
# ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

# Что такое Паттерн?

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

# 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?

# Три основные группы паттернов

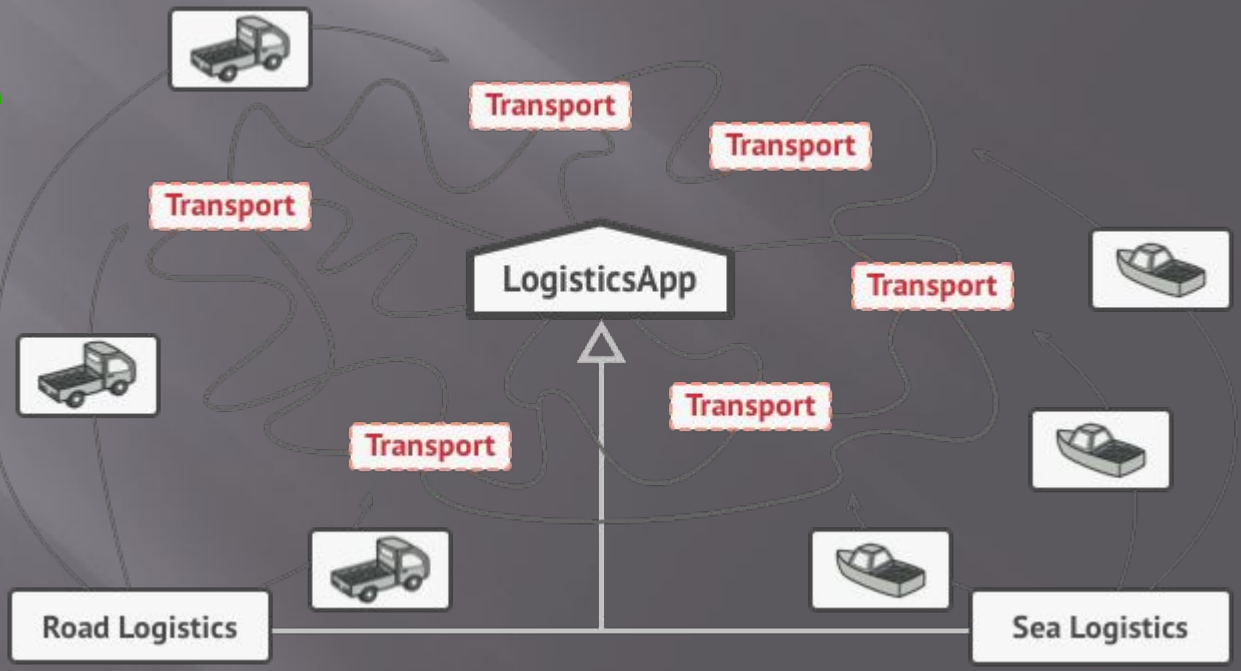
- **Порождающие паттерны** беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.
- **Структурные паттерны** показывают различные способы построения связей между объектами.
- **Поведенческие паттерны** заботятся об эффективной коммуникации между объектами.

# Фабричный метод

Порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

# Проблемы


# Решение



# Преимущества и недостатки

Избавляет класс от привязки к конкретным классам продуктов.

Выделяет код производства продуктов в одно место, упрощая поддержку кода.

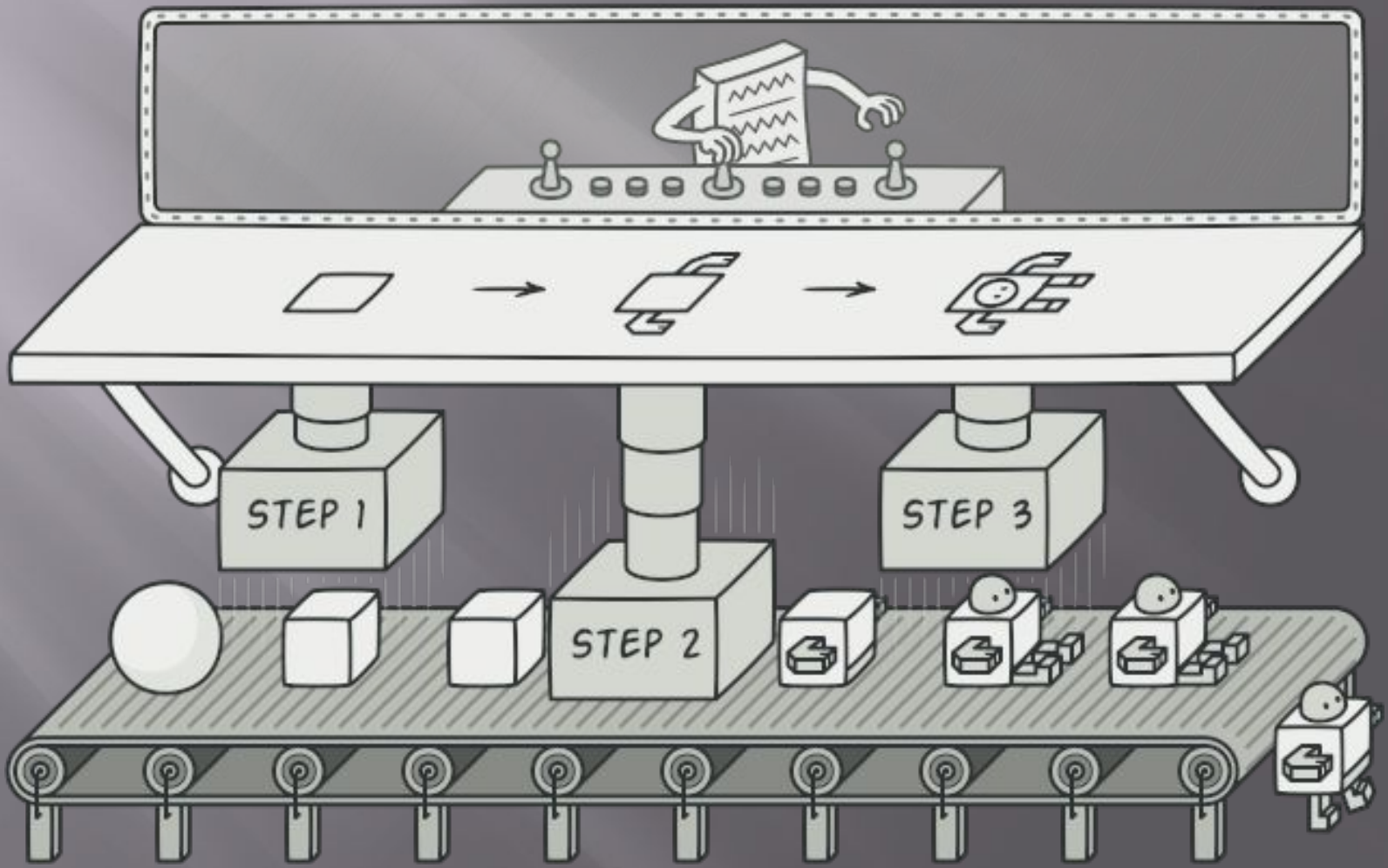
Упрощает добавление новых продуктов в программу.

Реализует *принцип открытости/закрытости*.

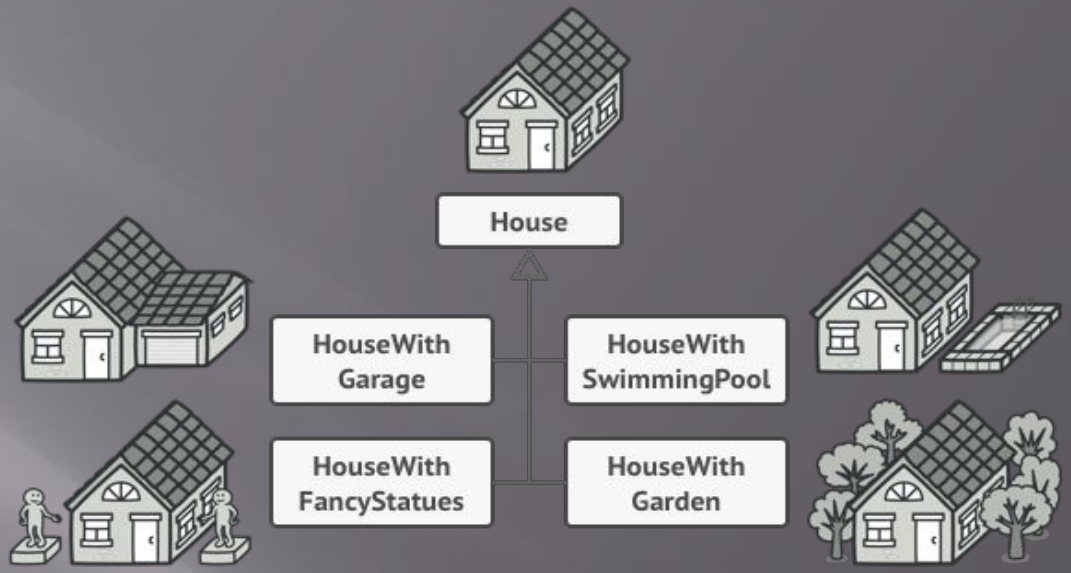
Может привести к созданию больших **параллельных иерархий классов**, так как для каждого класса продукта надо создать свой подкласс создателя.

# Строитель

**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


House
HouseWith Garage
HouseWith SwimmingPool
HouseWith FancyStatues
HouseWith Garden


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

# Решение


Я ПРИНЁС ТЕБЕ СТРОИТЕЛЯ.
ТЫ ЗНАЕШЬ, ЧТО ДЕЛАТЬ.
ОК, БОСС!


РЕБЯТА, МНЕ НУЖЕН ДОМ С 4-МЯ СТЕНАМИ, 1-Й ДВЕРЬЮ, 2-МЯ ОКНАМИ, ...

# Преимущества и недостатки

Позволяет создавать продукты пошагово.

