать любое количество снимков объекта и хранить их, независимо от объекта, с которого делают снимок. Снимки часто используют не только для реализации операции отмены, но и для транзакций, когда состояние объекта нужно «откатить», если операция не удалась.

Когда прямое получение состояния объекта раскрывает приватные детали его реализации, нарушая инкапсуляцию.

Паттерн предлагает изготовить снимок самому исходному объекту, поскольку ему доступны все поля, даже приватные.

Шаги реализации

1. Определите класс создателя, объекты которого должны создавать снимки своего состояния.
2. Создайте класс снимка и опишите в нём все те же поля, которые имеются в оригинальном классе-создателе.
3. Сделайте объекты снимков неизменяемыми. Они должны получать начальные значения только один раз, через свой конструктор.
4. Если ваш язык программирования это позволяет, сделайте класс снимка вложенным в класс создателя. Если нет, извлеките из класса снимка пустой интерфейс, который будет доступен остальным объектам программы. Впоследствии вы можете добавить в этот интерфейс некоторые вспомогательные методы, дающие доступ к метаданным снимка, однако прямой доступ к данным создателя должен быть исключён.
5. Добавьте в класс создателя метод получения снимков. Создатель должен создавать новые объекты снимков, передавая значения своих полей через конструктор.

Сигнатура метода должна возвращать снимки через ограниченный интерфейс, если он у вас есть. Сам класс должен работать с конкретным классом снимка.

6. Добавьте в класс создателя метод восстановления из снимка. Что касается привязки к типам, руководствуйтесь той же логикой, что и в пункте 4.
7. Опекуны, будь то история операций, объекты команд или нечто иное, должны знать о том, когда запрашивать снимки у создателя, где их хранить и когда восстанавливать.
8. Связь опекунов с создателями можно перенести внутрь снимков. В этом случае каждый снимок будет привязан к своему создателю и должен будет сам восстанавливать его состояние. Но это будет работать либо если классы снимков вложены в классы создателей, либо если создатели имеют соответствующие сеттеры для установки значений своих полей.

Преимущества и недостатки

Не нарушает инкапсуляции исходного объекта.

Упрощает структуру исходного объекта. Ему не нужно хранить историю версий своего состояния.

Требует много памяти, если клиенты слишком часто создают снимки.

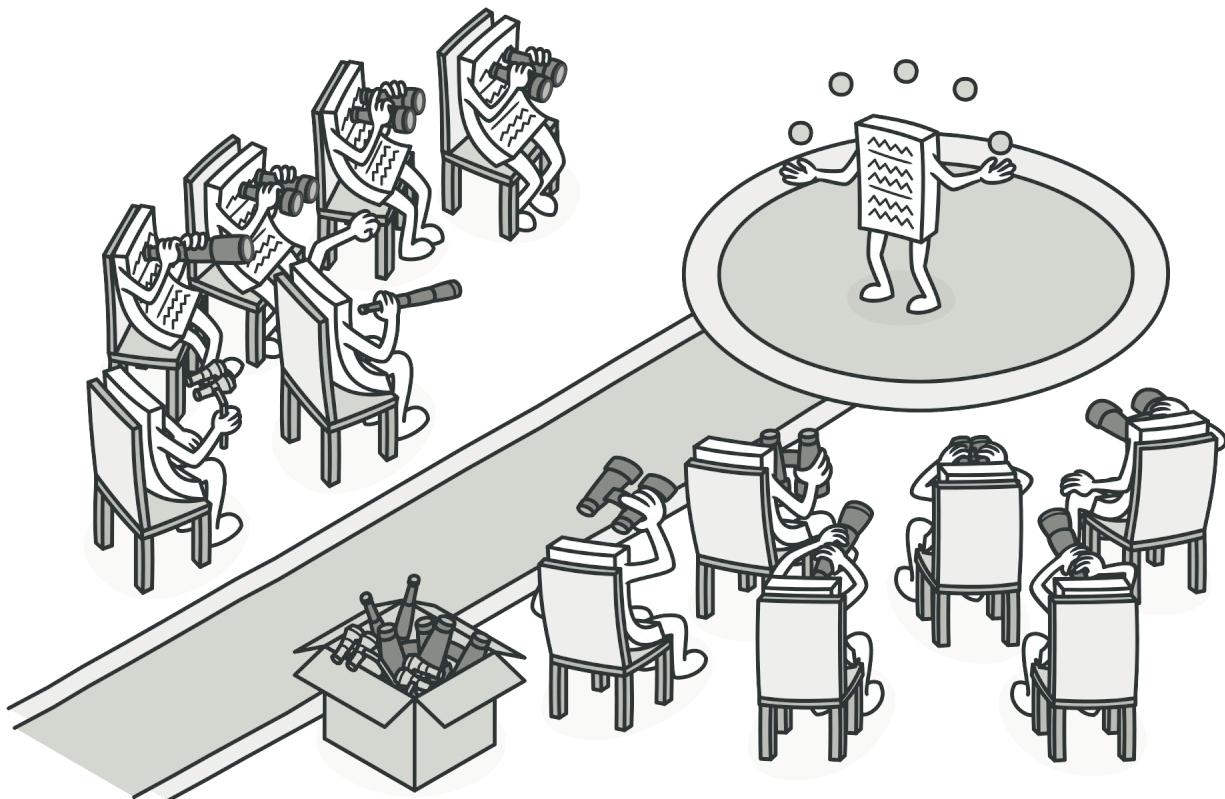
Может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не освобождают ресурсы, занятые устаревшими снимками.

В некоторых языках (например, PHP, Python, JavaScript) сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка.

Отношения с другими паттернами

- **Команду** и **Снимок** можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.

- **Снимок** можно использовать вместе с **Итератором**, чтобы сохранить текущее состояние обхода структуры данных и вернуться к нему в будущем, если потребуется.
- **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.



НАБЛЮДАТЕЛЬ

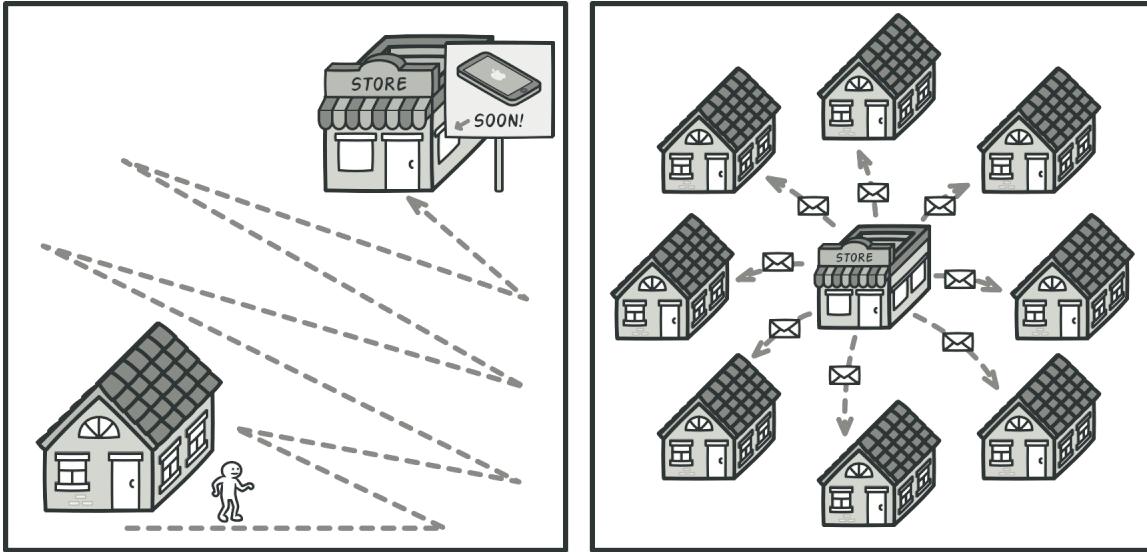
Также известен как: *Издатель-Подписчик, Слушатель, Observer*

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

Проблема

Представьте, что вы имеете два объекта: **Покупатель** и **Магазин**. В магазин вот-вот должны завезти новый товар, который интересен покупателю.

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.



Постоянное посещение магазина или спам?

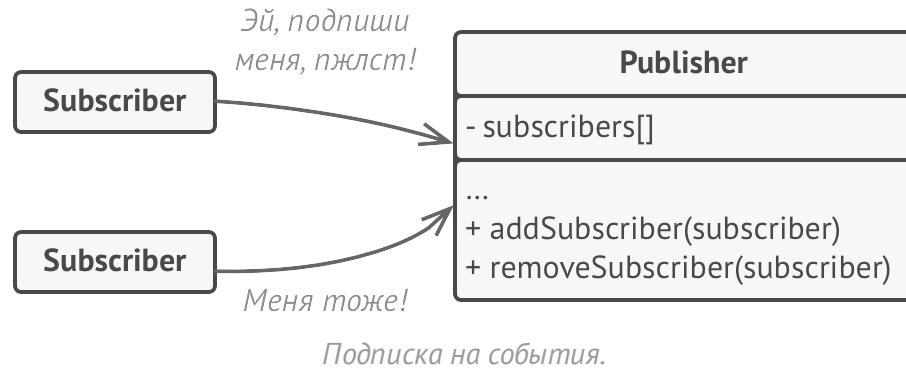
С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

Решение

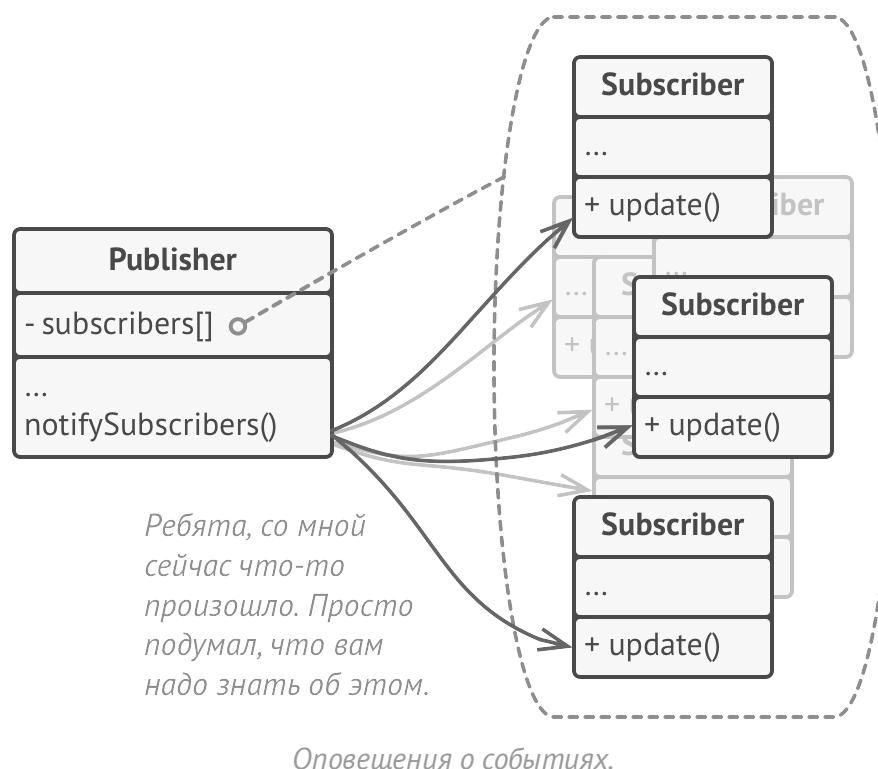
Давайте называть **Издателями** те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём **Подписчиками**.

Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.



Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходить по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.



Увидев, как складно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.

Аналогия из жизни

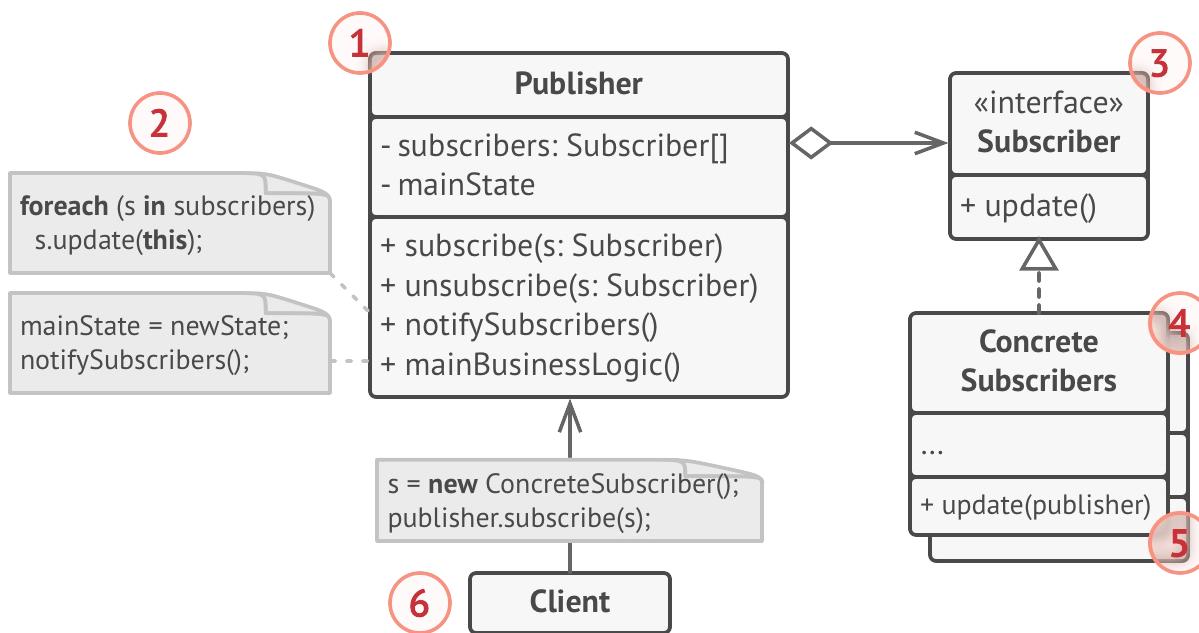


Подписка на газеты и их доставка.

