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 №6
із дисципліни «Технології Розробки Програмного Забезпечення»
Тема: ШАБЛОНИ «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»
Варіант-28

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 обробки та оновлення метаданих JSON. У процесі реалізації застосувати шаблон проектування Observer для автоматичного сповіщення підписаних компонентів про зміни в метаданих.

Теоритичні відомості:

Шаблон «Abstract Factory»

Абстрактна фабрика — це породжувальний патерн проектування, що дає змогу створювати сімейства пов’язаних об’єктів, не прив’язуючись до конкретних класів створюваних об’єктів

Проблема

Уявіть, що ви пишете симулятор меблевого магазину.
Ваш код містить:
1. Сімейство залежних продуктів. Скажімо, Крісло + Диван + Столик .
2. Кілька варіацій цього сімейства. Наприклад, продукти Крісло , Диван та Столик представлені в трьох різних стилях: Ар-деко , Вікторіанському і Модерн

Рис.2.1 Сімейства продуктів та їхніх варіацій

Вам потрібно створювати об’єкти продуктів у такий спосіб, щоб вони завжди пасували до інших продуктів того самого сімейства. Це дуже важливо, адже клієнти засмучуються, коли отримують меблі, що не можна поєднати між собою.
Крім того, ви не хочете вносити зміни в існуючий код під час додавання в програму нових продуктів або сімейств. Постачальники часто оновлюють свої каталоги, але ви б не хотіли змінювати вже написаний код кожен раз при надходженні нових моделей меблів.

Рішення
Для початку, патерн Абстрактна фабрика пропонує виділити загальні інтерфейси для окремих продуктів, що складають одне сімейство, і описати в них спільну для цих продуктів поведінку. Так, наприклад, усі варіації крісел отримають спільний інтерфейс Крісло, усі дивани реалізують інтерфейс Диван тощо.

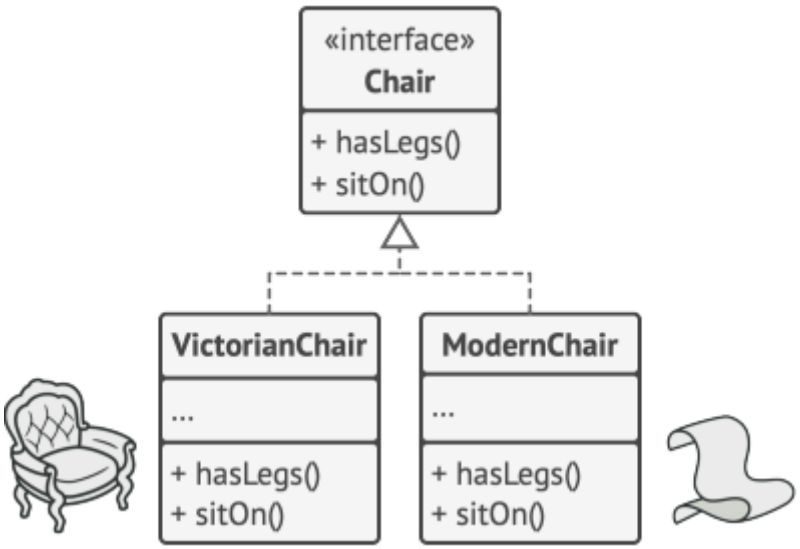

Рис.2.2 Всі варіації одного й того самого об’єкта мають жити в одній ієрархії класів.

Далі ви створюєте абстрактну фабрику — загальний інтерфейс, який містить методи створення всіх продуктів сімейства (наприклад, створитиКрісло , створитиДиван і створитиСтолик). Ці операції повинні повертати абстрактні типи продуктів, представлені інтерфейсами, які ми виділили раніше — Крісла, Дивани і Столики.

Як щодо варіацій продуктів? Для кожної варіації сімейства продуктів ми повинні створити свою власну фабрику, реалізувавши абстрактний інтерфейс. Фабрики створюють продукти однієї варіації. Наприклад, ФабрикаМодерн буде повертати тільки КріслаМодерн, ДиваниМодерн і СтоликиМодерн .

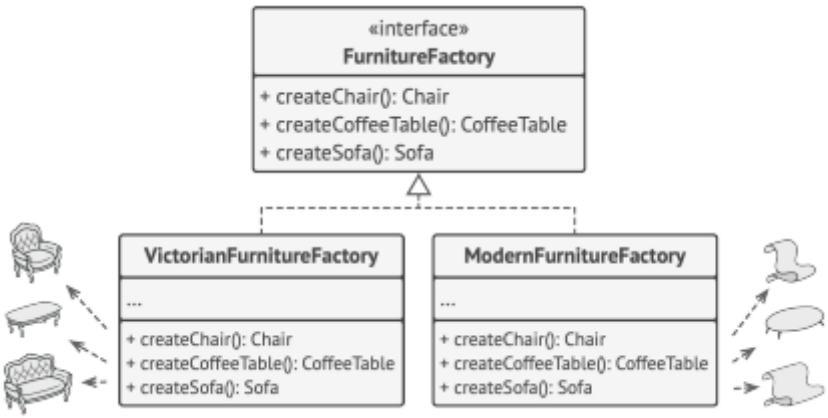

Рис.2.3 Конкретні фабрики відповідають певній варіації сімейства продуктів.

Клієнтський код повинен працювати як із фабриками, так і з продуктами тільки через їхні загальні інтерфейси. Це дозволить подавати у ваші класи будь-які типи фабрик і виробляти будь-які типи продуктів, без необхідності вносити зміни в існуючий код.

Наприклад, клієнтський код просить фабрику зробити стілець. Він не знає, якому типу відповідає ця фабрика. Він не знає, отримає вікторіанський або модерновий стілець. Для нього важливо, щоб на цьому стільці можна було сидіти та щоб цей стілець відмінно виглядав поруч із диваном тієї ж фабрики.

Залишилося прояснити останній момент: хто ж створює об’єкти конкретних фабрик, якщо клієнтський код працює лише із загальними інтерфейсами? Зазвичай програма створює конкретний об’єкт фабрики під час запуску, причому тип фабрики вибирається на підставі параметрів оточення або конфігурації.

Структура

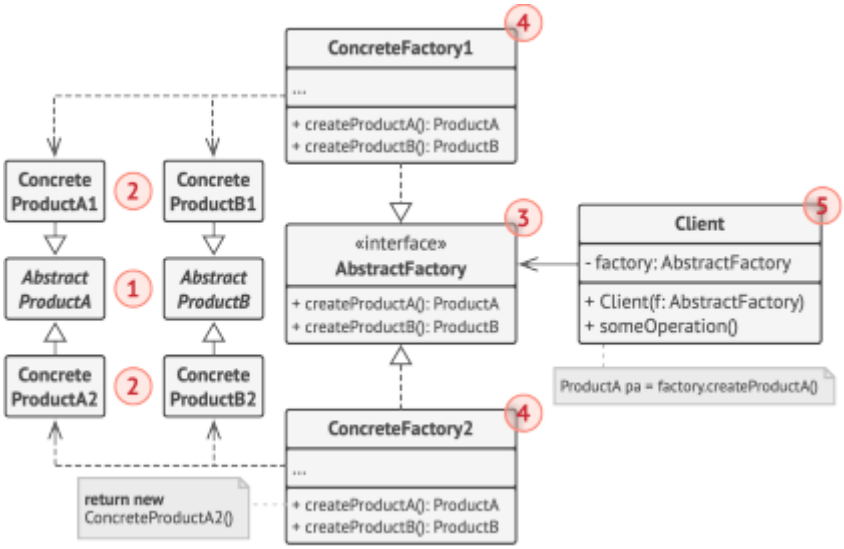

Рис.2.4 Структура патерну Абстрактна фабрика

1. Абстрактні продукти оголошують інтерфейси продуктів, що пов'язані один з одним за змістом, але виконують різні функції.
2. Конкретні продукти — великий набір класів, що належать до різних абстрактних продуктів (крісло/столик), але мають одні й ті самі варіації (Вікторіанський/Модерн).
3. Абстрактна фабрика оголошує методи створення різних абстрактних продуктів (крісло/столик).
4. Конкретні фабрики кожна належить до своєї варіації продуктів (Вікторіанський/Модерн) і реалізує методи абстрактної фабрики, даючи змогу створювати всі продукти певної варіації.
5. Незважаючи на те, що конкретні фабрики породжують конкретні продукти, сигнатури їхніх методів мусять повертати відповідні абстрактні продукти. Це дозволить клієнтського коду, що використовує фабрику, не прив'язуватися до конкретних класів продуктів. Клієнт зможе працювати з будь-якими варіаціями продуктів через абстрактні інтерфейси.