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

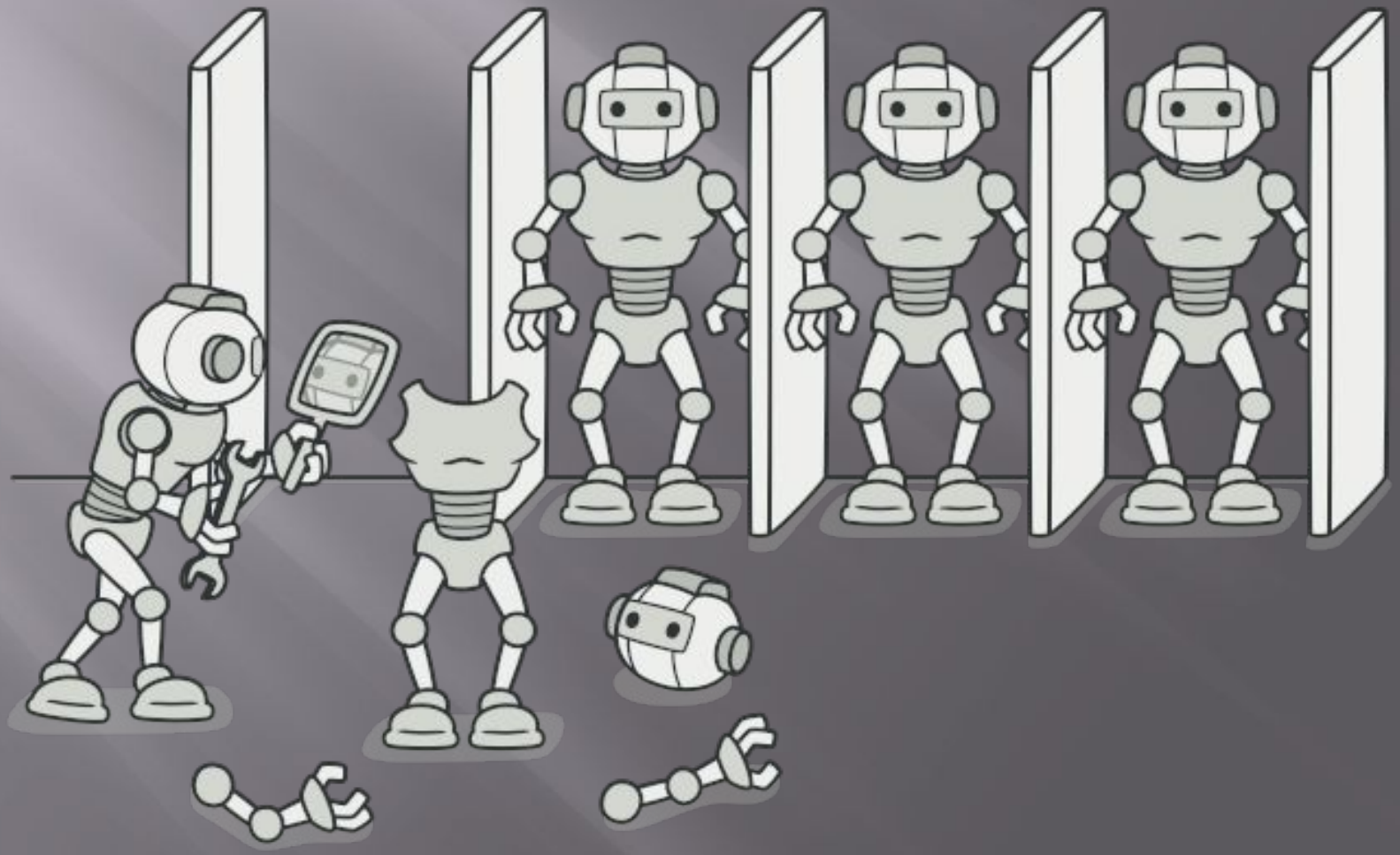
Изолирует сложный код сборки продукта от его основной бизнес-логики.

Усложняет код программы из-за введения дополнительных классов.

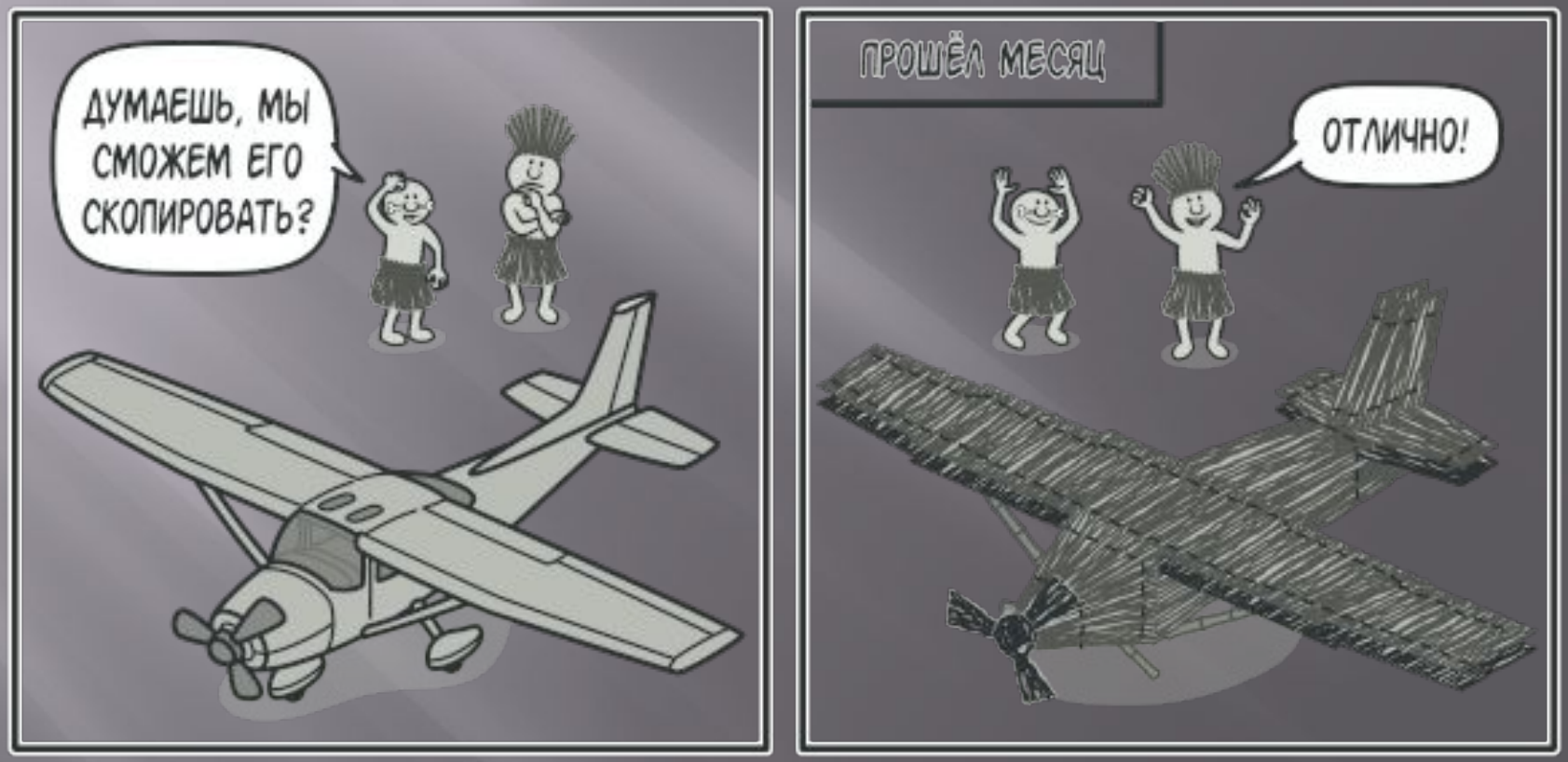
Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя может не быть метода получения результата.

# Прототип

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

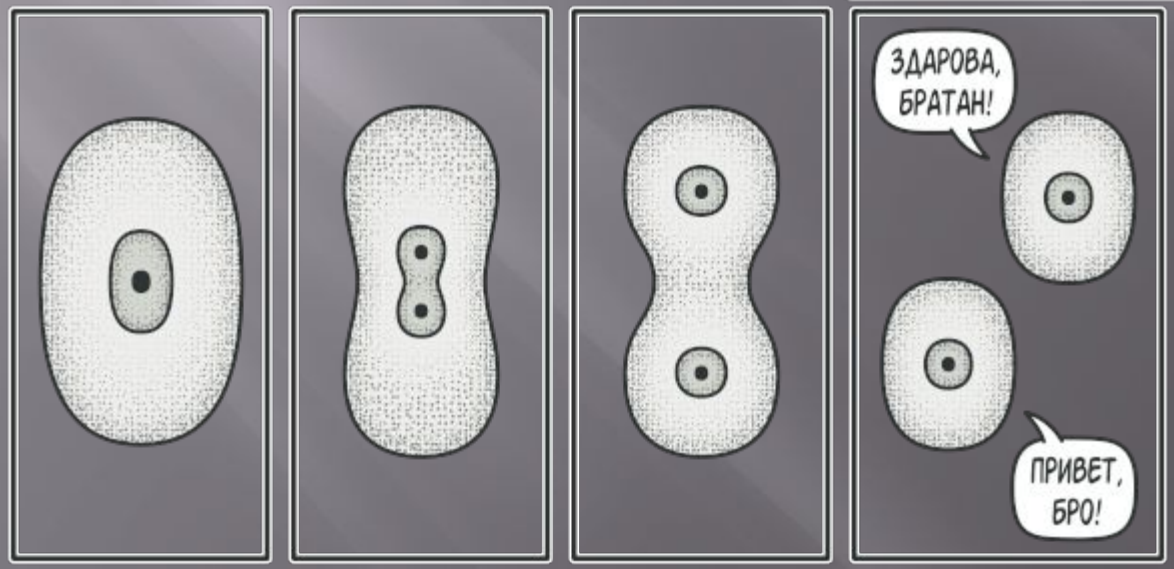


# Проблема



# Решение


ОКЕЙ, СЕЙЧАС СДЕЛАЕМ ПАРОЧКУ КЛОНОВ.
CONFIG A
CONFIG B
CONFIG C


ЗДАРОВА, БРАТАН!
ПРИВЕТ, БРО!

# Преимущества и недостатки

Позволяет клонировать объекты, не привязываясь к их конкретным классам.

Меньше повторяющегося кода инициализации объектов.