После того как вы оформили подписку на газету или журнал, вам больше не нужно ездить в супермаркет и проверять, не вышел ли очередной номер. Вместо этого издательство будет присыпать новые номера по почте прямо к вам домой сразу после их выхода.

Издательство ведёт список подписчиков и знает, кому какой журнал высылать. Вы можете в любой момент отказаться от подписки, и журнал перестанет вам приходить.

Структура

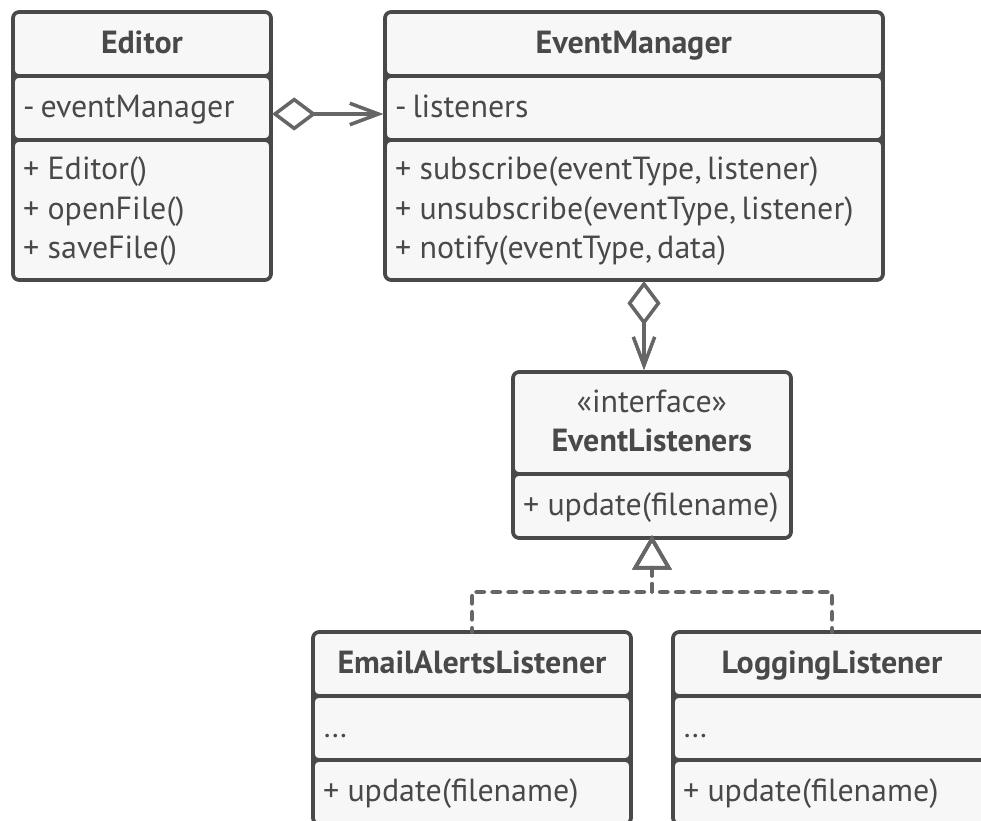


1. **Издатель** владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам. Издатель содержит механизм подписки: список подписчиков и методы подписки/отписки.
2. Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков. Для этого издатель проходит по списку подписчиков и вызывает их метод оповещения, заданный в общем интерфейсе подписчиков.
3. **Подписчик** определяет интерфейс, которым пользуется издатель для отправки оповещения. В большинстве случаев для этого достаточно единственного метода.
4. **Конкретные подписчики** выполняют что-то в ответ на оповещение, пришедшее от издателя. Эти классы должны следовать общему интерфейсу подписчиков, чтобы издатель не зависел от конкретных классов подписчиков.
5. По приходу оповещения подписчику нужно получить обновлённое состояние издателя. Издатель может передать это состояние через параметры метода оповещения. Более гибкий вариант – передавать через параметры весь объект издателя, чтобы подписчик мог сам получить требуемые данные. Как вариант, подписчик может постоянно хранить ссылку на объект издателя, переданный ему в конструкторе.

6. **Клиент** создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.

Псевдокод

В этом примере **Наблюдатель** позволяет объекту текстового редактора оповещать другие объекты об изменениях своего состояния.



Пример оповещения объектов о событиях в других объектах.

Список подписчиков составляется динамически, объекты могут, как подписываться на определённые события, так и отписываться от них прямо во время выполнения программы.

В этой реализации редактор не ведёт список подписчиков самостоятельно, а делегирует это вложенному объекту. Это даёт возможность использовать механизм подписки не только в классе редактора, но и в других классах программы.

Для добавления в программу новых подписчиков не нужно менять классы издателей, пока они работают с подписчиками через общий интерфейс.

```
1 // Базовый класс-издатель. Содержит код управления подписчиками
2 // и их оповещения.
3 class EventManager is
4     private field listeners: hash map of event types and listeners
5
6     method subscribe(eventType, listener) is
7         listeners.add(eventType, listener)
8
9     method unsubscribe(eventType, listener) is
10        listeners.remove(eventType, listener)
11
12    method notify(eventType, data) is
13        foreach (listener in listeners.of(eventType)) do
14            listener.update(data)
15
16 // Конкретный класс-издатель, содержащий интересную для других
17 // компонентов бизнес-логику. Мы могли бы сделать его прямым
18 // потомком EventManager, но в реальной жизни это не всегда
19 // возможно (например, если у класса уже есть родитель). Поэтому
20 // здесь мы подключаем механизм подписки при помощи композиции.
21 class Editor is
22     private field events: EventManager
23     private field file: File
24
25     constructor Editor() is
26         events = new EventManager()
27
28     // Методы бизнес-логики, которые оповещают подписчиков об
29     // изменениях.
30     method openFile(path) is
31         this.file = new File(path)
32         events.notify("open", file.name)
33
34     method saveFile() is
35         file.write()
36         events.notify("save", file.name)
37     // ...
```

```
38
39
40 // Общий интерфейс подписчиков. Во многих языках, поддерживающих
41 // функциональные типы, можно обойтись без этого интерфейса и
42 // конкретных классов, заменив объекты подписчиков функциями.
43 interface EventListener is
44     method update(filename)
45
46 // Набор конкретных подписчиков. Они реализуют добавочную
47 // функциональность, реагируя на извещения от издателя.
48 class LoggingListener is
49     private field log: File
50     private field message
51
52 constructor LoggingListener(log_filename, message) is
53     this.log = new File(log_filename)
54     this.message = message
55
56 method update(filename) is
57     log.write(replace('%s',filename,message))
58
59 class EmailAlertsListener is
60     private field email: string
61
62 constructor EmailAlertsListener(email, message) is
63     this.email = email
64     this.message = message
65
66 method update(filename) is
67     system.email(email, replace('%s',filename,message))
68
69
70 // Приложение может сконфигурировать издателей и подписчиков как
71 // угодно, в зависимости от целей и окружения.
72 class Application is
73     method config() is
74         editor = new TextEditor()
75
76         logger = new LoggingListener(
77             "/path/to/log.txt",
78             "Someone has opened file: %s");
```

```
79     editor.events.subscribe("open", logger)
80
81     emailAlers = new EmailAlertsListener(
82         "admin@example.com",
83         "Someone has changed the file: %s")
84     editor.events.subscribe("save", emailAlers)
```

Применимость

Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.

Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.

Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.

Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.

Шаги реализации

1. Разбейте вашу функциональность на две части: независимое ядро и optionalные зависимые части. Независимое ядро станет издателем. Зависимые части станут подписчиками.

2. Создайте интерфейс подписчиков. Обычно в нём достаточно определить единственный метод оповещения.
3. Создайте интерфейс издателей и опишите в нём операции управления подпиской. Помните, что издатель должен работать только с общим интерфейсом подписчиков.
4. Вам нужно решить, куда поместить код ведения подписки, ведь он обычно бывает одинаков для всех типов издателей. Самый очевидный способ – вынести этот код в промежуточный абстрактный класс, от которого будут наследоваться все издатели.

Но если вы интегрируете паттерн в существующие классы, то создать новый базовый класс может быть затруднительно. В этом случае вы можете поместить логику подписки во вспомогательный объект и делегировать ему работу из издателей.

5. Создайте классы конкретных издателей. Реализуйте их так, чтобы после каждого изменения состояния они отправляли оповещения всем своим подписчикам.
6. Реализуйте метод оповещения в конкретных подписчиках. Не забудьте предусмотреть параметры, через которые издатель мог бы отправлять какие-то данные, связанные с произошедшим событием.

Возможен и другой вариант, когда подписчик, получив оповещение, сам возьмёт из объекта издателя нужные данные. Но в этом случае вы будете вынуждены привязать класс подписчика к конкретному классу издателя.

7. Клиент должен создавать необходимое количество объектов подписчиков и подписывать их у издателей.

Преимущества и недостатки

Издатели не зависят от конкретных классов подписчиков и наоборот.

Вы можете подписывать и отписывать получателей на лету.

Реализует принцип открытости/закрытости.

Подписчики оповещаются в случайном порядке.

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.
- Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

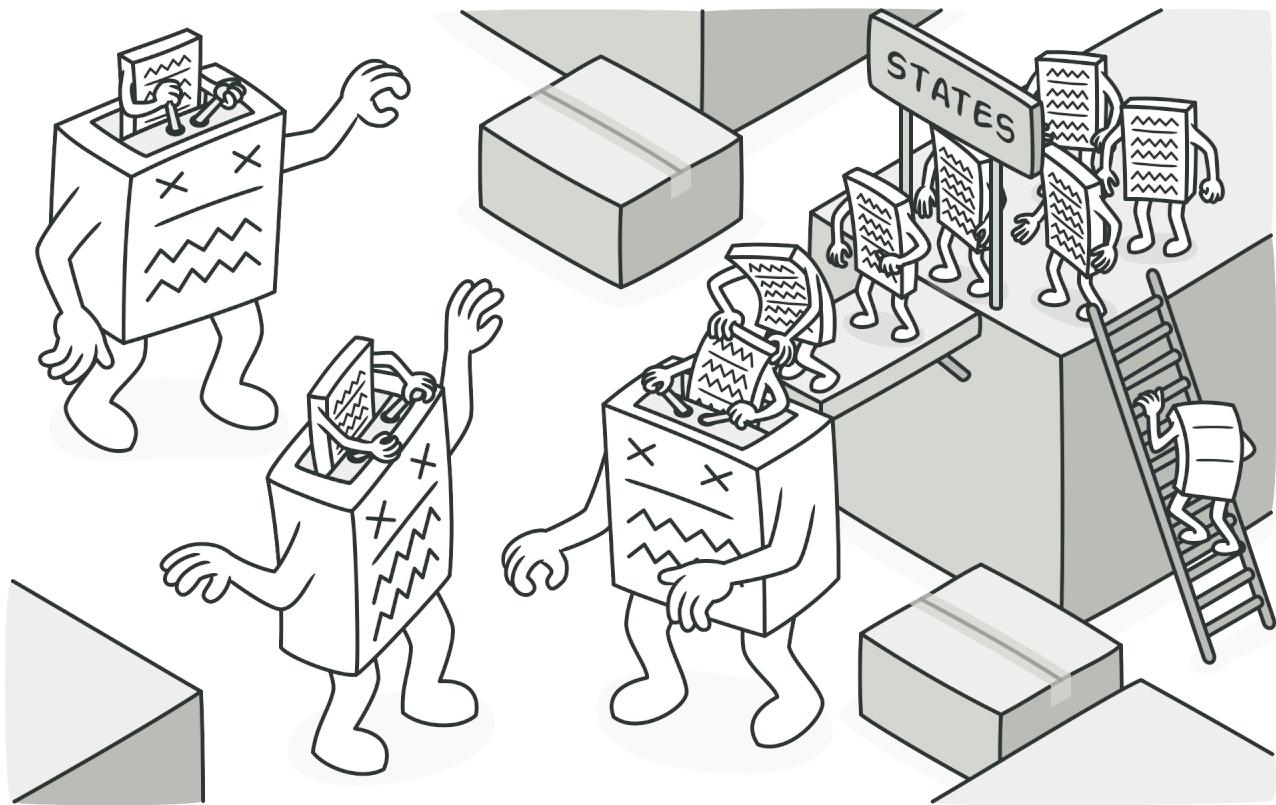
Цель *Посредника* – убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель *Наблюдателя* – обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

Довольно популярна реализация *Посредника* при помощи *Наблюдателя*. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.

Но *Посредник* имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать *Наблюдателя*,

но всё же останется *Посредником*.

Напротив, в случае реализации посредника с помощью *Наблюдателя* представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети *Наблюдателей*, не имея центрального объекта-*Посредника*.



