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 №7
із дисципліни «Технології Розробки Програмного Забезпечення»
Тема: ШАБЛОНИ «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD»
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 динамічного форматування даних. У процесі реалізації застосувати шаблон проектування Template method для досягнення гнучкості та масштабованості системи

Теоритичні відомості:

Шаблон «MEDIATOR»

Посередник — це поведінковий патерн проектування, що дає змогу зменшити зв’язаність великої кількості класів між собою, завдяки переміщенню цих зв’язків до одного класу-посередника.

Проблема

Припустімо, що у вас є діалог створення профілю користувача. Він складається з різноманітних елементів керування: текстових полів, чекбоксів, кнопок.

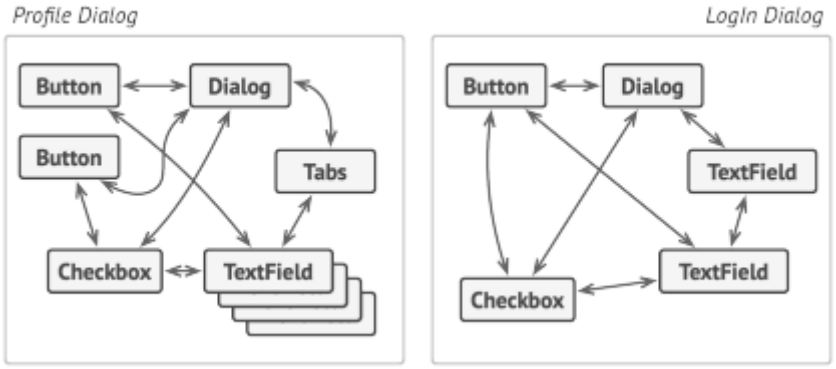

Рис.2.1 Безладні зв’язки між елементами інтерфейсу користувача.

Окремі елементи діалогу повинні взаємодіяти одне з одним. Так, наприклад, чекбокс «у мене є собака» відкриває приховане поле для введення імені домашнього улюбленця, а клік по кнопці збереження запускає перевірку значень усіх полів форми.

Рис.2.2 Код елементів потрібно правити під час зміни кожного діалогу.

Прописавши цю логіку безпосередньо в коді елементів керування, ви поставите хрест на їхньому повторному використанні в інших місцях програми. Вони стануть занадто тісно пов’язаними з елементами діалогу редагування профілю, які не потрібні в інших контекстах. Отже ви зможете або використовувати всі елементи відразу, або не використовувати жоден.

Рішення

Патерн Посередник змушує об’єкти спілкуватися через окремий об’єкт-посередник, який знає, кому потрібно перенаправити той або інший запит. Завдяки цьому компоненти системи залежатимуть тільки від посередника, а не від десятків інших компонентів.

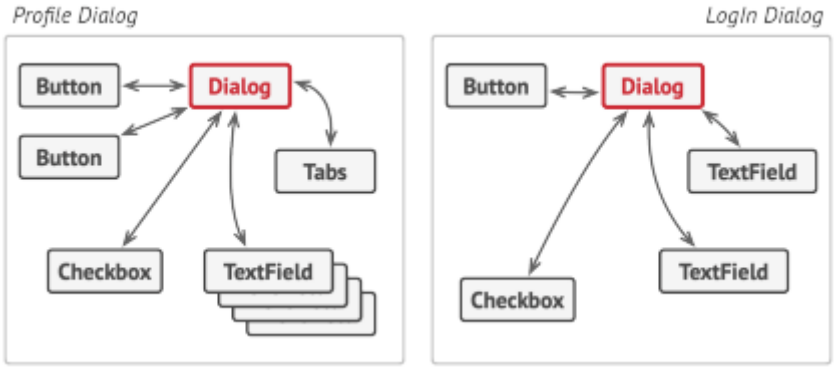

Рис.2.3 Елементи інтерфейсу спілкуються через посередника

У нашому прикладі посередником міг би стати діалог. Імовірно, клас діалогу вже знає, з яких елементів він скла дається. Тому жодних нових зв'язків додавати до нього не доведеться. Основні зміни відбудуться всередині окремих елементів діалогу. Якщо раніше при отриманні кліка від користувача об'єкт кнопки самостійно перевіряв значення полів діалогу, то тепер його єдиний обов'язок — повідомити діалогу про те, що відбувся клік. Отримавши повідомлення, діалог виконає всі необхідні перевірки полів. Таким чином, замість кількох залежностей від інших елементів кнопка отримає лише одну — від самого діалогу.