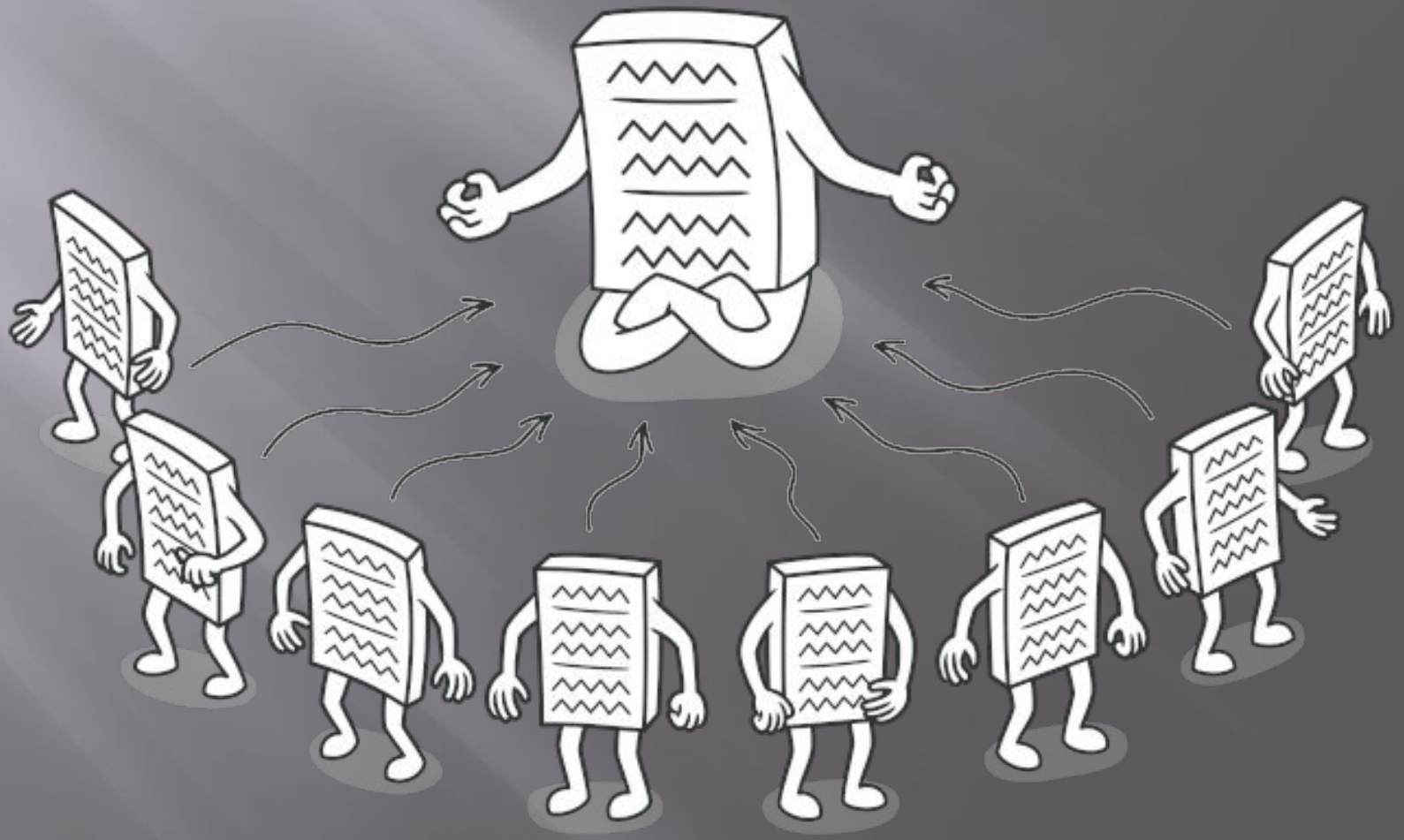
Ускоряет создание объектов.

Альтернатива созданию подклассов для конструирования сложных объектов.

Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

# Одиночка

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



**Гарантирует наличие единственного экземпляра класса**

- 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.

**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.

# Преимущества и недостатки

Гарантирует наличие единственного экземпляра класса.

Предоставляет к нему глобальную точку доступа.

Реализует отложенную инициализацию объекта-одиночки.

Нарушает *принцип единственной ответственности класса*.

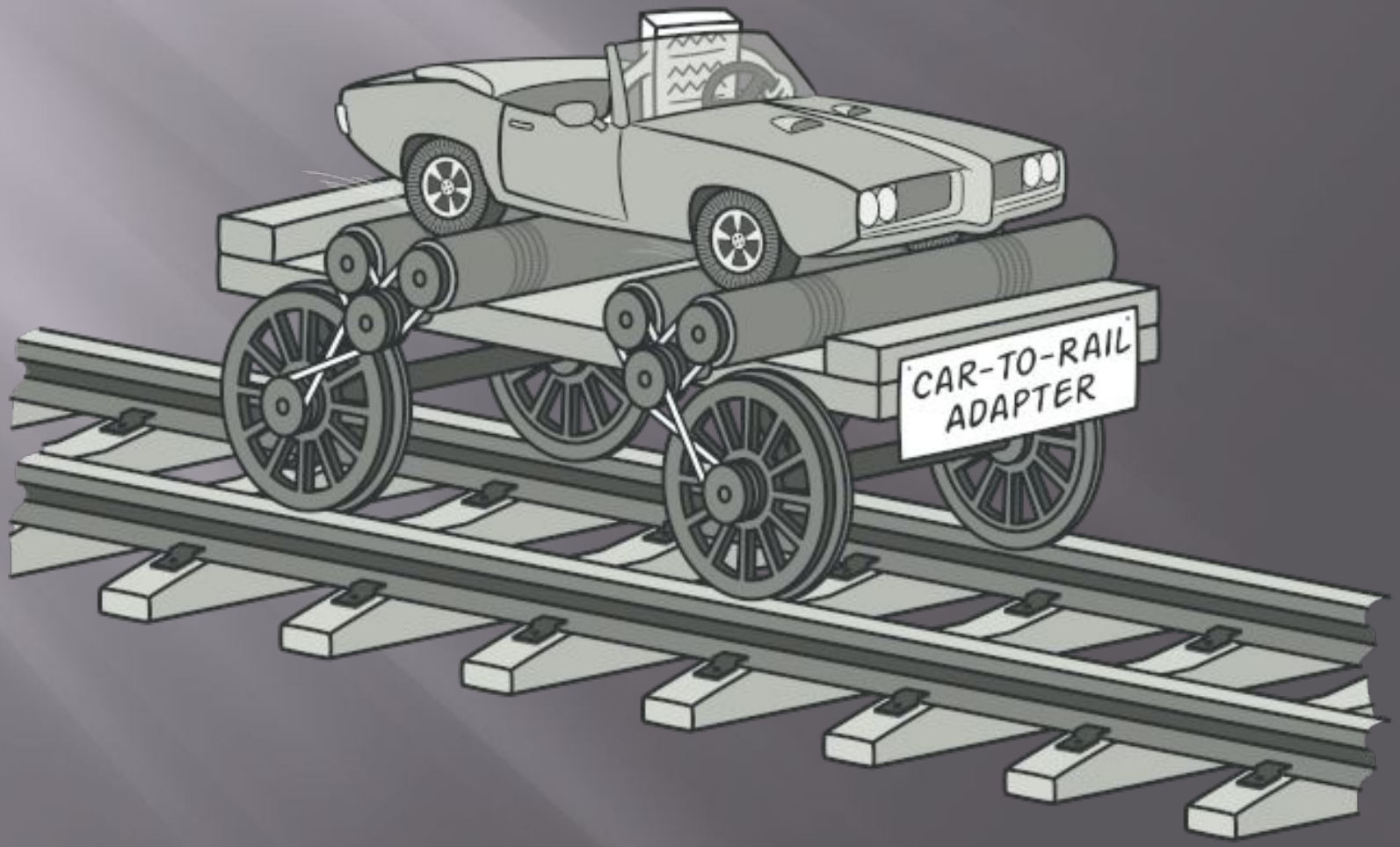
Маскирует плохой дизайн.

Проблемы мультипоточности.

Требует постоянного создания Mock-объектов при юнит-тестировании.

