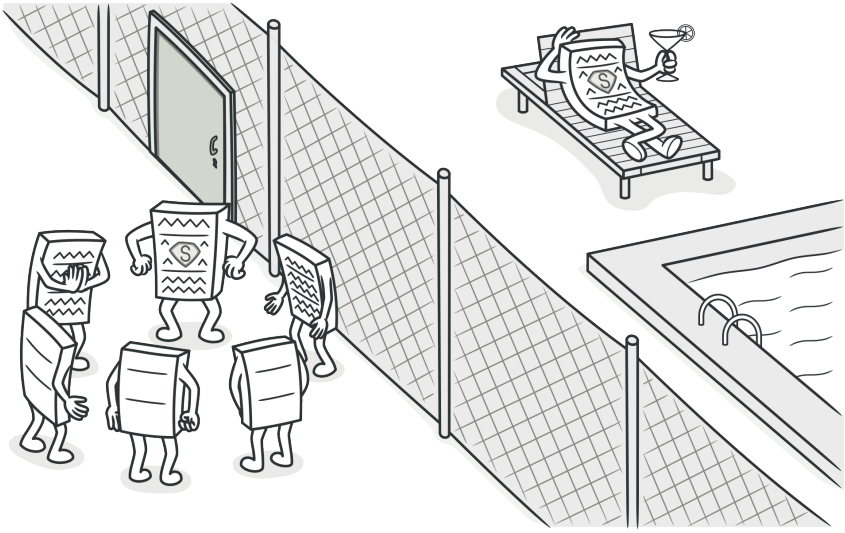


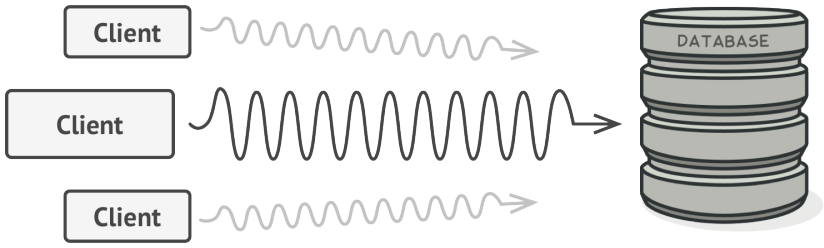
ЗАМЕСТИТЕЛЬ

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.



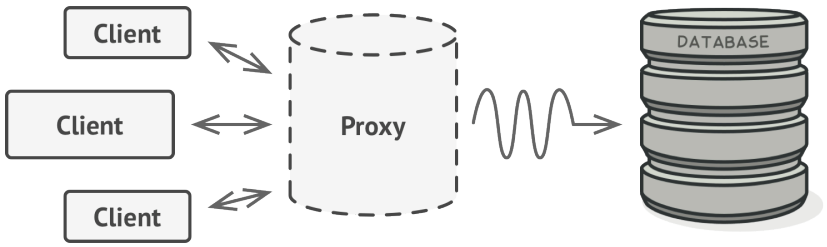
Запросы к базе данных могут быть очень медленными.

Мы могли бы не создавать этот объект в самом начале программы, а только 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 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,

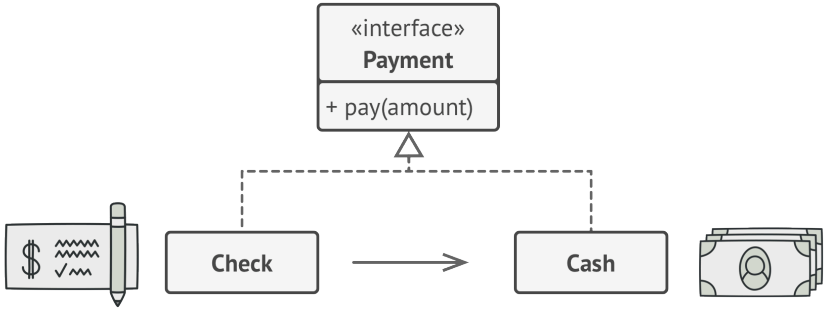
объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.



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 одинаковому интерфейсу, объект заместитель можно передать в любой код, ожидающий сервисный объект.