Щоб зробити код ще гнучкішим, можна виділити єдиний інтерфейс для всіх посередників, тобто діалогів програми. Наша кнопка стане залежною не від конкретного діалогу створення користувача, а від абстрактного, що дозволить використовувати її і в інших діалогах. Таким чином, посередник приховує у собі всі складні зв'язки й залежності між класами окремих компонентів програми. А чим менше зв'язків мають класи, тим простіше їх змінювати, розширювати й повторно використовувати.

Структура

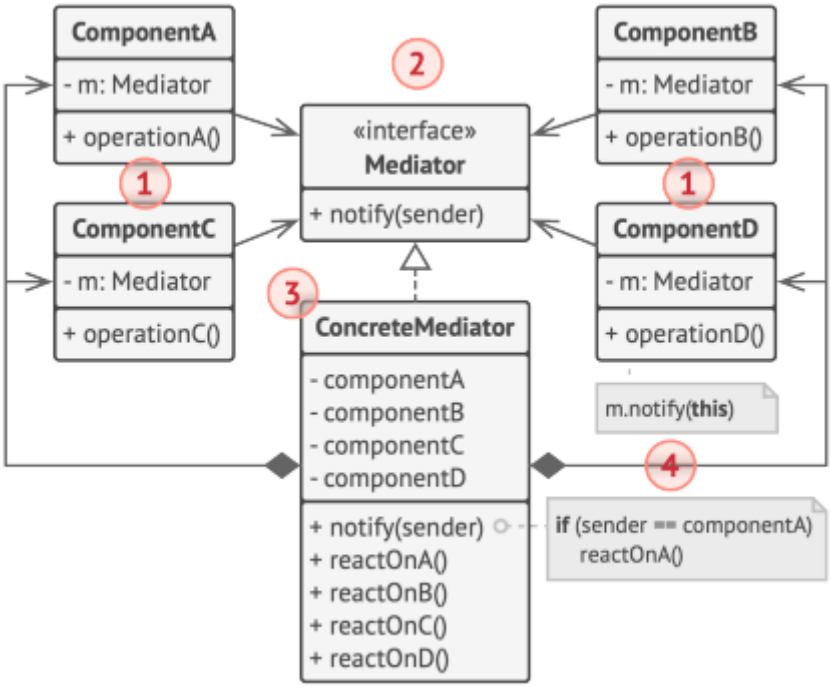

Рис.2.4 Структура патерну Посередник

1. Компоненти — це різнорідні об'єкти, що містять бізнес-логіку програми. Кожен компонент має посилання на об'єкт посередника, але працює з ним тільки через абстрактний інтерфейс посередників. Завдяки цьому компоненти можна повторно використовувати в інших програмах, зв'язавши їх з посередником іншого типу.
2. Посередник визначає інтерфейс для обміну інформацією з компонентами. Зазвичай достатньо одного методу, щоби повідомляти посередника про події, що відбулися в компонентах. У

параметрах цього методу можна передавати дета лі події: посилання на компонент, в якому вона відбулася, та будь-які інші дані.
3. Конкретний посередник містить код взаємодії кількох компонентів між собою. Найчастіше цей об’єкт не тільки зберігає посилання на всі свої компоненти, але й сам їх створює, керуючи подальшим життєвим циклом
4. Компоненти не повинні спілкуватися один з одним безпосередньо. Якщо в компоненті відбувається важлива подія, він повинен повідомити свого посередника, а той сам вирішить, чи стосується подія інших компонентів, і чи треба їх сповістити. При цьому компонент-відправник не знає, хто обробить його запит, а компонент-одержувач не знає, хто його надіслав.