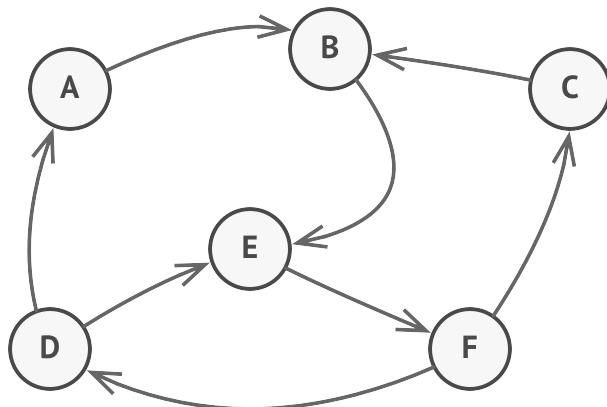
СОСТОЯНИЕ

Также известен как: *State*

Состояние – это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.

Проблема

Паттерн Состояние невозможно рассматривать в отрыве от концепции **машины состояний**, также известной как *стейт-машина* или *конечный автомат*.

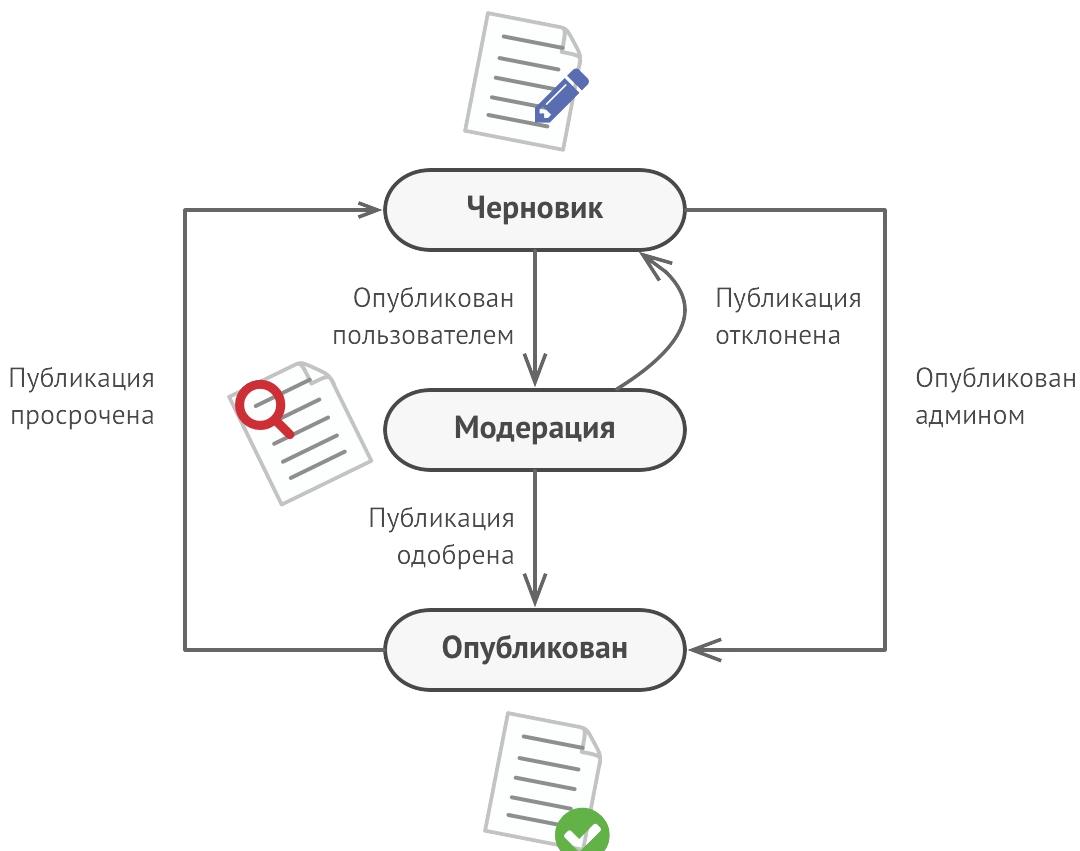


Конечный автомат.

Основная идея в том, что программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и *конечен*. Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.

Такой подход можно применить и к отдельным объектам. Например, объект `Документ` может принимать три состояния: `Черновик`, `Модерация` или `Опубликован`. В каждом из этих состояний метод `опубликовать` будет работать по-разному:

- Из черновика он отправит документ на модерацию.
- Из модерации – в публикацию, но при условии, что это сделал администратор.
- В опубликованном состоянии метод не будет делать ничего.



Возможные состояния документа и переходы между ними.

Машину состояний чаще всего реализуют с помощью множества условных операторов, `if` либо `switch`, которые проверяют текущее состояние объекта и выполняют соответствующее поведение. Наверняка вы уже реализовали хотя бы одну машину состояний в своей жизни, даже не зная об этом. Как насчёт вот такого кода, выглядит знакомо?

```
1 class Document
2     string state;
3     // ...
4     method publish() {
5         switch (state) {
6             "draft":
7                 state = "moderation";
8                 break;
9             "moderation":
10                if (currentUser.role == 'admin')
11                    state = "published"
12                    break;
13            "published":
14                // Do nothing.
15                break;
16        }
17    }
18    // ...
```

Основная проблема такой машины состояний проявится в том случае, если в [Документ](#) добавить ещё десяток состояний. Каждый метод будет состоять из увесистого условного оператора, перебирающего доступные состояния. Такой код крайне сложно поддерживать. Малейшее изменение логики переходов заставит вас перепроверять работу всех методов, которые содержат условные операторы машины состояний.

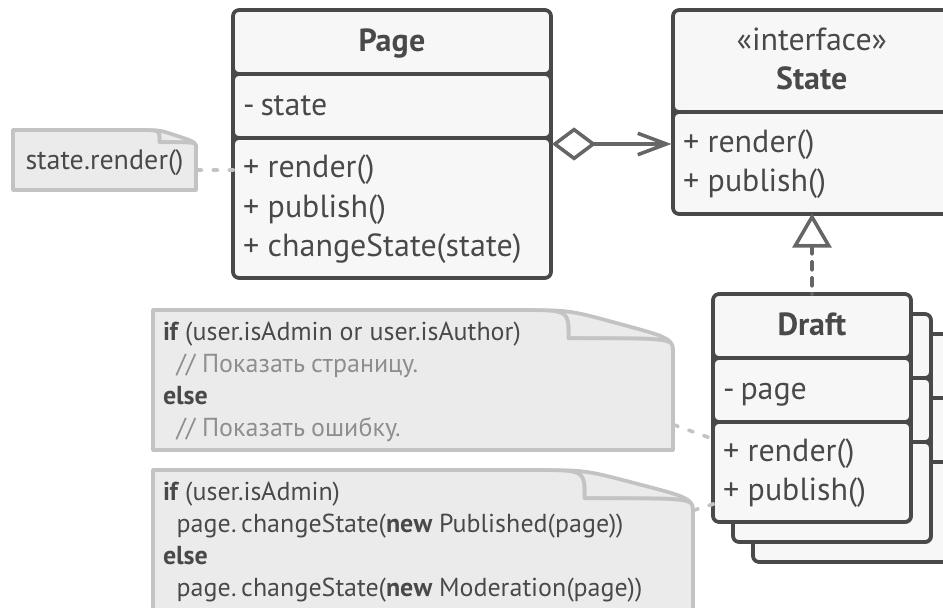
Путаница и нагромождение условий особенно сильно проявляется в старых проектах. Набор возможных состояний бывает трудно предопределить заранее, поэтому они всё время добавляются в процессе эволюции программы. Из-за этого решение, которое выглядело простым и эффективным в самом начале разработки, может впоследствии стать проекцией большого макаронного монстра.

Решение

Паттерн Состояние предлагает создать отдельные классы для каждого состояния, в котором может пребывать контекстный объект, а затем вынести туда поведения,

соответствующие этим состояниям.

Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.



Страница делегирует выполнение своему активному состоянию.

Благодаря тому, что объекты состояний будут иметь общий интерфейс, контекст сможет делегировать работу состоянию, не привязываясь к его классу. Поведение контекста можно будет изменить в любой момент, подключив к нему другой объект-состояние.

Очень важным нюансом, отличающим этот паттерн от [Стратегии](#), является то, что и контекст, и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

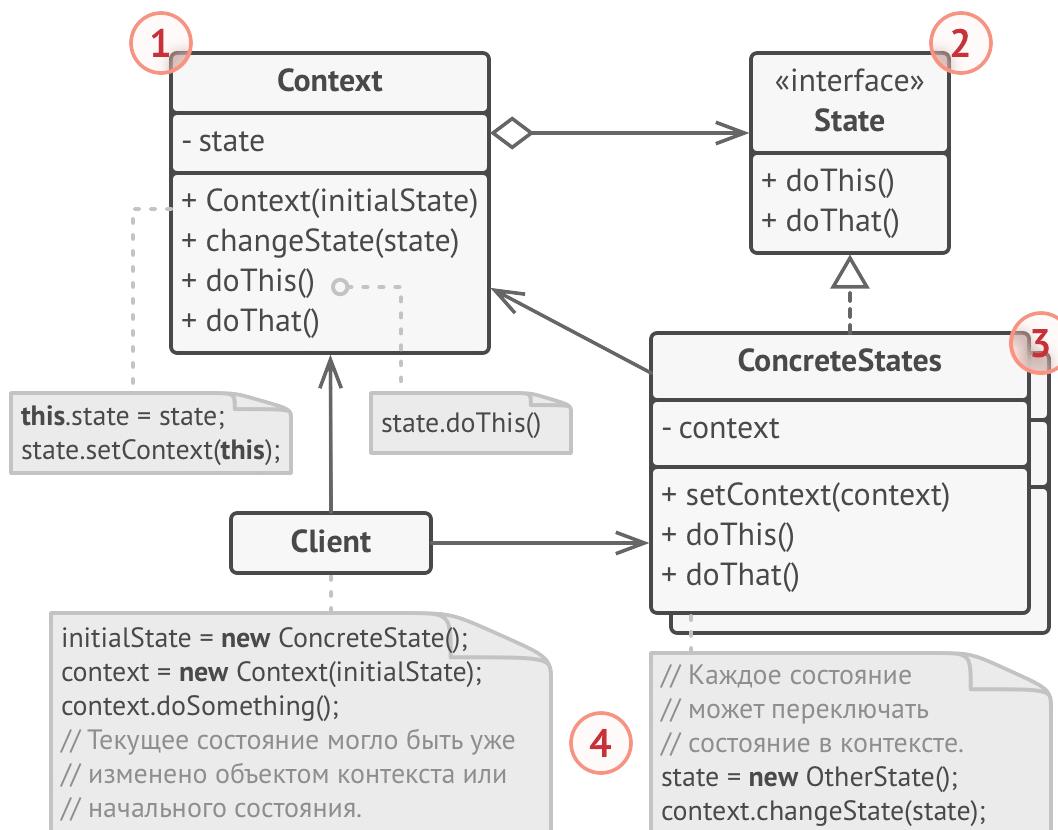
Аналогия из жизни

Ваш смартфон ведёт себя по-разному, в зависимости от текущего состояния:

- Когда телефон разблокирован, нажатие кнопок телефона приводит к каким-то действиям.

- Когда телефон заблокирован, нажатие кнопок приводит к экрану разблокировки.
- Когда телефон разряжен, нажатие кнопок приводит к экрану зарядки.

Структура



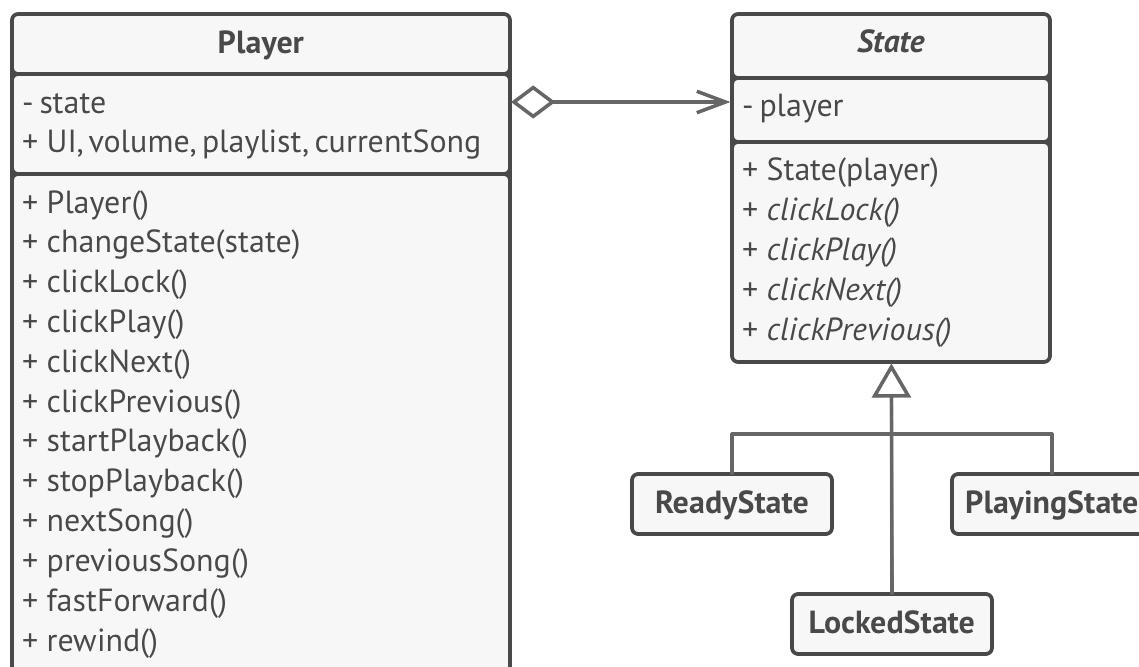
1. **Контекст** хранит ссылку на объект состояния и делегирует ему часть работы, зависящей от состояний. Контекст работает с этим объектом через общий интерфейс состояний. Контекст должен иметь метод для присваивания ему нового объекта-состояния.
2. **Состояние** описывает общий интерфейс для всех конкретных состояний.
3. **Конкретные состояния** реализуют поведения, связанные с определённым состоянием контекста. Иногда приходится создавать целые иерархии классов состояний, чтобы обобщить дублирующий код.