Застосування

Коли бізнес-логіка програми повинна працювати з різними видами пов'язаних один з одним продуктів, незалежно від конкретних класів продуктів.
Абстрактна фабрика приховує від клієнтського коду подробиці того, як і які конкретно об'єкти будуть створені. Внаслідок цього, клієнтський код може працювати з усіма типами створюваних продуктів, так як їхній загальний інтерфейс був визначений заздалегідь.

Коли в програмі вже використовується Фабричний метод, але чергові зміни передбачають введення нових типів продуктів.
У будь-якій добротній програмі кожен клас має відповідати лише за одну річ. Якщо клас має занадто багато фабричних методів, вони здатні затуманити його основну функцію. Тому є сенс у тому, щоб винести усю логіку створення продуктів в окрему ієрархію класів, застосувавши абстрактну фабрику.

Кроки реалізації

1. Створіть таблицю співвідношень типів продуктів до варіацій сімейств продуктів.
2. Зведіть усі варіації продуктів до загальних інтерфейсів.
3. Визначте інтерфейс абстрактної фабрики. Він повинен мати фабричні методи для створення кожного типу продуктів.
4. Створіть класи конкретних фабрик, реалізувавши інтерфейс абстрактної фабрики. Цих класів має бути стільки ж, скільки й варіацій сімейств продуктів.
5. Змініть код ініціалізації програми так, щоб вона створювала певну фабрику й передавала її до клієнтського коду.

6. Замініть у клієнтському коді ділянки створення продуктів через конструктор на виклики відповідних методів фабрики.

Переваги

- Гарантує поєднання створюваних продуктів.
- Звільняє клієнтський код від прив'язки до конкретних класів продукту.
- Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
- Спрощує додавання нових продуктів до програми.
- Реалізує принцип відкритості/закритості.

Недоліки

- Ускладнює код програми внаслідок введення великої кількості додаткових класів.
- Вимагає наявності всіх типів продукту в кожній варіації.

Шаблон «Factory Method»

Фабричний метод — це породжувальний патерн проектування, який визначає загальний інтерфейс для створення об'єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об'єктів.

Проблема

Уявіть, що ви створюєте програму керування вантажними перевезеннями. Спочатку ви плануєте перевезення товарів тільки вантажними автомобілями. Тому весь ваш код працює з об'єктами класу Вантажівка.
Згодом ваша програма стає настільки відомою, що морські перевізники шикуються в чергу і благають додати до програми підтримку морської логістики.
Чудові новини, чи не так?! Але як щодо коду? Велика частина існуючого коду жорстко прив'язана до класів Вантажівок. Щоб додати до програми класи морських Суден, знадобиться перелопачувати весь код. Якщо ж ви вирішите додати до програми ще один вид транспорту, тоді всю цю роботу доведеться повторити.
У підсумку ви отримаєте жахливий код, переповнений умовними операторами, що виконують ту чи іншу дію в залежності від вибраного класу транспорту.

Рішення

Патерн Фабричний метод пропонує відмовитись від безпосереднього створення об'єктів за допомогою оператора new, замінивши його викликом особливого фабричного методу. Не лякайтеся, об'єкти все одно будуть створюватися за допомогою new, але робити це буде фабричний метод

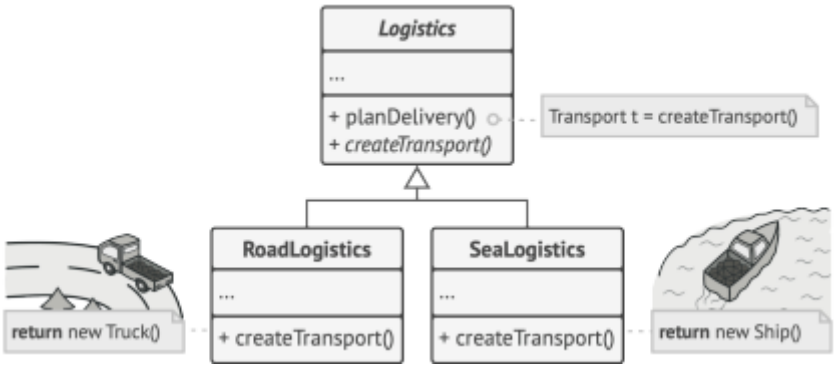

Рис.2.5 Підкласи можуть змінювати клас створюваних об'єктів.

На перший погляд це може здатись безглуздим — ми просто перемістили виклик конструктора з одного кінця програми в інший. Проте тепер ви зможете перевизначити фабричний метод у підкласі, щоб змінити тип створюваного продукту. Щоб ця система запрацювала, всі об’єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть виготовляти об’єкти різних класів, що відповідають одному і тому самому інтерфейсу

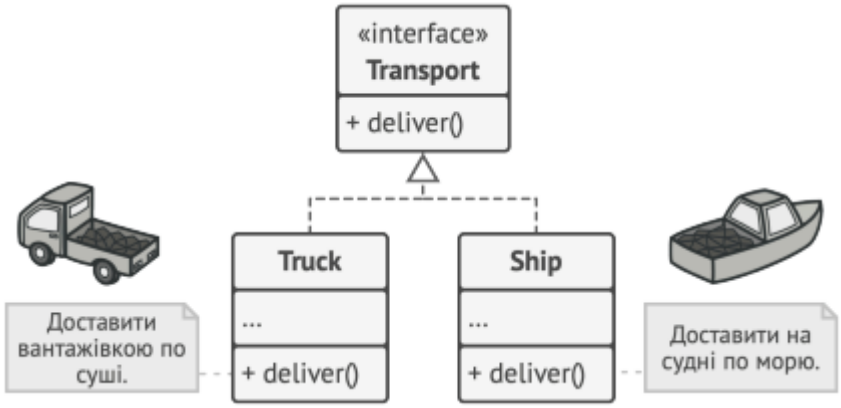

Рис.2.6 Всі об’єкти-продукти повинні мати спільний інтерфейс

Наприклад, класи Вантажівка і Судно реалізують інтерфейс Транспорт з методом доставити . Кожен з цих класів реалізує метод по-своєму: вантажівки перевозять вантажі сушею, а судна — морем. Фабричний метод класу ДорожноїЛогістики поверне об’єкт-вантажівку, а класу МорськоїЛогістики — об’єкт-судно.

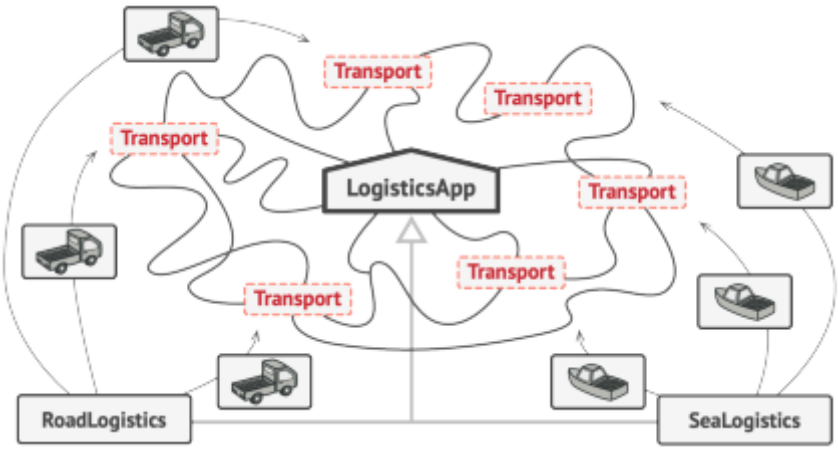

Рис.2.7. Допоки всі продукти реалізують спільний інтерфейс, їхні об’єкти можна змінювати один на інший у клієнтському коді.

Клієнт фабричного методу не відчує різниці між цими об’єктами, адже він трактуватиме їх як якийсь абстрактний Транспорт. Для нього буде важливим, щоб об’єкт мав метод доставити , а не те, як конкретно він працює.