Застосування

- Коли вам складно змінювати деякі класи через те, що вони мають величезну кількість хаотичних зв’язків з іншими класами.
- Посередник дозволяє розмістити усі ці зв’язки в одному класі. Після цього вам буде легше їх відрефакторити, зробити більш зрозумілими й гнучкими.
- Коли ви не можете повторно використовувати клас, оскільки він залежить від безлічі інших класів.
- Після застосування патерна компоненти втрачають колишні зв’язки з іншими компонентами, а все їхнє спілкування відбувається опосередковано, через об’єкт посередника.
- Коли вам доводиться створювати багато підкласів компонентів, щоб використовувати одні й ті самі компоненти в різних контекстах.
- Якщо раніше зміна відносин в одному компоненті могла призвести до лавини змін в усіх інших компонентах, то тепер вам достатньо створити підклас посередника та змінити в ньому зв’язки між компонентами.

Кроки реалізації

1. Знайдіть групу тісно сплетених класів, де можна отримати деяку користь, відв’язавши деякі один від одного. Наприклад, щоб повторно використовувати їхній код в іншій програмі
2. Створіть загальний інтерфейс посередників та опишіть в ньому методи для взаємодії з компонентами. У найпростішому випадку достатньо одного методу для отримання повідомлень від компонентів. Цей інтерфейс необхідний, якщо ви хочете повторно використовувати класи компонентів для інших завдань. У цьому випадку все, що потрібно зробити, — це створити новий клас конкретного посередника.
3. Реалізуйте цей інтерфейс у класі конкретного посередника. Помістіть до нього поля, які міститимуть посилання на всі об’єкти компонентів.
4. Ви можете піти далі і перемістити код створення компонентів до класу конкретного посередника, перетворивши його на фабрику.
5. Компоненти теж повинні мати посилання на об’єкт посередника. Зв’язок між ними зручніше всього встановити шляхом подання посередника до параметрів конструктора компонентів.
6. Змініть код компонентів так, щоб вони викликали метод повідомлення посередника, замість методів інших компонентів. З протилежного боку, посередник має викликати методи потрібного компонента, коли отримує повідомлення від компонента.

Переваги

- Усуває залежності між компонентами, дозволяючи використовувати їх повторно.
- Спрощує взаємодію між компонентами.
- Централізує керування в одному місці.

Недоліки

Посередник може сильно «роздутися».

Шаблон «FACADE»

Фасад — це структурний патерн проектування, який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку

Проблема

Вашому коду доводиться працювати з великою кількістю об’єктів певної складної бібліотеки чи фреймворка. Ви повинні самостійно ініціалізувати ці об’єкти, стежити за правильним порядком залежностей тощо. В результаті бізнес-логіка ваших класів тісно переплітається з деталями реалізації сторонніх класів. Такий код досить складно розуміти та підтримувати.

Рішення

Фасад — це простий інтерфейс для роботи зі складною підсистемою, яка містить безліч класів. Фасад може бути спрощеним відображенням системи, що не має 100% тієї функціональності, якої можна було б досягти, використовуючи складну підсистему безпосередньо. Разом з тим, він надає саме ті «фічі», які потрібні клієнтові, і приховує все інше. Фасад корисний у тому випадку, якщо ви використовуєте якусь складну бібліотеку з безліччю рухомих частин, з яких вам потрібна тільки частина. Наприклад, програма, що заливає в соціальні мережі відео з кошенятками, може використовувати професійну бібліотеку для стискання відео, але все, що потрібно клієнтському коду цієї програми, — це простий метод encode(filename, format). Створивши клас з таким методом, ви реалізуєте свій перший фасад.