Состояние может иметь обратную ссылку на объект контекста. Через неё не только удобно получать из контекста нужную информацию, но и осуществлять смену его состояния.

4. И контекст, и объекты конкретных состояний могут решать, когда и какое следующее состояние будет выбрано. Чтобы переключить состояние, нужно подать другой объект-состояние в контекст.

Псевдокод

В этом примере паттерн **Состояние** изменяет функциональность одних и тех же элементов управления музыкальным проигрывателем, в зависимости от того, в каком состоянии находится сейчас проигрыватель.



Пример изменения поведения проигрывателя с помощью состояний.

Объект проигрывателя содержит объект-состояние, которому и делегирует основную работу. Изменяя состояния, можно менять то, как ведут себя элементы управления проигрывателя.

```
1 // Общий интерфейс всех состояний.
2 abstract class State is
3     protected field player: Player
4
5     // Контекст передаёт себя в конструктор состояния, чтобы
6     // состояние могло обращаться к его данным и методам в
```

```
7 // будущем, если потребуется.
8 constructor State(player) is
9     this.player = player
10
11 abstract method clickLock()
12 abstract method clickPlay()
13 abstract method clickNext()
14 abstract method clickPrevious()
15
16
17 // Конкретные состояния реализуют методы абстрактного состояния
18 // по-своему.
19 class LockedState extends State is
20
21     // При разблокировке проигрывателя с заблокированными
22     // клавишами он может принять одно из двух состояний.
23     method clickLock() is
24         if (player.playing)
25             player.changeState(new PlayingState(player))
26         else
27             player.changeState(new ReadyState(player))
28
29     method clickPlay() is
30         // Ничего не делать.
31
32     method clickNext() is
33         // Ничего не делать.
34
35     method clickPrevious() is
36         // Ничего не делать.
37
38
39 // Конкретные состояния сами могут переводить контекст в другое
40 // состояние.
41 class ReadyState extends State is
42     method clickLock() is
43         player.changeState(new LockedState(player))
44
45     method clickPlay() is
46         player.startPlayback()
47         player.changeState(new PlayingState(player))
```

```
48
49     method clickNext() is
50         player.nextSong()
51
52     method clickPrevious() is
53         player.previousSong()
54
55
56 class PlayingState extends State is
57     method clickLock() is
58         player.changeState(new LockedState(player))
59
60     method clickPlay() is
61         player.stopPlayback()
62         player.changeState(new ReadyState(player))
63
64     method clickNext() is
65         if (event.doubleclick)
66             player.nextSong()
67         else
68             player.fastForward(5)
69
70     method clickPrevious() is
71         if (event.doubleclick)
72             player.previous()
73         else
74             player.rewind(5)
75
76
77 // Проигрыватель выступает в роли контекста.
78 class Player is
79     field state: State
80     field UI, volume, playlist, currentSong
81
82     constructor Player() is
83         this.state = new ReadyState(this)
84
85         // Контекст заставляет состояние реагировать на
86         // пользовательский ввод вместо себя. Реакция может быть
87         // разной, в зависимости от того, какое состояние сейчас
88         // активно.
```

```
89     UI = new UserInterface()
90     UI.lockButton.onClick(this.clickLock)
91     UI.playButton.onClick(this.clickPlay)
92     UI.nextButton.onClick(this.clickNext)
93     UI.prevButton.onClick(this.clickPrevious)
94
95     // Другие объекты тоже должны иметь возможность заменять
96     // состояние проигрывателя.
97     method changeState(state: State) is
98         this.state = state
99
100    // Методы UI будут делегировать работу активному состоянию.
101    method clickLock() is
102        state.clickLock()
103    method clickPlay() is
104        state.clickPlay()
105    method clickNext() is
106        state.clickNext()
107    method clickPrevious() is
108        state.clickPrevious()
109
110    // Сервисные методы контекста, вызываемые состояниями.
111    method startPlayback() is
112        // ...
113    method stopPlayback() is
114        // ...
115    method nextSong() is
116        // ...
117    method previousSong() is
118        // ...
119    method fastForward(time) is
120        // ...
121    method rewind(time) is
122        // ...
```

Применимость

Когда у вас есть объект, поведение которого кардинально меняется в зависимости от внутреннего состояния, причём типов состояний много, и их код часто меняется.

Паттерн предлагает выделить в собственные классы все поля и методы, связанные с определёнными состояниями. Первоначальный объект будет постоянно ссылаться на один из объектов-состояний, делегируя ему часть своей работы. Для изменения состояния в контекст достаточно будет подставить другой объект-состояние.

Когда код класса содержит множество больших, похожих друг на друга, условных операторов, которые выбирают поведения в зависимости от текущих значений полей класса.

Паттерн предлагает переместить каждую ветку такого условного оператора в собственный класс. Тут же можно поселить и все поля, связанные с данным состоянием.

Когда вы сознательно используете табличную машину состояний, построенную на условных операторах, но вынуждены мириться с дублированием кода для похожих состояний и переходов.

Паттерн Состояние позволяет реализовать иерархическую машину состояний, базирующуюся на наследовании. Вы можете отнаследовать похожие состояния от одного родительского класса и вынести туда весь дублирующий код.

Шаги реализации

1. Определитесь с классом, который будет играть роль контекста. Это может быть как существующий класс, в котором уже есть зависимость от состояния, так и новый класс, если код состояний размазан по нескольким классам.
2. Создайте общий интерфейс состояний. Он должен описывать методы, общие для всех состояний, обнаруженных в контексте. Заметьте, что не всё поведение контекста нужно переносить в состояние, а только то, которое зависит от состояний.

3. Для каждого фактического состояния создайте класс, реализующий интерфейс состояния. Переместите код, связанный с конкретными состояниями в нужные классы. В конце концов, все методы интерфейса состояния должны быть реализованы во всех классах состояний.

При переносе поведения из контекста вы можете столкнуться с тем, что это поведение зависит от приватных полей или методов контекста, к которым нет доступа из объекта состояния. Существует парочка способов обойти эту проблему.

Самый простой – оставить поведение внутри контекста, вызывая его из объекта состояния. С другой стороны, вы можете сделать классы состояний вложенными в класс контекста, и тогда они получат доступ ко всем приватным частям контекста. Но последний способ доступен только в некоторых языках программирования (например, Java, C#).

4. Создайте в контексте поле для хранения объектов-состояний, а также публичный метод для изменения значения этого поля.
5. Старые методы контекста, в которых находился зависимый от состояния код, замените на вызовы соответствующих методов объекта-состояния.
6. В зависимости от бизнес-логики, разместите код, который переключает состояние контекста либо внутри контекста, либо внутри классов конкретных состояний.

Преимущества и недостатки

Избавляет от множества больших условных операторов машины состояний.

Концентрирует в одном месте код, связанный с определённым состоянием.

Упрощает код контекста.

Может неоправданно усложнить код, если состояний мало и они редко меняются.

Отношения с другими паттернами

- **Мост, Стратегия и Состояние** (а также слегка и Адаптер) имеют схожие структуры классов – все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны – это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.
- **Состояние** можно рассматривать как надстройку над **Стратегией**. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в *Стратегии* эти объекты не знают друг о друге и никак не связаны. В *Состоянии* сами конкретные состояния могут переключать контекст.



СТРАТЕГИЯ

Также известен как: *Strategy*

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

Проблема

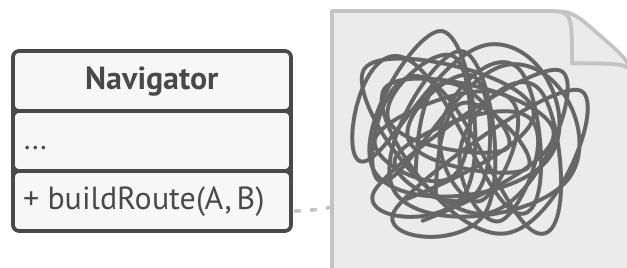
Вы решили написать приложение-навигатор для путешественников. Оно должно показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.

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

Первая версия вашего навигатора могла прокладывать маршрут лишь по дорогам, поэтому отлично подходила для путешествий на автомобиле. Но, очевидно, не все ездят в отпуск на машине. Поэтому следующим шагом вы добавили в навигатор прокладывание пеших маршрутов.

Через некоторое время выяснилось, что некоторые люди предпочитают ездить по городу на общественном транспорте. Поэтому вы добавили и такую опцию прокладывания пути.

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



Код навигатора становится слишком раздутым.

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

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

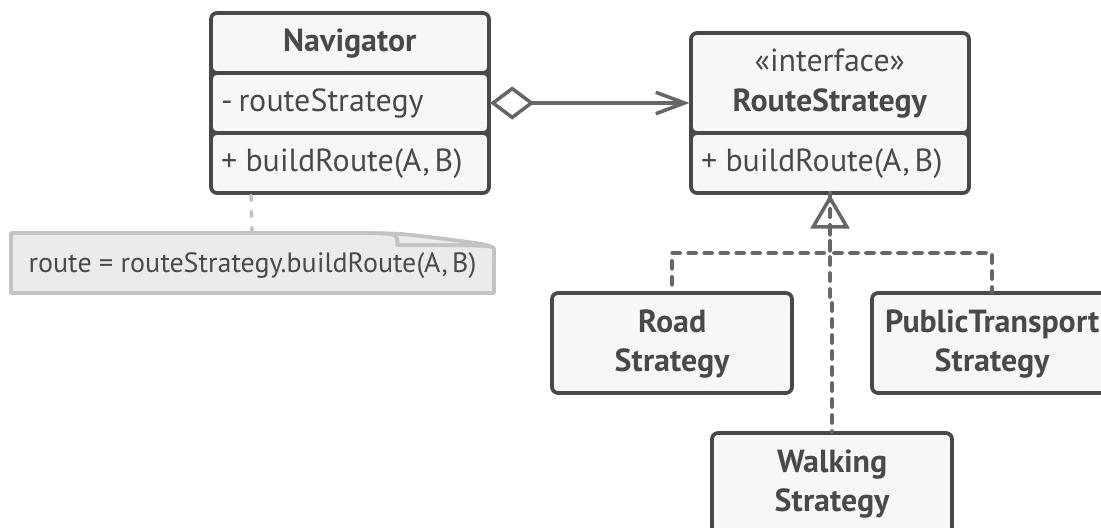
Кроме того, осложнялась командная работа с другими программистами, которых вы наняли после успешного релиза навигатора. Ваши изменения нередко затрагивали один и тот же код, создавая конфликты, которые требовали дополнительного времени на их разрешение.

Решение

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

Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.

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



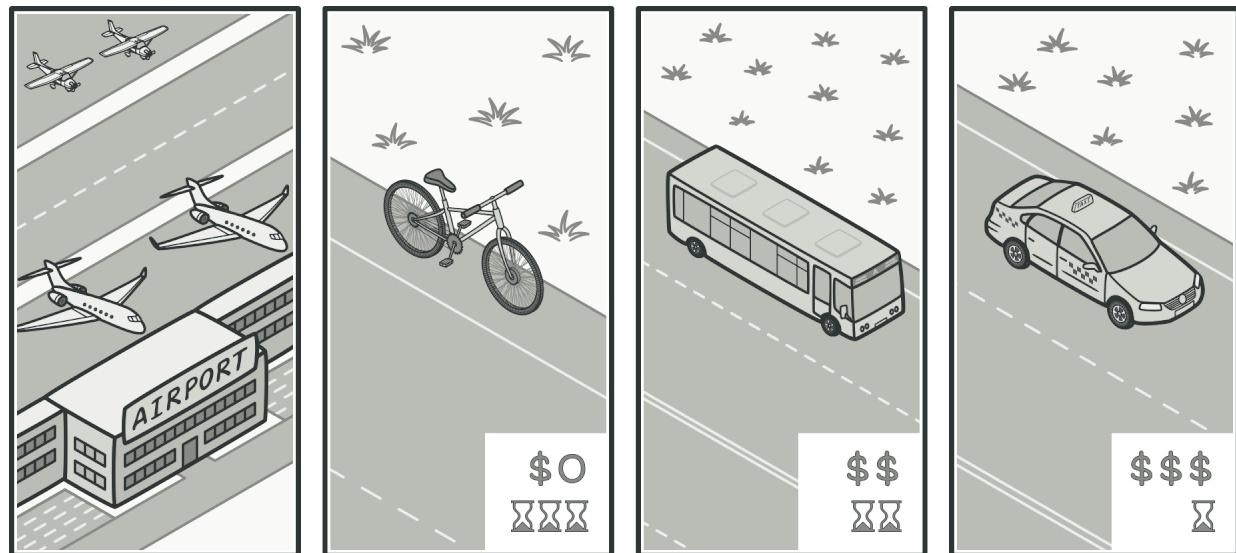
Стратегии построения пути.