Структура

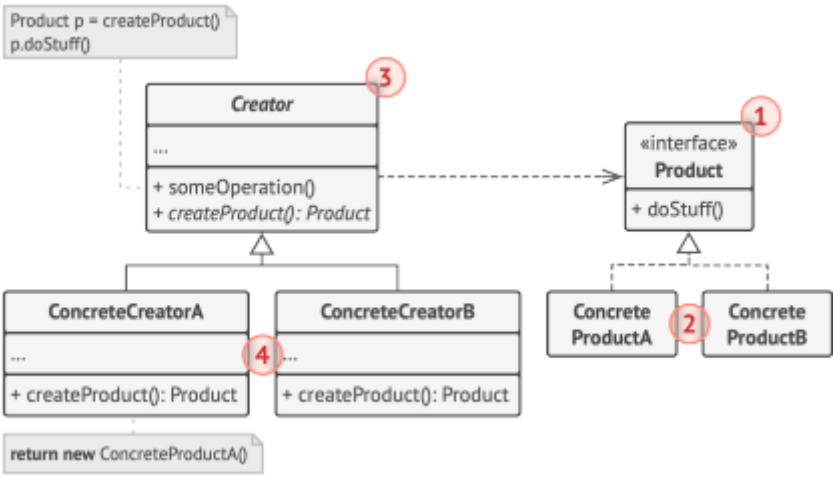

Рис.2.8 Структура патерну Фабричний метод

1. Продукт визначає загальний інтерфейс об'єктів, які може створювати творець та його підкласи.
2. Конкретні продукти містять код різних продуктів. Продукти відрізнятимуться реалізацією, але інтерфейс у них буде спільним.
3. Творець оголошує фабричний метод, який має повертати нові об'єкти продуктів. Важливо, щоб тип результату цього методу співпадав із загальним інтерфейсом продуктів. Зазвичай, фабричний метод оголошують абстрактним, щоб змусити всі підкласи реалізувати його по-своєму. Однак він може також повертати продукт за замовчуванням.
   Незважаючи на назву, важливо розуміти, що створення продуктів не є єдиною і головною функцією творця. Зазвичай він містить ще й інший корисний код для роботи з продуктом.
   Аналогія: у великій софтверній компанії може бути центр підготовки програмістів, але все ж таки основним завданням компанії залишається написання коду, а не навчання програмістів.
4. Конкретні творці по-своєму реалізують фабричний метод, виробляючи ті чи інші конкретні продукти. Фабричний метод не зобов'язаний створювати нові об'єкти увесь час. Його можна переписати так, аби повертати з якогось сховища або кешу вже існуючі об'єкти.

Застосування

Коли типи і залежності об'єктів, з якими повинен працювати ваш код, невідомі заздалегідь.
Фабричний метод відокремлює код виробництва продуктів від решти коду, який використовує ці продукти. Завдяки цьому код виробництва можна розширювати, не зачіпаючи основний код. Щоб додати підтримку нового продукту, вам потрібно створити новий підклас та визначити в ньому фабричний метод, повертаючи звідти екземпляр нового продукту.

Коли ви хочете надати користувачам можливість розширювати частини вашого фреймворку чи бібліотек
Користувачі можуть розширювати класи вашого фреймворку через успадкування. Але як же зробити так, аби фреймворк створював об'єкти цих класів, а не стандартних? Рішення полягає у тому, щоб надати користувачам можливість розширювати не лише бажані компоненти, але й класи, які їх створюють. Тому ці класи повинні мати конкретні створюючі методи, які можна буде перевизначити. Наприклад, ви використовуєте готовий UI-фреймворк для свого додатку. Але — от халепа — вам необхідно мати круглі кнопки, а не стандартні прямокутні. Ви створюєте клас RoundButton . Але як сказати головному класу фреймворку UIFramework , щоб він почав тепер створювати круглі кнопки замість стандартних прямокутних?
Для цього з базового класу фреймворку ви створюєте підклас UIWithRoundButtons , перевизначаєте в ньому метод створення кнопки (а-ля, createButton) і вписуєте туди створення свого класу кнопок. Потім використовуєте UIWithRoundButtons замість стандартного UIFramework .

Коли ви хочете зекономити системні ресурси, повторно використовуючи вже створені об'єкти, замість породження нових
Така проблема зазвичай виникає під час роботи з «важкими», вимогливими до ресурсів об'єктами, такими, як підключення до бази даних, файлової системи й подібними. Уявіть, скільки дій вам потрібно зробити, аби повторно використовувати вже існуючі об'єкти:

1. Спочатку слід створити загальне сховище, щоб зберігати в ньому всі створювані об'єкти
2. При запиті нового об'єкта потрібно буде подивитись у сховище та перевірити, чи є там невикористаний об'єкт.
3. Потім повернути його клієнтському коду
4. Але якщо ж вільних об'єктів немає, створити новий, не забувши додати його до сховища
   Увесь цей код потрібно десь розмістити, щоб не засмічувати клієнтський код.
   Найзручнішим місцем був би конструктор об'єкта, адже всі ці перевірки потрібні тільки під час створення об'єктів, але, на жаль, конструктор завжди створює нові об'єкти, тому він не може повернути існуючий екземпляр.
   Отже, має бути інший метод, який би віддавав як існуючі, так і нові об'єкти. Ним і стане фабричний метод.

Кроки реалізації

1. Приведіть усі створювані продукти до загального інтерфейсу.
2. Створіть порожній фабричний метод у класі, який виробляє продукти. В якості типу, що повертається, вкажіть загальний інтерфейс продукту
3. Пройдіться по коду класу й знайдіть усі ділянки, що створюють продукти. По черзі замініть ці ділянки викликами фабричного методу, переносячи в нього код створення різних продуктів.
   Можливо, доведеться додати до фабричного методу декілька параметрів, що контролюють, який з продуктів потрібно створити
   Імовірніше за все, фабричний метод виглядатиме гнітюче на цьому етапі. В ньому житиме великий умовний оператор, який вибирає клас створюваного продукту. Але не хвилюйтеся, ми ось-ось все це виправимо.
4. Для кожного типу продуктів заведіть підклас і перевизначте в ньому фабричний метод. З суперкласу перемістіть туди код створення відповідного продукту.
5. Якщо створюваних продуктів занадто багато для існуючих підкласів творця, ви можете подумати про введення параметрів до фабричного методу, аби повертати різні продукти в межах одного підкласу.
   Наприклад, у вас є клас Пошта з підкласами АвіаПошта і НаземнаПошта, а також класи продуктів Літак , Вантажівка й Потяг . Авіа відповідає Літакам , але для НаземноїПошти є відразу два продукти. Ви могли б створити новий підклас пошти й для потягів, але проблему можна вирішити по-іншому. Клієнтський код може передавати до фабричного методу НаземноїПошти аргумент, що контролює, який з продуктів буде створено.
6. Якщо після цих всіх переміщень фабричний метод став порожнім, можете зробити його абстрактним. Якщо ж у ньому щось залишилося — не страшно, це буде його типо□вою реалізацією (за замовчуванням)

Переваги

- Позбавляє клас від прив'язки до конкретних класів продуктів.
- Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
- Спрощує додавання нових продуктів до програми.
- Реалізує принцип відкритості/закритості.

Недоліки

Може призвести до створення великих паралельних ієрархій класів, адже для кожного класу продукту потрібно створити власний підклас творця.

Шаблон «Memento»

Знімок — це поведінковий патерн проектування, що дає змогу зберігати та відновлювати минулий стан об’єктів, не розкриваючи подробиць їхньої реалізації

Проблема

Припустімо, ви пишете програму текстового редактора. Крім звичайного редагування, ваш редактор дозволяє змінювати форматування тексту, вставляти малюнки та інше. В певний момент ви вирішили надати можливість скасовувати усі ці дії. Для цього вам потрібно зберігати поточний стан редактора перед тим, як виконати будь-яку дію. Якщо користувач вирішить скасувати свою дію, ви візьмете копію стану з історії та відновите попередній стан редактора.