Структура

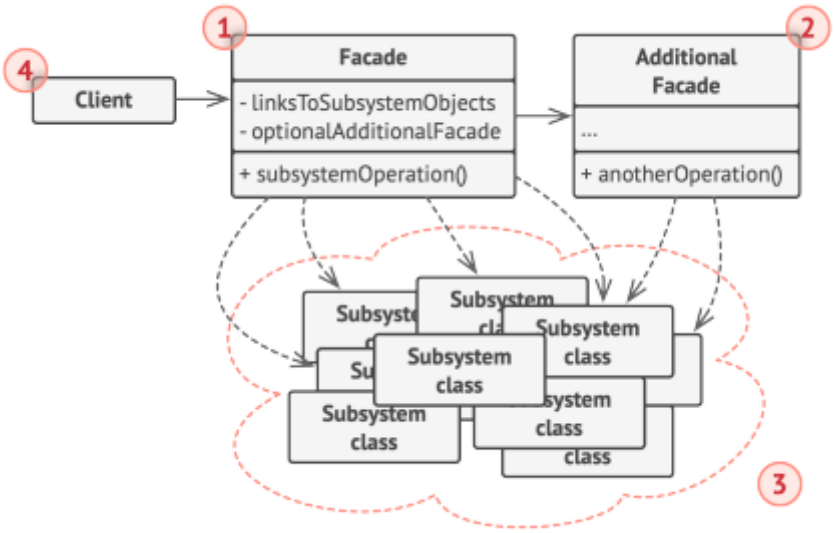

Рис.2.5 Структура патерну Фасад

1. Фасад надає швидкий доступ до певної функціональності підсистеми. Він «знає», яким класам потрібно переадресувати запит, і які дані для цього потрібні.
2. Додатковий фасад можна ввести, щоб не захаращувати єдиний фасад різнорідною функціональністю. Він може використовуватися як клієнтом, так й іншими фасадами.
3. Складна підсистема має безліч різноманітних класів. Для того, щоб примусити усіх їх щось робити, потрібно знати подробиці влаштування підсистеми, порядок ініціалізації об’єктів та інші деталі. Класи підсистеми не знають про існування фасаду і працюють один з одним безпосередньо
4. Клієнт використовує фасад замість безпосередньої роботи з об’єктами складної підсистеми.

Застосування

Якщо вам потрібно надати простий або урізаний інтерфейс до складної підсистеми.
Часто підсистеми ускладнюються в міру розвитку програми. Застосування більшості патернів призводить до появи менших класів, але у великій кількості. Таку підсистему простіше використовувати повторно, налаштовуючи її кожен раз під конкретні потреби, але, разом з тим, використовувати таку підсистему без налаштовування важче. Фасад пропонує певний вид системи за замовчуванням, який влаштовує більшість клієнтів.

Якщо ви хочете розкласти підсистему на окремі рівні.
Використовуйте фасади для визначення точок входу на кожен рівень підсистеми. Якщо підсистеми залежать одна від одної, тоді залежність можна спростити, дозволивши підсистемам обмінюватися інформацією тільки через фасади. Наприклад, візьмемо ту ж саму складну систему конвертації відео. Ви хочете розбити її на окремі шари для роботи з аудіо й відео. Можна спробувати створити фасад для кожної з цих частин і примусити класи аудіо та відео обробки спілкуватися один з одним через ці фасади, а не безпосередньо

Кроки реалізації

1. Визначте, чи можна створити більш простий інтерфейс, ніж той, який надає складна підсистема. Ви на правильному шляху, якщо цей інтерфейс позбавить клієнта від необхідності знати подробиці підсистеми.
2. Створіть клас фасаду, що реалізує цей інтерфейс. Він повинен переадресовувати виклики клієнта потрібним об’єктам підсистеми. Фасад повинен буде подбати про те, щоб правильно ініціалізувати об’єкти підсистеми.
3. Ви отримаєте максимум користі, якщо клієнт працюватиме тільки з фасадом. В такому випадку зміни в підсистемі стосуватимуться тільки коду фасаду, а клієнтський код залишиться робочим.
4. Якщо відповідальність фасаду стає розмитою, подумайте про введення додаткових фасадів.

Переваги

Ізолює клієнтів від компонентів складної підсистеми

Недоліки

Фасад ризикує стати божественим об’єктом, прив’язаним до всіх класів програми.

Шаблон «BRIDGE»

Міст — це структурний патерн проектування, який розділяє один або кілька класів на дві окремі ієрархії — абстракцію та реалізацію, дозволяючи змінювати код в одній гілці класів, незалежно від іншої

Проблема

Абстракція? Реалізація?! Звучить страхітливо! Розгляньмо простенький приклад, щоб зрозуміти про що йде мова.
У вас є клас геометричних Фігур , який має підкласи Круг та Квадрат . Ви хочете розширити ієрархію фігур за кольором, тобто мати Червоні та Сині фігури. Але для того, щоб все це об’єднати, доведеться створити 4 комбінації підкласів на зразок СиніКруги та ЧервоніКвадрати .