В нашем примере каждый алгоритм поиска пути переедет в свой собственный класс. В этих классах будет определён лишь один метод, принимающий в параметрах координаты начала и конца пути, а возвращающий массив точек маршрута.

Хотя каждый класс будет прокладывать маршрут по-своему, для навигатора это не будет иметь никакого значения, так как его работа заключается только в отрисовке маршрута. Навигатору достаточно подать в стратегию данные о начале и конце маршрута, чтобы получить массив точек маршрута в оговорённом формате.

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

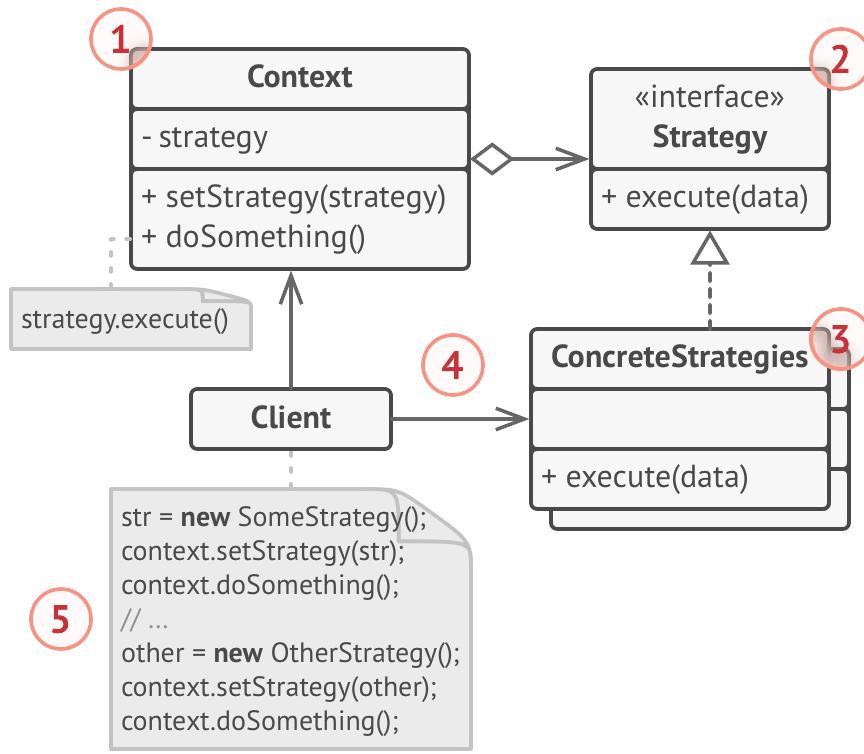
Аналогия из жизни



Различные стратегии попадания в аэропорт.

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

Структура



1. **Контекст** хранит ссылку на объект конкретной стратегии, работая с ним через общий интерфейс стратегий.
 2. **Стратегия** определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма.
- Для контекста неважно, какая именно вариация алгоритма будет выбрана, так как все они имеют одинаковый интерфейс.
3. **Конкретные стратегии** реализуют различные вариации алгоритма.
 4. Во время выполнения программы контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.
 5. Клиент должен создать объект конкретной стратегии и передать его в конструктор контекста. Кроме этого, клиент должен иметь возможность заменить стратегию на лету, используя сеттер. Благодаря этому, контекст не будет знать о том, какая именно стратегия сейчас выбрана.

Псевдокод

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

```
1 // Общий интерфейс всех стратегий.
2 interface Strategy is
3     method execute(a, b)
4
5 // Каждая конкретная стратегия реализует общий интерфейс своим
6 // способом.
7 class ConcreteStrategyAdd implements Strategy is
8     method execute(a, b) is
9         return a + b
10
11 class ConcreteStrategySubtract implements Strategy is
12     method execute(a, b) is
13         return a - b
14
15 class ConcreteStrategyMultiply implements Strategy is
16     method execute(a, b) is
17         return a * b
18
19 // Контекст всегда работает со стратегиями через общий
20 // интерфейс. Он не знает, какая именно стратегия ему подана.
21 class Context is
22     private strategy: Strategy
23
24     method setStrategy(Strategy strategy) is
25         this.strategy = strategy
26
27     method executeStrategy(int a, int b) is
28         return strategy.execute(a, b)
29
30
31 // Конкретная стратегия выбирается на более высоком уровне,
32 // например, конфигуратором всего приложения. Готовый
33 // объект-стратегия подаётся в клиентский объект, а затем может
34 // быть заменён другой стратегией в любой момент на лету.
```

```
35 class ExampleApplication is
36     method main() is
37         // 1. Создать объект контекста.
38         // 2. Получить первое число (n1).
39         // 3. Получить второе число (n2).
40         // 4. Получить желаемую операцию.
41         // 5. Затем, выбрать стратегию:
42
43     if (action == addition) then
44         context.setStrategy(new ConcreteStrategyAdd())
45
46     if (action == subtraction) then
47         context.setStrategy(new ConcreteStrategySubtract())
48
49     if (action == multiplication) then
50         context.setStrategy(new ConcreteStrategyMultiply())
51
52     // 6. Выполнить операцию с помощью стратегии:
53     result = context.executeStrategy(n1, n2)
54
55     // 7. Вывести результат на экран.
```

Применимость

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

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

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

Стратегия позволяет вынести отличающееся поведение в отдельную иерархию классов, а затем свести первоначальные классы к одному, сделав поведение этого класса

настраиваемым.

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

Стратегия позволяет изолировать код, данные и зависимости алгоритмов от других объектов, скрыв эти детали внутри классов-стратегий.

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

Стратегия помещает каждую лапу такого оператора в отдельный класс-стратегию. Затем контекст получает определённый объект-стратегию от клиента и делегирует ему работу. Если вдруг понадобится сменить алгоритм, в контекст можно подать другую стратегию.

Шаги реализации

1. Определите алгоритм, который подвержен частым изменениям. Также подойдёт алгоритм, имеющий несколько вариаций, которые выбираются во время выполнения программы.
2. Создайте интерфейс стратегий, описывающий этот алгоритм. Он должен быть общим для всех вариантов алгоритма.
3. Поместите вариации алгоритма в собственные классы, которые реализуют этот интерфейс.
4. В классе контекста создайте поле для хранения ссылки на текущий объект-стратегию, а также метод для её изменения. Убедитесь в том, что контекст работает с этим объектом только через общий интерфейс стратегий.
5. Клиенты контекста должны подавать в него соответствующий объект-стратегию, когда хотят, чтобы контекст вёл себя определённым образом.

Преимущества и недостатки

Горячая замена алгоритмов на лету.

Изолирует код и данные алгоритмов от остальных классов.

Уход от наследования к делегированию.

Реализует *принцип открытости/закрытости*.

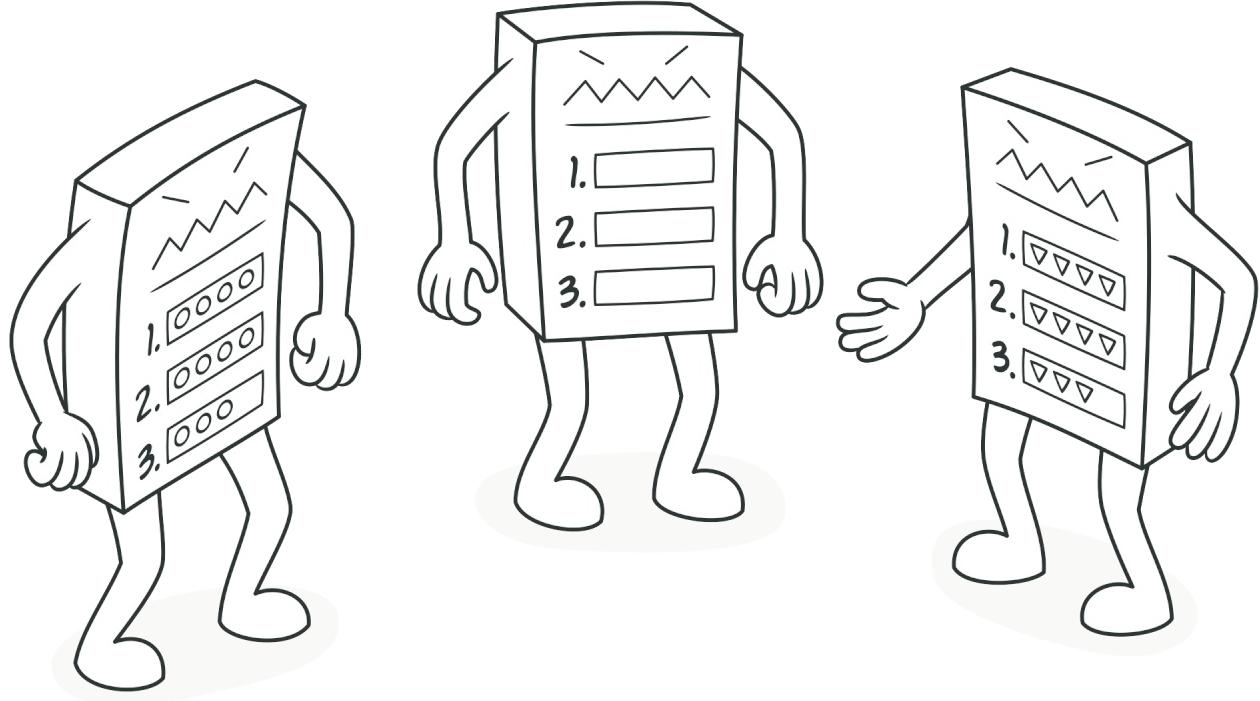
Усложняет программу за счёт дополнительных классов.

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

Отношения с другими паттернами

- **Мост, Стратегия и Состояние** (а также слегка и **Адаптер**) имеют схожие структуры классов – все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны – это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.
- **Команда** и **Стратегия** похожи по духу, но отличаются масштабом и применением:
 - Команду используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.
 - С другой стороны, Стратегия описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.
- **Стратегия** меняет поведение объекта «изнутри», а **Декоратор** изменяет его «снаружи».

- **Шаблонный метод** использует наследование, чтобы расширять части алгоритма. **Стратегия** использует делегирование, чтобы изменять выполняемые алгоритмы на лету. *Шаблонный метод* работает на уровне классов. *Стратегия* позволяет менять логику отдельных объектов.
- **Состояние** можно рассматривать как надстройку над **Стратегией**. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в *Стратегии* эти объекты не знают друг о друге и никак не связаны. В *Состоянии* сами конкретные состояния могут переключать контекст.



ШАБЛОННЫЙ МЕТОД

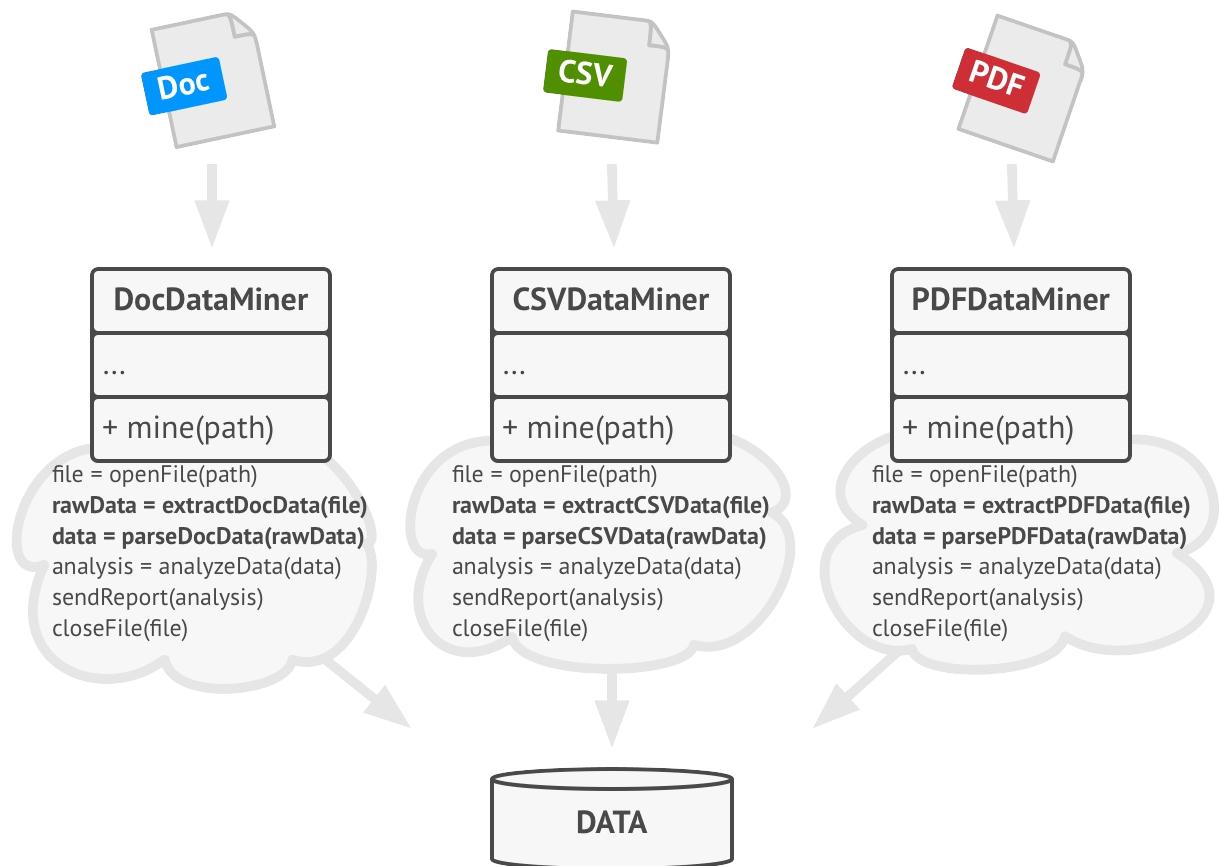
Также известен как: *Template Method*

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.



