Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №4
із дисципліни «Технології Розробки Програмного Забезпечення»
Тема: ШАБЛОНИ «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»
Варіант-11

Виконав:
Студент групи ІА-24
Коханчук Михайло Миколайович

Перевірив:
Мягкий Михайло Юрійович

Київ-2024

ЗМІСТ

МЕТА

Метою роботи є розробити частину функціоналу застосунку для роботи з JSON, реалізувавши класи та їхню взаємодію для забезпечення динамічного форматування даних. У процесі реалізації застосувати шаблон проектування Strategy для досягнення гнучкості та масштабованості системи.

Теоритичні відомості:

Будь-який патерн проектування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проектування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях. Крім того, патерн проектування обов'язково має загальновживане найменування. Правильно сформульований патерн проектування дозволяє, відшукавши одного разу вдале рішення, користуватися ним знову і знову.

Варто підкреслити, що важливим початковим етапом при роботі з патернами є адекватне моделювання розглянутої предметної області. Це є необхідним як для отримання належним чином формалізованої постановки задачі, так і для вибору відповідних патернів проектування.

Відповідне використання патернів проектування дає розробнику ряд незаперечних переваг. Наведемо деякі з них. Модель системи, побудована в межах патернів проектування, фактично є структурованим виокремленням тих елементів і зв'язків, які значимі при вирішенні поставленого завдання. Крім цього, модель, побудована з використанням патернів проектування, більш проста і наочна у вивченні, ніж стандартна модель. Проте, не дивлячись на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови

Застосування патернів проектування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід зазначити, що сукупність патернів проектування, по суті, являє собою єдиний словник проектування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним.

Таким чином шаблони представляють собою, підтверджені роками розробок в різних компаніях і на різних проектах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних обставинах.

Застосування шаблонів проектування не гарантує, що розроблена архітектура буде кристально чистою і зручною з точки зору програмування. Однак в потрібних місцях застосування шаблонів дозволить досягти наступних вигод:

- Зменшення трудовитрат і часу на побудову архітектури;
- Надання проектованій системі необхідних якостей (гнучкість, адаптованість, ін.);
- Зменшити накладні витрати на подальшу підтримку системи;
- Та інші.

Варто також зазначити, що знання шаблонів проектування допомагає не тільки архітекторам програмних систем, але і розробникам. Коли кожна людина в команді знає значення і властивості шаблонів, архітекторові простіше донести загальну ідею архітектури системи, а розробникам - простіше зрозуміти.

Оскільки, урешті-решт, кожен бізнес зводиться до грошей, шаблони проектування також є економічно виправданим вибором між побудовою власного «колеса», та реалізацією закріплених і гарантованих спільнотою розробників практик і підходів.

Це звичайно ж не означає, що їх необхідно використовувати в кожному проекті на кожну вимогу. Підходи не є догмою, їх потрібно використовувати з головою.

В межах даної та наступних робіт я посилатимусь та цитуватиму книгу Олександра Швецова «Занурення в патерни проектування».

Класифікація патернів

Патерни проектування відрізняються складністю, рівнем деталізації та масштабом застосування до всієї системи, що проектується. Мені подобається аналогія з будівництвом доріг: можна зробити перехрестя безпечнішим, встановивши кілька світлофорів або побудувавши цілу багаторівневу розв'язку з підземними переходами для пішоходів.

Найпростіші та низькорівневі патерни часто називають ідіомами. Зазвичай вони стосуються лише однієї мови програмування.

Найбільш універсальні та високорівневі патерни - це архітектурні патерни. Розробники можуть реалізувати ці патерни практично на будь-якій мові. На відміну від інших патернів, їх можна використовувати для проектування архітектури цілого додатку.

Крім того, всі патерни можна класифікувати за їхнім призначенням. Розглянемо три основні групи патернів:

- Породжувальні патерни(creational patterns) надають механізми створення об'єктів, які підвищують гнучкість і повторне використання існуючого коду.
- Структурні патерни(structural patterns) пояснюють, як збирати об'єкти та класи у більші структури, зберігаючи ці структури гнучкими та ефективними.
- Поведінкові патерни(behavioral) дбають про ефективну комунікацію та розподіл обов'язків між об'єктами.

Шаблон «SINGLETON»

Одинак(Singleton) - це шаблон проектування, який дозволяє гарантувати, що клас має лише один екземпляр, забезпечуючи при цьому глобальну точку доступу до цього екземпляра.

Проблема

Паттерн Одинак(Singleton) вирішує дві проблеми одночасно, порушуючи принцип єдиної відповідальності(Single Responsibility Principle):

- Гарантує, що клас має лише один екземпляр. Найчастіше за все це корисно для доступу до якогось спільного ресурсу, наприклад, бази даних.
  Уявіть собі, що ви створили об’єкт, а через деякий час намагаєтесь створити ще один. У цьому випадку хотілося б отримати старий об’єкт замість створення нового.
  Таку поведінку неможливо реалізувати за допомогою звичайного конструктора, оскільки конструктор класу завжди повертає новий об’єкт

- Створює глобальну точку доступу до цього екземпляра. Це не просто глобальна змінна, через яку можна дістатися до певного об’єкта. Глобальні змінні не захищені від запису, тому будь-який код може підмінити їхнє значення без вашого відома.
  Проте, є ще одна особливість. Було б непогано й зберігати в одному місці код, який вирішує проблему №1, і мати до нього простий та доступний інтерфейс.

У наш час патерн Singleton став настільки популярним, що люди можуть називати щось синглтоном, навіть якщо воно вирішує лише одну з перерахованих проблем.

Рішення

Всі реалізації Одинака зводяться до того, аби приховати типовий конструктор та створити публічний статичний метод, який і контролюватиме життєвий цикл об’єкта-одинака. Якщо у вас є доступ до класу одинака, отже, буде й доступ до цього статичного методу. З якої точки коду ви б його не викликали, він завжди віддаватиме один і той самий об’єкт.

Аналогія з життя

Уряд держави є чудовим прикладом патерну «синглтон». Країна може мати лише один офіційний уряд. Незалежно від особистої ідентичності людей, які формують уряд, назва «Уряд Х» є глобальною точкою доступу, яка ідентифікує групу відповідальних осіб.