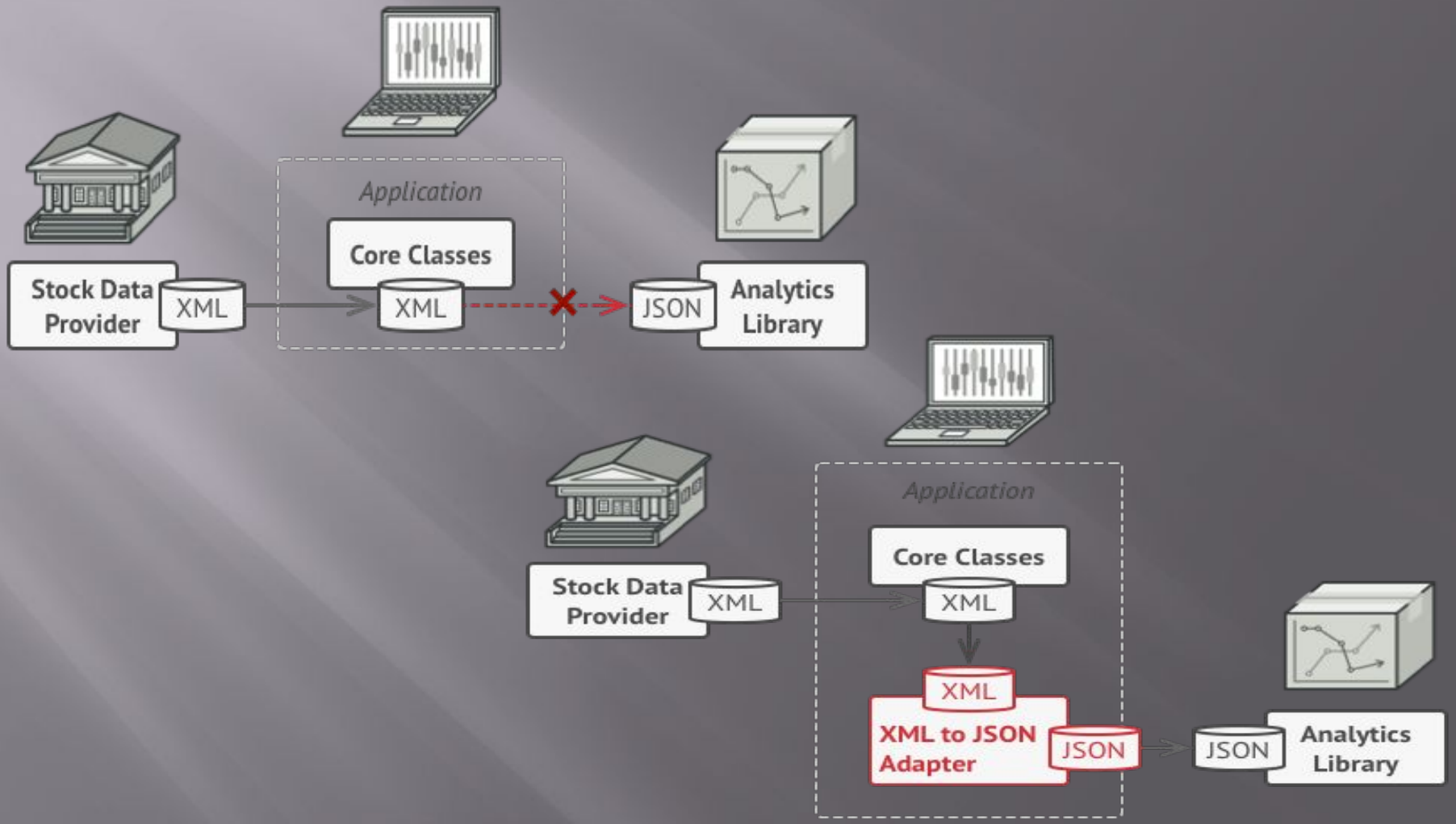
# Адаптер

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



# Проблема

# Решение



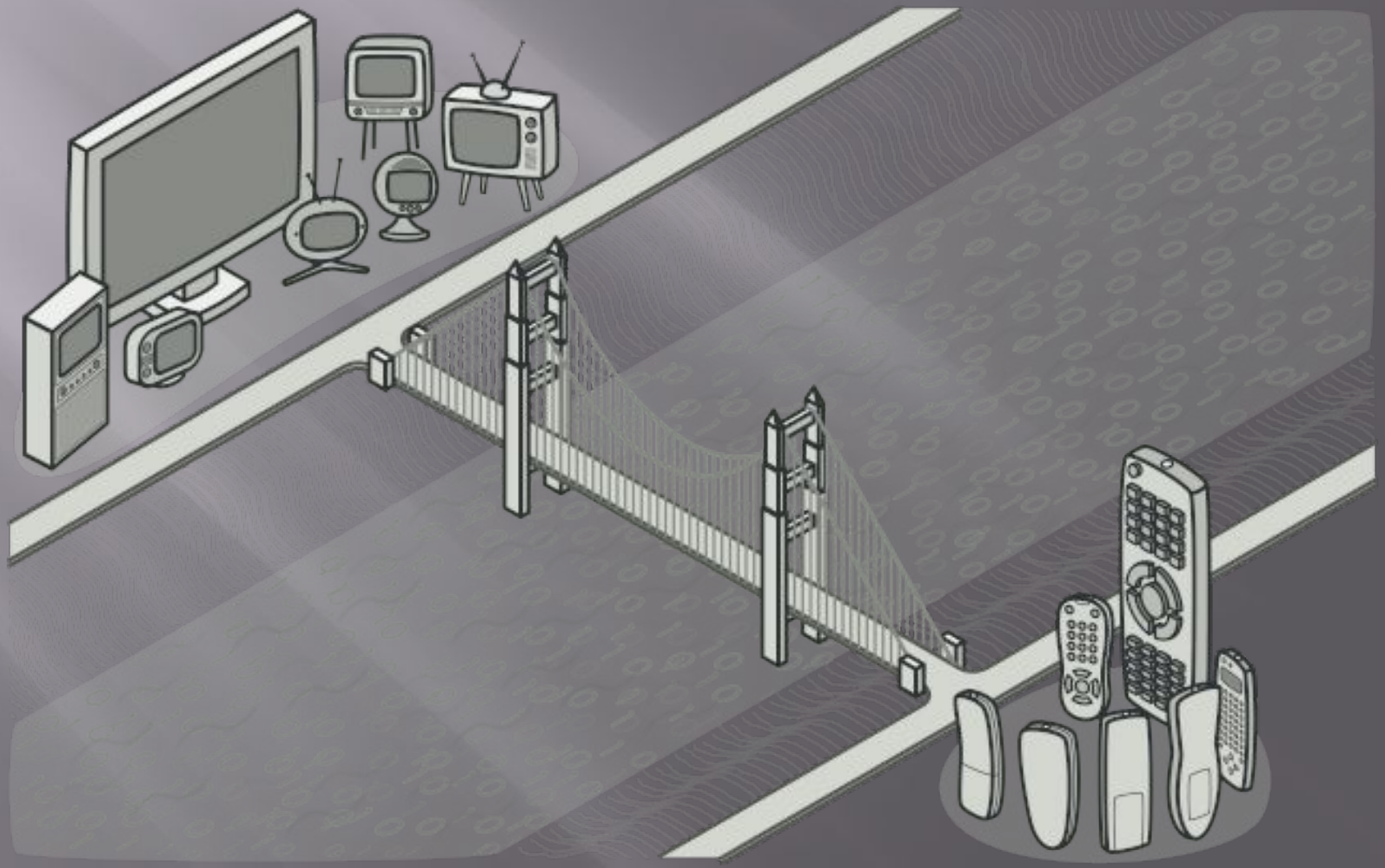
# Преимущества и недостатки

Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

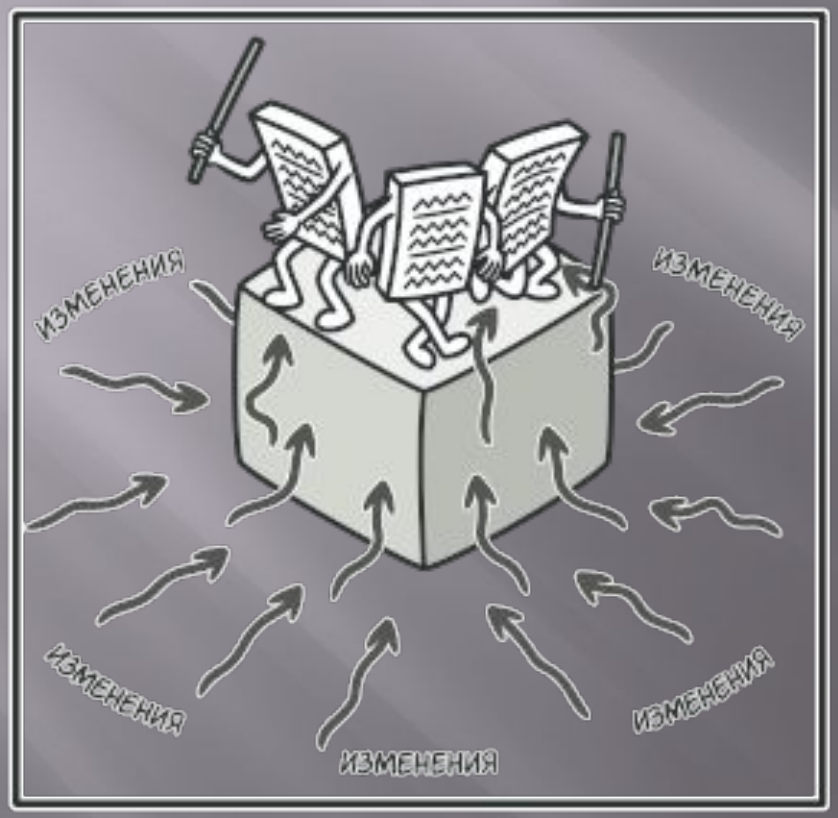
Усложняет код программы из-за введения дополнительных классов.

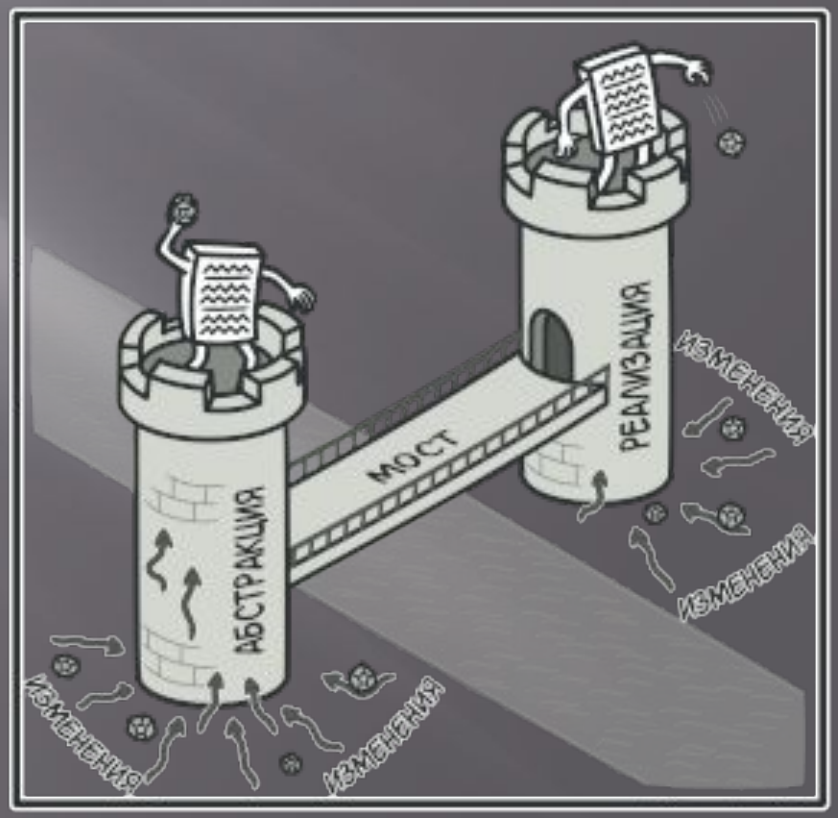
# Мост

**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.

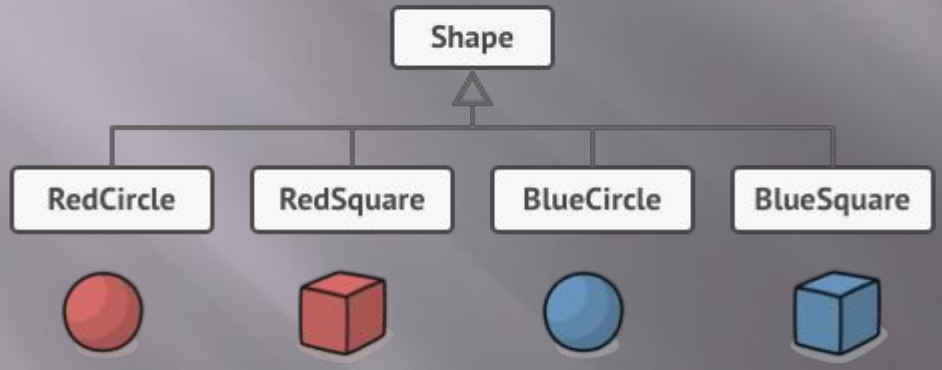


# Абстракция и Реализация

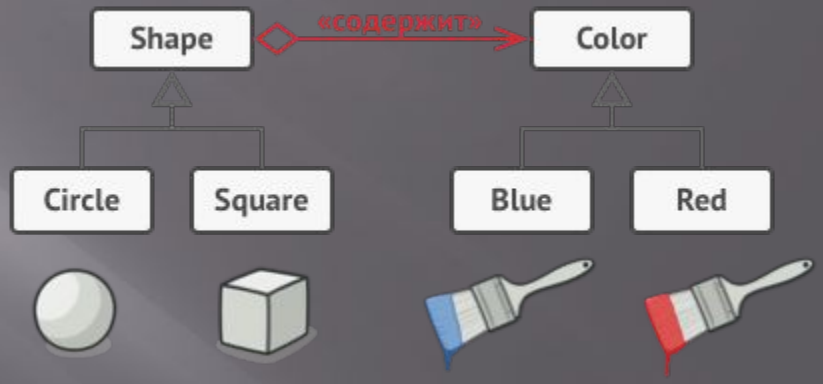


# Проблема


# Решение



# Преимущества и недостатки

Позволяет строить платформо-независимые программы.

