Патерн “Замісник”

Що таке патерн проектування?

Патерн проектування — це типовий спосіб вирішення певної задачі, яка часто виникає під час створення архітектури програм.

На відміну від готових бібліотек чи функцій, патерн не можна просто скопіювати у програму. Він не є конкретним шматком коду, а радше узагальненою ідеєю або підходом, який потрібно адаптувати під особливості кожного проєкту.

Часто патерни плутають з алгоритмами. Насправді між ними є суттєва різниця:

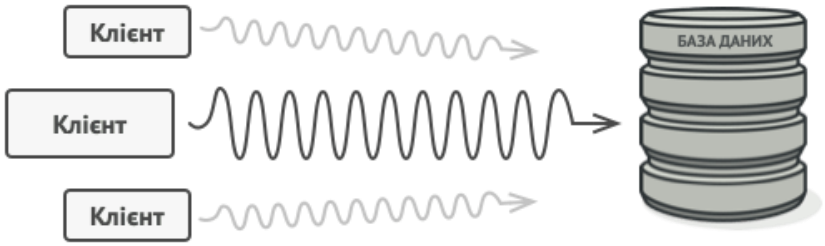
- Алгоритм — це точна послідовність дій, що завжди приводить до результату.
- Патерн — це високорівневий опис рішення, який може реалізовуватися по-різному у різних програмах.

Аналогія: алгоритм схожий на кулінарний рецепт із чіткими кроками приготування, тоді як патерн нагадує креслення інженера, що описує загальну конструкцію, але не диктує конкретний спосіб її реалізації.

Замісник (Proxy) — це структурний шаблон проектування, що дозволяє використовувати спеціальні об’єкти-замінники замість справжніх. Такі об’єкти перехоплюють виклики до оригінального об’єкта й можуть виконати певні дії до або після передачі виклику основному елементу.

Проблема

Для чого взагалі контролювати доступ до об’єктів? Розглянемо такий приклад: у вас є зовнішній ресурсоємний об’єкт, який потрібен не весь час, а лише зрідка.



Запити до бази даних можуть бути дуже повільними.

Ми могли б створювати цей об’єкт не на самому початку програми, а тільки тоді, коли він реально кому-небудь знадобиться. Кожен клієнт об’єкта отримав би деякий код відкладеної ініціалізації. Це, ймовірно, призвело б до дублювання великої кількості коду.

В ідеалі цей код хотілося б помістити безпосередньо до службового класу, але це не завжди можливо. Наприклад, код класу може знаходитися в закритій сторонній бібліотеці.

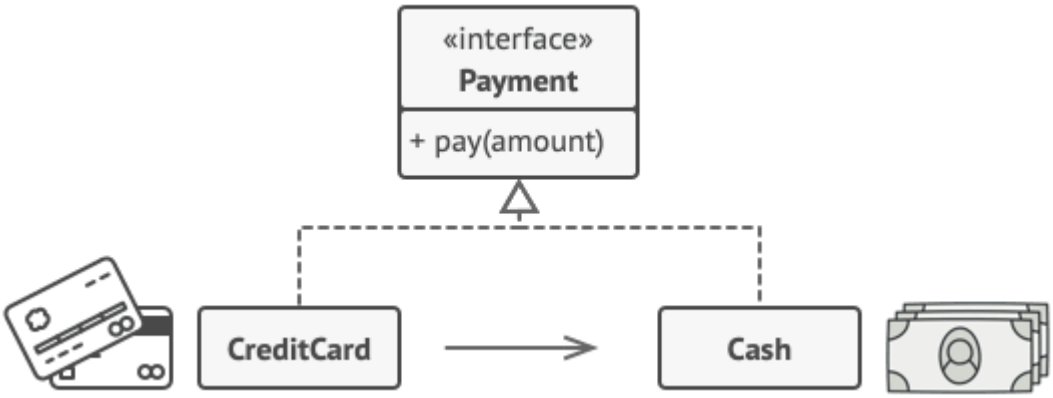
Рішення

Патерн Замісник пропонує створити новий клас-дублер, який має той самий інтерфейс, що й оригінальний службовий об’єкт. При отриманні запиту від клієнта об’єкт-замісник сам би створював примірник службового об’єкта та переадресовував би йому всю реальну роботу.

