

# **Паттерны #2**

**Поведенческие: команда, шаблонный метод, стратегия,  
посредник**

**11.10.2025, ФКН ПИ**

# Паттерны

## Порождающие

Помогают реализовать гибкое создание объектов без внесения в программу лишних зависимостей

## Поведенческие

Заботятся об эффективной коммуникации между объектами

## Структурные

Показывают различные способы построения связей между объектами.

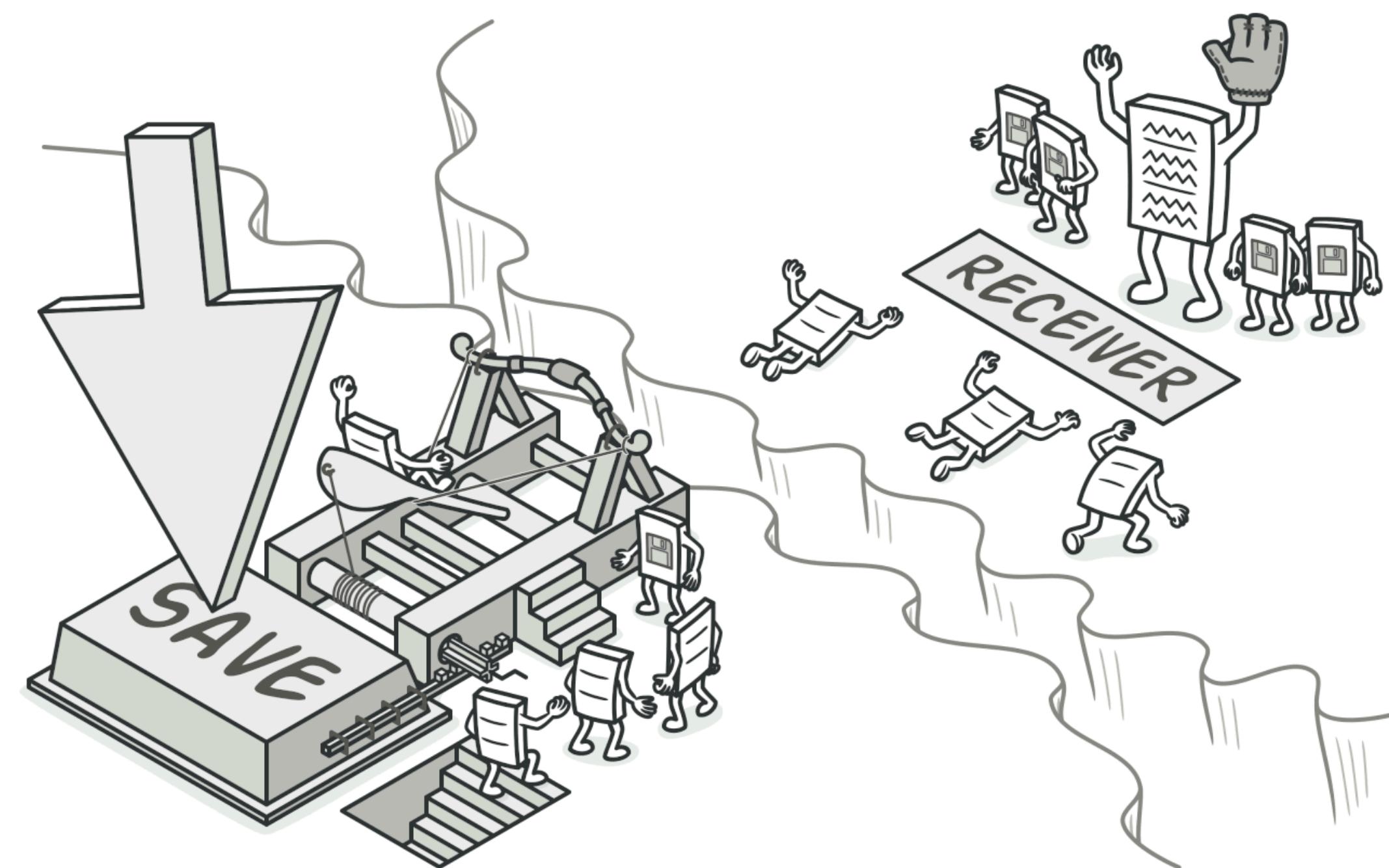
мы тут

# Поведенческие паттерны

- Команда
- Шаблонный метод
- Стратегия
- Посетитель
- Посредник

# Команда (Command)

Паттерн "Команда" позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие называется **командой**. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.



# Как реализовать?

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.