Структура

Одинак визначає статичний метод getInstance , який повертає один екземпляр свого класу. Конструктор Одинака повинен бути прихований від клієнтів. Виклик методу getInstance повинен стати єдиним способом отримати об’єкт цього класу

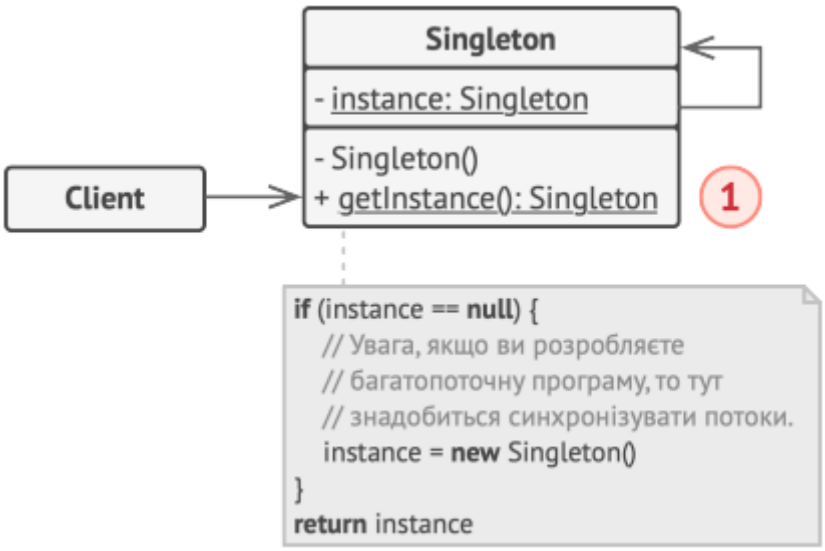

Рис.1.1 Структура патерну Singleton

Застосування

1. *Коли в програмі повинен бути єдиний екземпляр якогонебудь класу, доступний усім клієнтам (наприклад, спільний доступ до бази даних з різних частин програми).*

   Коли в програмі повинен бути єдиний екземпляр якого-небудь класу, доступний усім клієнтам (наприклад, спільний доступ до бази даних з різних частин програми). Одинак приховує від клієнтів всі способи створення нового об'єкта, окрім спеціального методу. Цей метод або створює об'єкт, або віддає існуючий об'єкт, якщо він вже був створений

2. *Коли ви хочете мати більше контролю над глобальними змінними.*

   На відміну від глобальних змінних, Одинак гарантує, що жоден інший код не замінить створений екземпляр класу, тому ви завжди впевнені в наявності лише одного об'єкта-одинака. Тим не менше, будь-коли ви можете розширити це обмеження і дозволити будь-яку кількість об'єктів-одинаків, змінивши код в одному місці (метод getInstance).

.

Кроки реалізації

1. Додайте до класу приватне статичне поле, котре міститиме одиночний об'єкт.
2. Оголосіть статичний створюючий метод, що використовуватиметься для отримання Одинака.
3. Додайте «лініву ініціалізацію» (створення об'єкта під час першого виклику методу) до створюючого методу одинака
4. Зробіть конструктор класу приватним.
5. У клієнтському коді замініть прямі виклики конструктора одинака на виклики його створюючого методу.

Переваги

- Гарантує наявність єдиного екземпляра класу.
- Надає глобальну точку доступу до нього.
- Реалізує відкладену ініціалізацію об'єкта-одинака.

Недоліки

- Порушує принцип єдиного обов'язку класу.
- Маскує поганий дизайн.

- Проблеми багатопоточності.
- Вимагає постійного створення Mock-об’єктів при юніт-тестуванні.

Шаблон «ITERATOR»

Ітератор — це поведінковий патерн проектування, що дає змогу послідовно обходити елементи складових об’єктів, не розкриваючи їхньої внутрішньої організації

Проблема

Колекції — це найпоширеніша структура даних, яку ви можете зустріти в програмуванні. Це набір об’єктів, зібраний в одну купу за якимись критеріями.

Більшість колекцій виглядають як звичайний список елементів. Але є й екзотичні колекції, побудовані на основі дерев, графів та інших складних структур даних.

Незважаючи на те, яким чином структуровано колекцію, користувач повинен мати можливість послідовно обходити її елементи, щоб виконувати з ними певні дії.

У який же спосіб слід переміщатися складною структурою даних? Наприклад, сьогодні може бути достатнім обхід дерева в глибину, але завтра виникне необхідність переміщуватися деревом по ширині. А на наступному тижні, хай йому грець, знадобиться можливість обходу колекції у випадковому порядку.

Додаючи все нові алгоритми до коду колекції, ви потроху розмиваєте її основну задачу, що полягає в ефективному зберіганні даних. Деякі алгоритми можуть бути аж занадто «заточені» під певну програму, а тому виглядатимуть неприродно в загальному класі колекції.

Рішення

Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий об’єкт.

Об’єкт-ітератор відстежуватиме стан обходу, поточну позицію в колекції та кількість елементів, які ще залишилося обійти. Одну і ту саму колекцію зможуть одночасно обходити різні ітератори, а сама колекція навіть не знатиме про це.

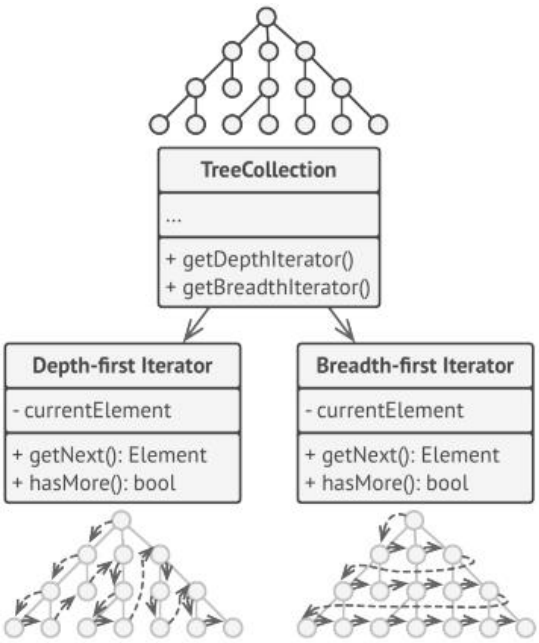

Рис.1.2 Ітератори містять код обходу колекції. Одну колекцію можуть обходити відразу декілька ітераторів.

До того ж, якщо вам потрібно буде додати новий спосіб обходу, ви зможете створите окремий клас ітератора, не змінюючи існуючого коду колекції.

Аналогія з життя

Ви плануєте полетіти до Риму та обійти всі визначні пам'ятки за кілька днів. Але по приїзді ви можете довго блукати вузькими вуличками, намагаючись знайти один тільки Колізей.

Якщо у вас обмежений бюджет, ви можете скористатися віртуальним гідом, встановленим у смартфоні, який дозволить відфільтрувати тільки цікаві вам об'єкти. А можете плюнути на все та найняти місцевого гіда, який хоч і обійдеться в копієчку, але знає все місто, як свої п'ять пальців, і зможе «занурити» вас в усі міські легенди.

Таким чином, Рим виступає колекцією пам'яток, а ваш мозок, навігатор чи гід — ітератором колекції. Ви як клієнтський код можете вибрати одного з ітераторів, відштовхуючись від вирішуваного завдання та доступних ресурсів.

