

Паттерны #3

Структурные: фасад, компоновщик, заместитель, мост, адаптер

Неделя 7

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.

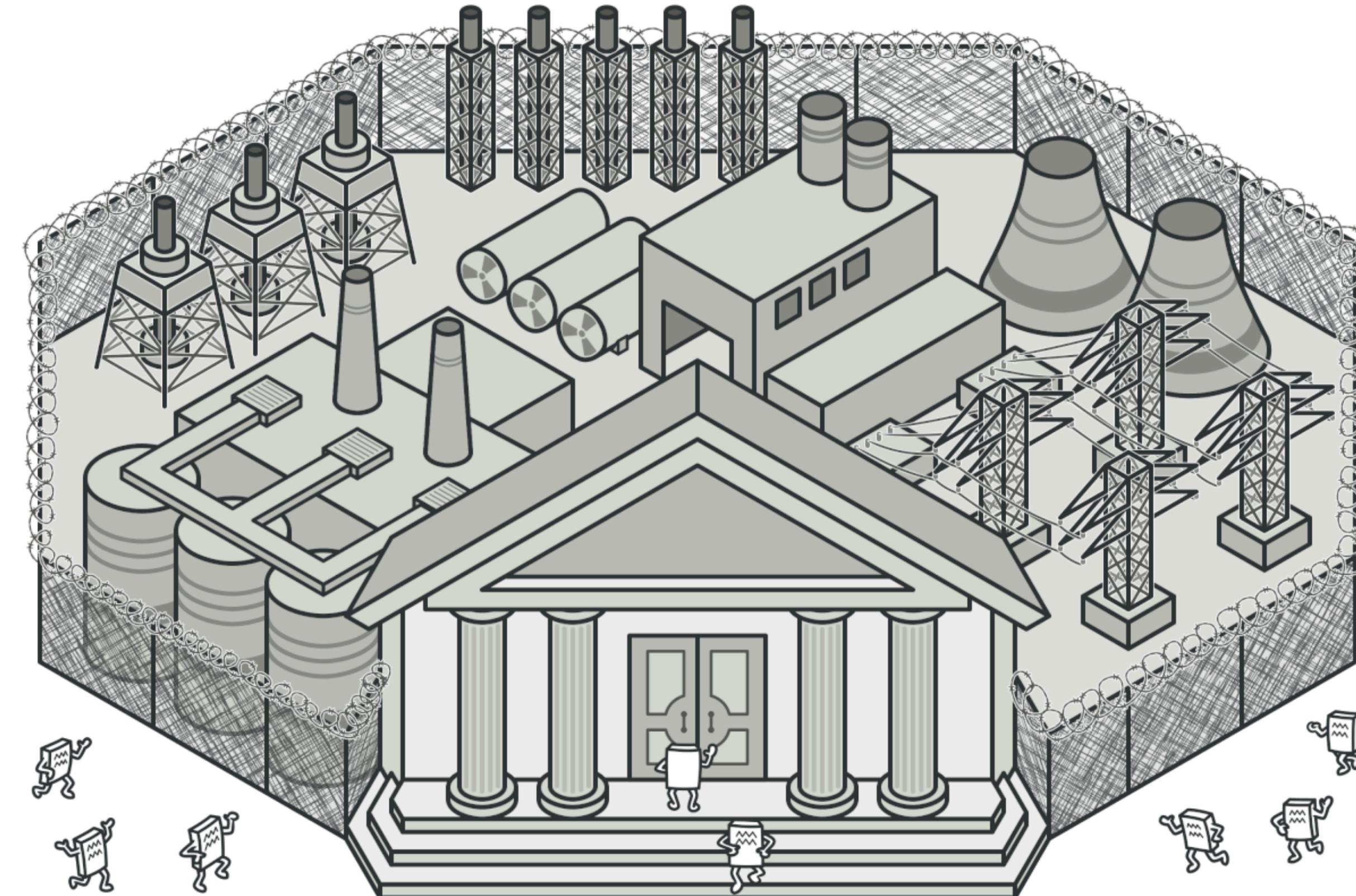
мы тут

Структурные паттерны

- Фасад
- Компоновщик
- Заместитель
- Мост
- Адаптер

Фасад

Фасад (Facade) представляет шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней.



Как реализовать?