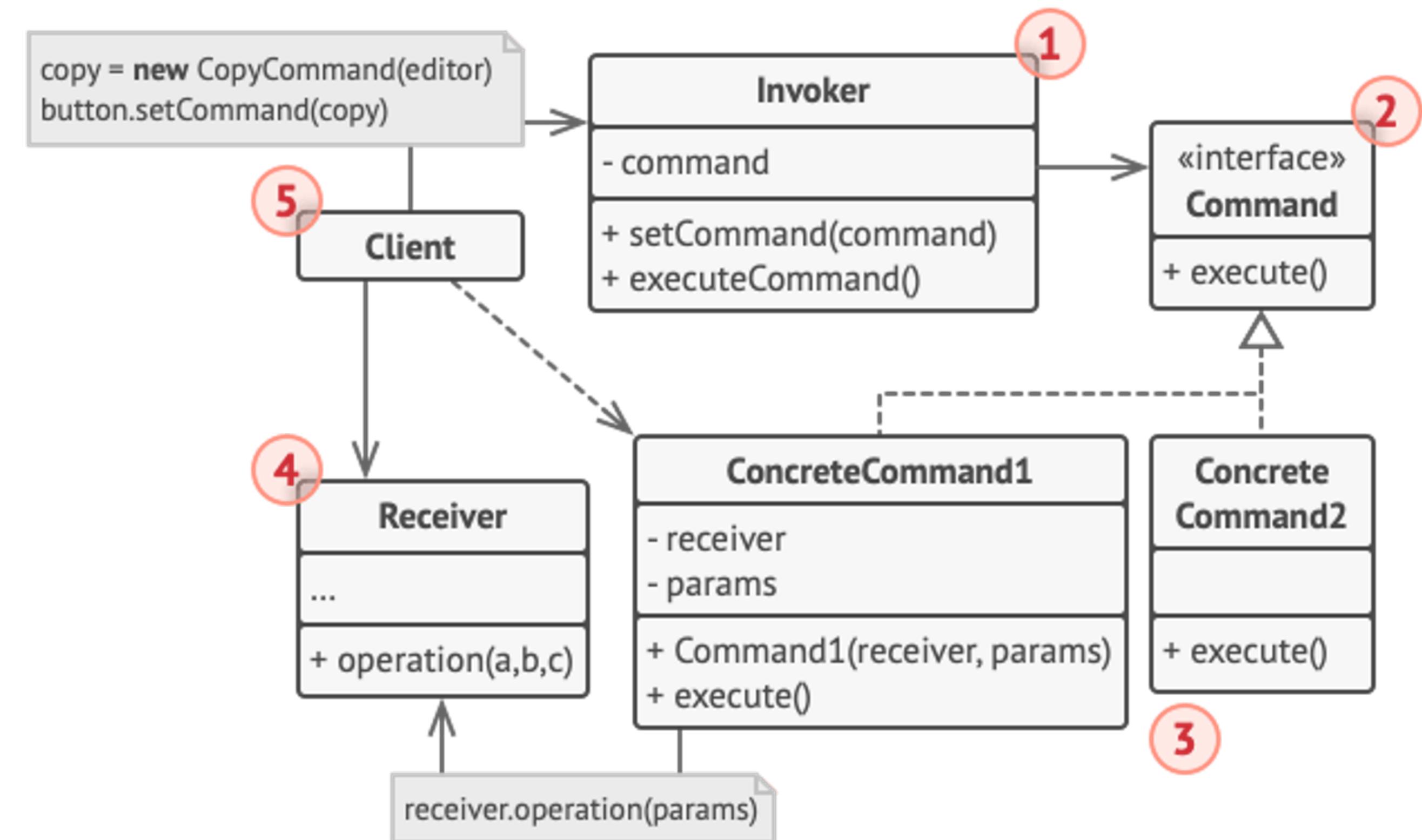
2. **Команда** описывает общий для всех конкретных команд интерфейс. Обычно здесь описан всего один метод для запуска команды.

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.



## Кейсы для использования

- Когда надо передавать в качестве параметров определенные действия, вызываемые в ответ на другие действия. То есть когда **необходимы функции обратного действия** в ответ на определенные действия.
- Когда необходимо обеспечить выполнение **очереди запросов**, а также их возможную отмену.
- Когда надо поддерживать **логгирование** изменений в результате запросов. Использование логов может помочь восстановить состояние системы - для этого необходимо будет использовать последовательность запротоколированных команд.

# Кейсы для использования

## Когда вы хотите параметризовать объекты выполняемым действием.

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

## 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.

## **Плюсы/минусы**

- + 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.

# Дополнительные моменты

## Стоит отличать от Стратегии

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.

## Но в принципе он похож на Посетитель

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

# Пример. Сеанс работы с системой учета

## ***Тот случай, когда нужно реализовать отмену операций***

Студенты ВШЭ, занимающиеся продажей автомобилей, заметили, что часто из-за спешки и невнимательности допускаются ошибки при учете автомобилей и покупателей в системе. Для решения данной проблемы они просят добавить сеансы работы с системой учета. Сеанс должен позволять следующее:

- Добавлять новые автомобили и покупателей;
- Отменить последнее действие;
- Просмотреть список внесенных изменений;
- Сохранить внесенные изменения.