Структура

Одинак визначає статичний метод getInstance , який повертає один екземпляр свого класу. Конструктор Одинака повинен бути прихований від клієнтів. Виклик методу getInstance повинен стати єдиним способом отримати об'єкт цього класу

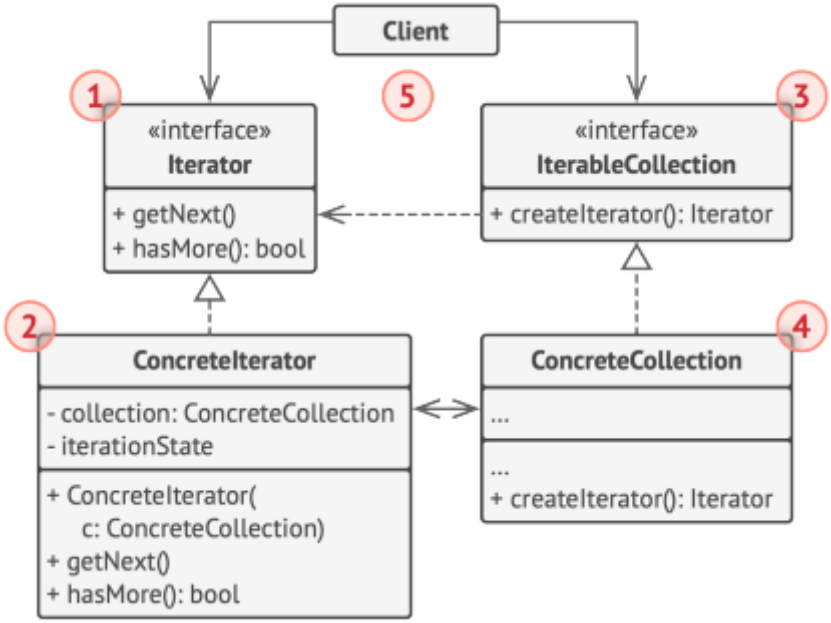

Рис.1.3 Структура патерну Iterator

1. Ітератор описує інтерфейс для доступу та обходу елементів колекцій.
2. Конкретний ітератор реалізує алгоритм обходу якоїсь конкретної колекції. Об'єкт ітератора повинен сам відстежувати поточну позицію при обході колекції, щоб окремі ітератори могли обходити одну і ту саму колекцію незалежно.
3. Колекція описує інтерфейс отримання ітератора з колекції. Як ми вже говорили, колекції не завжди є списком. Це може бути і база даних, і віддалене API, і навіть дерево Компонувальника. Тому сама колекція може створювати ітератори, оскільки вона знає, які саме ітератори здатні з нею працювати.
4. Конкретна колекція повертає новий екземпляр певного конкретного ітератора, зв'язавши його з поточним об'єктом колекції. Зверніть увагу на те, що сигнатура методу повертає інтерфейс ітератора. Це дозволяє клієнтові не залежати від конкретних класів ітераторів.
5. Клієнт працює з усіма об'єктами через інтерфейси колекції та ітератора. Через це клієнтський код не залежить від конкретних класів, що дозволяє застосовувати різні ітератори, не змінюючи існуючого коду програми.

В загальному випадку клієнти не створюють об’єкти ітераторів, а отримують їх з колекцій. Тим не менше, якщо клієнтові потрібний спеціальний ітератор, він завжди може створити його самостійно.

Застосування

Якщо у вас є складна структура даних, і ви хочете приховати від клієнта деталі її реалізації (з питань складності абобезпеки).

Ітератор надає клієнтові лише кілька простих методів перебору елементів колекції. Це не тільки спрощує доступ до колекції, але й захищає її від необережних або злочинних дій.

Якщо вам потрібно мати кілька варіантів обходу однієї і тієї самої структури даних.

Нетривіальні алгоритми обходу структури даних можуть мати досить об’ємний код. Цей код буде захаращувати все навкруги — чи то самий клас колекції, чи частина бізнеслогіки програми. Застосувавши ітератор, ви можете виділити код обходу структури даних в окремий клас, спростивши підтримку решти коду.

Якщо вам хочеться мати єдиний інтерфейс обходу різних структур даних.

Ітератор дозволяє винести реалізації різних варіантів обходу в підкласи. Це дозволить легко взаємозаміняти об’єкти ітераторів в залежності від того, з якою структурою даних доводиться працювати.

Кроки реалізації

1. Створіть загальний інтерфейс ітераторів. Обов’язковий мінімум — це операція отримання наступного елемента. Але для зручності можна передбачити й інше. Наприклад, методи отримання попереднього елементу, поточної позиції, перевірки закінчення обходу тощо
2. Створіть інтерфейс колекції та опишіть у ньому метод отримання ітератора. Важливо, щоб сигнатура методу повертала загальний інтерфейс ітераторів, а не один з конкретних ітераторів.
3. Створіть класи конкретних ітераторів для тих колекцій, які потрібно обходити за допомогою патерна. Ітератор повинен бути прив’язаний тільки до одного об’єкта колекції. Зазвичай цей зв’язок встановлюється через конструктор.
4. Реалізуйте методи отримання ітератора в конкретних класах колекцій. Вони повинні створювати новий ітератор того класу, який здатен працювати з даним типом колекції. Колекція повинна передавати посилання на власний об’єкт до конструктора ітератора.
5. У клієнтському коді та в класах колекцій не повинно залишитися коду обходу елементів. Клієнт повинен отримувати новий ітератор з об’єкта колекції кожного разу, коли йому потрібно перебрати її елементи

Переваги

- Спрощує класи зберігання даних.
- Дозволяє реалізувати різні способи обходу структури даних.
- Дозволяє одночасно переміщуватися структурою даних у різних напрямках.

Недоліки

- Невиправданий, якщо можна обійтися простим циклом.

Шаблон «PROXY»

Замісник(Proxy) — це структурний патерн проектування, що дає змогу підставляти замість реальних об’єктів спеціальні об’єкти-замінники. Ці об’єкти перехоплюють виклики до оригінального об’єкта, дозволяючи зробити щось до чи після передачі виклику оригіналові.

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.

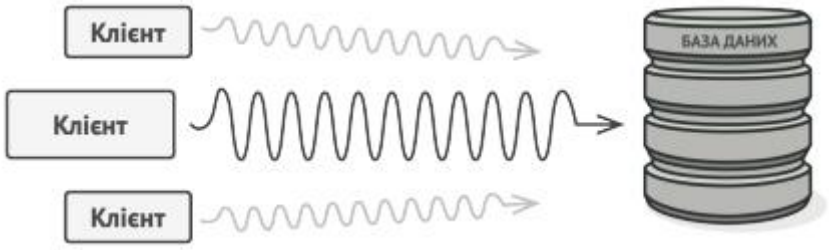

Рис.1.4 Запити до бази даних можуть бути дуже повільними.

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.

Рис.1.5 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

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.