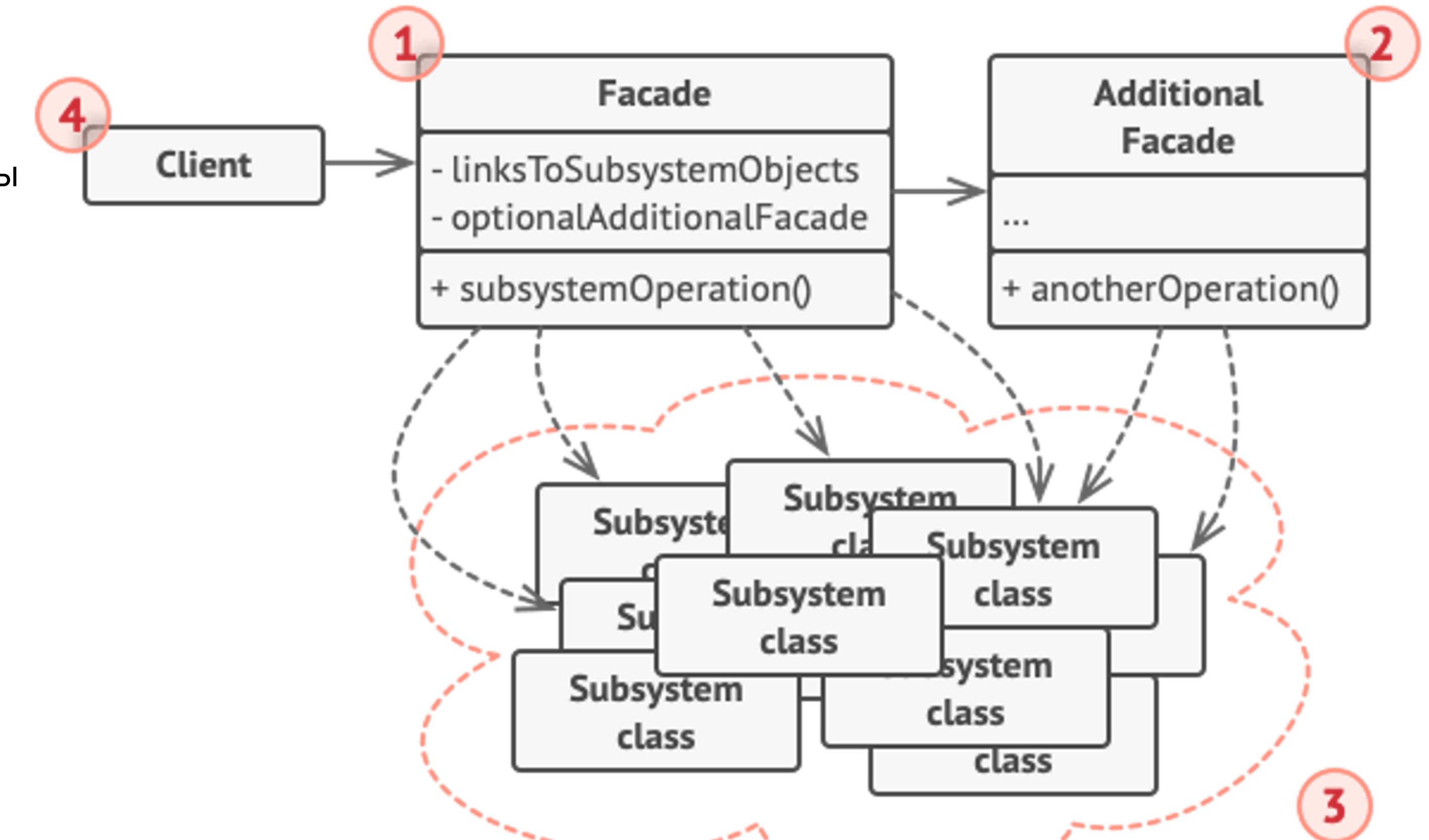
1. **Фасад** предоставляет быстрый доступ к определённой функциональности подсистемы. Он «знает», каким классам нужно переадресовать запрос, и какие данные для этого нужны.

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

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

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

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



Кейсы для использования

- Когда имеется сложная система, и необходимо упростить с ней работу.**

Фасад позволит определить одну точку взаимодействия между клиентом и системой.

- Когда надо уменьшить количество зависимостей между клиентом и сложной системой.**

Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.

- Когда нужно определить подсистемы компонентов в сложной системе.**

Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.

Кейсы для использования

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 обмечаться друг с другом через эти фасады, а не напрямую.

Плюсы/минусы

- + Изолирует клиентов от компонентов сложной подсистемы
- Фасад рискует стать божественным объектом, привязанным ко всем классам программы

Дополнительные моменты

Стоит отличать от Адаптера

Фасад задаёт новый интерфейс, тогда как Адаптер повторно использует старый. Адаптер обворачивает только один класс, а Фасад обворачивает целую подсистему. Кроме того, Адаптер позволяет двум существующим интерфейсам работать сообща, вместо того, чтобы задать полностью новый.

Но в принципе он похож на Прокси

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.

Пример. Фасад для подсистемы

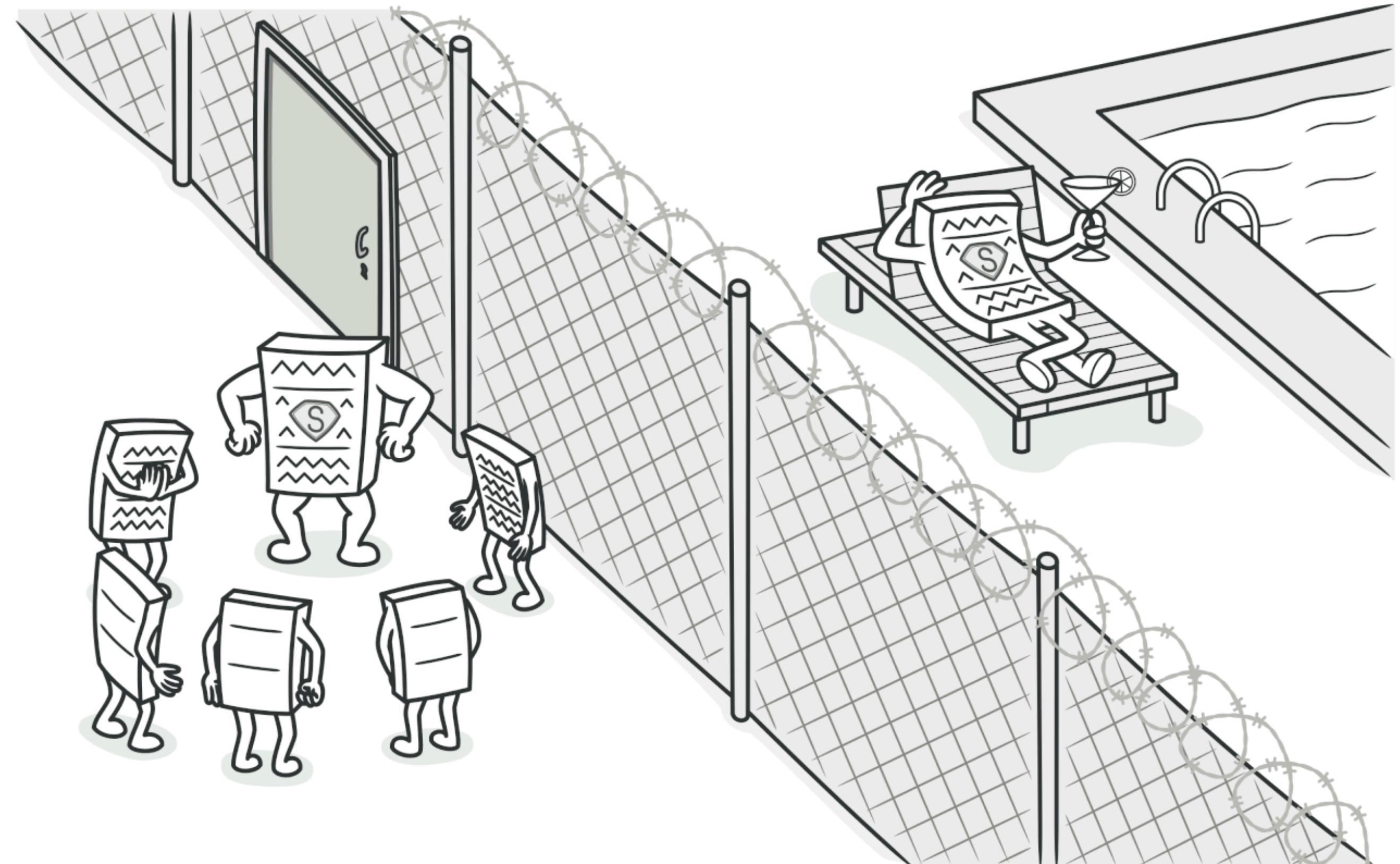
В основной программе сейчас происходит инициализация всех нужных зависимостей, их получение из контейнера, много дуалирующегося кода..