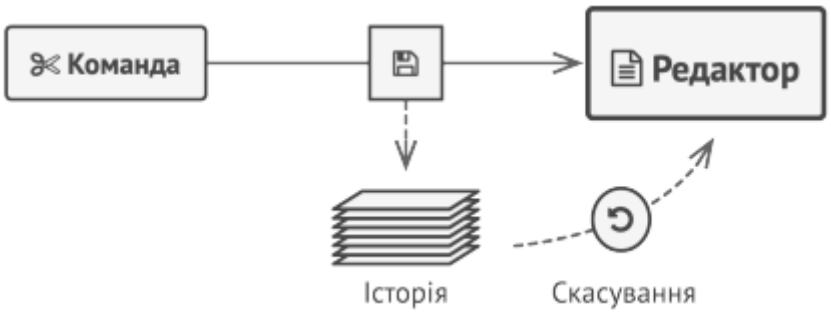

Рис.2.9 Перед виконанням команди ви можете зберегти копію стану редактора, щоб потім мати можливість скасувати операцію.

Щоб зробити копію стану об’єкта, достатньо скопіювати значення полів. Таким чином, якщо ви зробили клас редактора достатньо відкритим, то будь-який інший клас зможе зазирнути всередину, щоб скопіювати його стан. Здавалося б, які проблеми? Тепер будь-яка операція зможе зробити резервну копію редактора перед виконанням своєї дії. Але такий наївний підхід забезпечить вам безліч про□блем у майбутньому. Адже, якщо ви вирішите провести рефакторинг — прибрати або додати кілька полів до класу редактора — доведеться змінювати код усіх класів, які могли копіювати стан редактора.

Рис.2.10 Як команді створити знімок стану редактора, якщо всі його поля приватні?

Але це ще не все. Давайте тепер поглянемо безпосередньо на копії стану, які ми створювали. З чого складається стан редактора? Навіть найпримітивніший редактор повинен мати декілька полів для зберігання поточного тексту, позиції курсора та прокручування екрану. Щоб зробити копію стану, вам потрібно додати значення всіх цих полів до деякого «контейнера». Імовірно, вам знадобиться зберігати масу таких контейнерів в якості історії операцій, тому зручніше за все зробити їх об’єктами одного класу. Цей клас повинен мати багато полів, але практично жодного методу. Щоб інші об’єкти могли записувати та читати з нього дані, вам доведеться зробити його поля публічними. Проте це призведе до тієї ж проблеми, що й з відкритим класом редактора. Інші класи стануть залежними від будь-яких змін класу контейнера, який схильний до таких самих змін, що і клас редактора.

Виходить, що нам доведеться або відкрити класи для всіх бажаючих, отримавши постійний клопіт з підтримкою коду, або залишити класи закритими, відмовившись від ідеї скасування операцій. Чи немає тут альтернативи?

Рішення

Усі проблеми, описані вище, виникають через порушення інкапсуляції, коли одні об'єкти намагаються зробити роботу за інших, проникаючи до їхньої приватної зони, щоб зібрати необхідні для операції дані.
Патерн Знімок доручає створення копії стану об'єкта самому об'єкту, який цим станом володіє. Замість того, щоб робити знімок «ззовні», наш редактор сам зробить копію своїх полів, адже йому доступні всі поля, навіть приватні.
Патерн пропонує тримати копію стану в спеціальному об'єкті-знімку з обмеженим інтерфейсом, що дозволяє, наприклад, дізнатися дату виготовлення або назву знімка. Проте, знімок повинен бути відкритим для свого творця і дозволяти прочитати та відновити його внутрішній стан.

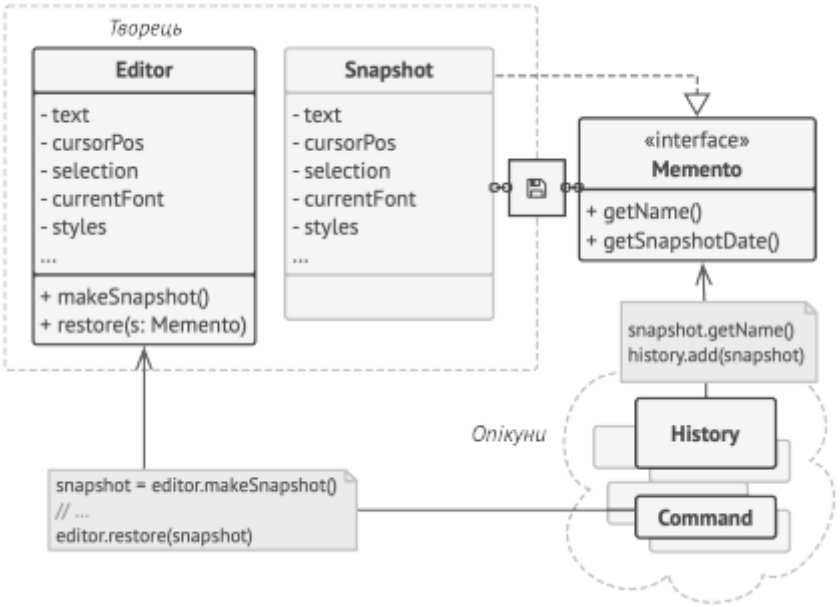

Рис.2.11 Знімок повністю відкритий для творця, але лише частково відкритий для опікунів.

Така схема дозволяє творцям робити знімки та віддавати їх на зберігання іншим об'єктам, що називаються опікунами. Опікунам буде доступний тільки обмежений інтерфейс знімка, тому вони ніяк не зможуть вплинути на «нутрощі» самого знімку. У потрібний момент опікун може попросити творця відновити свій стан, передавши йому відповідний знімок.

У нашому прикладі з редактором опікуном можна зробити окремий клас, який зберігатиме список виконаних операцій. Обмежений інтерфейс знімків дозволить демонструвати користувачеві гарний список з назвами й датами виконаних операцій. Коли ж користувач вирішить скасувати операцію, клас історії візьме останній знімок зі стека та надішле його об'єкту редактора для відновлення.

Структура

Класична реалізація патерна покладається на механізм вкладених класів, який доступний тільки в деяких мовах програмування (C++, C#, Java).

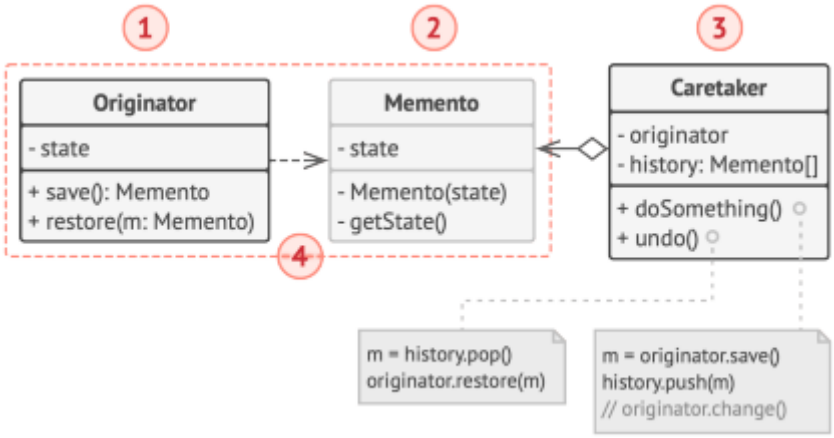

Рис.2.12 Класична реалізація на вкладених класах

1. Творець може створювати знімки свого стану, а також відтворювати минулий стан, якщо до нього подати готовий знімок. Команда описує інтерфейс, спільний для всіх конкретних команд. Зазвичай тут описується лише один метод запуску команди.
2. Знімок — це простий об’єкт даних, який містить стан творця. Надійніше за все зробити об’єкти знімків незмінними, встановлюючи в них стан тільки через конструктор.
3. Опікун повинен знати, коли робити знімок творця та коли його потрібно відновлювати. Опікун може зберігати історію минулих станів творця у вигляді стека знімків. Коли треба буде скасувати останню операцію, він візьме «верхній» знімок зі стеку та передасть його творцеві для відновлення
4. У даній реалізації знімок — це внутрішній клас по відношенню до класу творця. Саме тому він має повний доступ до всіх полів та методів творця, навіть приватних. З іншого боку, опікун не має доступу ані до стану, ані до методів знімків, а може лише зберігати посилання на ці об’єкти.