Классы дата-майнинга содержат много дублирования.

В какой-то момент вы заметили, что код всех трёх классов обработки документов хоть и отличается в части работы с файлами, но содержит довольно много общего в части самого извлечения данных. Было бы здорово избавится от повторной реализации алгоритма извлечения данных в каждом из классов.

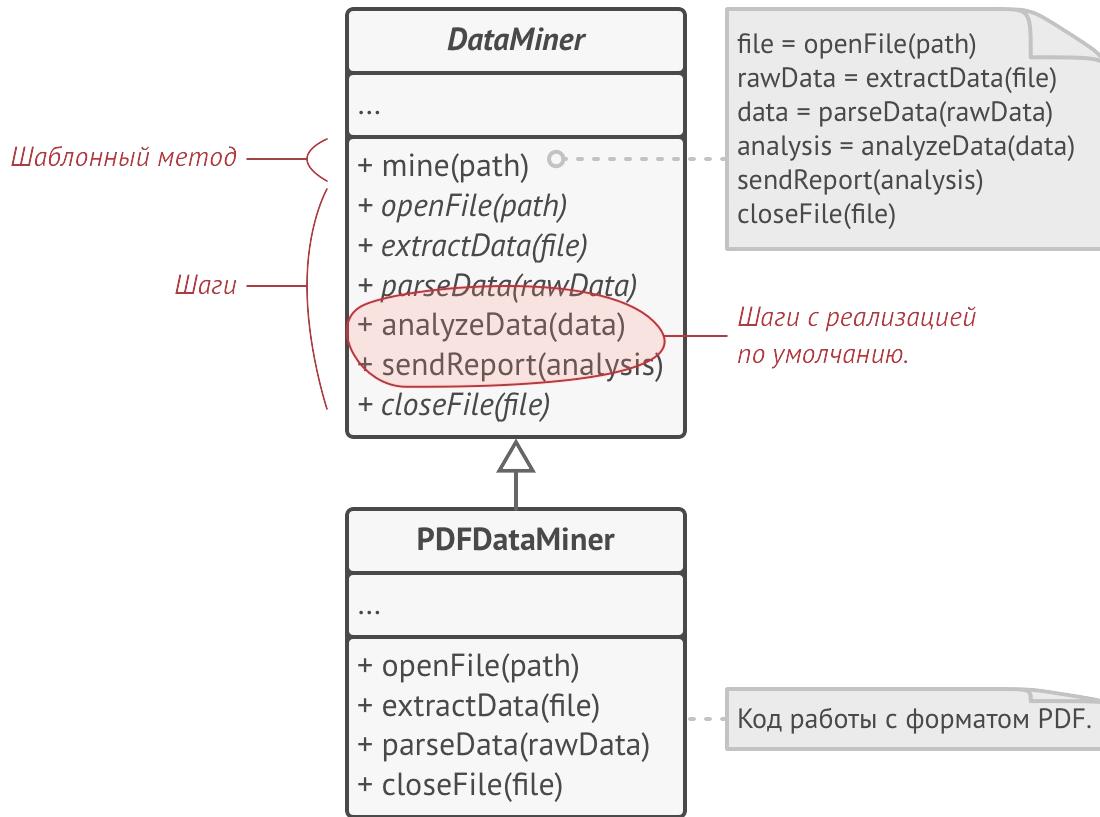
К тому же остальной код, работающий с объектами этих классов, наполнен условиями, проверяющими тип обработчика перед началом работы. Весь этот код можно упростить, если слить все три класса воедино либо свести их к общему интерфейсу.

Решение

Паттерн Шаблонный метод предлагает разбить алгоритм на последовательность шагов, описать эти шаги в отдельных методах и 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.



Шаблонный метод разбивает алгоритм на шаги, позволяя подклассам переопределить некоторые из них.

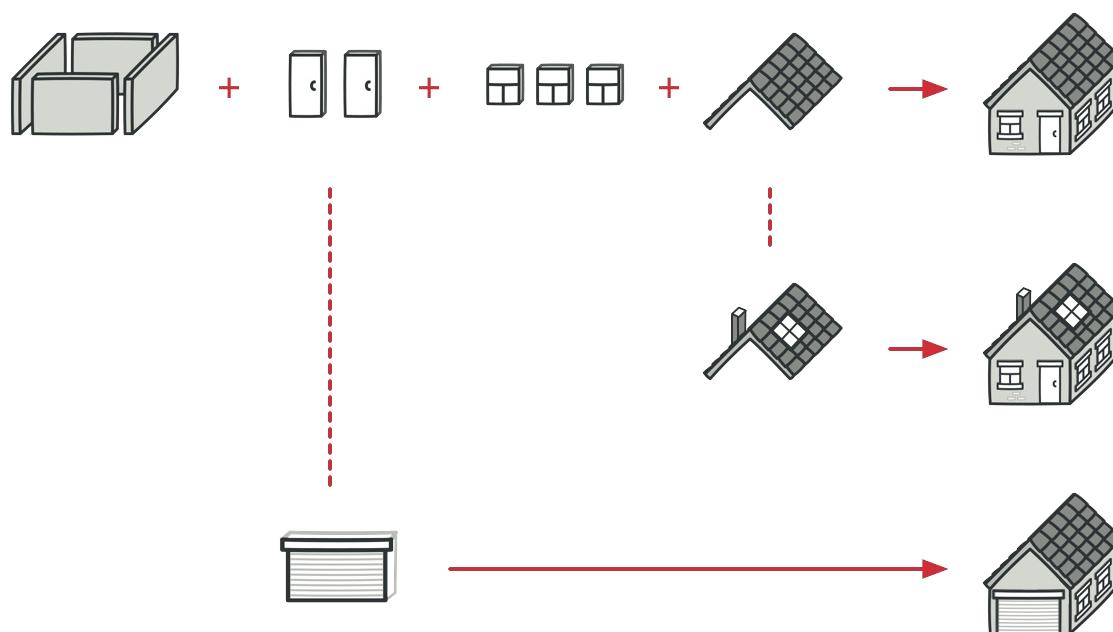
Для начала шаги шаблонного метода можно сделать абстрактными. Из-за этого все подклассы должны будут реализовать каждый из шагов по-своему. В нашем случае все подклассы и так содержат реализацию каждого из шагов, поэтому ничего дополнительного делать не нужно.

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*: 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. Однако, хук даёт подклассам дополнительные точки «вклинивания» в шаблонный метод.

Аналогия из жизни

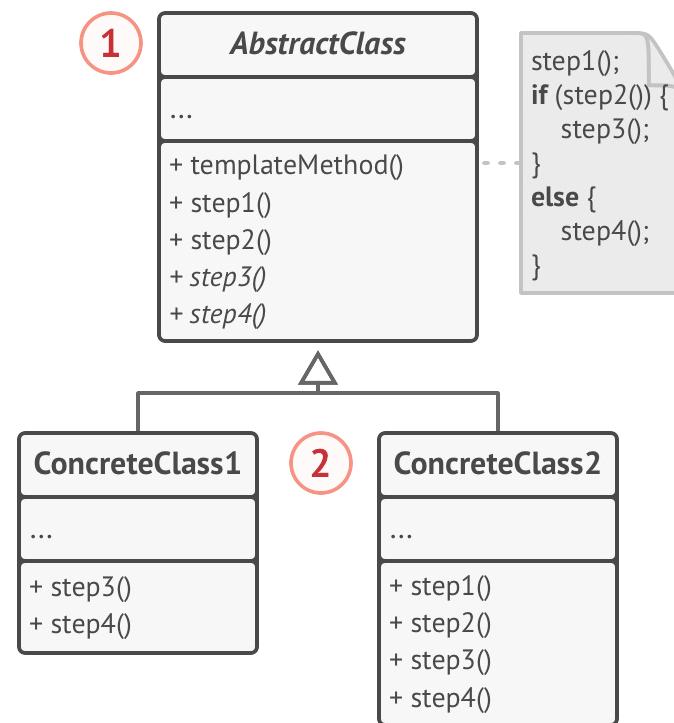


Проект типового дома могут немного изменить по желанию клиента.

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 чуточку непохожим на другие.

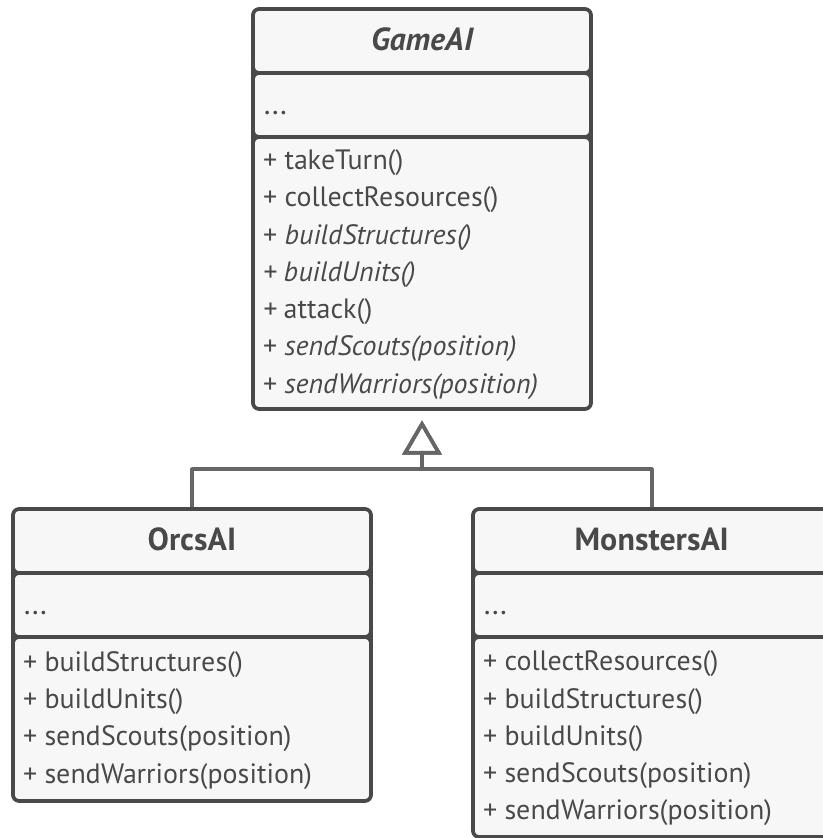
Структура



1. **Абстрактный класс** определяет шаги алгоритма и содержит шаблонный метод, состоящий из вызовов этих шагов. Шаги могут быть как абстрактными, так и содержать реализацию по умолчанию.
2. **Конкретный класс** переопределяет некоторые (или все) шаги алгоритма. Конкретные классы не переопределяют сам шаблонный метод.

Псевдокод

В этом примере **Шаблонный метод** используется как заготовка для стандартного искусственного интеллекта в простой игре-стратегии. Для введения в игру новой расы достаточно создать подкласс и реализовать в нём недостающие методы.



Пример классов искусственного интеллекта для простой игры.

Все расы игры будут содержать примерно такие же типы юнитов и строений, поэтому структура ИИ будет одинаковой. Но разные расы могут по-разному реализовать эти шаги. Так, например, орки будут агрессивней в атаке, люди – более активны в защите, а дикие монстры вообще не будут заниматься строительством.

```

1 class GameAI is
2     // Шаблонный метод должен быть задан в базовом классе. Он
3     // состоит из вызовов методов в определённом порядке. Чаще
4     // всего эти методы являются шагами некоего алгоритма.
5     method turn() is
6         collectResources()
7         buildNewStructures()
8         buildUnits()
9         attack()
10
11     // Некоторые из этих методов могут быть реализованы прямо в
12     // базовом классе.
13     method collectResources() is
14         foreach (s in this.builtStructures) do
15             s.collect()

```

```
16 // А некоторые могут быть полностью абстрактными.
17 abstract method buildStructures()
18 abstract method buildUnits()
19
20
21 // Кстати, шаблонных методов в классе может быть несколько.
22 method attack() is
23     enemy = closestEnemy()
24     if (enemy == null)
25         sendScouts(map.center)
26     else
27         sendWarriors(enemy.position)
28
29 abstract method sendScouts(position)
30 abstract method sendWarriors(position)
31
32 // Подклассы могут предоставлять свою реализацию шагов
33 // алгоритма, не изменяя сам шаблонный метод.
34 class OrcsAI extends GameAI is
35     method buildStructures() is
36         if (there are some resources) then
37             // Строить фермы, затем бараки, а потом цитадель.
38
39     method buildUnits() is
40         if (there are plenty of resources) then
41             if (there are no scouts)
42                 // Построить раба и добавить в группу
43                 // разведчиков.
44             else
45                 // Построить пехотинца и добавить в группу
46                 // воинов.
47
48     // ...
49
50     method sendScouts(position) is
51         if (scouts.length > 0) then
52             // Отправить разведчиков на позицию.
53
54     method sendWarriors(position) is
55         if (warriors.length > 5) then
56             // Отправить воинов на позицию.
```

```
57
58 // Подклассы могут не только реализовывать абстрактные шаги, но
59 // и переопределять шаги, уже реализованные в базовом классе.
60 class MonstersAI extends GameAI {
61     method collectResources() {
62         // Ничего не делать.
63
64     method buildStructures() {
65         // Ничего не делать.
66
67     method buildUnits() {
68         // Ничего не делать.
```

Применимость

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.