Вынесем все это в отдельную подсистему

Заместитель

Что такое Прокси?

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



Как использовать?

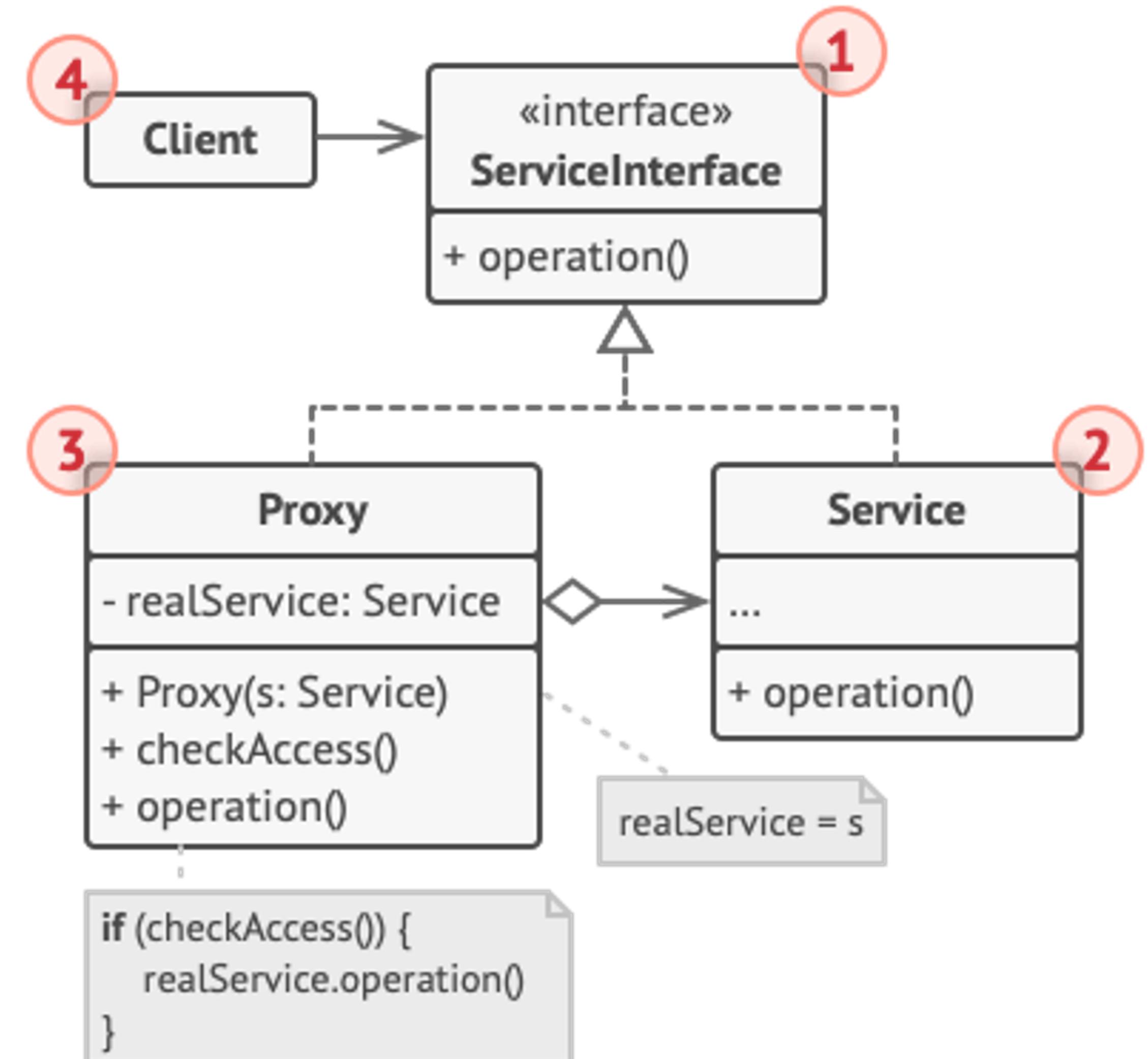
1. **Интерфейс сервиса** определяет общий интерфейс для сервиса и заместителя. Благодаря этому, объект заместителя можно использовать там, где ожидается объект сервиса.

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.



Когда использовать?

1. Когда надо осуществлять взаимодействие по сети, а объект-проси должен имитировать поведения объекта в другом адресном пространстве. Использование прокси позволяет снизить накладные издержки при передачи данных через сеть. Подобная ситуация еще называется **удалённый заместитель (remote proxies)**
2. Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобится, а до этого все запросы к нему обрабатывает прокси-объект. Подобная ситуация еще называется **виртуальный заместитель (virtual proxies)**
3. Когда необходимо разграничить доступ к вызываемому объекту в зависимости от прав вызывающего объекта. Подобная ситуация еще называется **защищающий заместитель (protection proxies)**
4. Когда нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом. Подобная ситуация называется **"умные ссылки" (smart reference)**

Кейсы для использования

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.

Кейсы для использования

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.