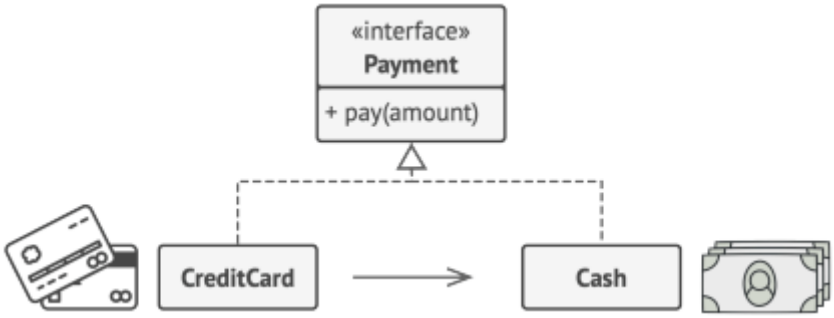

Рис. 1.6 Платіжною карткою можна розраховуватися так само, як і готівкою.

Структура

Одинак визначає статичний метод getInstance , який повертає один екземпляр свого класу. Конструктор Одинака повинен бути прихований від клієнтів. Виклик методу getInstance повинен стати єдиним способом отримати об’єкт цього класу

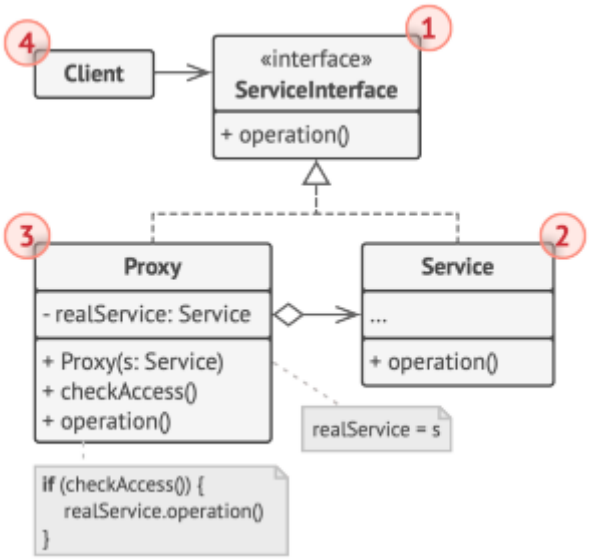

Рис.1.7 Структура патерну PROXY

1. Інтерфейс сервісу визначає загальний інтерфейс для сервісу й замісника. Завдяки цьому об’єкт замісника можна використовувати там, де очікується об’єкт сервісу.
2. Сервіс містить корисну бізнес-логіку.
3. Замісник зберігає посилання на об’єкт сервісу. Після того, як замісник закінчує свою роботу (наприклад, ініціалізацію, логування, захист або інше), він передає виклики вкладеному сервісу. Замісник може сам відповідати за створення й видалення об’єкта сервісу.
4. Клієнт працює з об’єктами через інтерфейс сервісу. Завдяки цьому його можна «обдурити», підмінивши об’єкт сервісу об’єктом замісника.

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 лініву ініціалізацію сервісного об'єкта при першому зверненні клієнта до методів замісника.

Переваги

- Дозволяє контролювати сервісний об'єкт непомітно для клієнта.
- Може працювати, навіть якщо сервісний об'єкт ще не створено.
- Може контролювати життєвий цикл службового об'єкта.

Недоліки

- Ускладнює код програми внаслідок введення додаткових класів.
- Збільшує час отримання відклику від сервісу.

Шаблон «STATE»

Стан(State) — це поведінковий патерн проектування, що дає змогу об'єктам змінювати поведінку в залежності від їхнього стану. Ззовні створюється враження, ніби змінився клас об'єкта.

Проблема
Патерн Стан неможливо розглядати у відриві від концепції машини станів, також відомої як стейт-машина або скінченний автомат.

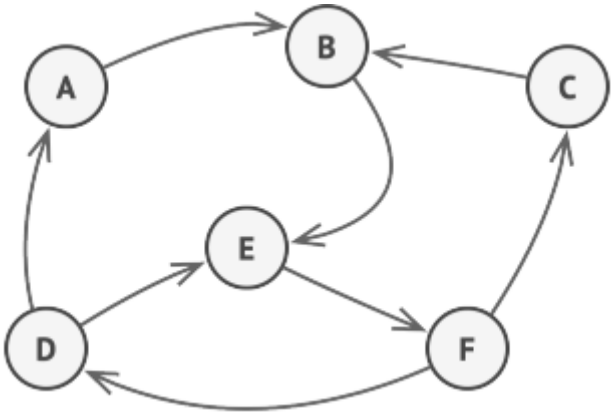

Рис.1.8 Скінченний автомат.

Основна ідея в тому, що програма може знаходитися в одному з кількох станів, які увесь час змінюють один одного. Набір цих станів, а також переходів між ними, визначений наперед та скінченний. Перебуваючи в різних станах, програма може по-різному реагувати на одні і ті самі події, що відбуваються з нею.

Такий підхід можна застосувати і до окремих об’єктів. Наприклад, об’єкт Документ може приймати три стани: Чернетка , Модерація або Опублікований . У кожному з цих станів метод опублікувати працюватиме по-різному:

- З чернетки він надішле документ на модерацію.
- З модерації — в публікацію, але за умови, що це зробив адміністратор.
- В опублікованому стані метод не буде робити нічого

Рис.1.9 Можливі стани документу та переходи між ними.

Машину станів найчастіше реалізують за допомогою множини умовних операторів, if або switch , які перевіряють поточний стан об’єкта та виконують відповідну поведінку. Ймовірніше за все, ви вже реалізували у своєму житті хоча б одну машину станів, навіть не знаючи про це. Не вірите? Як щодо такого коду, виглядає знайомо?

```
class Document is
  field state: string
  // ...
  method publish() is
    switch (state)
      "draft":
        state = "moderation"
        break
      "moderation":
        if (currentUser.role == "admin")
          state = "published"
        break
      "published":
        // Do nothing.
        break
  // ...
```

Рис.1.10 Псевдокод машини станів

Побудована таким чином машина станів має критичну ваду, яка покаже себе, якщо до Документа додати ще з десяток станів. Кожен метод буде складатися з об'ємного умовного оператора, який перебирає доступні стани.

Такий код дуже складно підтримувати. Навіть найменша зміна логіки переходів змусить вас перевіряти роботу всіх методів, які містять умовні оператори машини станів.

Плутанина та нагромадження умов особливо сильно проявляється в старих проектах. Набір можливих станів буває важко визначити заздалегідь, тому вони увесь час додаються в процесі еволюції програми. Через це рішення, що здавалося простим і ефективним на початку розробки проекту, може згодом стати проекцією величезного макаронного монстра.

Рішення

Патерн Стан пропонує створити окремі класи для кожного стану, в якому може перебувати контекстний об'єкт, а потім винести туди поведінки, що відповідають цим станам.