*При этом все производимые действия должны попадать в отчет о работе системы.*

# Стратегия

# Что такое Стратегия?

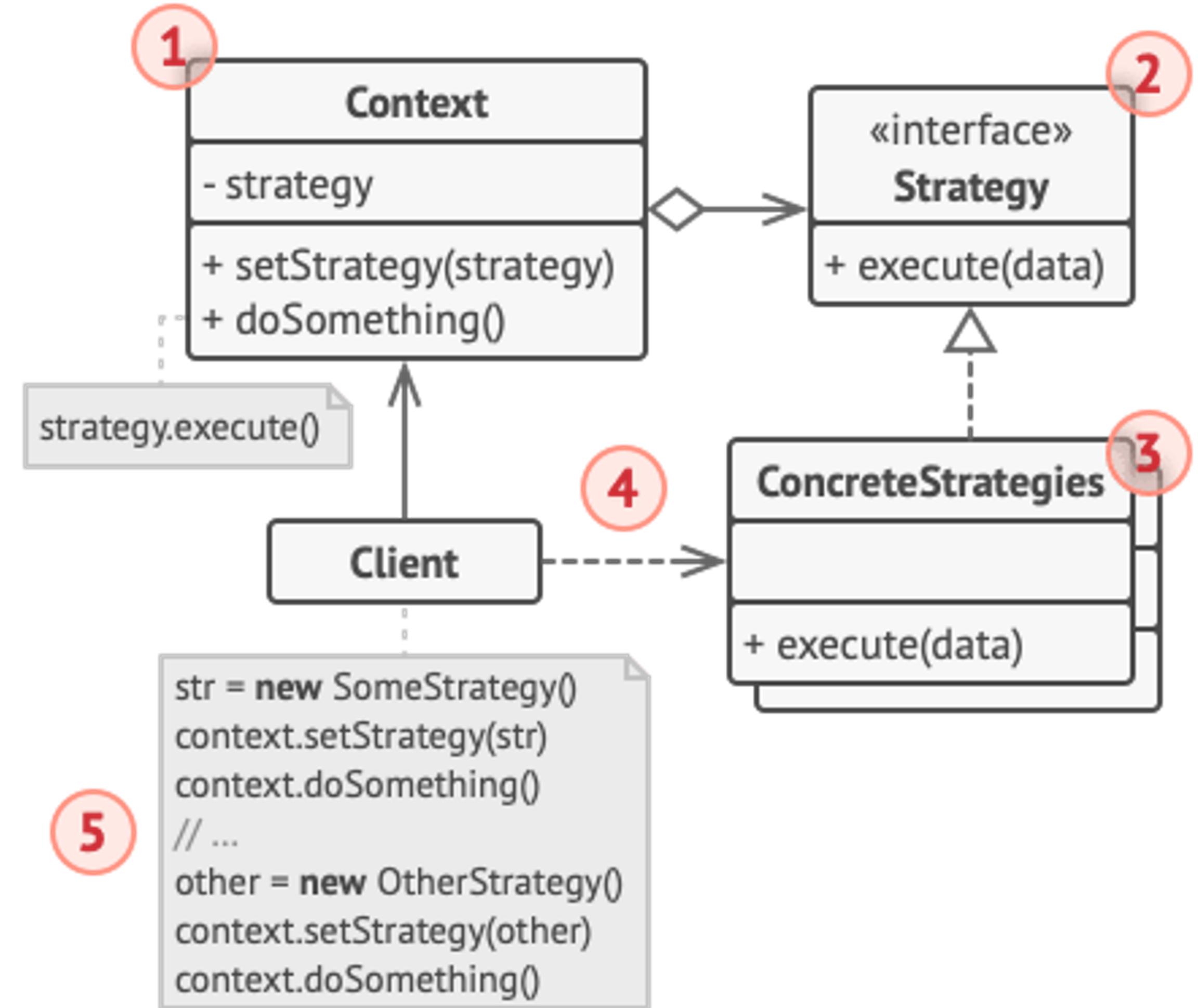
**Паттерн Стратегия (Strategy)** представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость.

В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.



# Как использовать?

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.



## Кейсы для использования

- Когда есть **несколько родственных классов, которые отличаются поведением.**  
Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять
- Когда необходимо **обеспечить выбор** из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий
- Когда необходимо менять поведение объектов **на стадии выполнения** программы
- Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

## Плюсы/минусы

- + 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.