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

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

Плюсы/минусы

- + 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.

Дополнительные моменты

Стоит отличать от Декоратора

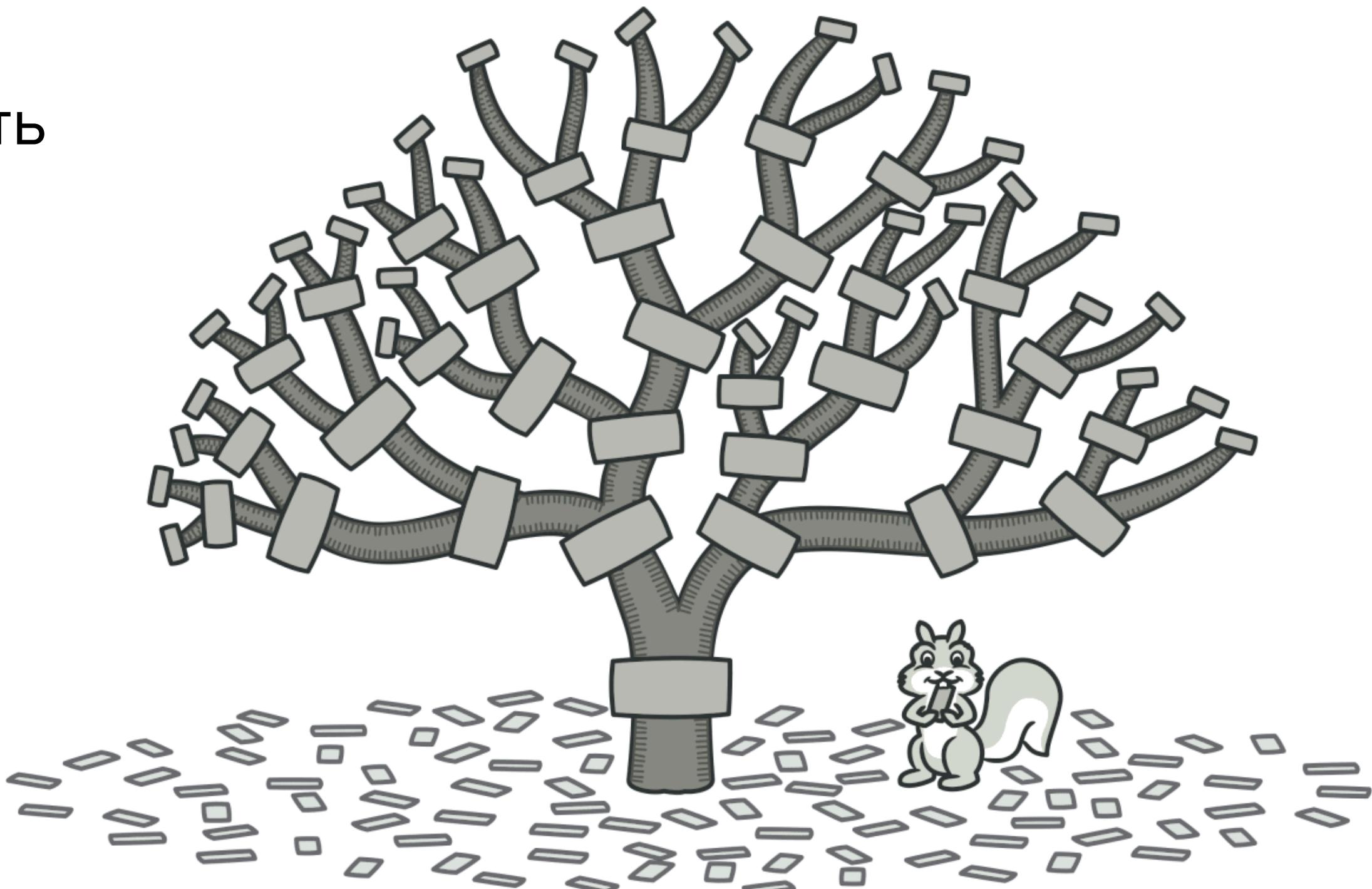
Декоратор и Заместитель имеют схожие структуры, но разные назначения. Они похожи тем, что оба построены на принципе композиции и делегируют работу другим объектам. Паттерны отличаются тем, что Заместитель сам управляет жизнью сервисного объекта, а обёртывание Декораторов контролируется клиентом.

Компоновщик

Что такое Composite?