Когда у вас есть несколько классов, делающих одно и то же с незначительными различиями. 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

1. Изучите алгоритм и подумайте, можно ли его разбить на шаги. Прикиньте, какие шаги будут стандартными для всех вариаций алгоритма, а какие – изменяющимися.
2. 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).
3. 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.
4. Подумайте о введении в алгоритм хуков. Чаще всего, хуки располагают между основными шагами алгоритма, а также до и после всех шагов.
5. Создайте конкретные классы, унаследовав их от абстрактного класса. Реализуйте в них все недостающие шаги и хуки.

Преимущества и недостатки

Облегчает повторное использование кода.

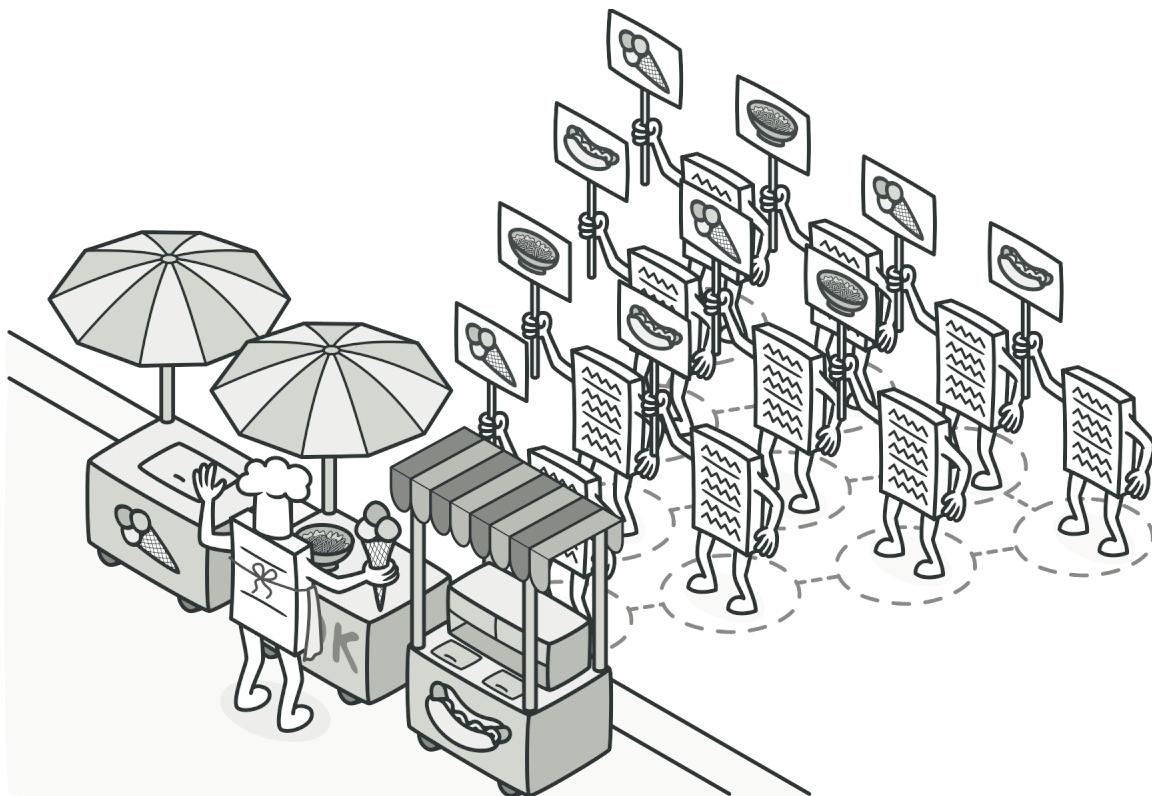
Вы жёстко ограничены скелетом существующего алгоритма.

Вы можете нарушить *принцип подстановки Барбары Лисков*, изменяя базовое поведение одного из шагов алгоритма через подкласс.

С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.

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.



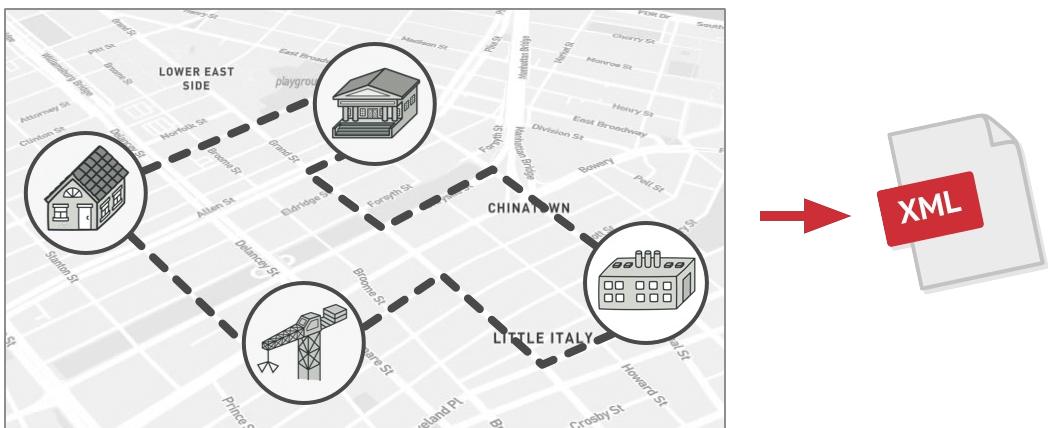
ПОСЕТИТЕЛЬ

Также известен как: *Visitor*

Посетитель – это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

Проблема

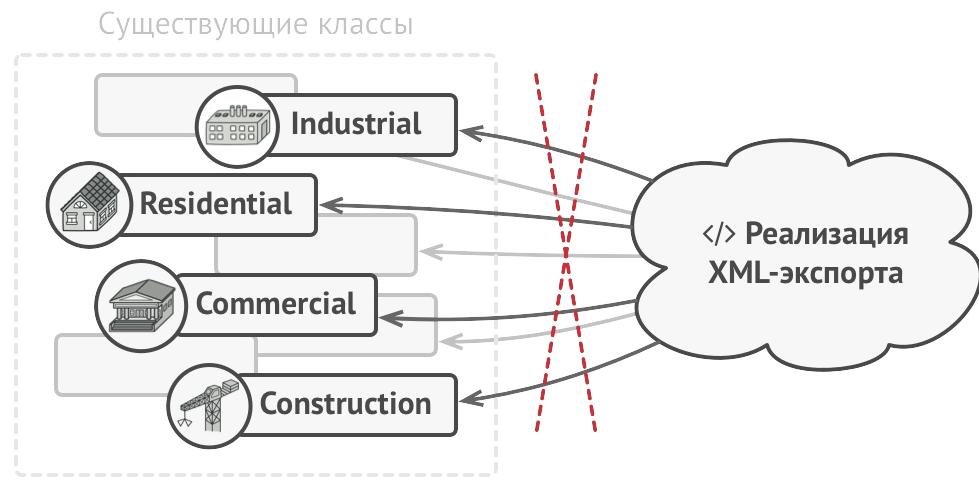
Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлами графа являются городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом.



Экспорт геоузлов в XML.

Ваша задача – сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. Достаточно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы графа, вызывать этот метод для каждого узла. Благодаря полиморфизму, решение получилось бы изящным, так как вам не пришлось бы привязываться к конкретным классам узлов.

Но, к сожалению, классы узлов вам изменить не удалось. Системный архитектор сослался на то, что код классов узлов сейчас очень стабилен, и от него многое зависит, поэтому он не хочет рисковать и позволять кому-либо его трогать.



Код XML-экспорта придётся добавить во все классы узлов, а это слишком накладно.

К тому же он сомневался в том, что экспорт в XML вообще уместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит в рамках этих классов чужеродно.

Была и ещё одна причина запрета. Если на следующей неделе вам бы понадобился экспорт

в какой-то другой формат данных, то эти классы снова пришлось бы менять.

Решение

Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.

Код поведения, скорее всего, должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будет схож, но основное отличие будет в типе принимаемого в параметрах объекта, например:

```
1 class ExportVisitor implements Visitor is
2     method doForCity(City c) { ... }
3     method doForIndustry(Industry f) { ... }
4     method doForSightSeeing(SightSeeing ss) { ... }
5     // ...
```

Здесь возникает вопрос: как подавать узлы в объект-посетитель? Так как все методы имеют отличающуюся сигнатуру, использовать полиморфизм при переборе узлов не получится. Придётся проверять тип узлов для того, чтобы выбрать соответствующий метод посетителя.

```
1 foreach (Node node in graph)
2     if (node instanceof City)
3         exportVisitor.doForCity((City) node);
4     if (node instanceof Industry)
5         exportVisitor.doForIndustry((Industry) node);
6     // ...
```

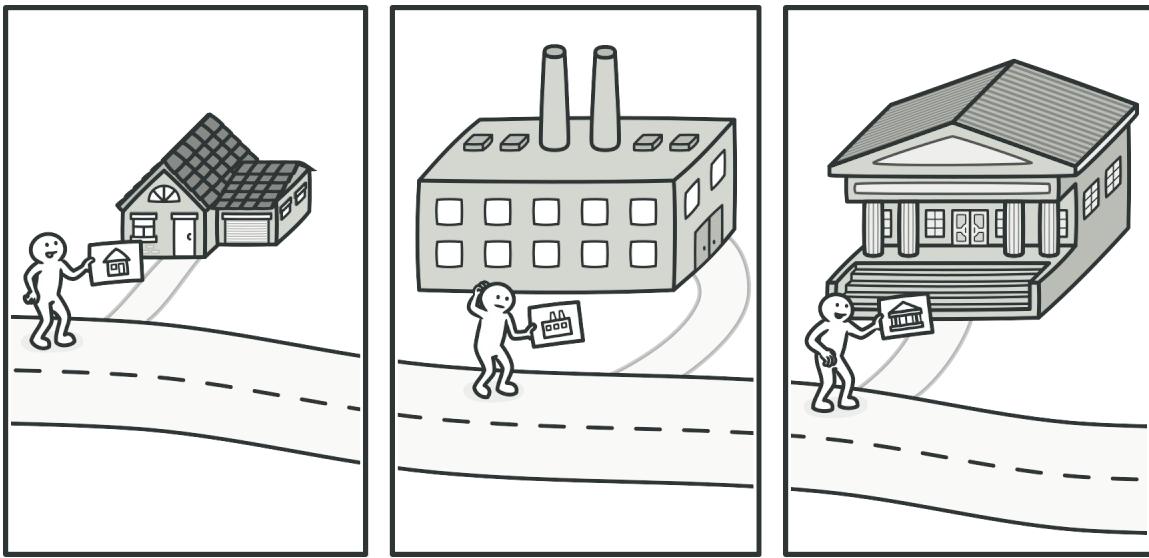
Тут не поможет даже механизм перегрузки методов (доступный в Java и C#). Если назвать все методы одинаково, то неопределённость реального типа узла всё равно не даст вызвать правильный метод. Механизм перегрузки всё время будет вызывать метод посетителя, соответствующий типу `Node`, а не реального класса поданного узла.

Но паттерн Посетитель решает и эту проблему, используя механизм [двойной диспетчеризации](#). Вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

```
1 // Client code
2 foreach (Node node in graph)
3     node.accept(exportVisitor);
4
5 // City
6 class City is
7     method accept(Visitor v) is
8         v.doForCity(this);
9     // ...
10
11 // Industry
12 class Industry is
13     method accept(Visitor v) is
14         v.doForIndustry(this);
15     // ...
```

Как видите, изменить классы узлов всё-таки придётся. Но это простое изменение позволит применять к объектам узлов и другие поведения, ведь классы узлов будут привязаны не к конкретному классу посетителей, а к их общему интерфейсу. Поэтому если придётся добавить в программу новое поведение, вы создадите новый класс посетителей и будете передавать его в методы узлов.