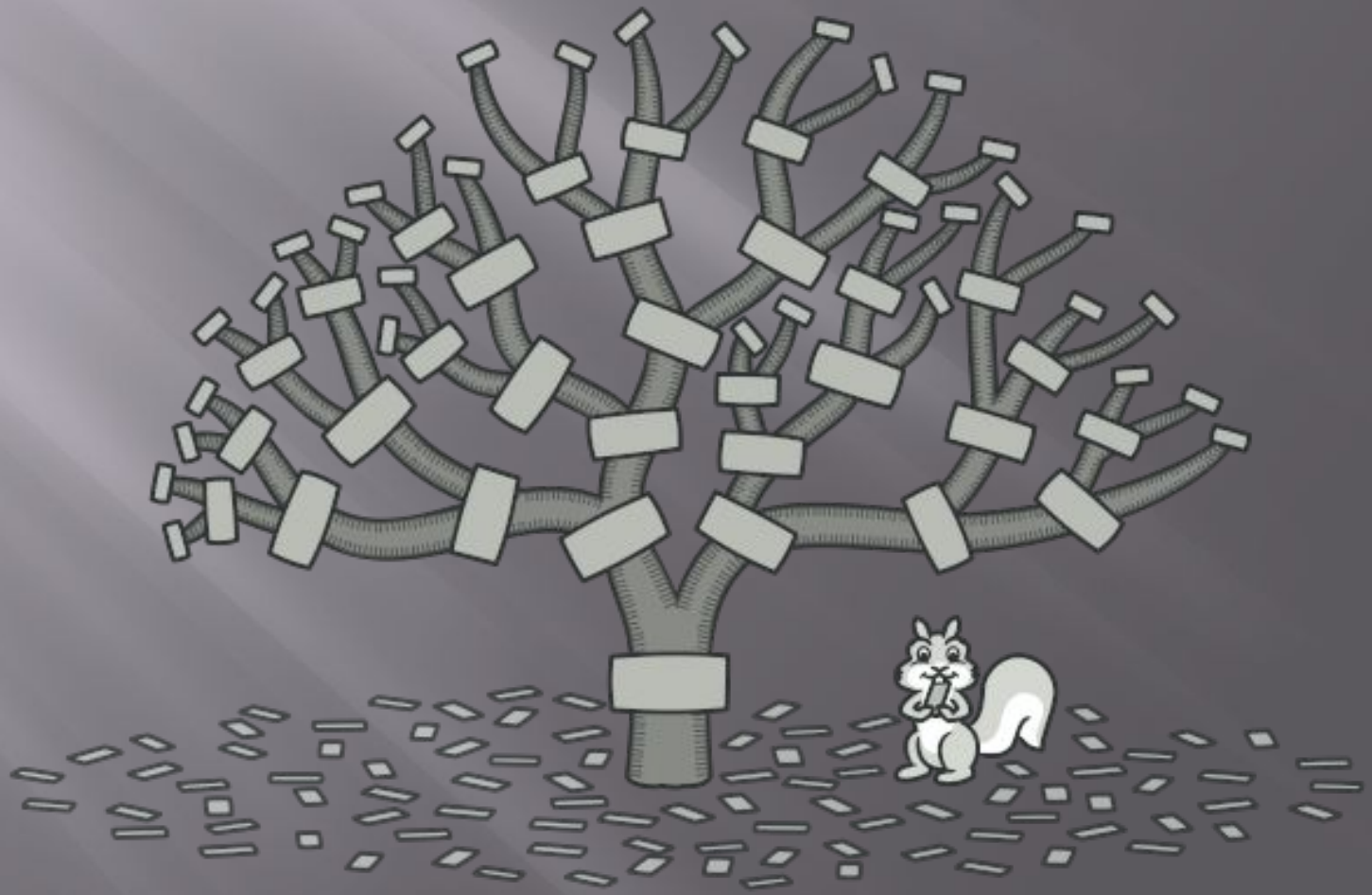
Скрывает лишние или опасные детали реализации от клиентского кода.

Реализует *принцип открытости/закрытости*.

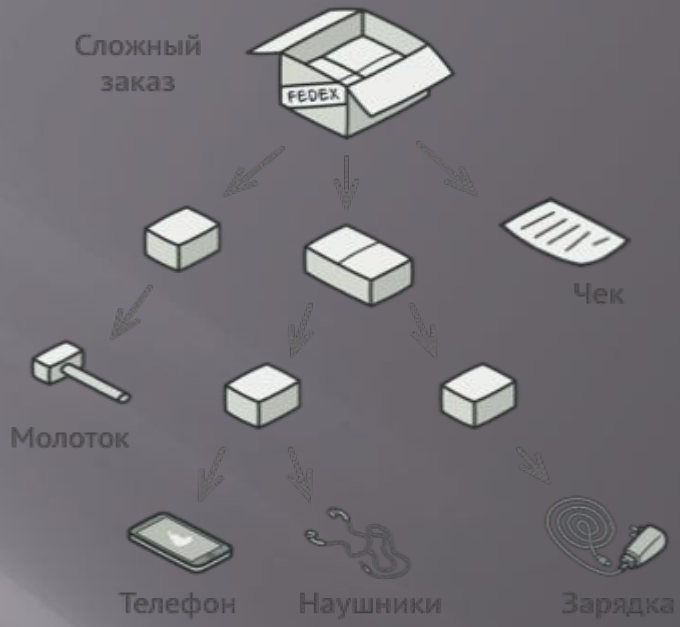
Усложняет код программы из-за введения дополнительных классов.

# Компоновщик

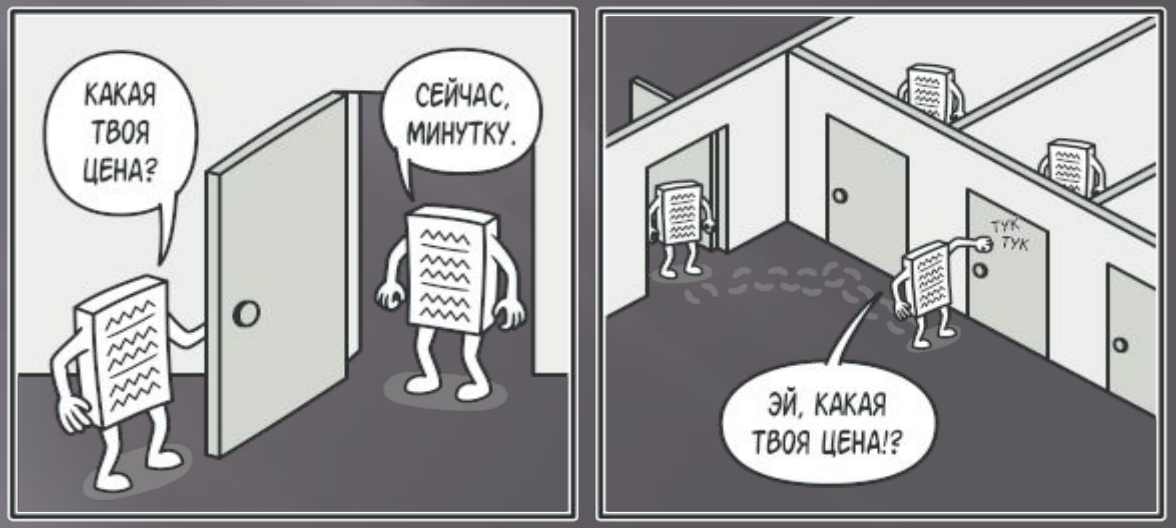
**Компоновщик** — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.