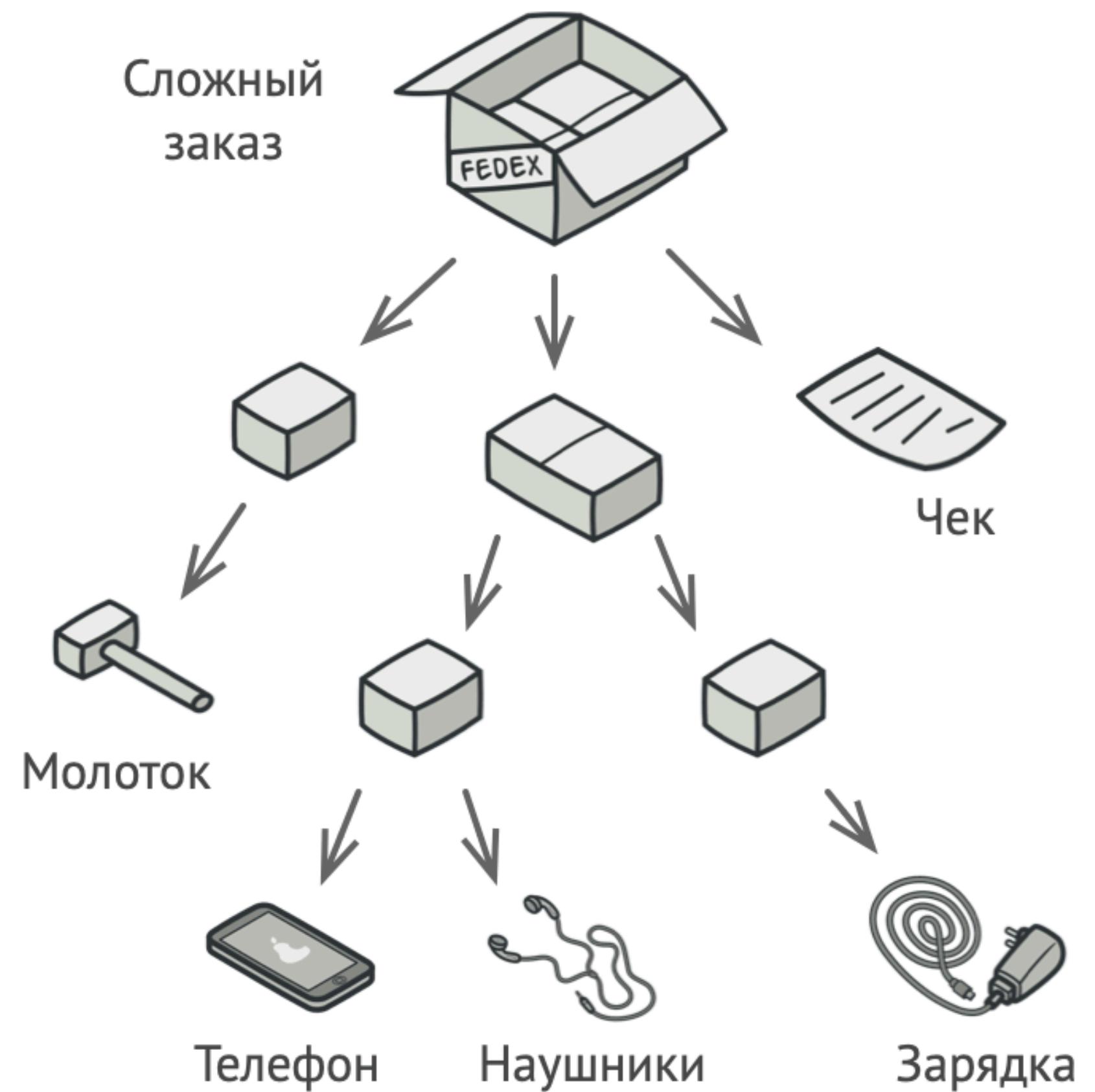
Паттерн Компоновщик (Composite) объединяет группы объектов в древовидную структуру по принципу «часть-целое» и позволяет клиенту одинаково работать как с отдельными объектами, так и с группой объектов.

Образно реализацию паттерна можно представить в виде меню, которое имеет различные пункты. Эти пункты могут содержать подменю, в которых, в свою очередь, также имеются пункты. То есть пункт меню служит с одной стороны частью меню, а с другой стороны еще одним меню. В итоге мы однообразно можем работать как с пунктом меню, так и со всем меню в целом.



Когда использовать?

- Когда объекты должны быть реализованы в виде иерархической древовидной структуры
- Когда клиенты единообразно должны управлять как целыми объектами, так и их составными частями. То есть целое и его части должны реализовать один и тот же интерфейс



Как использовать?

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

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

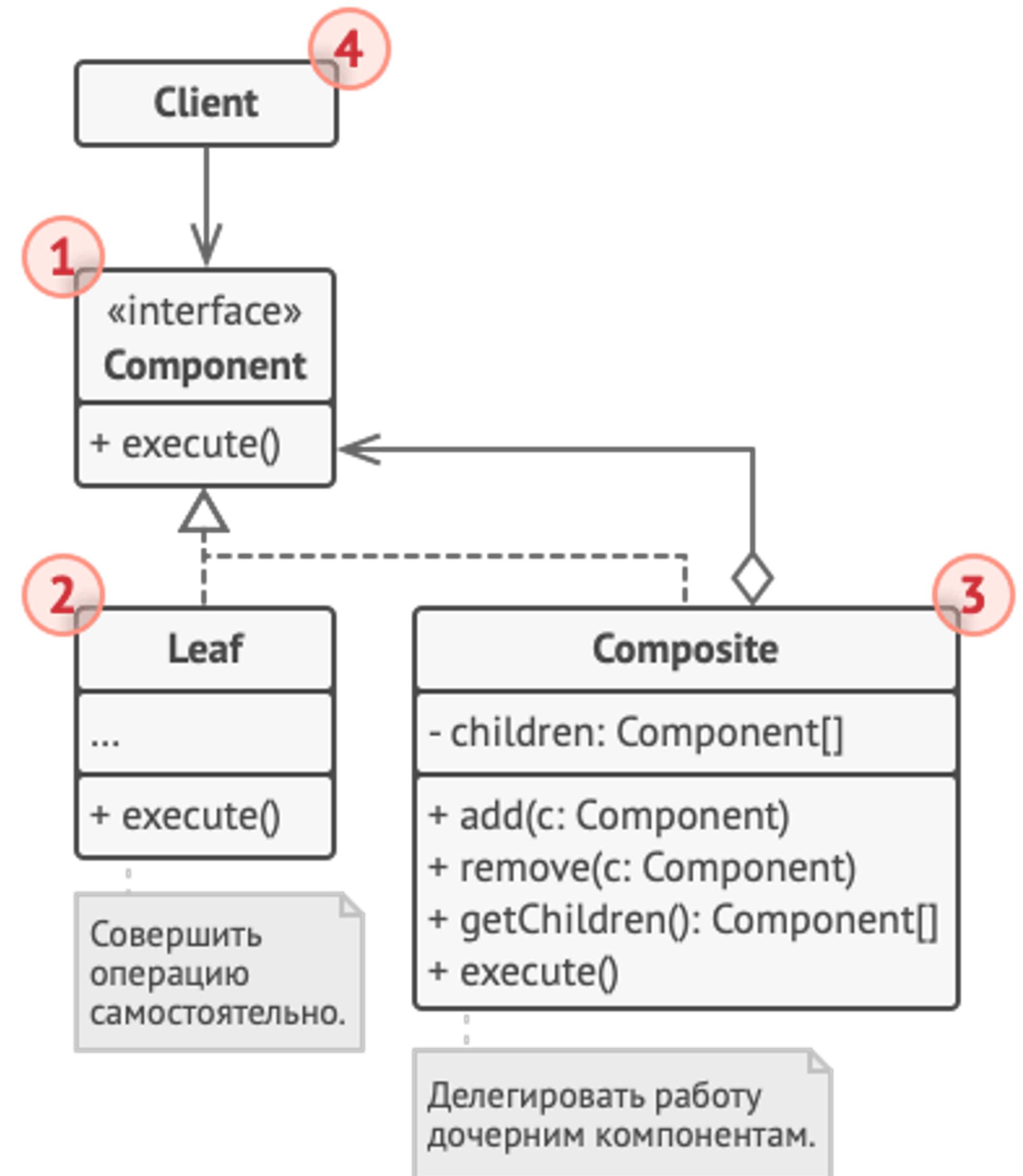
Из-за того, что им некому больше передавать выполнение, классы листьев будут содержать большую часть полезного кода.