# Дополнительные моменты

## Стоит отличать от Декоратора

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

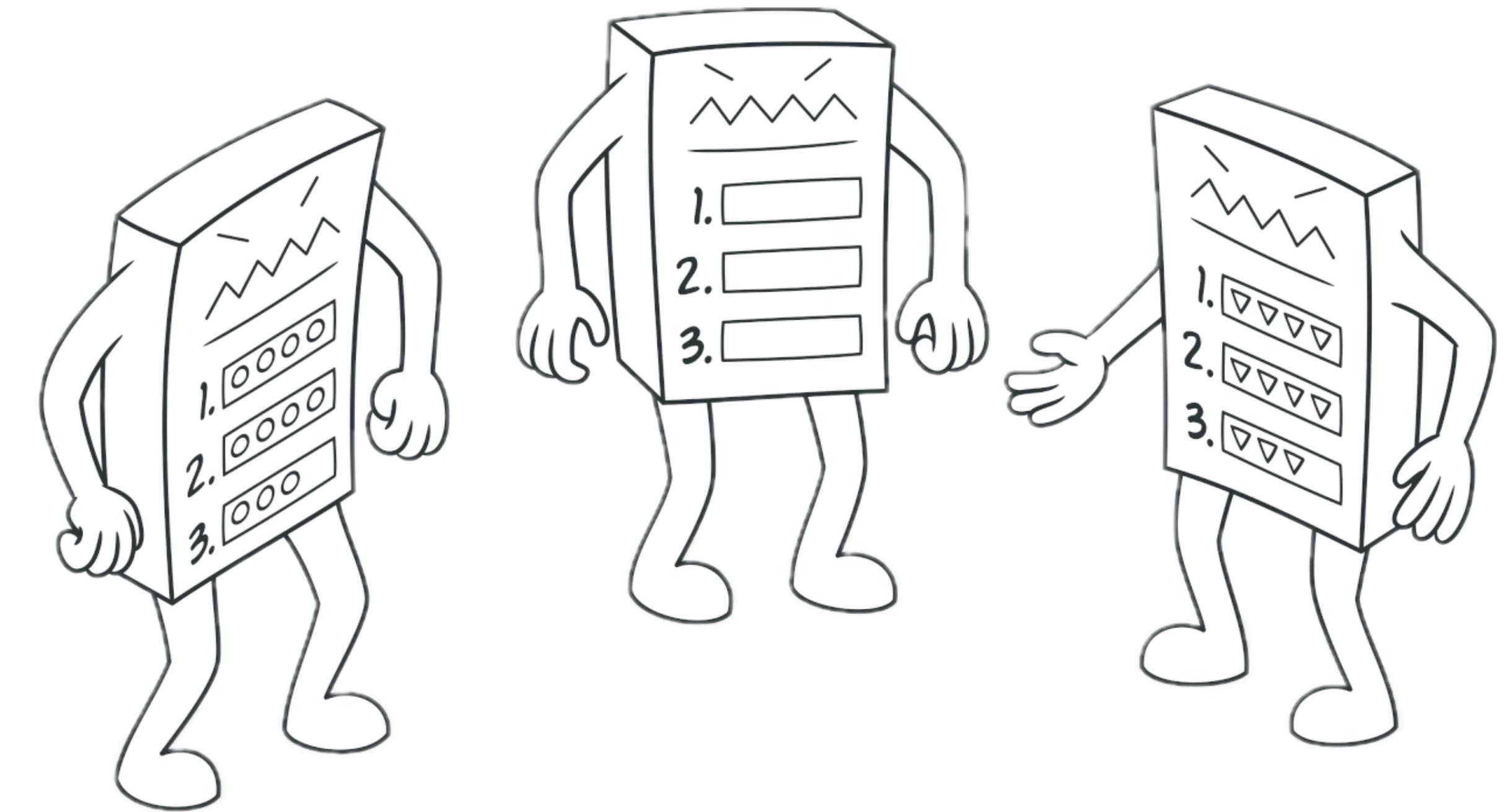
## Отношение к State

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.

# Template Method

# Что такое Template Method?

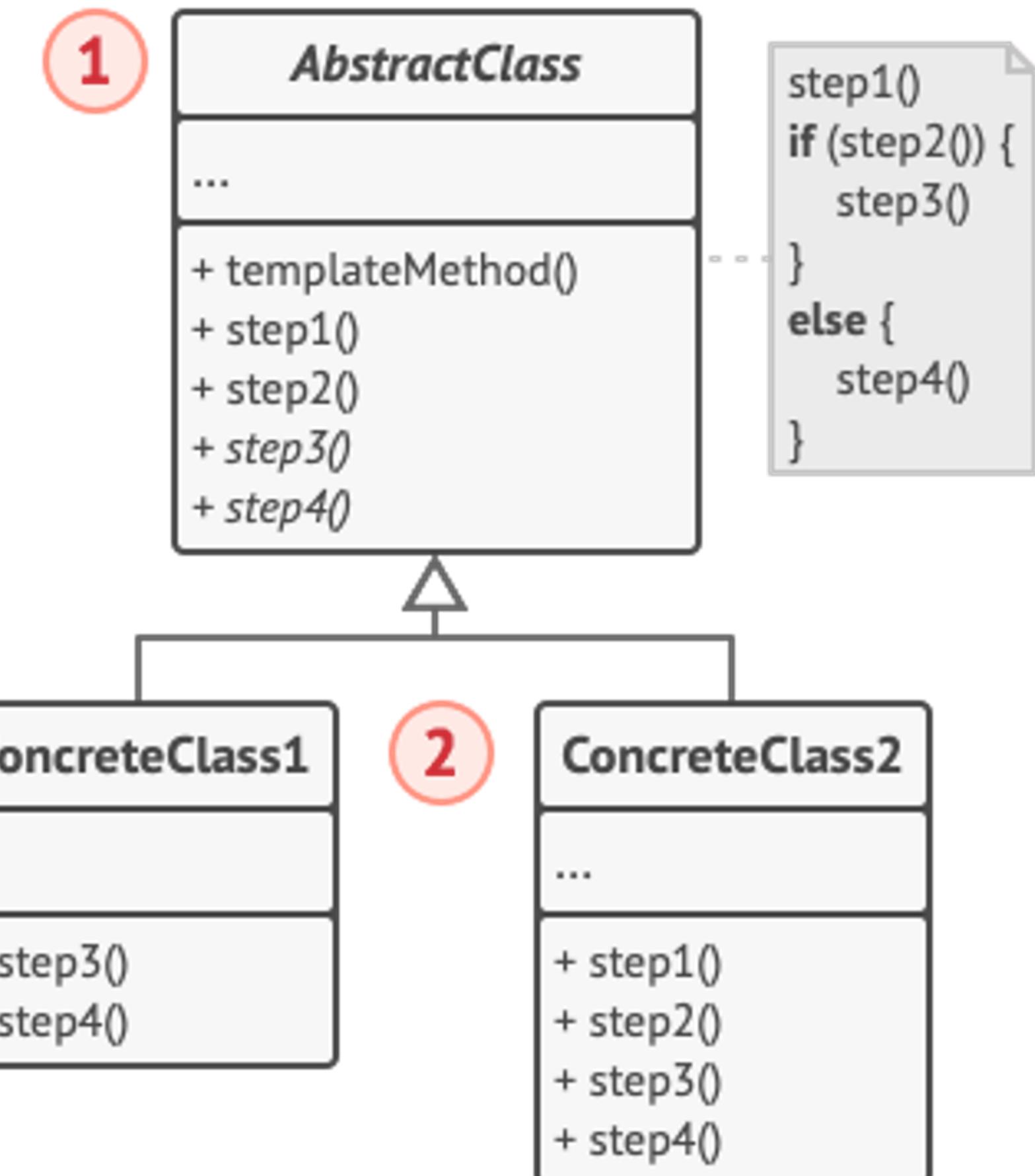
**Шаблонный метод (Template Method)** определяет общий алгоритм поведения подклассов, позволяя им переопределить отдельные шаги этого алгоритма без изменения его структуры.