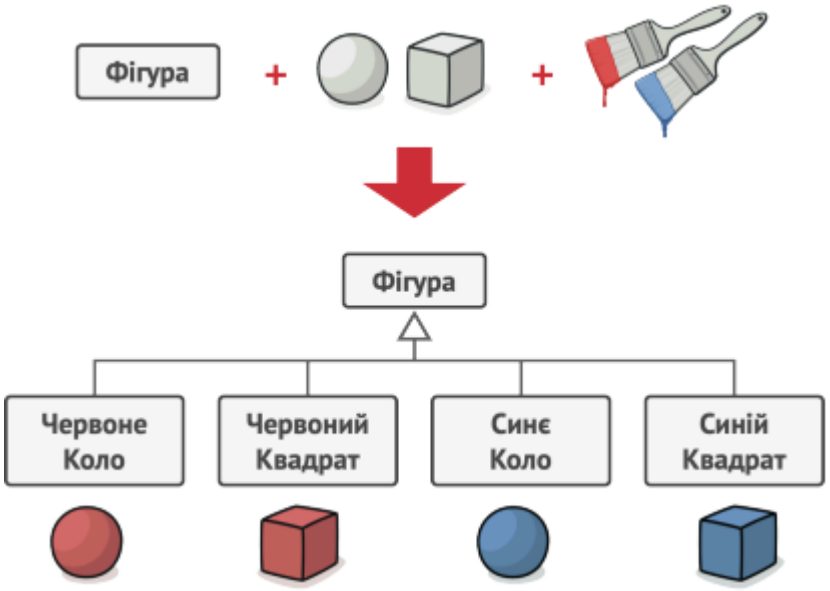

Рис.2.6 Кількість підкласів зростає в геометричній прогресії

При додаванні нових видів фігур і кольорів кількість комбінацій зростатиме в геометричній прогресії. Наприклад, щоб ввести в програму фігури трикутників, доведеться створити відразу два нових класи трикутників, по одному для кожного кольору. Після цього введення нового кольору вимагатиме створення вже трьох класів, по одному для кожного виду фігур. Чим далі, тим гірше.

Рішення

Корінь проблеми полягає в тому, що ми намагаємося розширити класи фігур одразу в двох незалежних площинах — за видом та кольором. Саме це призводить до розростання дерева класів. Патерн Міст пропонує замінити спадкування на делегування. Для цього потрібно виділити одну з таких «площин» в окрему ієрархію і посилатися на об’єкт цієї ієрархії, замість зберігання його стану та поведінки всередині одного класу.

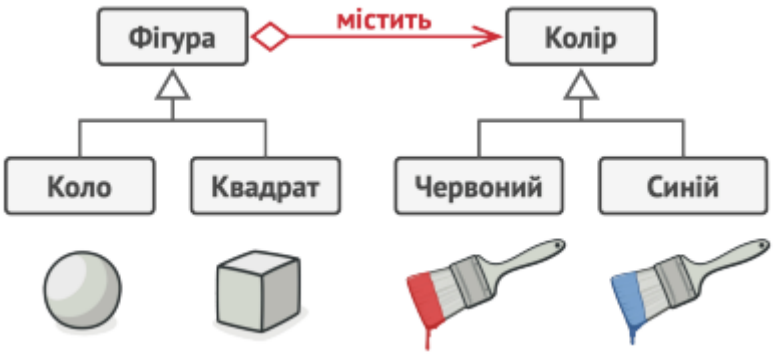

Рис.2.7 Розмноження підкласів можна зупинити, розбивши класи на кілька ієрархій.

Таким чином, ми можемо зробити Колір окремим класом з підкласами Червоний та Синій . Клас Фігур отримає посилання на об’єкт Кольору і зможе делегувати йому роботу, якщо виникне така необхідність. Такий зв’язок і стане мостом між Фігурами та Кольором . При додаванні нових класів кольорів не потрібно буде звертатись до класів фігур і навпаки

Абстракція і Реалізація

Ці терміни було введено в книзі GoF при описі Мосту. На мій погляд, вони виглядають занадто академічними та показують патерн складнішим, ніж він є насправді. Пам’ятаючи про приклад з фігурами й кольорами, давайте все ж таки розберемося, що мали на увазі автори патерна. Отже, абстракція (або інтерфейс) — це уявний рівень керування чим-небудь, що не виконує роботу самостійно, а делегує її рівню реалізації (який зветься платформою)

Якщо говорити про реальні програми, то абстракцією може виступати графічний інтерфейс програми (GUI), а реалізацією — низькорівневий код операційної системи (API), до якого графічний інтерфейс звертається, реагуючи на дії користувача.