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.



Кейсы для использования

Когда вам нужно представить древовидную структуру объектов.

Паттерн Компонент 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.

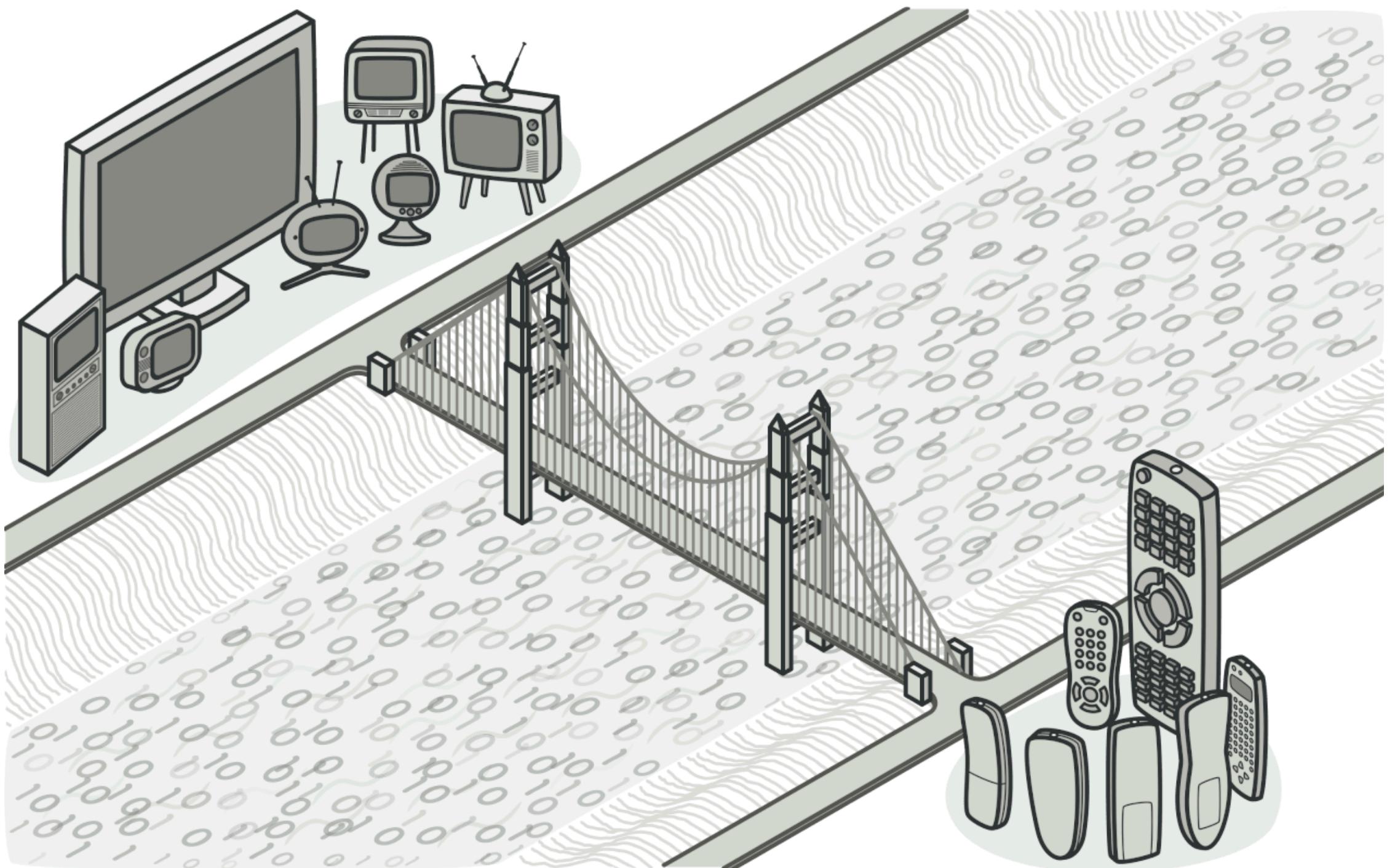
Благодаря тому, что простые и составные объекты реализуют общий интерфейс, клиенту безразлично, с каким именно объектом ему предстоит работать.

Плюсы/минусы

- + Упрощает архитектуру клиента при работе со сложным деревом компонентов.
- + Облегчает добавление новых видов компонентов.
- Создаёт слишком общий дизайн классов.

Мост

Что такое Bridge?



Мост (Bridge) - структурный шаблон проектирования, который позволяет отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга.

Даже если мы отделим абстракцию от конкретных реализаций, то у нас все равно все наследуемые классы будут жестко привязаны к интерфейсу, определяемому в базовом абстрактном классе. Для преодоления жестких связей и служит паттерн Мост.

Как использовать?

1. **Абстракция** содержит управляющую логику. Код абстракции делегирует реальную работу связанному объекту реализации.