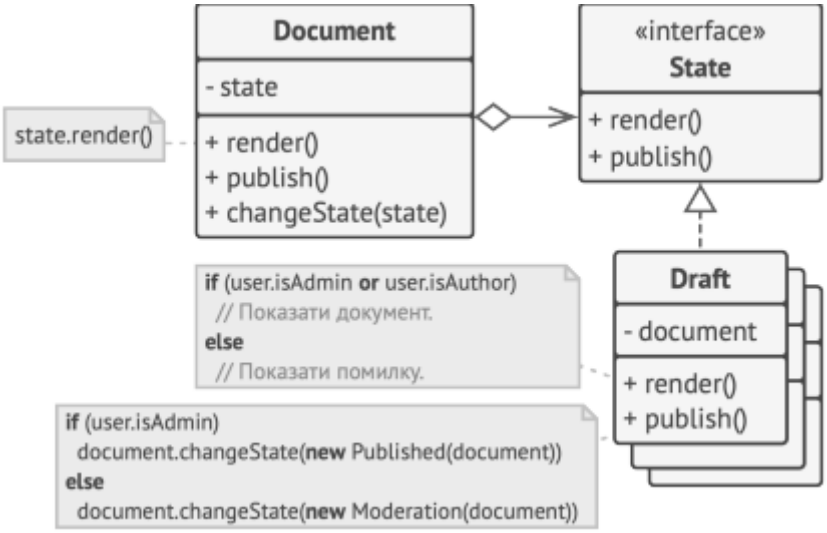

Рис.1.11 Сторінка делегує виконання своєму активному стану.

Замість того, щоб зберігати код всіх станів, початковий об'єкт, який зветься контекстом, міститиме посилання на один з об'єктів-станів і делегуватиме йому роботу в залежності від стану.

Завдяки тому, що об’єкти станів матимуть спільний інтерфейс, контекст зможе делегувати роботу стану, не прив’язуючись до його класу. Поведінку контексту можна буде змінити в будь-який момент, підключивши до нього інший об’єкт-стан.

Дуже важливим нюансом, який відрізняє цей патерн від Стратегії, є те, що і контекст, і конкретні стани можуть знати один про одного та ініціювати переходи від одного стану до іншого.

Аналогія з життя

Ваш смартфон поводиться по-різному в залежності від поточного стану:

- Якщо телефон розблоковано, натискання кнопок телефону призведе до якихось дій.
- Якщо телефон заблоковано, натискання кнопок призведе до появи екрану розблокування.
- Якщо телефон розряджено, натискання кнопок призведе до появи екрану зарядки.

Структура

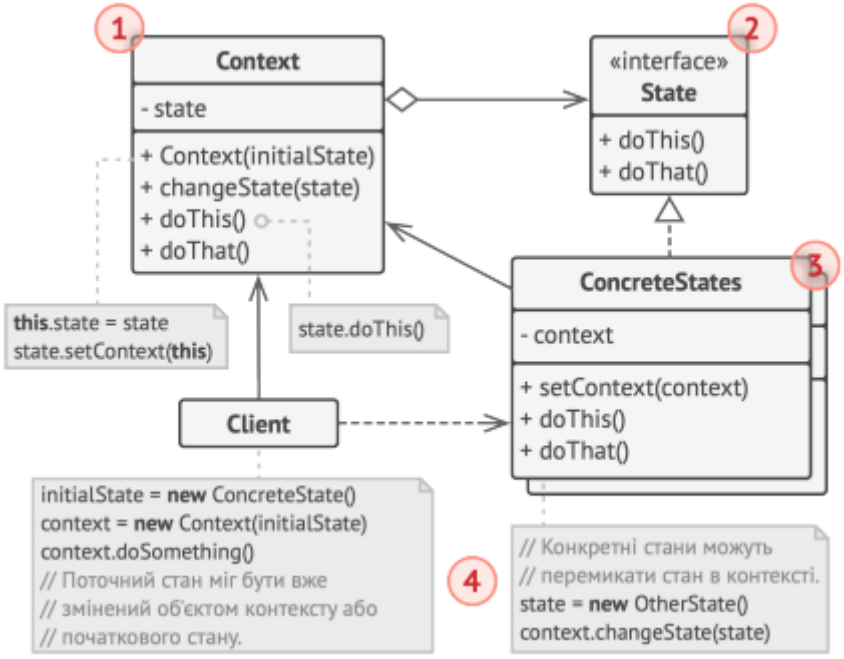

Рис.1.12 Структура патерну State

1. Контекст зберігає посилання на об’єкт стану та делегує йому частину роботи, яка залежить від станів. Контекст працює з цим об’єктом через загальний інтерфейс станів. Контекст повинен мати метод для присвоєння йому нового об’єктастану
2. Стан описує спільний для всіх конкретних станів інтерфейс.
3. Конкретні стани реалізують поведінки, пов’язані з певним станом контексту. Іноді доводиться створювати цілі ієрархії класів станів, щоб узагальнити дублюючий код. Стан може мати зворотнє посилання на об’єкт контексту. Через нього не тільки зручно отримувати з контексту потрібну інформацію, але й здійснювати зміну стану.
4. І контекст, і об’єкти конкретних станів можуть вирішувати, коли і який стан буде обрано наступним. Щоб перемкнути стан, потрібно подати інший об’єкт-стан до контексту.

Застосування

Якщо у вас є об’єкт, поведінка якого кардинально змінюється в залежності від внутрішнього стану, причому типів станів багато, а їхній код часто змінюється.

Патерн пропонує виділити в окремі класи всі поля й методи, пов’язані з визначеним станом. Початковий об’єкт буде постійно посилатися на один з об’єктів-станів, делегуючи йому частину своєї роботи. Для зміни стану до контексту достатньо буде підставляти інший об’єкт-стан.

Якщо код класу містить безліч великих, схожих один на одного умовних операторів, які вибирають поведінки в залежності від поточних значень полів класу.
Патерн пропонує перемістити кожну гілку такого умовного оператора до власного класу. Сюди ж можна поселити й усі поля, пов'язані з цим станом

Якщо ви свідомо використовуєте табличну машину станів, побудовану на умовних операторах, але змушені миритися з дублюванням коду для схожих станів та переходів.
Патерн Стан дозволяє реалізувати ієрархічну машину станів, що базується на наслідуванні. Ви можете успадкувати схожі стани від одного батьківського класу та винести туди весь дублюючий код.