# Проблема




# Решение



# Преимущества и недостатки

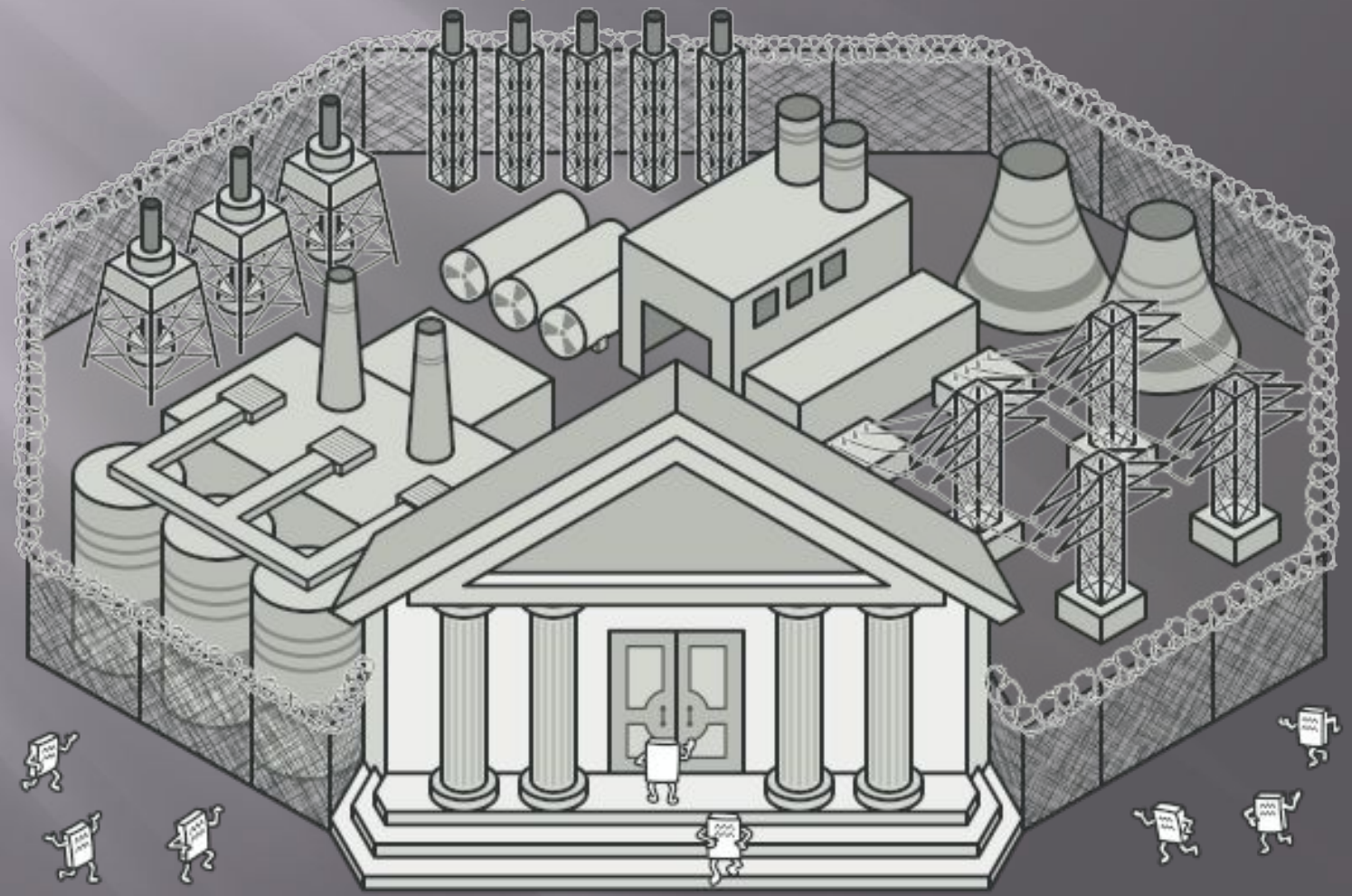
Упрощает архитектуру клиента при работе со сложным деревом компонентов.

Облегчает добавление новых видов компонентов.

Создаёт слишком общий дизайн классов.

# Фасад

**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.

# 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.

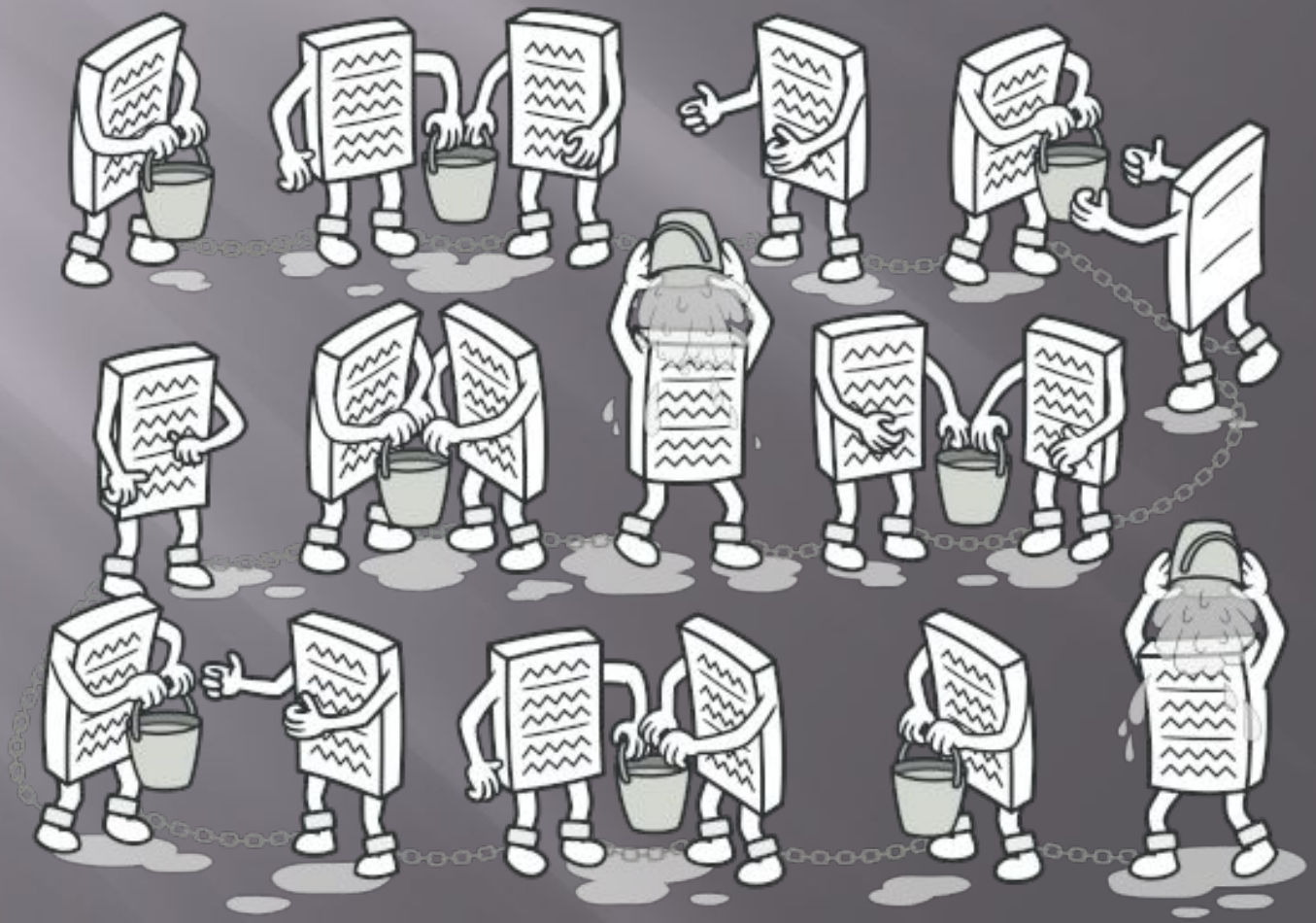
# Преимущества и недостатки

Изолирует клиентов от компонентов сложной подсистемы.

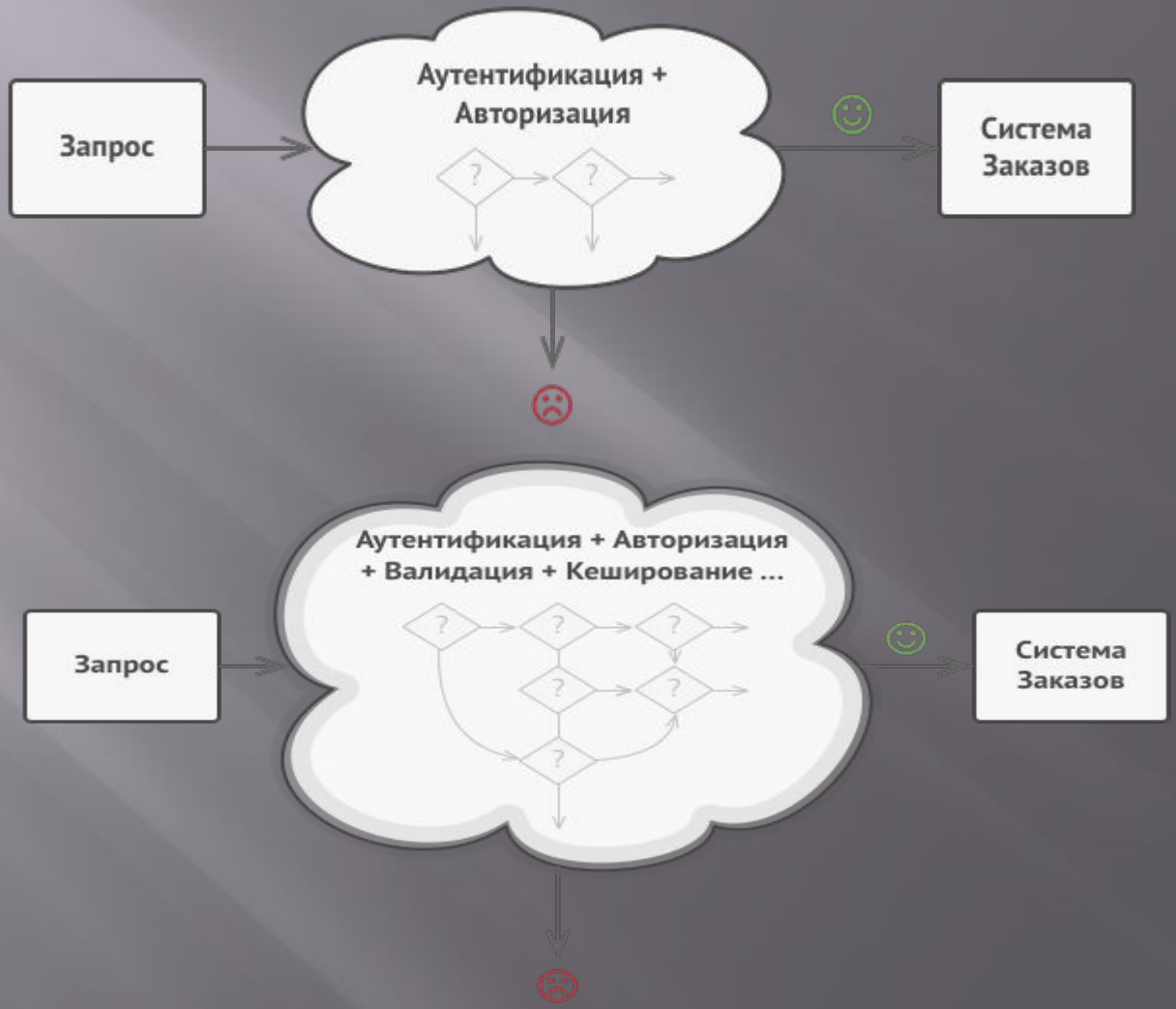
Фасад рискует стать **божественным объектом**, привязанным ко всем классам программы.

# Цепочка обязанностей

**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


Запрос
Аутентификация +
Авторизация
Система
Заказов
Запрос
Аутентификация + Авторизация
+ Валидация + Кеширование ...
Система
Заказов

# Решение


АЛЁ, МНЕ НУЖНА ТЕХ. ПОДДЕРЖКА.
НАЖМИТЕ 1, ЕСЛИ...
ВЫ ПРОБОВАЛИ ВЫКЛЮЧИТЬ И ВКЛЮЧИТЬ?
ОКЕЙ, ВОТ ГДЕ ВЫ МОЖЕТЕ СКАЧАТЬ НОВЫЕ ДРАЙВЕРА.