Ви можете розвивати програму у двох різних напрямках:
• мати кілька різних GUI (наприклад, для звичайних користувачів та адміністраторів).
• підтримувати багато видів API (наприклад, працювати під Windows, Linux і macOS). Така програма може виглядати як один великий клубок коду, в якому змішано умовні оператори рівнів GUI та API.

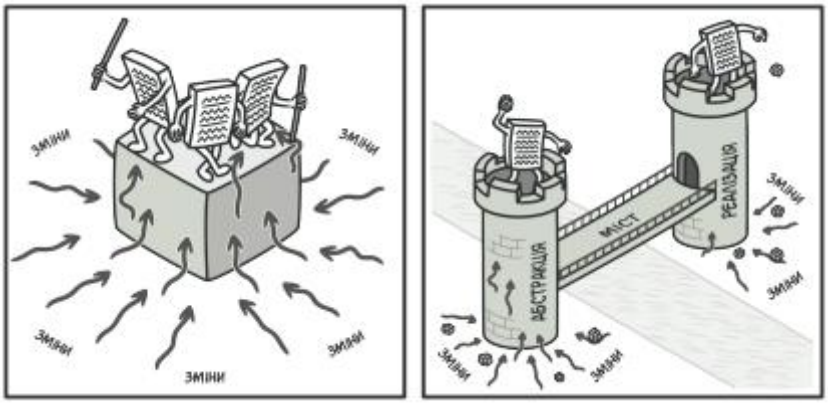

Рис.2.8 Коли зміни беруть проект в «осаду», вам легше відбиватися, якщо розділити монолітний код на частини

Ви можете спробувати структурувати цей хаос, створивши для кожної з варіацій інтерфейсу-платформи свої підкласи. Але такий підхід призведе до зростання класів комбінацій, і з кожною новою платформою їх буде все більше й більше.
Ми можемо вирішити цю проблему, застосувавши Міст. Патерн пропонує розплутати цей код, розділивши його на дві частини:

• Абстракцію: рівень графічного інтерфейсу програми.
• Реалізацію: рівень взаємодії з операційною системою.

Абстракція делегуватиме роботу одному з об’єктів реалізації. Причому, реалізації можна буде взаємозаміняти, але тільки за умови, що всі вони слідуватимуть єдиному інтерфейсу.
Таким чином, ви зможете змінювати графічний інтерфейс програми, не чіпаючи низькорівневий код роботи з операційною системою. І навпаки, ви зможете додавати підтримку нових операційних систем, створюючи нові підкласи реалізації, без необхідності правити код у класах графічного інтерфейсу.