Замісник «прикидається» базою даних, прискорюючи роботу внаслідок ледачої ініціалізації і кешування запитів, що повторюються.

Але в чому ж його користь? Ви могли б помістити до класу замісника якусь проміжну логіку, що виконувалася б до або після викликів цих самих методів чинного об’єкта. А завдяки однаковому інтерфейсу об’єкт-замісник можна передати до будь-якого коду, що очікує на сервісний об’єкт.

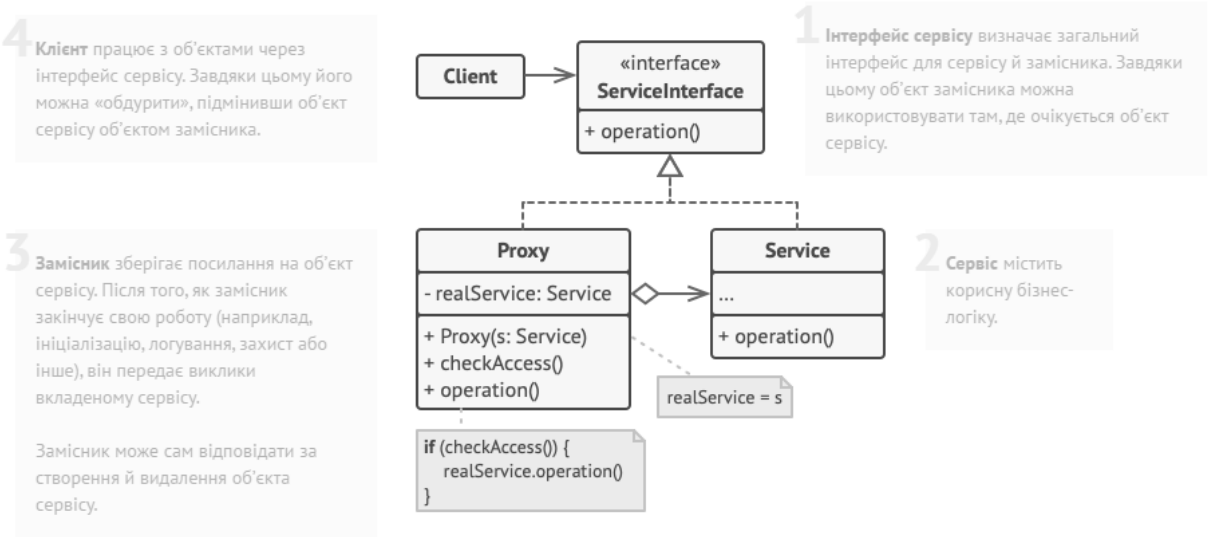
Аналогія з життя



Платіжною карткою можна розраховуватися так само, як і готівкою.

Платіжна картка — це замісник пачки готівки. І чек, і готівка мають спільний інтерфейс — ними обома можна оплачувати товари. Вигода покупця в тому, що не потрібно носити з собою «тонни» готівки. З іншого боку власник магазину не змушений замовляти клопітку інкасацію коштів з магазину, бо вони потрапляють безпосередньо на його банківський рахунок.

Структура



Застосування

– Лінива ініціалізація (віртуальний проксі). Коли у вас є важкий об’єкт, який завантажує дані з файлової системи або бази даних.

Замість того, щоб завантажувати дані відразу після старту програми, можна заощадити ресурси й створити об’єкт тоді, коли він дійсно знадобиться.

– Захист доступу (захищаючий проксі). Коли в програмі є різні типи користувачів, і вам хочеться захистити об’єкт від неавторизованого доступу. Наприклад, якщо ваші об’єкти — це важлива частина операційної системи, а користувачі — сторонні програми (корисні чи шкідливі).

Проксі може перевіряти доступ під час кожного виклику та передавати виконання службовому об’єкту, якщо доступ дозволено.

– Локальний запуск сервісу (віддалений проксі).

Коли справжній сервісний об’єкт знаходиться на віддаленому сервері. У цьому випадку замісник транслює запити клієнта у виклики через мережу по протоколу, який є зрозумілим віддаленому сервісу.

– Логування запитів (логуючий проксі). Коли потрібно зберігати історію звернень до сервісного об’єкта.

Замісник може зберігати історію звернення клієнта до сервісного об’єкта.