# Как использовать?

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.



## Кейсы для использования

Когда планируется, что в будущем подклассы должны **будут переопределять различные этапы алгоритма** без изменения его структуры

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

Когда в классах, реализующим схожий алгоритм, **происходит дублирование кода**. Вынесение общего кода в шаблонный метод уменьшит его дублирование в подклассах.

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..

## Плюсы/минусы

- + 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.

# Дополнительные моменты

## **Стоит отличать от Фабричного метода.**

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

## **Отличается от стратегии**

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.

## Пример. Форматы выгрузки отчетов

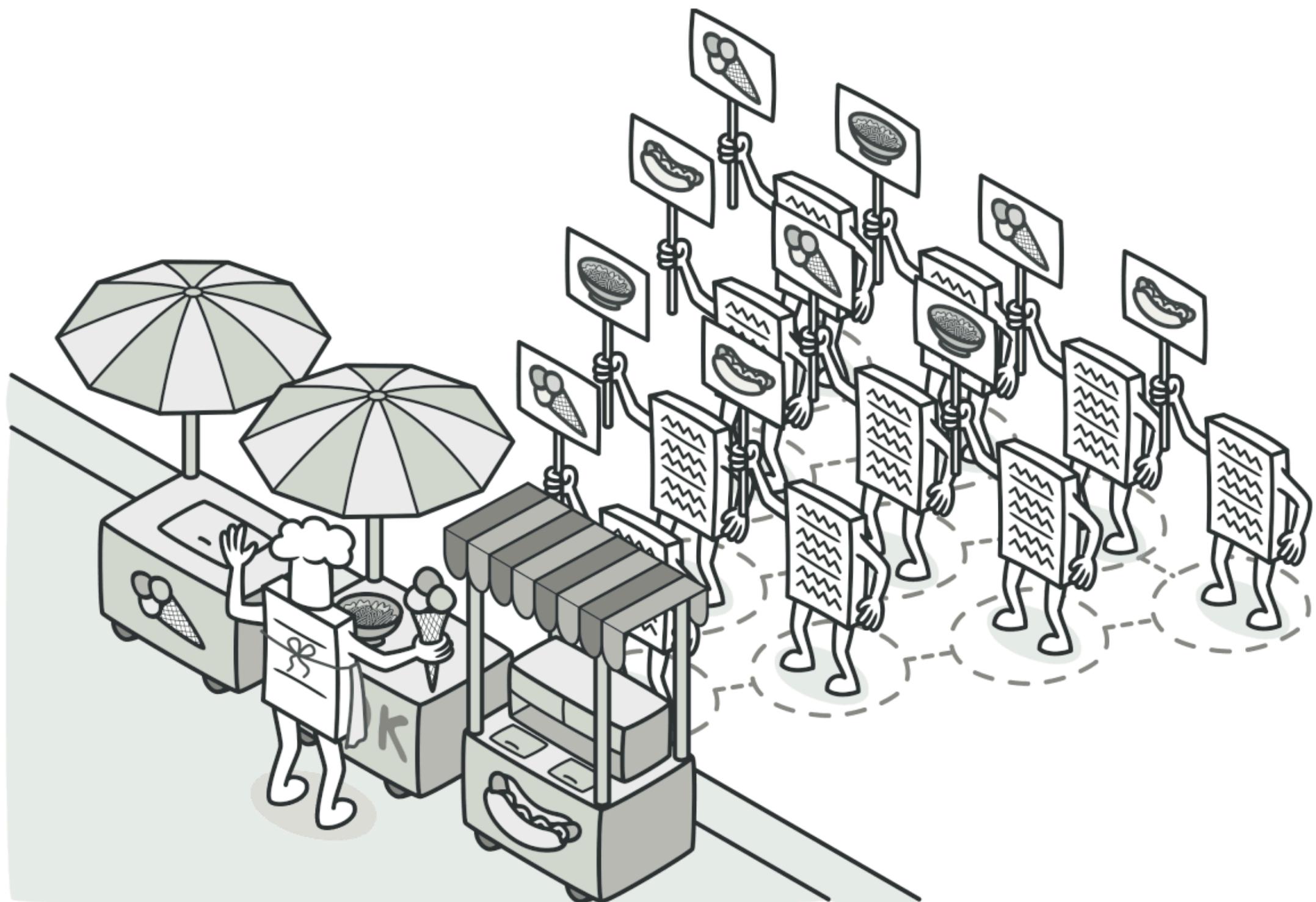
В нашем приложении мы можем использовать несколько форматов выгрузки отчета.