Кроки реалізації

1. Визначтеся з класом, який відіграватиме роль контексту. Це може бути як існуючий клас, який вже має залежність від стану, так і новий клас, якщо код станів «розмазаний» по кількох класах.
2. Створіть загальний інтерфейс станів. Він повинен описувати методи, спільні для всіх станів, виявлених у контексті. Зверніть увагу, що не всю поведінку контексту потрібно переносити до стану, а тільки ту, яка залежить від станів.
3. Для кожного фактичного стану створіть клас, який реалізує інтерфейс стану. Перемістіть код, пов'язаний з конкретними станами, до потрібних класів. Зрештою, всі методи інтерфейсу стану повинні бути реалізовані в усіх класах станів.
   При перенесенні поведінки з контексту ви можете зіткнутися з тим, що ця поведінка залежить від приватних полів або методів контексту, до яких немає доступу з об'єкта стану. Є кілька способів, щоб обійти цю проблему.
   Найпростіший — залишити поведінку всередині контексту, викликаючи його з об'єкта стану. З іншого боку, ви можете зробити класи станів вкладеними до класу контексту, і тоді вони отримають доступ до всіх приватних частин контексту. Останній спосіб, щоправда, доступний лише в деяких мовах програмування (наприклад, Java, C#).
4. Створіть в контексті поле для зберігання об'єктів-станів, а також публічний метод для зміни значення цього поля.
5. Старі методи контексту, в яких перебував залежний від стану код, замініть на виклики відповідних методів об'єкта-стану.
6. В залежності від бізнес-логіки, розмістіть код, який перемикає стан контексту, або всередині контексту, або всередині класів конкретних станів.

Переваги

- Позбавляє від безлічі великих умовних операторів машини станів.
- Концентрує в одному місці код, пов'язаний з певним станом.
- Спрощує код контексту.

Недоліки

- Може невиправдано ускладнити код, якщо станів мало, і вони рідко змінюються.

Шаблон «STRATEGY»

Стратегія — це поведінковий патерн проектування, який визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі. Після цього алгоритми можна заміняти один на інший прямо під час виконання програми

Проблема
Ви вирішили написати програму-навігатор для подорожуючих. Вона повинна показувати гарну й зручну карту, яка дозволяла б з легкістю орієнтуватися в незнайомому місті. Однією з найбільш очікуваних

функцій був пошук та прокладання маршрутів. Перебуваючи в невідомому йому місті, користувач повинен мати можливість вказати початкову точку та пункт призначення, а навігатор, в свою чергу, прокладе оптимальний шлях. Перша версія вашого навігатора могла прокладати маршрут лише автомобільними шляхами, тому чудово підходила для подорожей автомобілем. Але, вочевидь, не всі їздять у відпустку автомобілями. Тому наступним кроком ви додали до навігатора можливість прокладання піших маршрутів. Через деякий час з'ясувалося, що частина туристів під час пересування містом віддають перевагу громадському транспорту. Тому ви додали ще й таку опцію прокладання шляху. Але й це ще не все. У найближчій перспективі ви хотіли б додати прокладку маршрутів велодоріжками, а у віддаленому майбутньому — маршрути, пов'язані з відвідуванням цікавих та визначних місць.

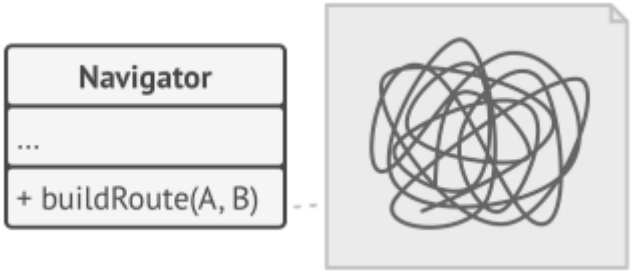

Рис.1.13 Код навігатора стає занадто роздутим.

Якщо з популярністю навігатора не було жодних проблем, то технічна частина викликала запитання й періодичний головний біль. З кожним новим алгоритмом код основного класу навігатора збільшувався вдвічі. В такому великому класі стало важкувато орієнтуватися. Будь-яка зміна алгоритмів пошуку, чи то виправлення багів, чи додавання нового алгоритму, зачіпала основний клас. Це підвищувало ризик створення помилки шляхом випадкового внесення змін до робочого коду. Крім того, ускладнювалася командна робота з іншими програмістами, яких ви найняли після успішного релізу навігатора. Ваші зміни нерідко торкалися одного і того самого коду, створюючи конфлікти, які вимагали додаткового часу на їхнє вирішення.

Рішення

Патерн Стратегія пропонує визначити сімейство схожих алгоритмів, які часто змінюються або розширюються, й винести їх до власних класів, які називають стратегіями. Замість того, щоб початковий клас сам виконував той чи інший алгоритм, він відіграватиме роль контексту, посилаючись на одну зі стратегій та делегуючи їй виконання роботи. Щоб змінити алгоритм, вам буде достатньо підставити в контекст інший об'єкт-стратегію. Важливо, щоб всі стратегії мали єдиний інтерфейс. Використовуючи цей інтерфейс, контекст буде незалежним від конкретних класів стратегій. З іншого боку, ви зможете змінювати та додавати нові види алгоритмів, не чіпаючи код контексту

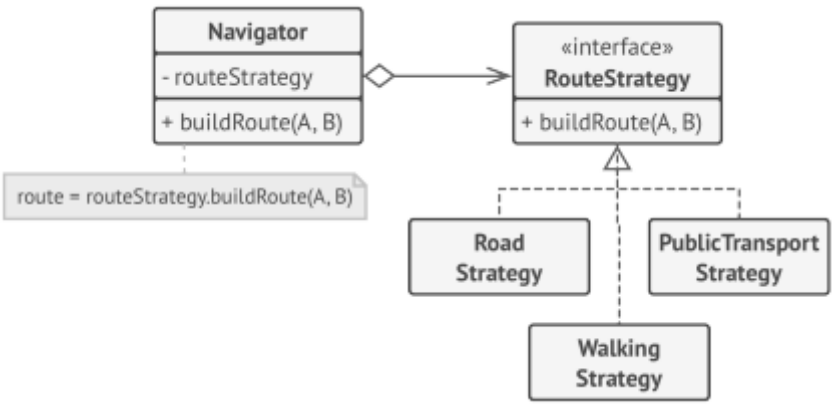

Рис.1.14 Стратегії побудови шляху

У нашому прикладі кожен алгоритм пошуку шляху переїде до свого власного класу. В цих класах буде визначено лише один метод, що приймає в параметрах координати початку та кінця маршруту, а повертає масив всіх точок маршруту. Хоча кожен клас прокладатиме маршрут на свій розсуд, для

навігатора це не буде мати жодного значення, оскільки його робота полягає тільки у зображенні маршруту. Навігатору достатньо подати до стратегії дані про початок та кінець маршруту, щоб отримати масив точок маршруту в обумовленому форматі. Клас навігатора буде мати метод для встановлення стратегії, що дозволить змінювати стратегію пошуку шляху «на льоту». Цей метод стане у нагоді клієнтському коду навігатора, наприклад, кнопкам-перемикачам типів маршрутів в інтерфейсі користувача.

Аналогія з життя

Вам потрібно дістатися аеропорту. Можна доїхати автобусом, таксі або велосипедом. Тут вид транспорту є стратегією. Ви вибираєте конкретну стратегію в залежності від контексту — наявності грошей або часу до відльоту.