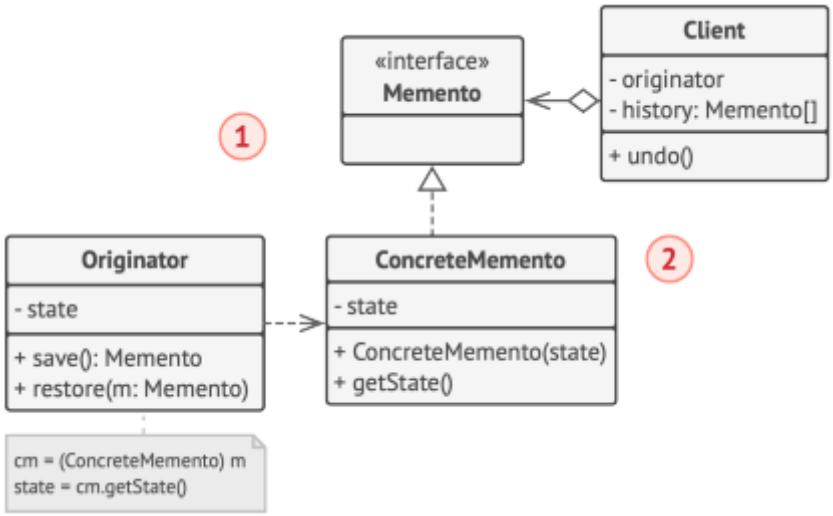

Рис.2.13 Реалізація з проміжним порожнім інтерфейсом

Підходить для мов, що не мають механізму вкладених класів (наприклад, PHP)

1. У цій реалізації творець працює безпосередньо з конкретним класом знімка, а опікун — тільки з його обмеженим інтерфейсом.
2. Завдяки цьому досягається той самий ефект, що і в класичній реалізації. Творець має повний доступ до знімка, а опікун — ні.

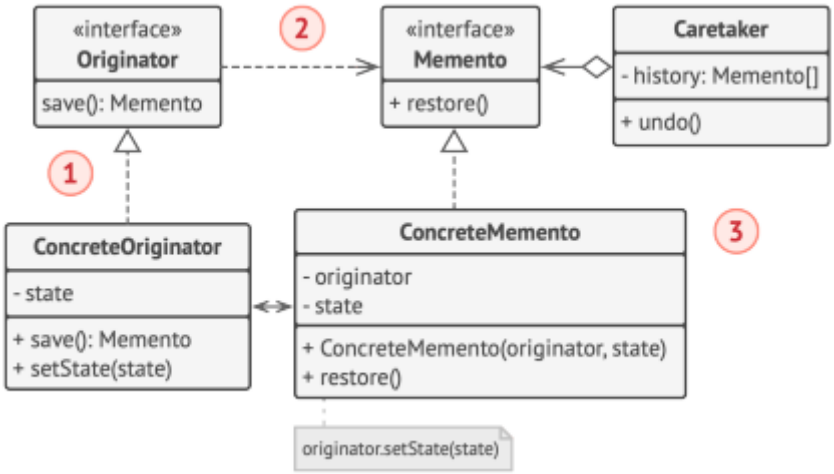

Рис.2.14 Знімки з підвищеним захистом

Якщо потрібно повністю виключити можливість доступу до стану творців та знімків

1. Ця реалізація дозволяє мати кілька видів творців та знімків. Кожному класу творців відповідає власний клас знімків. Ані творці, ані знімки не дозволяють іншим об'єктам читати свій стан.
2. Тут опікун ще жорсткіше обмежений у доступі до стану творців та знімків, але, з іншого боку, опікун стає незалежним від творців, оскільки метод відновлення тепер знаходиться в самих знімках.
3. Знімки тепер пов'язані з тими творцями, з яких вони зроблені. Вони, як і раніше, отримують стан через конструктор. Завдяки близькому зв'язку між класами, знімки знають, як відновити стан своїх творців.

Застосування

Коли вам потрібно зберігати миттєві знімки стану об'єкта (або його частини) для того, щоб об'єкт можна було відновити в тому самому стані.
Патерн Знімок дозволяє створювати будь-яку кількість знімків об'єкта і зберігати їх незалежно від об'єкта, з якого роблять знімок. Знімки часто використовують не тільки для реалізації операції скасування, але й для транзакцій, коли стан об'єкта потрібно «відкотити», якщо операція не була вдалою.

Коли пряме отримання стану об'єкта розкриває приватні деталі його реалізації, порушуючи інкапсуляцію.
Патерн пропонує виготовити знімок саме вихідному об'єкту, тому що йому доступні всі поля, навіть приватні.

Кроки реалізації

1. Визначте клас творця, об'єкти якого повинні створювати знімки свого стану.
2. Створіть клас знімка та опишіть в ньому ті ж самі поля, які є в оригінальному класі-творці.
3. Зробіть об'єкти знімків незмінними. Вони повинні одержувати початкові значення тільки один раз, через власний конструктор
4. Якщо ваша мова програмування це дозволяє, зробіть клас знімка вкладеним у клас творця. Якщо ні, вийміть з класу знімка порожній інтерфейс, який буде доступним іншим об'єктам програми. Згодом ви можете додати до цього інтерфейсу деякі допоміжні методи, що дають доступ до метаданих знімка, але прямий доступ до даних творця повинен бути виключеним.
5. Додайте до класу творця метод одержання знімків. Творець повинен створювати нові об'єкти знімків, передаючи значення своїх полів через конструктор. Сигнатура методу повинна повертати знімки через обмежений інтерфейс, якщо він у вас є. Сам клас повинен працювати з конкретним класом знімка.

6. Додайте до класу творця метод відновлення зі знімка. Щодо прив'язки до типів, керуйтеся тією ж логікою, що і в пункті 4.
7. Опікуни, незалежно від того, чи це історія операцій, чи об'єкти команд, чи щось інше, повинні знати про те, коли запитувати знімки у творця, де їх зберігати та коли відновлювати
8. Зв'язок опікунів з творцями можна перенести всередину знімків. У цьому випадку кожен знімок буде прив'язаний до свого творця і повинен буде сам відновлювати його стан. Але це працюватиме або якщо класи знімків вкладені до класів творців, або якщо творці мають відповідні сетери для встановлення значень своїх полів.

Переваги

- Не порушує інкапсуляцію вихідного об'єкта.
- Спрощує структуру вихідного об'єкта. Йому не потрібно зберігати історію версій свого стану

Недоліки

- Вимагає багато пам'яті, якщо клієнти дуже часто створюють знімки.
- Може спричинити додаткові витрати пам'яті, якщо об'єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.
- В деяких мовах (наприклад, PHP, Python, JavaScript) складно гарантувати, щоб лише вихідний об'єкт мав доступ до стану знімка.

Шаблон «Observer»

Спостерігач — це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які відбуваються в інших об'єктах.

Проблема

Уявіть, що ви маєте два об'єкти: Покупець і Магазин . До магазину мають ось-ось завезти новий товар, який цікавить покупця. Покупець може щодня ходити до магазину, щоб перевіряти наявність товару. Але через це він буде дратуватися, даремно витрачаючи свій дорогоцінний час

З іншого боку, магазин може розсилати спам кожному своєму покупцеві. Багатьох покупців це засмутить, оскільки товар специфічний і потрібний не всім. Виходить конфлікт: або покупець гає час на періодичні перевірки, або магазин розтрачує ресурси на непотрібні сповіщення.

Рішення

Давайте називати Видавцями ті об'єкти, які містять важливий або цікавий для інших стан. Решту об'єктів, які хотіли б відстежувати зміни цього стану, назвемо Підписниками. Патерн Спостерігач пропонує зберігати всередині об'єкта видавця список посилань на об'єкти підписників. Причому видавець не повинен вести список підписки самостійно. Він повинен надати методи, за допомогою яких підписники могли б додавати або прибирати себе зі списку.