Цепочка обязанностей
Запрос
Обработчик
Обработчик
Обработчик
Система Заказов

# Преимущества и недостатки

Уменьшает зависимость между клиентом и обработчиками.

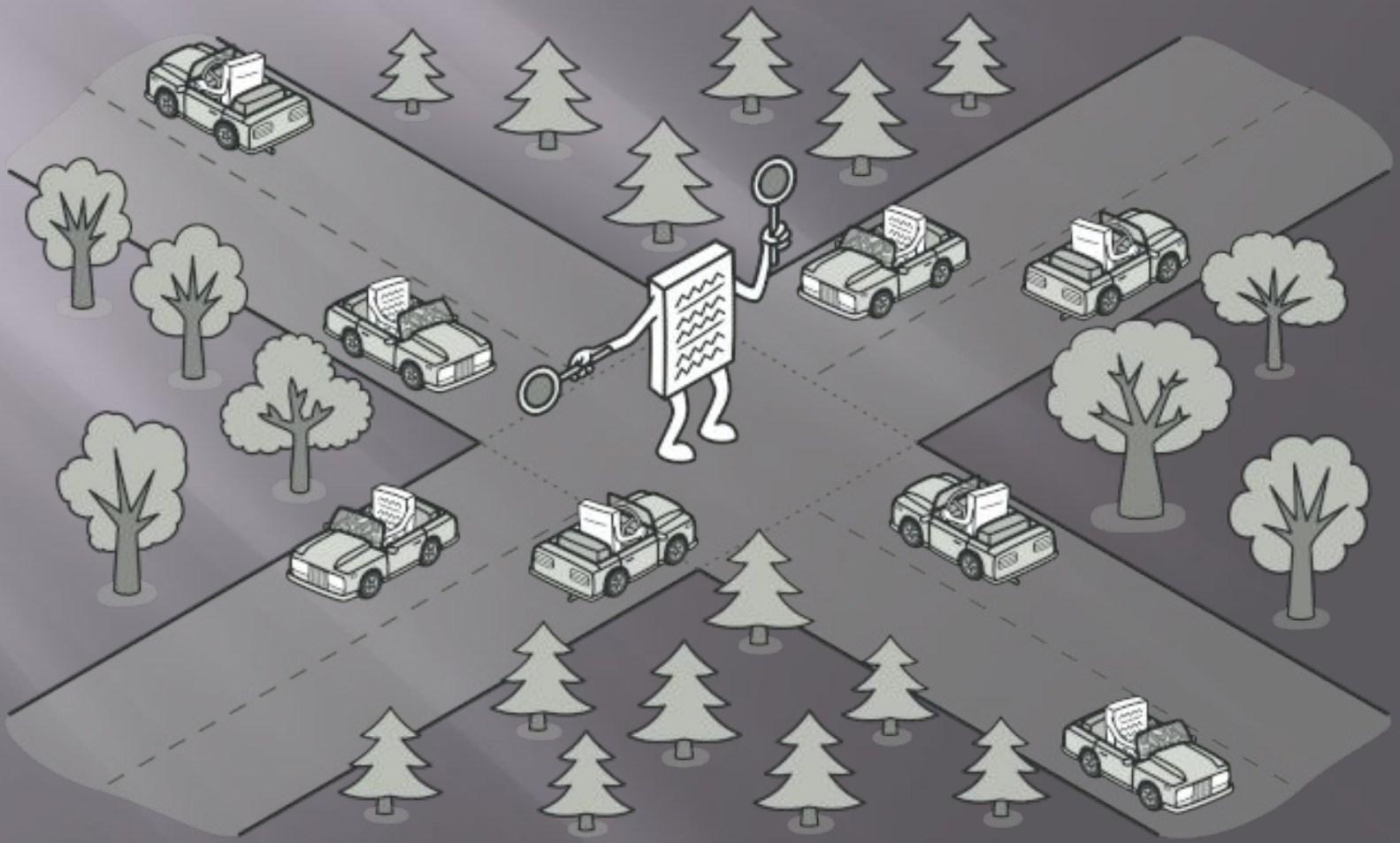
Реализует *принцип единственной обязанности*.

Реализует *принцип открытости/закрытости*.

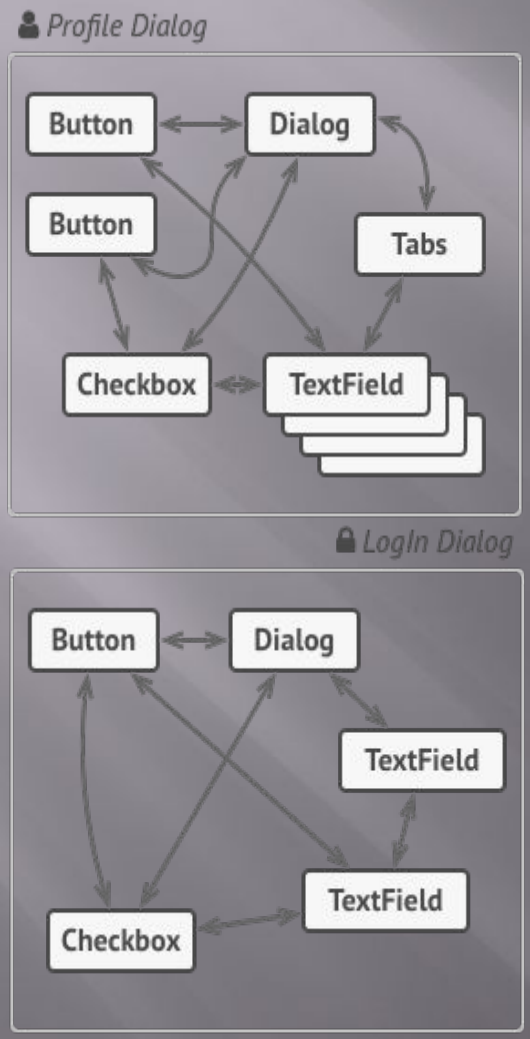
Запрос может остаться никем не обработанным.

# Посредник

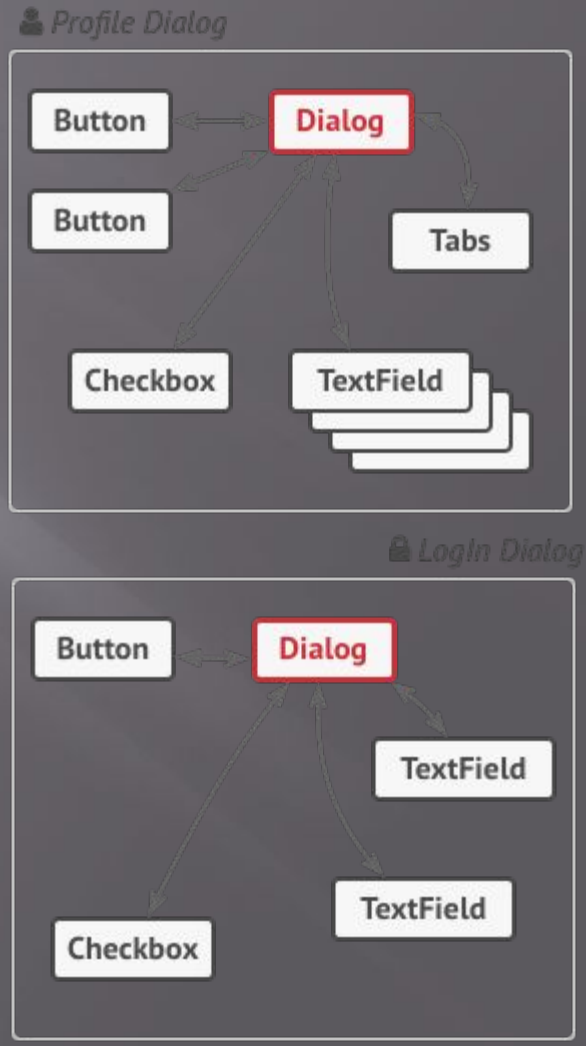
**Посредник** — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.(аэропорт)



# Проблема




# Решение



# Преимущества и недостатки

Устраняет зависимости между компонентами, позволяя повторно их использовать.

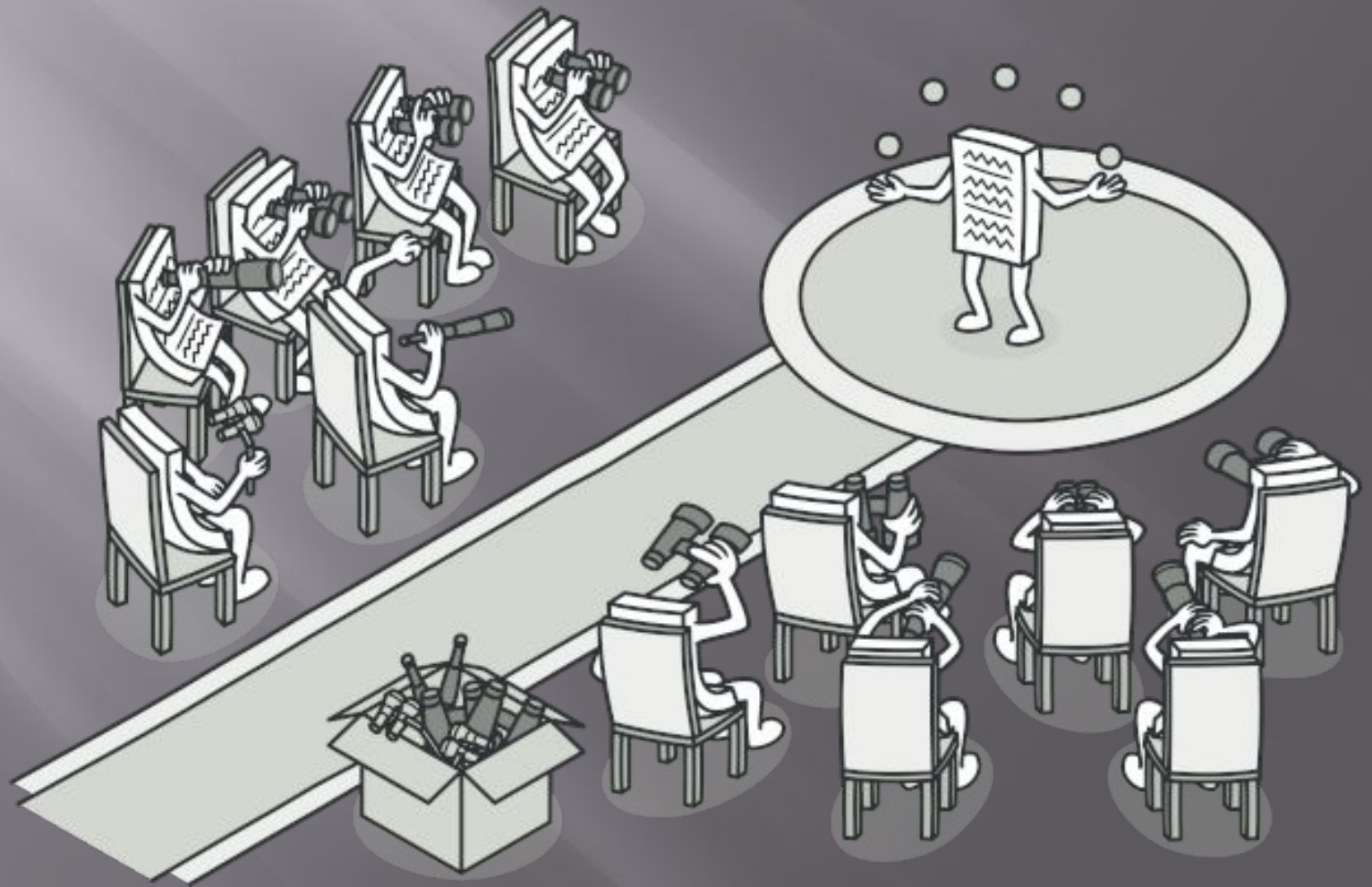
Упрощает взаимодействие между компонентами.