Структура

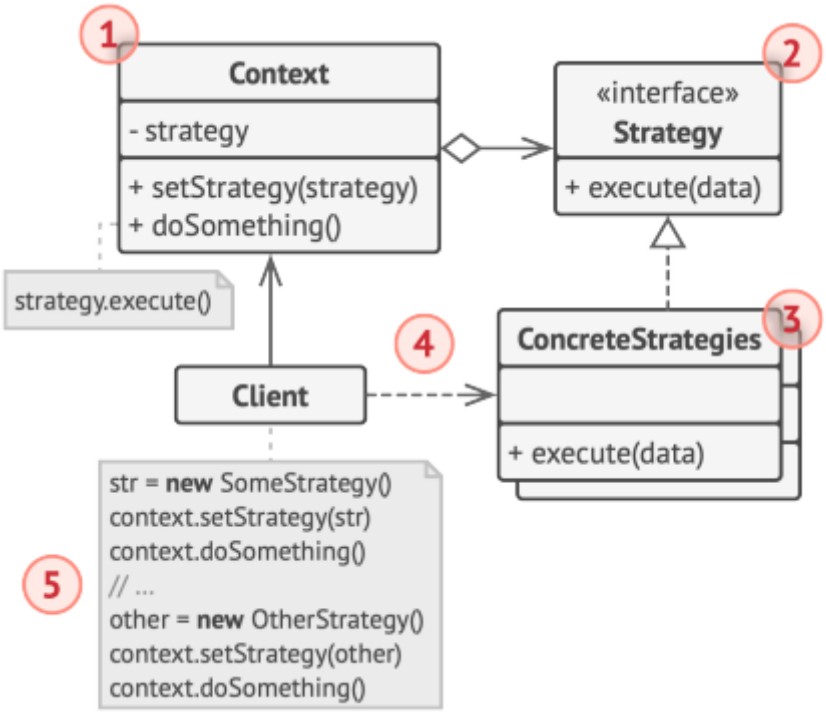

Рис.1.15 Структура патерну Strategy

1. Контекст зберігає посилання на об’єкт конкретної стратегії, працюючи з ним через загальний інтерфейс стратегій.
2. Стратегія визначає інтерфейс, спільний для всіх варіацій алгоритму. Контекст використовує цей інтерфейс для виклику алгоритму Для контексту неважливо, яка саме варіація алгоритму буде обрана, оскільки всі вони мають однаковий інтерфейс.
3. Конкретні стратегії реалізують різні варіації алгоритму
4. Під час виконання програми контекст отримує виклики від клієнта й делегує їх об’єкту конкретної стратегії
5. Клієнт повинен створити об’єкт конкретної стратегії та передати його до конструктора контексту. Крім того, клієнт повинен мати можливість замінити стратегію на льоту, використовуючи сетер поля стратегії. Завдяки цьому, контекст не знатиме про те, яку саме стратегію зараз обрано.

Застосування

Якщо вам потрібно використовувати різні варіації якогонебудь алгоритму всередині одного об’єкта. Стратегія дозволяє варіювати поведінку об’єкта під час виконання програми, підставляючи до нього різні об’єкти-поведінки (наприклад, що відрізняються балансом швидкості та споживання ресурсів).

Якщо у вас є безліч схожих класів, які відрізняються лише деякою поведінкою.
Стратегія дозволяє відокремити поведінку, що відрізняється, у власну ієрархію класів, а потім звести початкові класи до одного, налаштовуючи його поведінку стратегіями.

Якщо ви не хочете оголювати деталі реалізації алгоритмів для інших класів.
Стратегія дозволяє ізолювати код, дані й залежності алгоритмів від інших об'єктів, приховавши ці деталі всередині класів-стратегій.

Якщо різні варіації алгоритмів реалізовано у вигляді розлогого умовного оператора. Кожна гілка такого оператора є варіацією алгоритму.
Стратегія розміщує кожну лапу такого оператора до окремого класу-стратегії. Потім контекст отримує певний об'єктстратегію від клієнта й делегує йому роботу. Якщо раптом знадобиться змінити алгоритм, до контексту можна подати іншу стратегію.

Кроки реалізації

1. Визначте алгоритм, що схильний до частих змін. Також підійде алгоритм, який має декілька варіацій, які обираються під час виконання програми.
2. Створіть інтерфейс стратегій, що описує цей алгоритм. Він повинен бути спільним для всіх варіантів алгоритму
3. Помістіть варіації алгоритму до власних класів, які реалізують цей інтерфейс.
4. У класі контексту створіть поле для зберігання посилання на поточний об'єкт-стратегію, а також метод для її зміни. Переконайтеся в тому, що контекст працює з цим об'єктом тільки через загальний інтерфейс стратегій.
5. Клієнти контексту мають подавати до нього відповідний об'єкт-стратегію, коли хочуть, щоб контекст поводився певним чином.

Переваги

- Гаряча заміна алгоритмів на льоту.
- Ізолює код і дані алгоритмів від інших класів.
- Заміна спадкування делегуванням.
- Реалізує принцип відкритості/закритості.

Недоліки

- Ускладнює програму внаслідок додаткових класів.
- Клієнт повинен знати, в чому полягає різниця між стратегіями, щоб вибрати потрібну.

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.

2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

3. Застосування одного з розглянутих шаблонів при реалізації програми

Хід роботи:

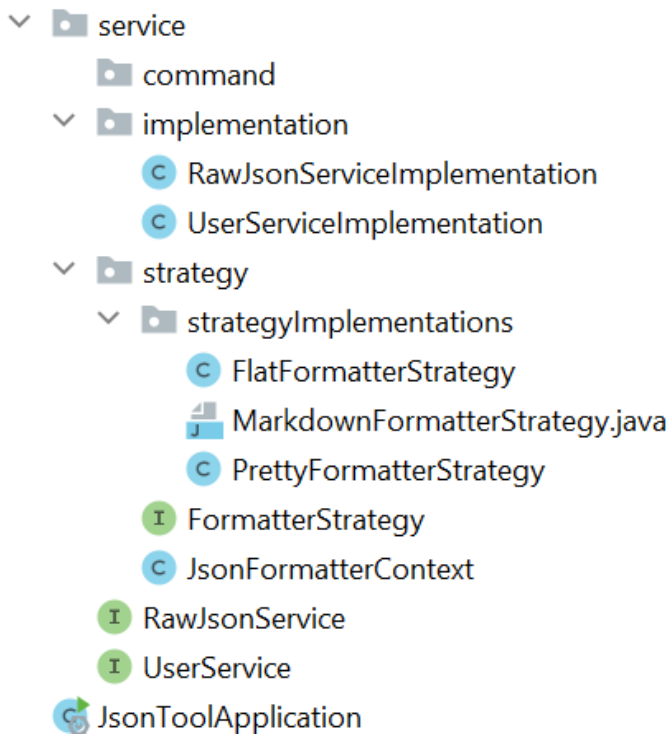

Рис. 1. Ієрархія класів. Реалізація патерну Strategy на рівні сервісів.

```
7 usages   2 implementations
public interface FormatterStrategy {
    2 implementations
    String format(RawJsonDto rawJsonDto);
};
```