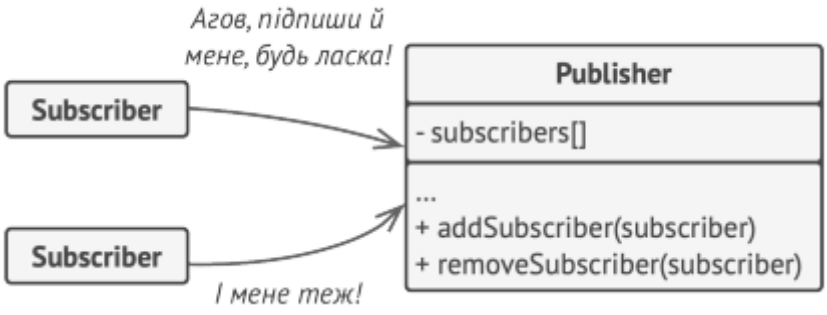

Рис. 2.15 Підписка на події.

Тепер найцікавіше. Коли у видавця відбуватиметься важлива подія, він буде проходитися за списком передплатників та сповіщувати їх про подію, викликаючи певний метод об'єктів-передплатників.

Видавцю байдуже, якого класу буде той чи інший підписник, бо всі вони повинні слідувати загальному інтерфейсу й мати єдиний метод оповіщення.

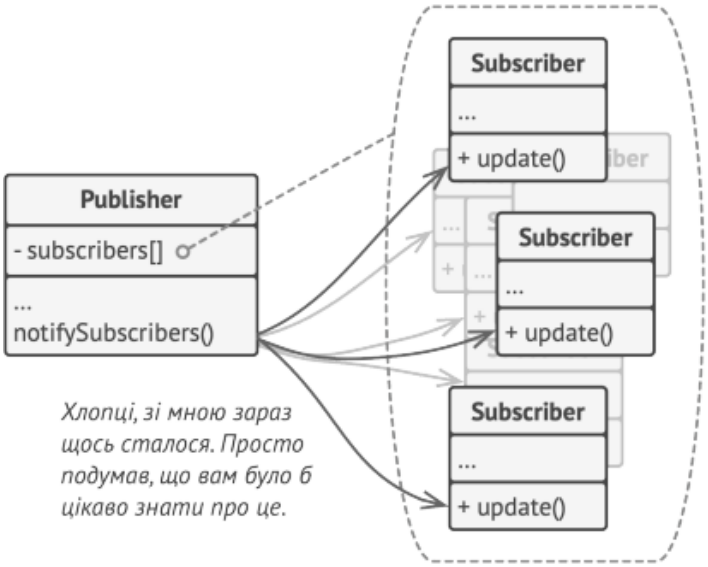

Рис.2.16 Сповіщення про події.

Побачивши, як добре все працює, ви можете виділити загальний інтерфейс і для всіх видавців, який буде складатися з методів підписки та відписки. Після цього підписники зможуть працювати з різними типами видавців, і отримувати від них сповіщення через єдиний метод.

Аналогія з життя
Після того, як ви оформили підписку на журнал, вам більше не потрібно їздити до супермаркета та дізнаватись, чи вже вийшов черговий номер. Натомість видавництво надсилатиме нові номери поштою прямо до вас додому, відразу після їхнього виходу.
Видавництво веде список підписників і знає, кому який журнал слати. Ви можете в будь-який момент відмовитися від підписки, й журнал перестане до вас надходити.

Структура

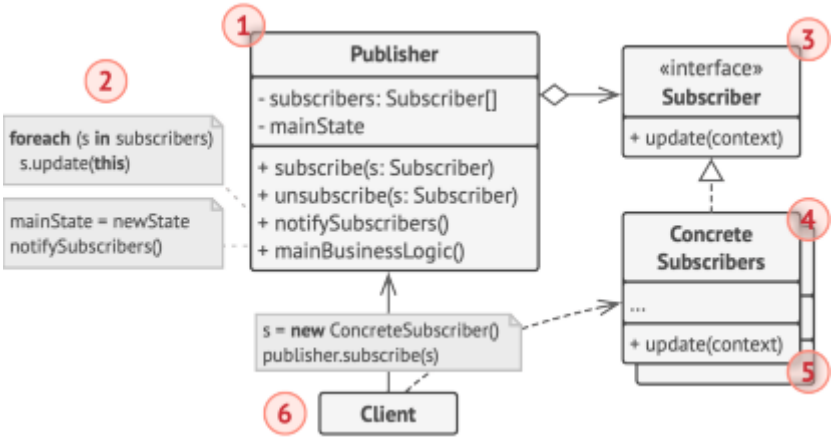

Рис.2.17 Адаптер об'єктів

1. Видавець володіє внутрішнім станом, зміни якого цікаво відслідковувати підписникам. Видавець містить механізм підписки: список підписників та методи підписки/відписки.
2. Коли внутрішній стан видавця змінюється, він сповіщає своїх підписників. Для цього видавець проходиться за списком підписників і викликає їхній метод сповіщення, який описаний в загальному інтерфейсі підписників.
3. Підписник визначає інтерфейс, яким користується видавець для надсилання сповіщень. Здебільшого для цього досить одного методу.

4. Конкретні підписники виконують щось у відповідь на сповіщення, яке надійшло від видавця. Ці класи мають дотримуватися загального інтерфейсу, щоб видавець не залежав від конкретних класів підписників.
5. Після отримання сповіщення підписнику необхідно отримати оновлений стан видавця. Видавець може передати цей стан через параметри методу сповіщення. Більш гнучкий варіант — передавати через параметри весь об'єкт видавця, щоб підписник міг сам отримати необхідні дані. Як варіант, підписник може постійно зберігати посилання на об'єкт видавця, переданий йому через конструктор.
6. Клієнт створює об'єкти видавців і підписників, а потім реєструє підписників на оновлення у видавцях.

Застосування

Якщо після зміни стану одного об'єкта потрібно щось зробити в інших, але ви не знаєте наперед, які саме об'єкти мають відреагувати.
Описана проблема може виникнути при розробленні бібліотек користувацького інтерфейсу, якщо вам необхідно надати можливість стороннім класам реагувати на кліки по кнопках. Патерн Спостерігач надає змогу будь-якому об'єкту з інтерфейсом підписника зареєструватися для отримання сповіщень про події, що трапляються в об'єктах-видавцях.

Якщо одні об'єкти мають спостерігати за іншими, але тільки у визначених випадках.
Видавці ведуть динамічні списки. Усі спостерігачі можуть підписуватися або відписуватися від отримання сповіщень безпосередньо під час виконання програми

Кроки реалізації

1. Розбийте вашу функціональність на дві частини: незалежне ядро та опціональні залежні частини. Незалежне ядро стане видавцем. Залежні частини стануть підписниками.
2. Створіть інтерфейс підписників. Зазвичай достатньо визначити в ньому лише один метод сповіщення
3. Створіть інтерфейс видавців та опишіть у ньому операції керування підпискою. Пам'ятайте, що видавці повинні працювати з підписниками тільки через їхній загальний інтерфейс
4. Вам потрібно вирішити, куди помістити код ведення підписки, адже він зазвичай буває однаковим для всіх типів видавців. Найочевидніший спосіб — це винесення коду до проміжного абстрактного класу, від якого будуть успадковуватися всі видавці. Якщо ж ви інтегруєте патерн до існуючих класів, то створити новий базовий клас може бути важко. У цьому випадку ви можете помістити логіку підписки в допоміжний об'єкт та делегувати йому роботу з видавцями. Програма повинна використовувати адаптер тільки через клієнтський інтерфейс. Це дозволить легко змінювати та додавати адаптери в майбутньому.
5. Створіть класи конкретних видавців. Реалізуйте їх таким чином, щоб після кожної зміні стану вони слали сповіщення всім своїм підписникам.
6. Реалізуйте метод сповіщення в конкретних підписниках. Не забудьте передбачити параметри, через які видавець міг би відправляти якісь дані, пов'язані з подією, що відбулась. Можливий і інший варіант, коли підписник, отримавши сповіщення, сам візьме потрібні дані з об'єкта видавця. Але в цьому разі ви будете змушені прив'язати клас підписника до конкретного класу видавця.
7. Клієнт повинен створювати необхідну кількість об'єктів підписників та підписувати їх у видавці

Переваги

- Видавці не залежать від конкретних класів підписників і навпаки.
- Ви можете підписувати і відписувати одержувачів «на льоту».
- Реалізує принцип відкритості/закритості.

Недоліки

Підписники сповіщуються у випадковій послідовності.

Шаблон «Decorator»

Декоратор — це структурний патерн проектування, що дає змогу динамічно додавати об’єктам нову функціональність, загортаючи їх у корисні «обгортки».