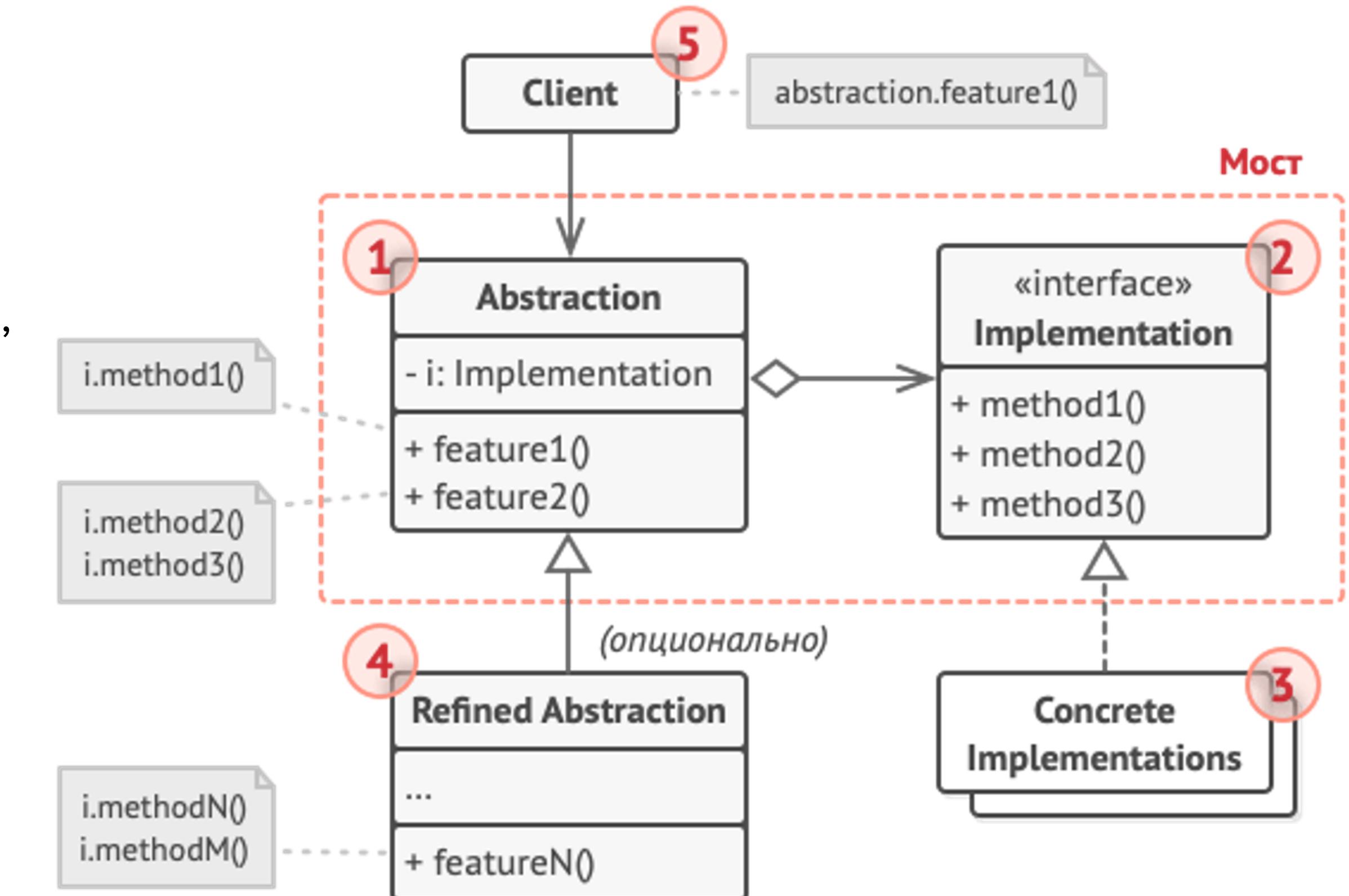
2. **Реализация** задаёт общий интерфейс для всех реализаций. Все методы, которые здесь описаны, будут доступны из класса абстракции и его подклассов.

Интерфейсы абстракции и реализаций 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.



Кейсы для использования

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 «вшиита» в класс абстракции.

Плюсы/минусы

- + 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.

Дополнительные моменты

Стоит отличать от Адаптера

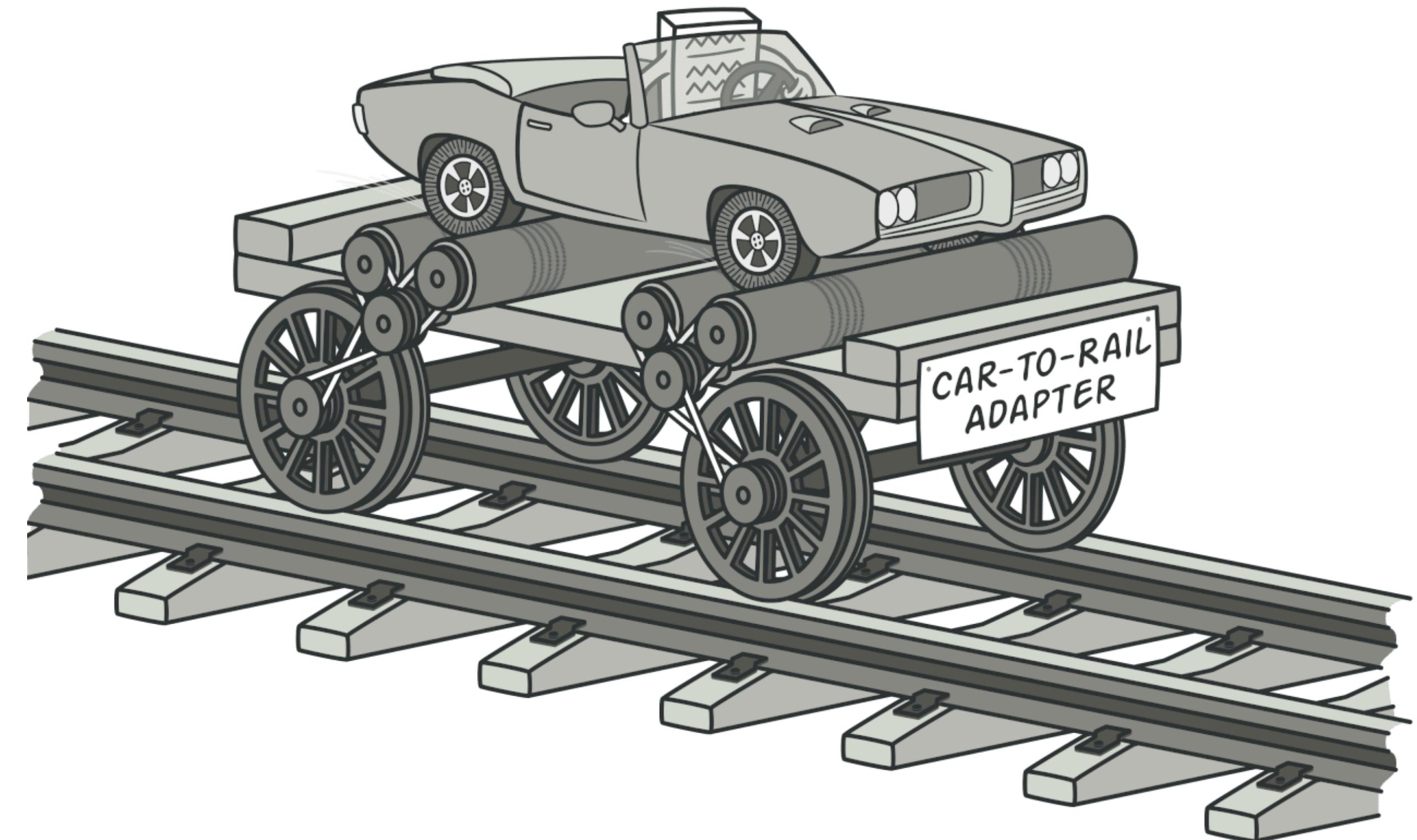
Мост проектируют загодя, чтобы развивать большие части приложения отдельно друг от друга. Адаптер применяется постфактум, чтобы заставить несовместимые классы работать вместе.