– Кешування об’єктів («розумне» посилання). Коли потрібно кешувати результати запитів клієнтів і керувати їхнім життєвим циклом.

Замісник може підраховувати кількість посилань на сервісний об’єкт, які були віддані клієнту та залишаються активними. Коли всі посилання звільняться, можна буде звільнити і сам сервісний об’єкт (наприклад, закрити підключення до бази даних).

Крім того, Замісник може відстежувати, чи клієнт не змінював сервісний об’єкт. Це дозволить повторно використовувати об’єкти й суттєво заощаджувати ресурси, особливо якщо мова йде про великі «ненажерливі» сервіси.

Кроки реалізації

1. Визначте інтерфейс, який би зробив замісника та оригінальний об’єкт взаємозамінними.
2. Створіть клас замісника. Він повинен містити посилання на сервісний об’єкт. Частіше за все сервісний об’єкт створюється самим замісником. У рідкісних випадках замісник отримує готовий сервісний об’єкт від клієнта через конструктор.
3. Реалізуйте методи замісника в залежності від його призначення. У більшості випадків, виконавши якусь корисну роботу, методи замісника повинні передати запит сервісному об’єкту.
4. Подумайте про введення фабрики, яка б вирішувала, який з об’єктів створювати: замісника або реальний сервісний об’єкт. Проте, з іншого боку, ця логіка може бути вкладена до створюючого методу самого замісника.
5. Подумайте, чи не реалізувати вам ліниву ініціалізацію сервісного об’єкта при першому зверненні клієнта до методів замісника.

Розглянемо приклад реалізації патерна «Замісник (Proxy)» мовою програмування Ruby. У цьому прикладі є реальний сервіс, який отримує дані з YouTube, і замісник, що додає кешування для зменшення кількості звернень до сервісу. Клієнтський код працює з об’єктом через єдиний інтерфейс і не помічає різниці між справжнім сервісом та його замісником. Це демонструє головну ідею патерна — можливість додавання нової поведінки без зміни вихідного коду основного об’єкта.

```
# Реальний сервіс YouTube
class ThirdPartyYouTubeClass
  def list_videos
    puts "Отримуємо список відео з YouTube..."
```

```
    ["video1", "video2", "video3"]
  end

  def get_video_info(id)
    puts "Отримуємо інформацію про відео #{id}..."
    { id: id, title: "Відео #{id}" }
  end
end

# Замісник із кешуванням запитів
class CachedYouTubeClass
  def initialize(service)
    @service = service
    @list_cache = nil
    @video_cache = {}
  end

  def list_videos
    @list_cache ||= @service.list_videos
  end

  def get_video_info(id)
    @video_cache[id] ||= @service.get_video_info(id)
  end
end

# Клієнтський код
class YouTubeManager
  def initialize(service)
    @service = service
```

```
  end

  def show_video(id)
    info = @service.get_video_info(id)
    puts "Відображаємо: #{info[:title]}"
  end
end

service = ThirdPartyYouTubeClass.new
proxy = CachedYouTubeClass.new(service)
manager = YouTubeManager.new(proxy)

manager.show_video("video1") # перший виклик — звертається до YouTube
manager.show_video("video1") # другий виклик — бере з кешу
```

Лістинг 1 – фрагмент реалізації патерна

Його суть у тому, що клієнт працює не безпосередньо зі справжнім об’єктом, а через спеціальний клас-замісник, який має такий самий інтерфейс.

Клас ThirdPartyYouTubeClass — це реальний сервіс, який виконує «важкі» операції, наприклад отримує дані з YouTube. Клас CachedYouTubeClass є замісником, який зберігає результати запитів у кеші. Якщо клієнт повторно звертається до того самого відео, проксі повертає вже збережені дані, не звертаючись до справжнього сервісу.

Клас YouTubeManager виступає клієнтом і працює через спільний інтерфейс, не помічаючи різниці між реальним сервісом і його замісником. Завдяки цьому можна додати кешування або інші дії (наприклад, перевірку доступу) без зміни основного коду. Такий підхід робить програму гнучкішою та ефективнішою.

Переваги:

- контроль доступу без змін клієнтського коду;
- економія ресурсів завдяки відкладеній ініціалізації;
- можливість додаткової логіки (логування, кешування).

Недоліки:

- ускладнення структури програми;
- уповільнення роботи через додатковий рівень викликів.