Проблема

Ви працюєте над бібліотекою сповіщень, яку можна підключати до різноманітних програм, щоб отримувати сповіщення про важливі події. Основою бібліотеки є клас Notifier з методом send , який приймає на вхід рядок-повідомлення і надсилає його всім адміністраторам електронною поштою. Стороння програма повинна створити й налаштувати цей об’єкт, вказавши, кому надсилати сповіщення, та використовувати його щоразу, коли щось відбувається.

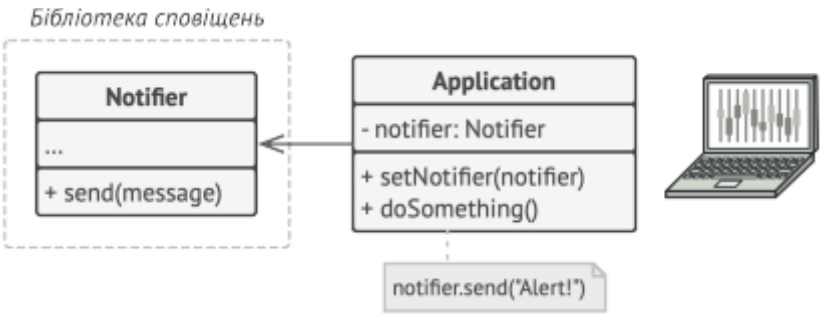

Рис.2.18 Сторонні програми використовують головний клас сповіщень.

В якийсь момент стало зрозуміло, що користувачам не вистачає одних тільки email-сповіщень. Деякі з них хотіли б отримувати сповіщення про критичні проблеми через SMS. Інші хотіли б отримувати їх у вигляді Facebook-повідомлень. Корпоративні користувачі хотіли би бачити повідомлення у Slack.

Рис.2.19 Кожен тип сповіщення живе у власному підкласі.

Спершу ви додали кожен з типів сповіщень до програми, успадкувавши їх від базового класу Notifier. Тепер користувачі могли вибрати один з типів сповіщень, який і використовувався надалі. Але потім хтось резонно запитав, чому не можна увімкнути кілька типів сповіщень одночасно? Адже, якщо у вашому будинку раптом почалася пожежа, ви б хотіли отримати сповіщення по всіх каналах, чи не так

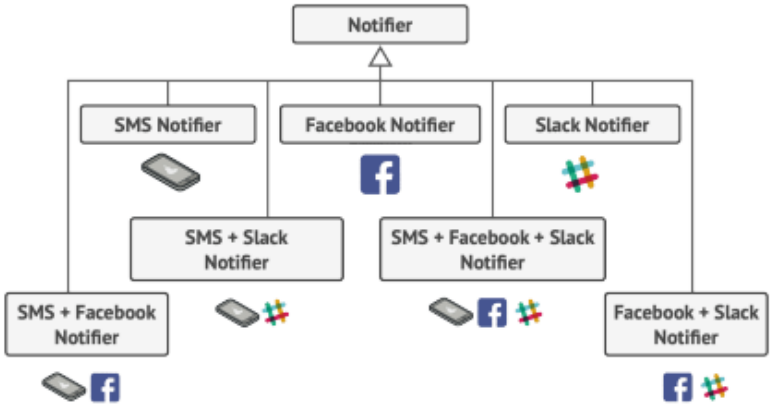

Рис.2.20 Комбінаторний вибух підкласів при поєднанні типів сповіщень

Ви зробили спробу реалізувати всі можливі комбінації підкласів сповіщень, але після того, як додали перший десяток класів, стало зрозуміло, що такий підхід неймовірно роздуває код програми. Отже, потрібен інший спосіб комбінування поведінки об'єктів, який не призводить до збільшення кількості підкласів

Рішення

Спадкування — це перше, що приходить в голову багатьом програмістам, коли потрібно розширити яку-небудь чинну поведінку. Проте механізм спадкування має кілька прикрих проблем.

- Він статичний. Ви не можете змінити поведінку об'єкта, який вже існує. Для цього необхідно створити новий об'єкт, вибравши інший підклас.
- Він не дозволяє наслідувати поведінку декількох класів одночасно. Тому доведеться створювати безліч підкласів комбінацій, щоб досягти поєднання поведінки.

Одним зі способів, що дозволяє обійти ці проблеми, є заміна спадкування агрегацією або композицією. Це той випадок, коли один об'єкт утримує інший і делегує йому роботу замість того, щоб самому успадкувати його поведінку. Саме на цьому принципі побудовано патерн Декоратор

Рис.2.21 Спадкування проти Агрегації

Декоратор має альтернативну назву — обгортка. Вона більш вдало описує суть патерна: ви розміщуєте цільовий об'єкт у іншому об'єкті-обгортці, який запускає базову поведінку об'єкта, а потім додає до результату щось своє. Обидва об'єкти мають загальний інтерфейс, тому для користувача немає жодної різниці, з чим працювати — з чистим чи загорнутим об'єктом. Ви можете використовувати кілька різних обгорток одночасно — результат буде мати об'єднану поведінку всіх обгорток.

В нашому прикладі зі сповіщеннями залишимо в базовому класі просте надсилання сповіщень електронною поштою, а розширені способи зробимо декораторами. Стороння програма, яка виступає клієнтом, під час початкового налаштовування буде загортати об'єкт сповіщення в ті обгортки, які відповідають бажаному способу сповіщення.

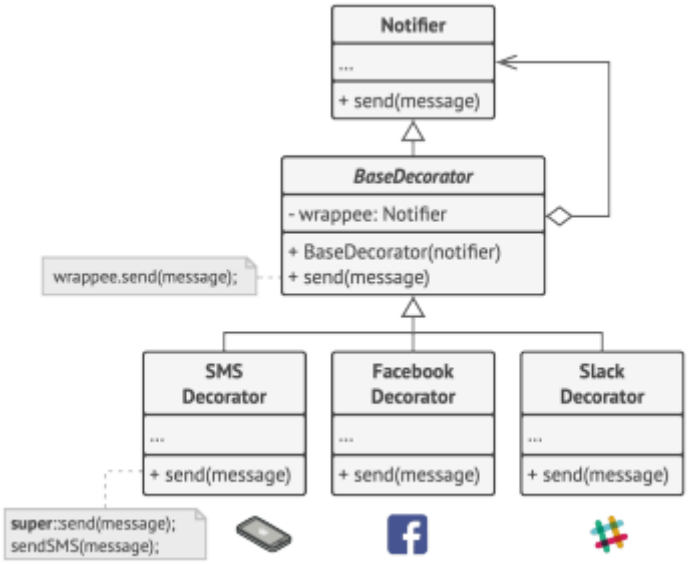

Рис.2.22 Розширені способи надсилання сповіщень стають декораторами.

Остання обгортка у списку буде саме тим об'єктом, з яким клієнт працюватиме увесь інший час. Для решти клієнтського коду нічого не зміниться, адже всі обгортки мають такий самий інтерфейс, що і базовий клас сповіщень.

.