- JSON;
- Markdown.

Для реализации данного функционала можно использовать паттерн “Шаблонный метод”.

# Visitor

# Что такое Visitor?



**Паттерн Посетитель (Visitor)** позволяет определить операцию для объектов других классов без изменения этих классов.

При использовании паттерна Посетитель определяются две иерархии классов: одна для элементов, для которых надо определить новую операцию, и вторая иерархия для посетителей, описывающих данную операцию.

# Как использовать?

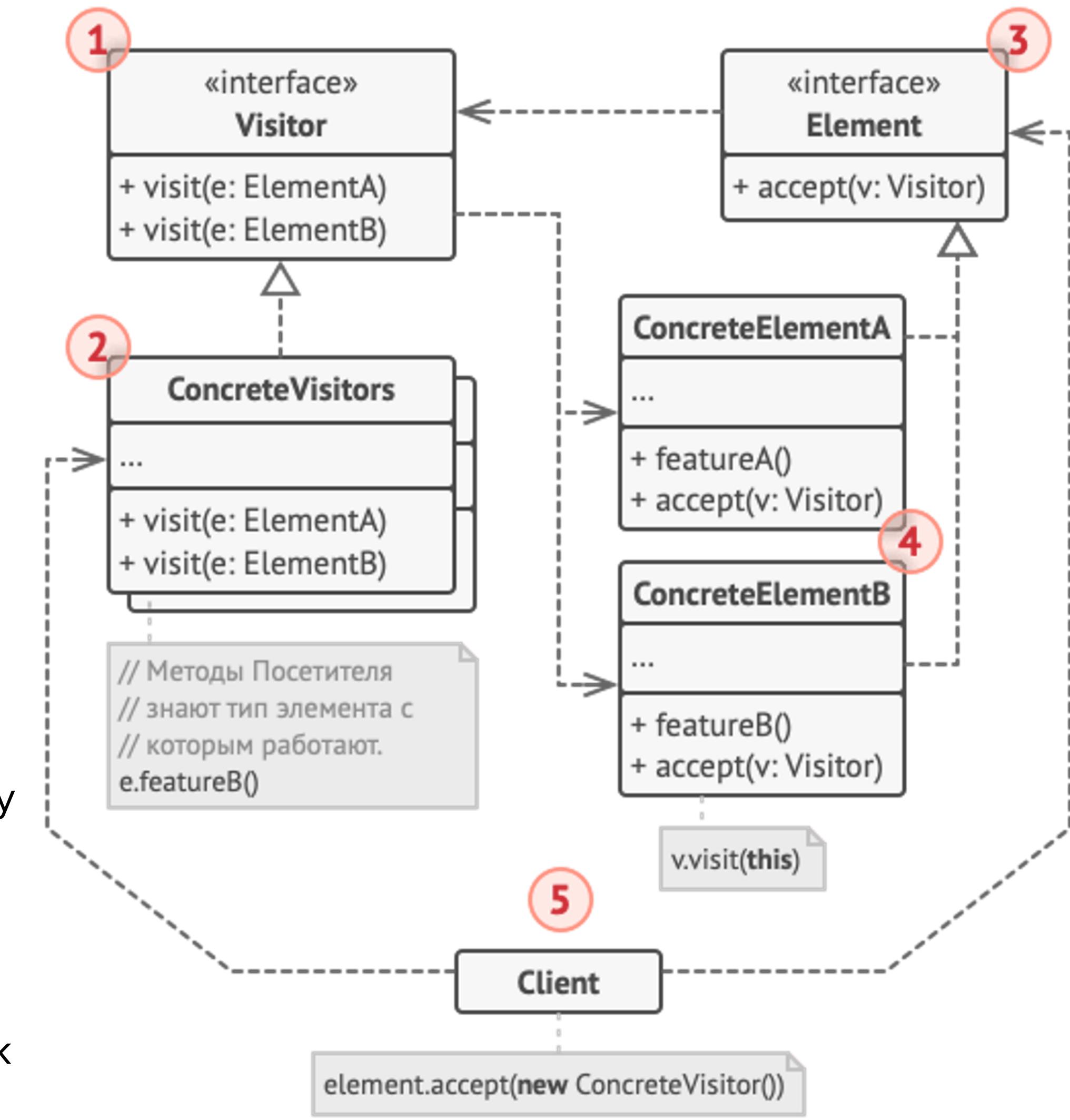
1. **Посетитель** описывает общий интерфейс для всех типов посетителей. Он объявляет набор методов, отличающихся типом входящего параметра, которые нужны для запуска операции для всех типов конкретных элементов. В языках, поддерживающих перегрузку методов, эти методы могут иметь одинаковые имена, но типы их параметров должны отличаться.