Централизует управление в одном месте

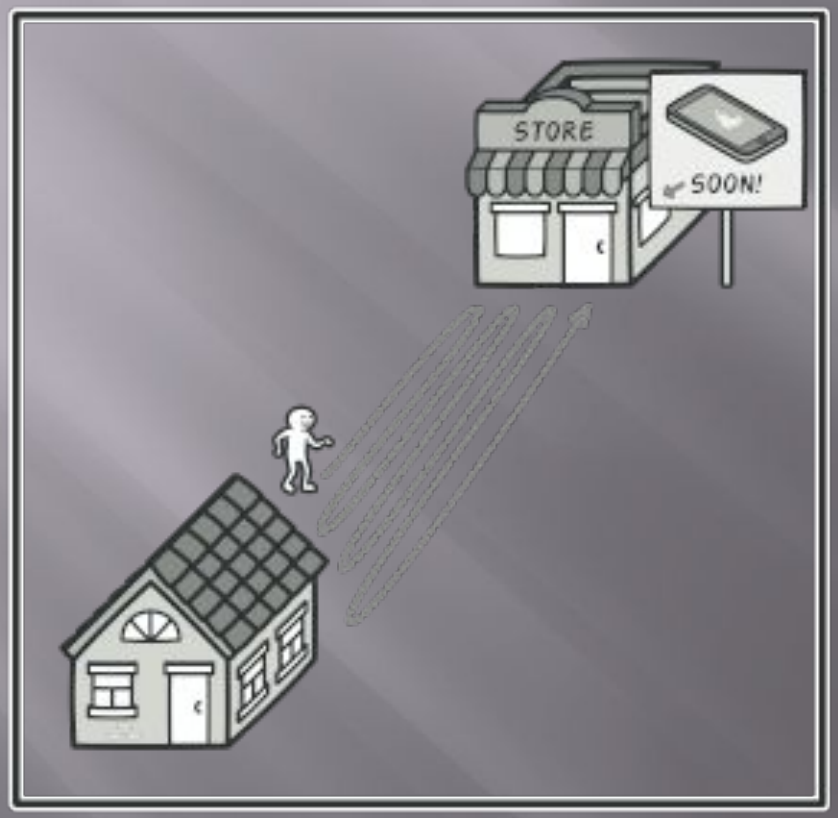
Посредник может **сильно раздуться**.

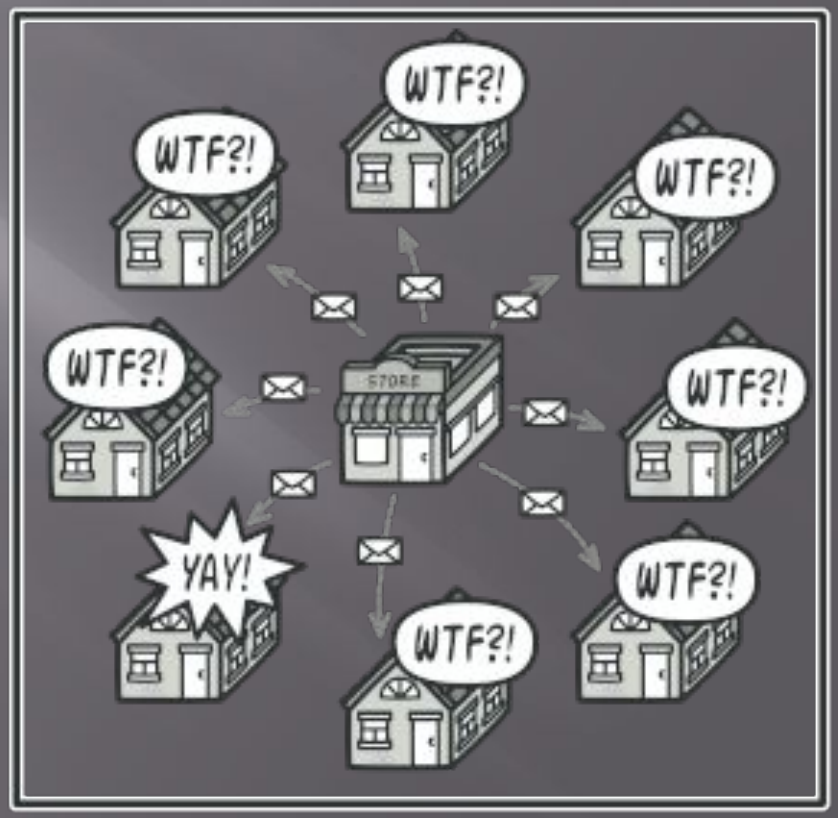
# Наблюдатель

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



# Проблема



# Решения


# Преимущества и недостатки

Издатели не зависят от конкретных классов подписчиков и наоборот.

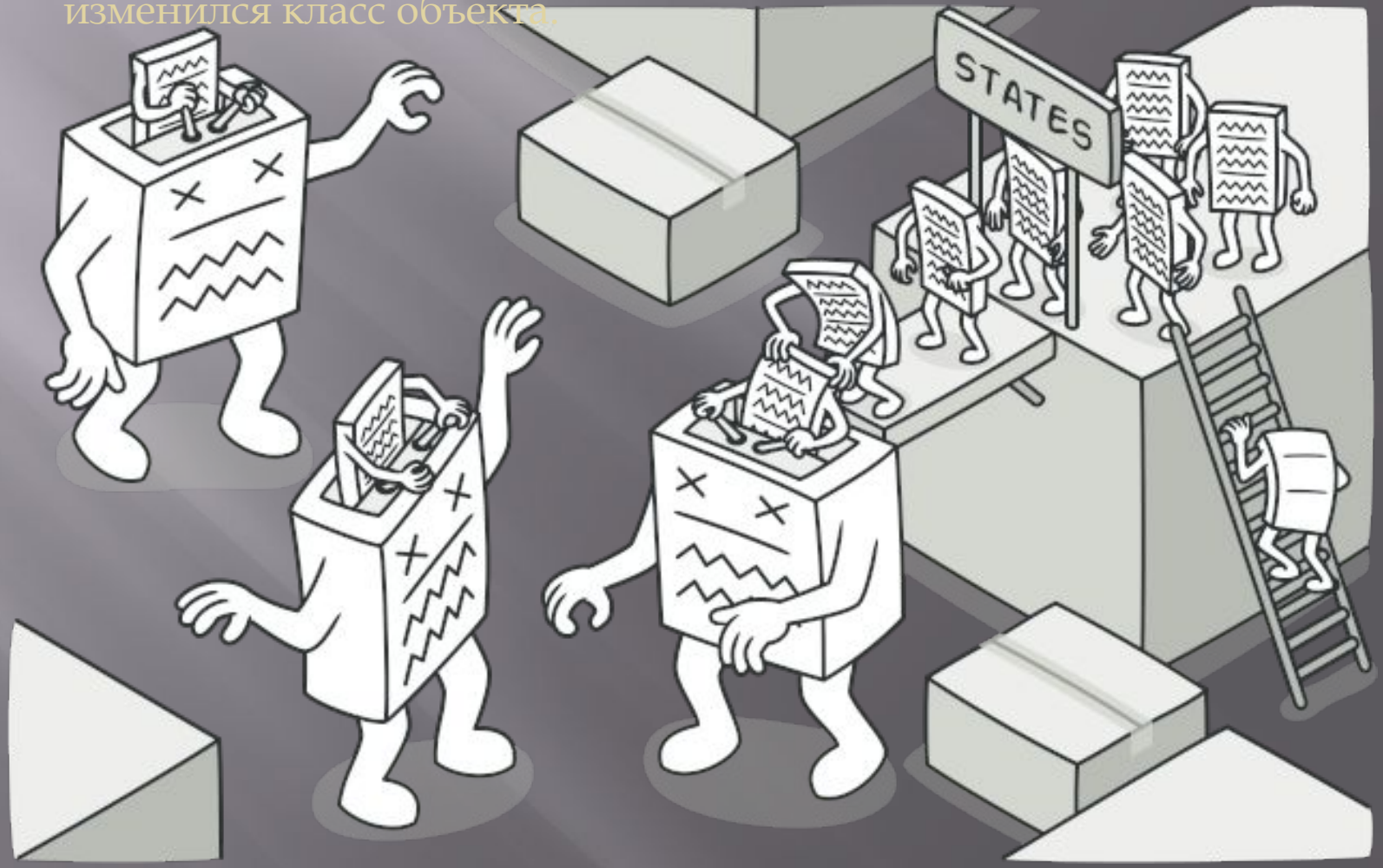
Вы можете подписывать и отписывать получателей на лету.

Реализует *принцип открытости/закрытости*.

Подписчики оповещаются в случайном порядке.

# Состояние

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



# 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.

# Принципы программирования

- KISS-Keep It Simple Stupid
- DRY(DIE)-Don’t repeat yourself (Dublication is evil)
- YAGNI – You Aren’t Gonna Need It
- SOLID