Структура

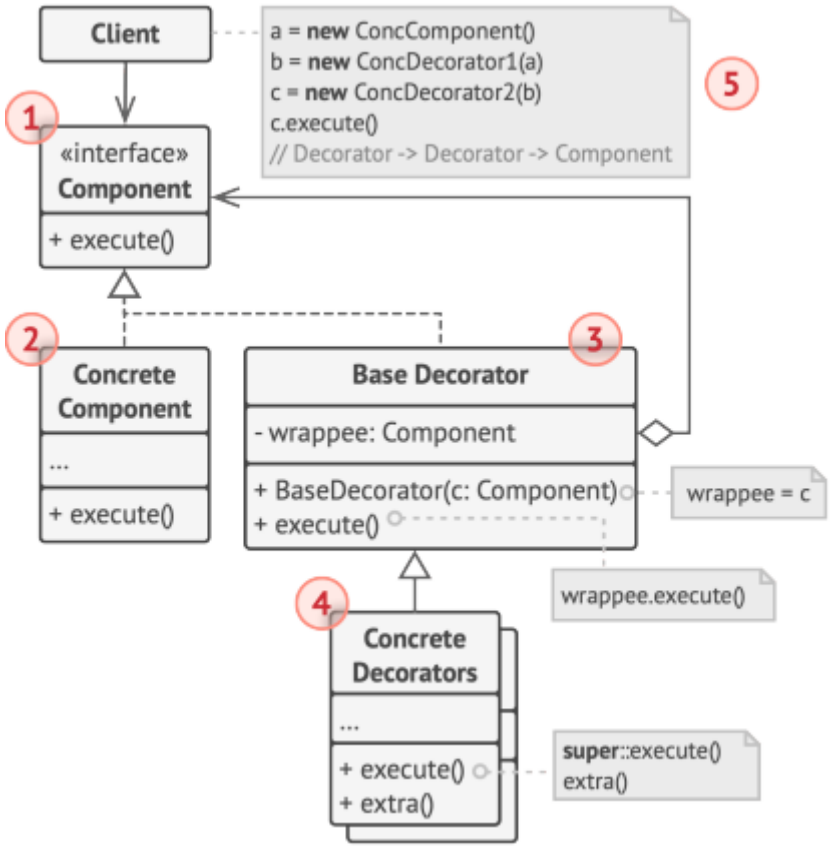

Рис.2.28 Базова реалізація патерну Декоратор

1. Компонент задає загальний інтерфейс обгорток та об'єктів, що загортаються.
2. Конкретний компонент визначає клас об'єктів, що загортаються. Він містить якусь базову поведінку, яку потім змінюють декоратори.
3. Базовий декоратор зберігає посилання на вкладений об'єкт компонент. Це може бути як конкретний компонент, так і один з конкретних декораторів. Базовий декоратор делегує всі свої операції вкладеному об'єкту. Додаткова поведінка житиме в конкретних декораторах.
4. Конкретні декоратори — це різні варіації декораторів, що містять додаткову поведінку. Вона виконується до або після виклику аналогічної поведінки загорнутого об'єкта.

5. Клієнт може обертати прості компоненти й декоратори в інші декоратори, працюючи з усіма об’єктами через загальний інтерфейс компонентів

Застосування

Якщо вам потрібно додавати об’єктам нові обов’язки «на льоту», непомітно для коду, який їх використовує
Об’єкти вкладаються в обгортки, які мають додаткові поведінки. Обгортки і самі об’єкти мають однаковий інтерфейс, тому клієнтам не важливо, з чим працювати — зі звичайним об’єктом чи з загорнутим.

Якщо не можна розширити обов’язки об’єкта за допомогою спадкування
У багатьох мовах програмування є ключове слово final, яке може заблокувати спадкування класу. Розширити такі класи можна тільки за допомогою Декоратора.

Кроки реалізації

1. Переконайтеся, що у вашому завданні присутні основний компонент і декілька опціональних доповнень-надбудов над ним.
2. Створіть інтерфейс компонента, який описував би загальні методи як для основного компонента, так і для його доповнень
3. Створіть клас конкретного компонента й помістіть в нього основну бізнес-логіку.
4. Створіть базовий клас декораторів. Він повинен мати поле для зберігання посилань на вкладений об’єкт-компонент. Усі методи базового декоратора повинні делегувати роботу вкладеному об’єкту
5. Конкретний компонент, як і базовий декоратор, повинні дотримуватися одного і того самого інтерфейсу компонента.
6. Створіть класи конкретних декораторів, успадковуючи їх від базового декоратора. Конкретний декоратор повинен виконувати свою додаткову функціональність, а потім (або перед цим) викликати цю ж операцію загорнутого об’єкта.
7. Клієнт бере на себе відповідальність за конфігурацію і порядок загортання об’єктів

Переваги

- Більша гнучкість, ніж у спадкування.
- Дозволяє додавати обов’язки «на льоту».
- Можна додавати кілька нових обов’язків одразу.
- Дозволяє мати кілька дрібних об’єктів, замість одного об’єкта «на всі випадки життя».

Недоліки

- Важко конфігурувати об’єкти, які загорнуто в декілька обгорток одночасно.
- Велика кількість крихітних класів.

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:

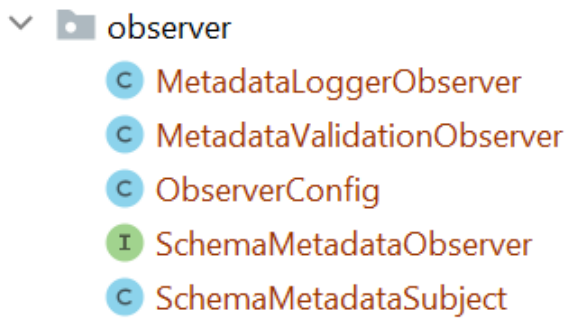

Рис. 1. Ієрархія класів. Реалізація патерну Спостерігач.

```
public interface SchemaMetadataObserver {
    1 usage  2 implementations
    void onMetadataUpdated(String key, String newValue);
}
```

Рис. 2. Інтерфейс для реалізації патерну Спостерігач.

```
@Component
public class MetadataLoggerObserver implements SchemaMetadataObserver {
    1 usage
    @Override
    public void onMetadataUpdated(String key, String newValue) {
        System.out.println("Metadata updated: Key = " + key + ", New Value = " + newValue);
    }
}
```

Рис. 2.1.1 Метод onMetadataUpdated класу MetadataLoggerObserver

```java
@Override
public void onMetadataUpdated(String key, String newValue) {
    if (newValue.length() > 255){
        System.out.println("Value: " + newValue + "with key: " + key + "must be less then 255 symbols in length");
    }
    System.out.println("Validating metadata for key: " + key);
}
```

Рис. 2.1.2 Метод onMetadataUpdated класу MetadataValidationObserver

```java
public class SchemaMetadataSubject {
    3 usages
    private final List<SchemaMetadataObserver> observers = new ArrayList<>();

    2 usages
    public void addObserver(SchemaMetadataObserver observer) {
        observers.add(observer);
    }

    no usages
    public void removeObserver(SchemaMetadataObserver observer) {
        observers.remove(observer);
    }

    1 usage
    public void notifyObservers(String key, String newValue) {
        for (SchemaMetadataObserver observer : observers) {
            observer.onMetadataUpdated(key, newValue);
        }
    }
}
```

Рис. 2.1.3 Клас SchemaMetadataSubject

```java
@Configuration
public class ObserverConfig {
    no usages
    public ObserverConfig(SchemaMetadataServiceImplementation metadataService,
                          MetadataLoggerObserver loggerObserver,
                          MetadataValidationObserver validationObserver) {
        metadataService.addObserver(loggerObserver);
        metadataService.addObserver(validationObserver);
    }
}
```

Рис. 2.1.4 Клас ObserverConfig

```java
@Service
public class SchemaMetadataServiceImplementation extends SchemaMetadataSubject {
    2 usages
    private final Map<String, String> metadata = new HashMap<>();

    1 usage
    public void updateMetadata(String key, String newValue) {
        metadata.put(key, newValue);
        notifyObservers(key, newValue);
    }

    1 usage
    public String getMetadata(String key) {
        return metadata.get(key);
    }
}
```

Рис. 2.1.5 Виклики методів execute класів патерну Спостерігач у сервісі SchemaMetadataServiceImplementation

```java
@RestController
@RequestMapping("/api/metadata")
public class MetadataController {
    3 usages
    private final SchemaMetadataServiceImplementation metadataService;

    no usages
    @Autowired
    public MetadataController(SchemaMetadataServiceImplementation metadataService) {
        this.metadataService = metadataService;
    }

    no usages
    @PostMapping("/update")
    public String updateMetadata(@RequestParam String key, @RequestParam String value) {
        metadataService.updateMetadata(key, value);
        return "Metadata updated successfully";
    }

    no usages
    @GetMapping("/get")
    public String getMetadata(@RequestParam String key) {
        return metadataService.getMetadata(key);
    }
}
```

Рис. 2.1.6 Виклики методів сервісу SchemaMetadataServiceImplementation у контролері

ВИСНОВОК

У процесі роботи було розроблено частину функціоналу застосунку для роботи з JSON, що включає механізм обробки та оновлення метаданих. Реалізовано шаблон проектування Observer, який забезпечує автоматичне оновлення компонентів після змін у метаданих. Це дозволило підвищити гнучкість і масштабованість застосунку, зробивши його придатним для інтеграції нових функцій без значного впливу на існуючу архітектуру.