Рис. 2. Інтерфейс FormatterStrategy для реалізації патерну Strategy.

```
public class FlatFormatterStrategy implements FormatterStrategy {

    1 usage
    private final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public String format(RawJsonDto rawJsonDto) {
        try {
            JsonNode rootNode = objectMapper.readTree(rawJsonDto.getSchemaData());
            StringBuilder builder = new StringBuilder();
            formatNode(rootNode, currentPath: "", builder);
            return builder.toString().trim();
        } catch (Exception e) {
            throw new InvalidJsonException("Invalid JSON input", e);
        }
    }
```

Рис. 2.1.1 Метод format класу FlatFormatterStrategy

```
private void formatNode(JsonNode node, String currentPath, StringBuilder builder) {
    if (node.isObject()) {
        Iterator<Map.Entry<String, JsonNode>> fields = node.fields();
        while (fields.hasNext()) {
            Map.Entry<String, JsonNode> field = fields.next();
            String newPath = currentPath.isEmpty() ? field.getKey() : currentPath + "." + field.getKey();
            formatNode(field.getValue(), newPath, builder);
        }
    } else if (node.isArray()) {
        for (int i = 0; i < node.size(); i++) {
            String newPath = currentPath + "[" + i + "]";
            formatNode(node.get(i), newPath, builder);
        }
    } else {
        builder.append(currentPath)
                .append(": ")
                .append(node.isTextual() ? "\"" + node.asText() + "\"" : node.toString())
                .append("\n");
    }
}
```

Рис. 2.1.2 Метод formatNode класу FlatFormatterStrategy

```java
public class PrettyFormatterStrategy implements FormatterStrategy {

    2 usages
    private final ObjectMapper objectMapper;

    1 usage
    public PrettyFormatterStrategy() {
        this.objectMapper = new ObjectMapper();
    }

    @Override
    public String format(RawJsonDto rawJsonDto) {
        try {
            JsonNode rootNode = objectMapper.readTree(rawJsonDto.getSchemaData());
            return formatNode(rootNode, level: 0);
        } catch (Exception e) {
            throw new InvalidJsonException("Invalid JSON input", e);
        }
    }
```

Рис. 2.2.1 Метод format класу PrettyFormatterStrategy

```java
    private String formatNode(JsonNode node, int level) {
        StringBuilder builder = new StringBuilder();

        if (node.isObject()) {
            builder.append("{\n");
            node.fields().forEachRemaining(entry -> {
                appendIndent(builder, level: level + 1);
                builder.append("\"").append(entry.getKey()).append("\": ");
                builder.append(formatNode(entry.getValue(), level: level + 1)).append(",\n");
            });
            removeTrailingComma(builder);
            builder.append("\n");
            appendIndent(builder, level);
            builder.append("}");
        } else if (node.isArray()) {
            builder.append("[\n");
            node.forEach(childNode -> {
                appendIndent(builder, level: level + 1);
                builder.append(formatNode(childNode, level: level + 1)).append(",\n");
            });
            removeTrailingComma(builder);
            builder.append("\n");
            appendIndent(builder, level);
            builder.append("]");
        } else {
            if (node.isTextual()) {
                builder.append("\"").append(node.asText()).append("\"");
            } else {
                builder.append(node.toString());
            }
        }

        return builder.toString();
    }
```

Рис. 2.2.2 Метод formatNode класу PrettyFormatterStrategy

```
private void appendIndent(StringBuilder builder, int level) {
    builder.append("  ".repeat(level));
}

2 usages
private void removeTrailingComma(StringBuilder builder) {
    int length = builder.length();
    if (length > 2 && builder.substring( start: length - 2).equals(",\n")) {
        builder.delete(length - 2, length);
    }
}
```

Рис. 2.2.3 Допоміжні методи класу PrettyFormatterStrategy

```
public class JsonFormatterContext {
    3 usages
    private FormatterStrategy formatterStrategy;

    2 usages
    public void setFormatterStrategy(FormatterStrategy formatterStrategy) {
        this.formatterStrategy = formatterStrategy;
    }

    public String format(RawJsonDto jsonInput) {
        if (formatterStrategy == null) {
            throw new IllegalStateException("No formatter strategy set");
        }
        try {
            return formatterStrategy.format(jsonInput);
        } catch (InvalidJsonException e) {
            System.err.println("Error formatting JSON: " + e.getMessage());
            throw e;
        }
    }
}
```

Рис. 2.3 Клас JsonFormatterContext

```
1 usage
@Override
public String formatJson(RawJsonDto rawJsonDto, String formatType) {
    try {
        switch (formatType.toLowerCase()) {
            case "flat":
                formatterContext.setFormatterStrategy(new FlatFormatterStrategy());
                break;
            case "pretty":
                formatterContext.setFormatterStrategy(new PrettyFormatterStrategy());
                break;
            default:
                throw new IllegalArgumentException("Unknown format type: " + formatType);
        }
        return formatterContext.format(rawJsonDto);
    } catch (InvalidJsonException e) {
        System.err.println("Error: " + e.getMessage());
        return "{}";
    }
}
```

Рис. 2.4.1 Метод formatJson сервісу RawJsonServiceImplementation

```
@PostMapping("/format/{type}")
public ResponseEntity<String> formatJsonPretty(@RequestBody RawJsonDto rawJsonDto, @PathVariable("type") String type) {
    try {
        String formattedJson = rawJsonService.formatJson(rawJsonDto, type);
        return ResponseEntity.ok(formattedJson);
    } catch (RuntimeException e) {
        return ResponseEntity.badRequest().body("Invalid JSON input: " + e.getMessage());
    }
}
```

Рис. 2.4.2 Виклик методу formatJson сервісу RawJsonServiceImplementation у контролері RawJsonController

У межах розробленого застосунку патерн Strategy використовується для реалізації динамічного вибору способів форматування JSON. Основою є контекстний клас JsonFormatterContext, який приймає стратегії форматування у вигляді інтерфейсу FormatterStrategy. Кожна стратегія реалізує метод format, адаптуючи алгоритм форматування до специфічних вимог. Наприклад, PrettyFormatterStrategy форматує JSON у вигляді дерева з чіткою ієрархією, а FlatFormatterStrategy — у компактний рядок. Логіка вибору стратегії визначається у сервісному класі RawJsonServiceImplementation, де на основі переданого параметра типу форматування (pretty, flat) контекст застосовує відповідну стратегію.

ВИСНОВОК

Реалізація патерну Strategy у межах даного застосунку забезпечила розширюваність і гнучкість системи форматування JSON. Класи і методи були побудовані так, щоб динамічно адаптувати поведінку залежно від вимог користувача, не порушуючи принципів SOLID. Цей підхід спрощує підтримку та інтеграцію нових стратегій, що робить розроблену систему ефективною та масштабованою.