2. **Конкретные посетители** реализуют какое-то особенное поведение для всех типов элементов, которые можно подать через методы интерфейса посетителя.

3. **Элемент** описывает метод принятия посетителя. Этот метод должен иметь единственный параметр, объявленный с типом интерфейса посетителя.

4. **Конкретные элементы** реализуют методы принятия посетителя. Цель этого метода — вызвать тот метод посещения, который соответствует типу этого элемента. Так посетитель узнает, с каким именно элементом он работает.

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



## Кейсы для использования

Когда имеется **много объектов разнородных классов с разными интерфейсами**, и требуется выполнить ряд операций над каждым из этих объектов

Когда классам необходимо добавить одинаковый набор операций **без изменения** этих классов

Когда **часто добавляются новые операции** к классам, при этом общая структура классов стабильна и практически не изменяется

# Кейсы для использования

**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.

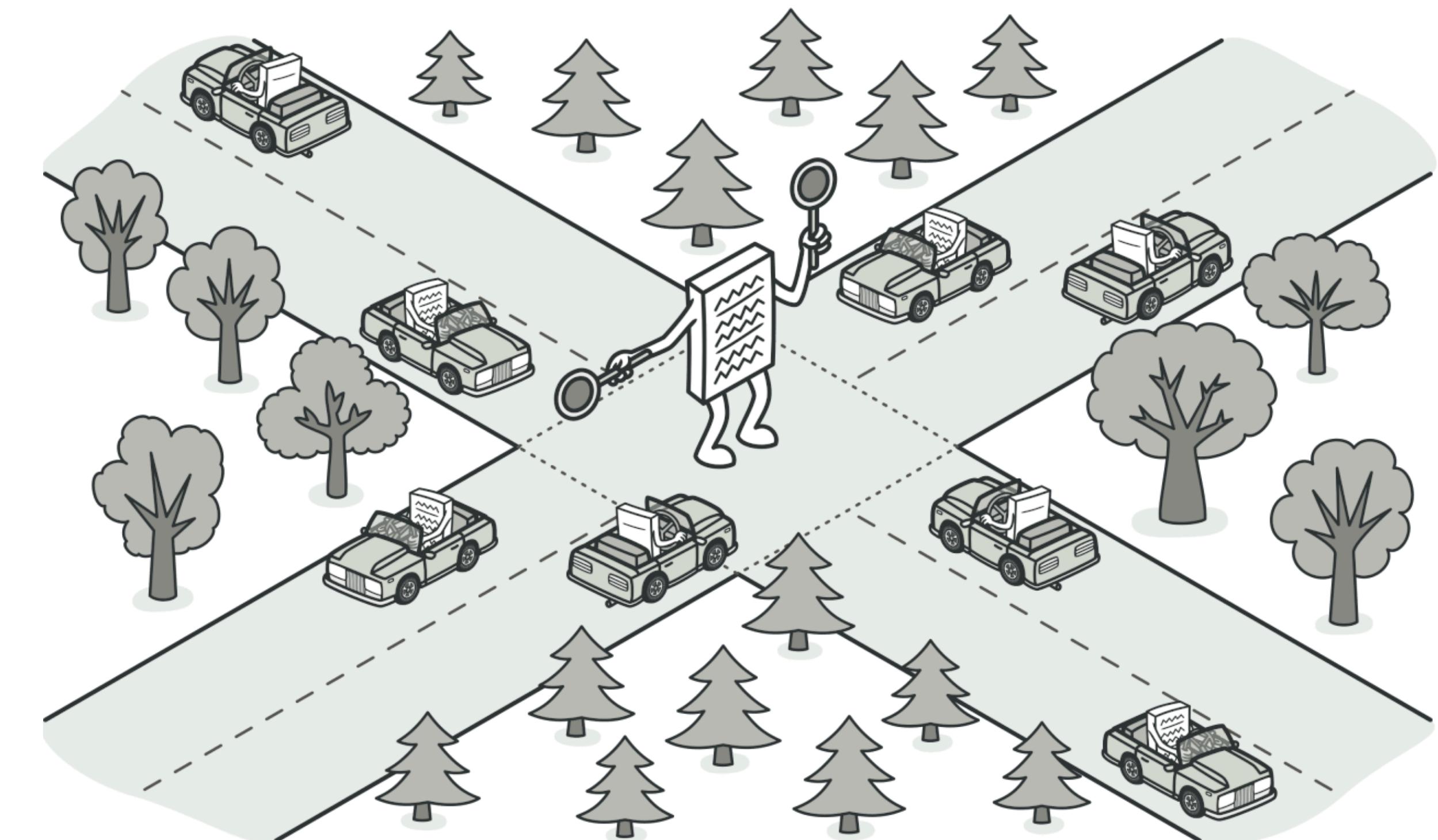
## Плюсы/минусы

- + Упрощает добавление операций, работающих со сложными структурами объектов.
- + Объединяет родственные операции в одном классе.
- + Посетитель может накапливать состояние при обходе структуры элементов.
- Паттерн не оправдан, если иерархия элементов часто меняется.
- Может привести к нарушению инкапсуляции элементов.