Адаптер

Что такое Adapter?

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого.

Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.



Как использовать?

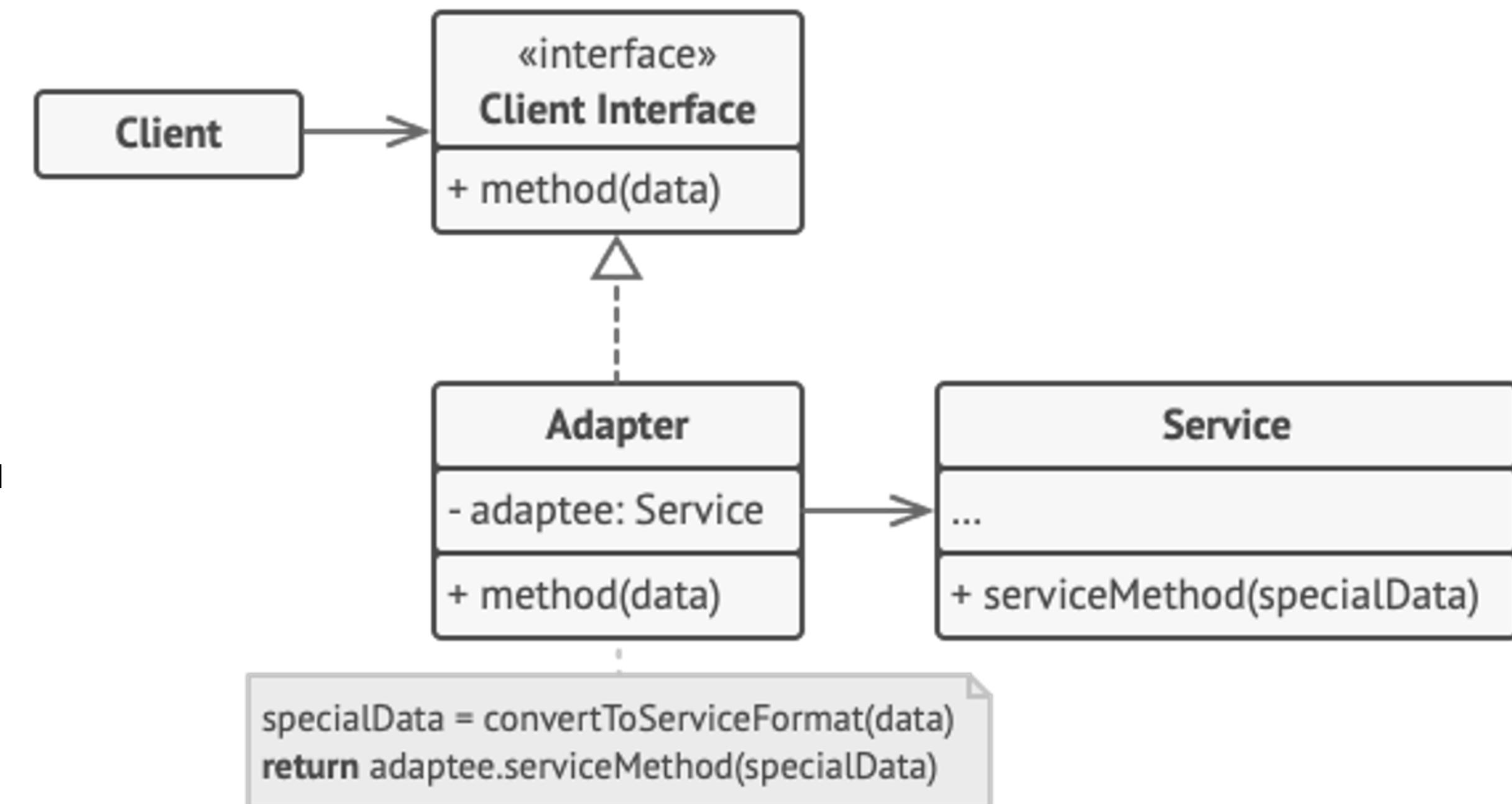
1. **Клиент** — это класс, который содержит существующую бизнес-логику программы.

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.



Кейсы для использования

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 **Декоратор**.

Плюсы/минусы

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

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.

Пример. Фасад для подсистемы

В основной программе сейчас происходит инициализация всех нужных зависимостей, их получение из контейнера, много дуалирующегося кода..

Вынесем все это в отдельную подсистему