Аналогия из жизни

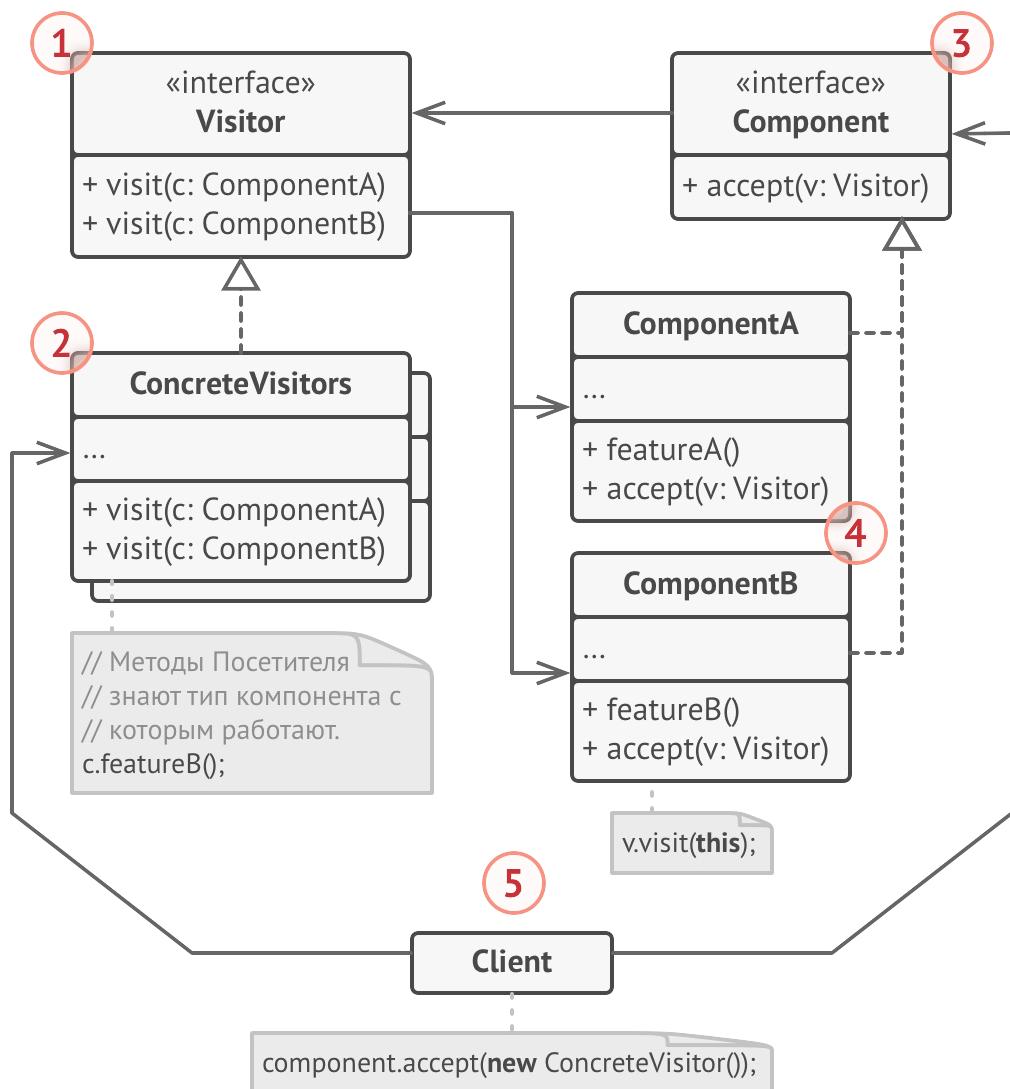


У страхового агента подготовлены полисы для разных видов организаций.

Представьте начинающего страхового агента, жаждущего получить новых клиентов. Он беспорядочно посещает все дома в округе, предлагая свои услуги. Но для каждого из посещаемых *типов* домов у него имеется особое предложение.

- Придя в дом к обычной семье, он предлагает оформить медицинскую страховку.
- Придя в банк, он предлагает страховку от грабежа.
- Придя на фабрику, он предлагает страховку предприятия от пожара и наводнения.

Структура

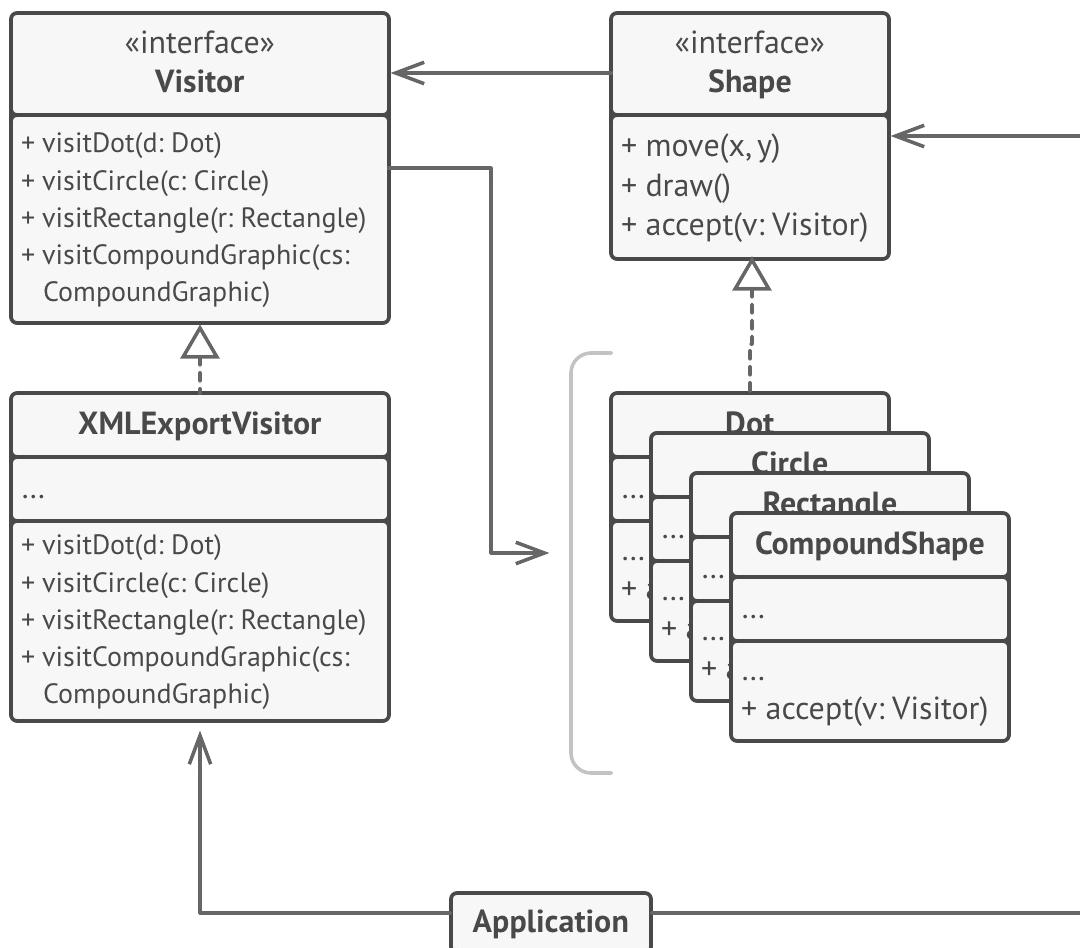


1. **Посетитель** описывает общий интерфейс для всех типов посетителей. Он объявляет набор методов, отличающихся типом входящего параметра, которые нужны для запуска операции для всех типов конкретных компонентов. В языках, поддерживающих перегрузку методов, эти методы могут иметь одинаковые имена, но типы их параметров должны отличаться.
2. **Конкретные посетители** реализуют какое-то особенное поведение для всех типов компонентов, которые можно подать через методы интерфейса посетителя.
3. **Компонент** описывает метод *принятия* посетителя. Этот метод должен иметь единственный параметр, объявленный с типом интерфейса посетителя.
4. **Конкретные компоненты** реализуют методы *принятия* посетителя. Цель этого метода – вызвать тот метод посещения, который соответствует типу этого компонента. Так посетитель узнает, с каким именно компонентом он работает.

5. **Клиентом** зачастую выступает коллекция или сложный составной объект, например, дерево **Компоновщика**. Зачастую клиент не привязан к конкретным классам компонентов, работая с ними через общий интерфейс компонентов.

Псевдокод

В этом примере **Посетитель** добавляет в существующую иерархию классов геометрических фигур возможность экспорта в XML.



Пример организации экспортации объектов в XML через отдельный класс-посетитель.

```
1 // Сложная иерархия компонентов.  
2 interface Shape is  
3     method move(x, y)  
4     method draw()  
5     method accept(v: Visitor)  
6  
7 // Метод принятия посетителя должен быть реализован в каждом
```

```
8 // компоненте, а не только в базовом классе. Это поможет
9 // программе определить, какой метод посетителя нужно вызвать,
10 // если вы не знаете тип компонента.
11 class Dot extends Shape is
12     // ...
13     method accept(v: Visitor) is
14         v.visitDot(this)
15
16 class Circle extends Dot is
17     // ...
18     method accept(v: Visitor) is
19         v.visitCircle(this)
20
21 class Rectangle extends Shape is
22     // ...
23     method accept(v: Visitor) is
24         v.visitRectangle(this)
25
26 class CompoundShape implements Shape is
27     // ...
28     method accept(v: Visitor) is
29         v.visitCompoundShape(this)
30
31
32 // Интерфейс посетителей должен содержать методы посещения
33 // каждого компонента. Важно, чтобы иерархия компонентов
34 // менялась редко, так как при добавлении нового компонента
35 // придётся менять всех существующих посетителей.
36 interface Visitor is
37     method visitDot(d: Dot)
38     method visitCircle(c: Circle)
39     method visitRectangle(r: Rectangle)
40     method visitCompoundShape(cs: CompoundShape)
41
42 // Конкретный посетитель реализует одну операцию для всей
43 // иерархии компонентов. Новая операция = новый посетитель.
44 // Посетитель выгодно применять, когда новые компоненты
45 // добавляются очень редко, а новые операции – часто.
46 class XMLExportVisitor is
47     method visitDot(d: Dot) is
48         // Экспорт id и координат центра точки.
```

```
49
50     method visitCircle(c: Circle) is
51         // Экспорт id, координат центра и радиуса окружности.
52
53     method visitRectangle(r: Rectangle) is
54         // Экспорт id, координат левого-верхнего угла, ширины и
55         // высоты прямоугольника.
56
57     method visitCompoundShape(cs: CompoundShape) is
58         // Экспорт id составной фигуры, а также списка id
59         // подфигур, из которых она состоит.
60
61
62 // Приложение может применять посетителя к любому набору
63 // объектов компонентов, даже не уточняя их типы. Нужный метод
64 // посетителя будет выбран благодаря проходу через метод accept.
65 class Application is
66     field allShapes: array of Shapes
67
68     method export() is
69         exportVisitor = new XMLExportVisitor()
70
71     foreach (shape in allShapes) do
72         shape.accept(exportVisitor)
```

Вам не кажется, что вызов метода `accept` – это лишнее звено? Если так, то ещё раз рекомендую вам ознакомиться с проблемой раннего и позднего связывания в статье [Посетитель и Double Dispatch](#).

Применимость

Когда вам нужно выполнить какую-то операцию над всеми элементами сложной структуры объектов, например, деревом.

Посетитель позволяет применять одну и ту же операцию к объектам различных классов.

Когда над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но вы не хотите «засорять» классы такими операциями.

Посетитель позволяет извлечь родственные операции из классов, составляющих структуру объектов, поместив их в один класс-посетитель. Если структура объектов является общей для нескольких приложений, то паттерн позволит в каждое приложение включить только нужные операции.

Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

Посетитель позволяет определить поведение только для этих классов, оставив его пустым для всех остальных.

Шаги реализации

1. Создайте интерфейс посетителя и объявите в нём методы «посещения» для каждого класса компонента, который существует в программе.
2. Опишите интерфейс компонентов. Если вы работаете с уже существующими классами, то объявите абстрактный метод принятия посетителей в базовом классе иерархии компонентов.
3. Реализуйте методы принятия во всех конкретных компонентах. Они должны переадресовывать вызовы тому методу посетителя, в котором тип параметра совпадает с текущим классом компонента.
4. Иерархия компонентов должна знать только о базовом интерфейсе посетителей. С другой стороны, посетители будут знать обо всех классах компонентов.
5. Для каждого нового поведения создайте свой конкретный класс. Приспособьте это поведение для работы со всеми типами компонентов, реализовав все методы

интерфейса посетителей.

Вы можете столкнуться с ситуацией, когда посетителю нужен будет доступ к приватным полям компонентов. В этом случае вы можете либо раскрыть доступ к этим полям, нарушив инкапсуляцию компонентов, либо сделать класс посетителя вложенным в класс компонента, если вам повезло писать на языке, который поддерживает вложенность классов.

6. Клиент будет создавать объекты посетителей, а затем передавать их компонентам, используя метод принятия.

Преимущества и недостатки

Упрощает добавление операций, работающих со сложными структурами объектов.

Объединяет родственные операции в одном классе.

Посетитель может накапливать состояние при обходе структуры компонентов.

Паттерн не оправдан, если иерархия компонентов часто меняется.

Может привести к нарушению инкапсуляции компонентов.

Отношения с другими паттернами

- **Посетитель** можно рассматривать как расширенный аналог **Команды**, который способен работать сразу с несколькими видами получателей.
- Вы можете выполнить какое-то действие над всем деревом **Компоновщика** при помощи **Посетителя**.
- **Посетитель** можно использовать совместно с **Итератором**. *Итератор* будет отвечать за обход структуры данных, а *Посетитель* — за выполнение действий над каждым её компонентом.

Заключение

Поздравляю! Вы добрались до конца!

Но в мире существует множество других паттернов. Надеюсь, эта книга станет вашей стартовой точкой к дальнейшему постижению паттернов и развитию сверхспособностей в проектировании программ.

Вот парочка идей для следующих шагов, если вы ещё не определились, что будете делать дальше.

- Не забывайте, что вместе с этой книгой поставляется архив с реальными примерами кода на разных языках программирования.
- Прочтите книгу Джошуа Кериевски «Рефакторинг с использованием паттернов проектирования».
- Не разбираетесь в рефакторинге? У меня есть хорошие материалы для вас.
- Распечатайте шпаргалки по паттернам и повесьте их где-то на видном месте.
- Оставьте отзыв об этой книге. Мне было бы очень интересно услышать ваше мнение, даже если это критика