# Mediator

# Что такое Mediator?

**Паттерн Посредник (Mediator)** представляет такой шаблон проектирования, который обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга. Тем самым достигается слабосвязанность взаимодействующих объектов.



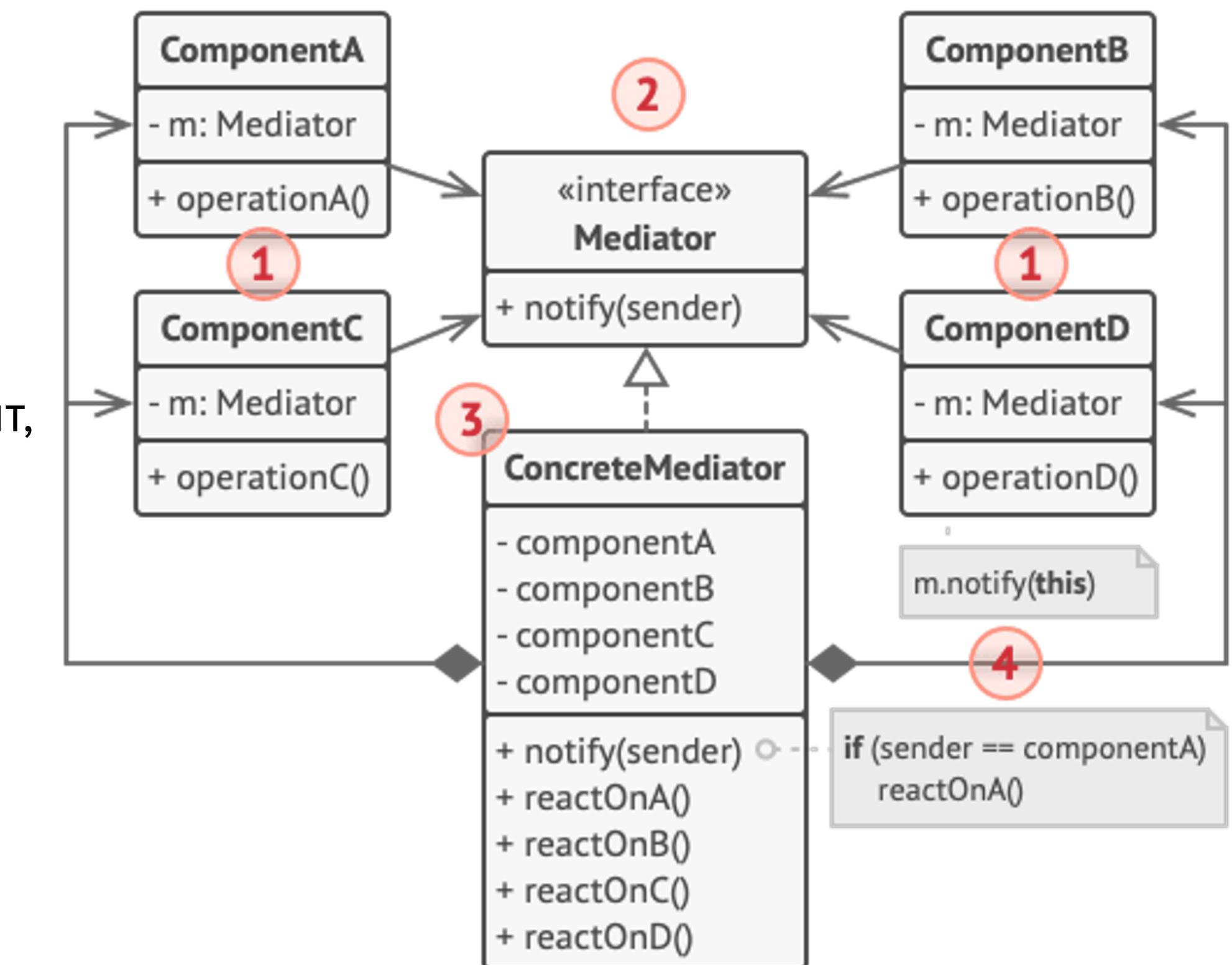
# Как использовать?

1. **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.

2. **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.

3. **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.

4. 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.



## Кейсы для использования

**Когда имеется множество взаимосвязанных объектов, связи между которыми сложны и запутаны.**

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

**Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.**

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.

## **Плюсы/минусы**

- + Устраняет зависимости между компонентами, позволяя повторно их использовать.
- + Упрощает взаимодействие между компонентами.
- + Централизует управление в одном месте.
  
- Посредник может сильно раздуться..

# Дополнительные моменты

## Стоит отличать от Наблюдателя

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

Цель Посредника — убрать обюдные 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.