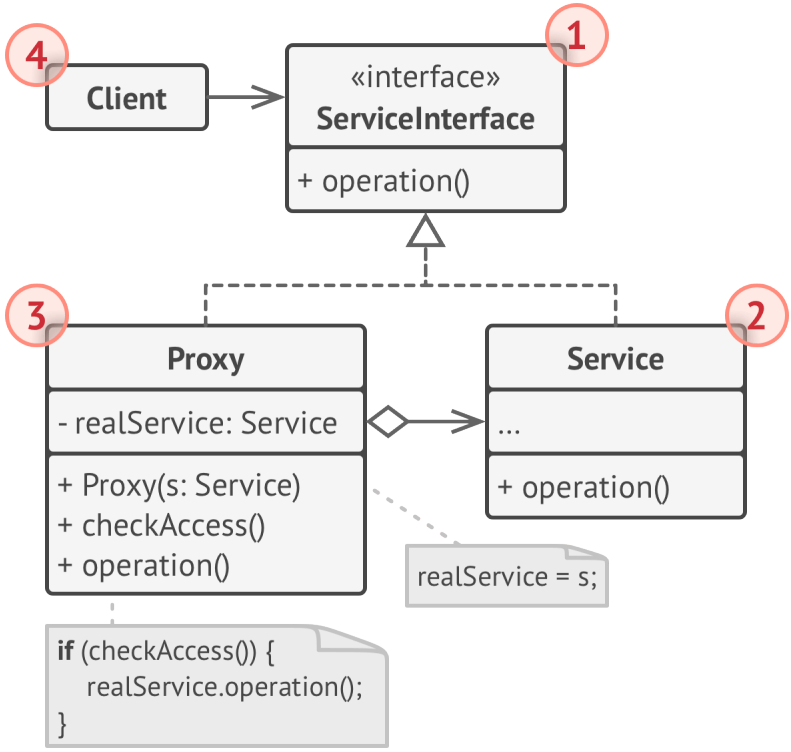
Аналогия из жизни



Банковским чеком можно расплачиваться, как и наличностью.

Банковский чек — это заместитель пачки наличности. И чек, и наличность имеют общий интерфейс — ими можно оплачивать товары. Для покупателя польза в том, что не надо таскать с собой тонны наличности. 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.

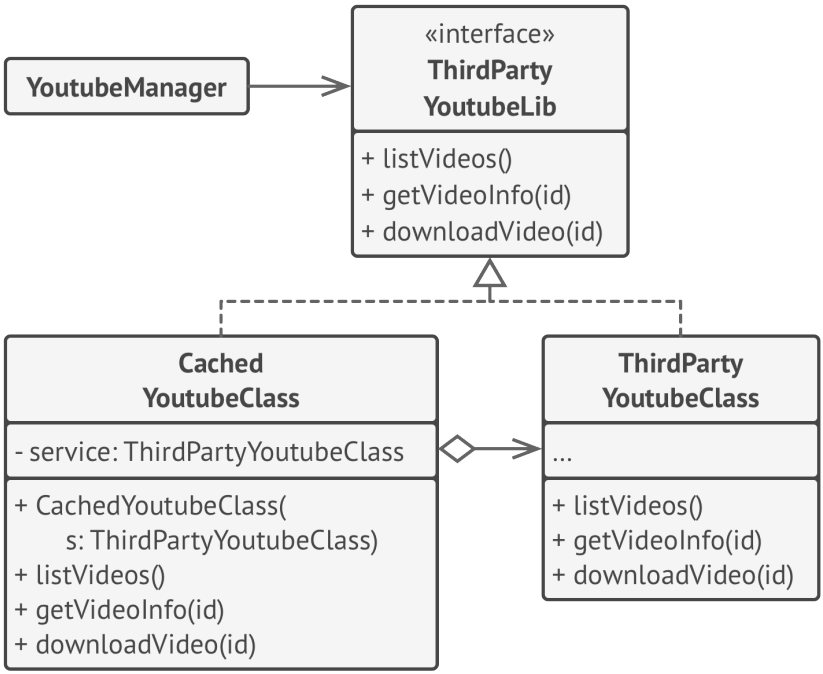
3. **Заместитель** хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису.

   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 тяжёлой служебной 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.

```
// Интерфейс удалённого сервиса.
interface ThirdPartyYoutubeLib is
  method listVideos()
  method getVideoInfo(id)
  method downloadVideo(id)

```

```
// Конкретная реализация сервиса. Методы этого класса
// запрашивают у ютуба различную информацию. Скорость
// запроса зависит от интернет-канала пользователя и
// состояния самого ютуба. Чем больше будет вызовов к
// сервису, тем менее отзывчивой будет программа.
class ThirdPartyYoutubeClass is
  method listVideos() is
    // Получить список видеороликов с помощью API Youtube.

  method getVideoInfo(id) is
    // Получить детальную информацию о каком-то видеоролике.

  method downloadVideo(id) is
    // Скачать видео с Youtube.

// С другой стороны, можно кешировать запросы к ютубу и не
// повторять их какое-то время, пока кеш не устареет. Но
// внести этот код напрямую в сервисный класс нельзя, так
// как он находится в сторонней библиотеке. Поэтому мы
// поместим логику кеширования в отдельный класс-обёртку. Он
// будет делегировать запросы к сервисному объекту, только
// если нужно непосредственно выслать запрос.
class CachedYoutubeClass implements ThirdPartyYoutubeLib is
  private field service: ThirdPartyYoutubeClass
  private field listCache, videoCache
  field needReset

  constructor CachedYoutubeClass(service: ThirdPartyYoutubeLib) is
    this.service = service

  method listVideos() is
    if (listCache == null || needReset)
      listCache = service.listVideos()
    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.
53  // Клиент ничего не заметит, так как заместитель имеет тот
54  // же 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
74  // клиентам объект заместителя.
```

```
class Application is
  method init() is
    youtubeService = new ThirdPartyYoutubeClass()
    youtubeProxy = new CachedYoutubeClass(youtubeService)
    manager = new YoutubeManager(youtubeProxy)
    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.

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

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

Кроме того, Заместитель может отслеживать, не менял ли клиент сервисный объект. Это позволит использовать

объекты повторно и здóрово 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 программу за счёт 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 похожие структуры, но разные назначения. Они похожи тем, что оба построены на композиции и делегировании работы другому объекту.

Паттерны отличаются тем, что *Заместитель* сам управляет жизнью сервисного объекта, а обёртывание *Декораторов* контролируется клиентом.