Структура

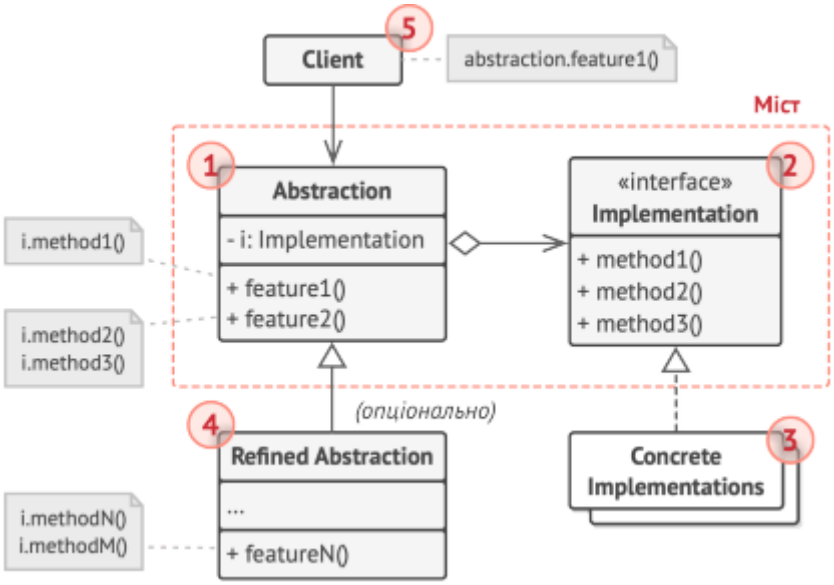

Рис.2.9 Реалізація патерну Міст

1. Абстракція містить керуючу логіку. Код абстракції делегує реальну роботу пов'язаному об'єктові реалізації.
2. Реалізація описує загальний інтерфейс для всіх реалізацій. Всі методи, які тут описані, будуть доступні з класу абстракції та його підкласів. Інтерфейси абстракції та реалізації можуть або збігатися, або бути абсолютно різними. Проте, зазвичай в реалізації живуть базові операції, на яких будуються складні операції абстракції.
3. Конкретні реалізації містять платформо-залежний код.
4. Розширені абстракції містять різні варіації керуючої логіки. Як і батьківский клас, працює з реалізаціями тільки через загальний інтерфейс реалізацій.
5. Клієнт працює тільки з об'єктами абстракції. Не рахуючи початкового зв'язування абстракції з однією із реалізацій, клієнтський код не має прямого доступу до об'єктів реалізації.

Застосування

Якщо ви хочете розділити монолітний клас, який містить кілька різних реалізацій якої-небудь функціональності (наприклад, якщо клас може працювати з різними системами баз даних)
Чим більший клас, тим важче розібратись у його коді, і тим більше це розтягує час розробки. Крім того, зміни, що вносяться в одну з реалізацій, призводять до редагування всього класу, що може викликати появу несподіваних помилок у коді. Міст дозволяє розділити монолітний клас на кілька окремих ієрархій. Після цього ви можете змінювати код в одній гілці класів незалежно від іншої. Це спрощує роботу над кодом і зменшує ймовірність внесення помилок.

Якщо клас потрібно розширювати в двох незалежних площинах.
Міст пропонує виділити одну з таких площин в окрему ієрархію класів, зберігаючи посилання на один з її об'єктів у початковому класі.

Якщо ви хочете мати можливість змінювати реалізацію під час виконання програми.
Міст дозволяє замінювати реалізацію навіть під час виконання програми, оскільки конкретна реалізація не «зашита» в клас абстракції.
До речі, через цей пункт Міст часто плутають із Стратегією. Зверніть увагу, що у Моста цей пункт займає останнє місце за значущістю, оскільки його головна задача — структурна

Кроки реалізації

1. Визначте, чи існують у ваших класах два непересічних виміри. Це може бути функціональність/платформа, предметна область/інфраструктура, фронт-енд/бек-енд або інтерфейс/реалізація
2. Продумайте, які операції будуть потрібні клієнтам, і опишіть їх у базовому класі абстракції.
3. Визначте поведінки, які доступні на всіх платформах, та виберіть з них ту частину, яка буде потрібна для абстракції. На підставі цього опишіть загальний інтерфейс реалізації.
4. Для кожної платформи створіть власний клас конкретної реалізації. Всі вони повинні дотримуватися загального інтерфейсу, який ми виділили перед цим.
5. Додайте до класу абстракції посилання на об'єкт реалізації. Реалізуйте методи абстракції, делегуючи основну роботу пов'язаному об'єкту реалізації.
6. Якщо у вас є кілька варіацій абстракції, створіть для кожної з них власний підклас.
7. Клієнт повинен подати об'єкт реалізації до конструктора абстракції, щоб зв'язати їх разом. Після цього він може вільно використовувати об'єкт абстракції, забувши про реалізацію.

Переваги

- Дозволяє будувати платформо-незалежні програми.
- Приховує зайві або небезпечні деталі реалізації від клієнтського коду.
- Реалізує принцип відкритості/закритості.

Недоліки

Ускладнює код програми внаслідок введення додаткових класів

Шаблон «TEMPLATE METHOD»

Шаблонний метод — це поведінковий патерн проектування, який визначає кістяк алгоритму, перекладаючи відповідальність за деякі його кроки на підкласи. Патерн дозволяє підкласам перевизначати кроки алгоритму, не змінюючи його загальної структури.

Проблема

Ви пишете програму для дата-майнінгу в офісних документах. Користувачі завантажуватимуть до неї документи різних форматів (PDF, DOC, CSV), а програма повинна видобути з них корисну інформацію. У першій версії ви обмежилися обробкою тільки DOC файлів. У наступній версії додали підтримку CSV. А через місяць «прикрутили» роботу з PDF документами

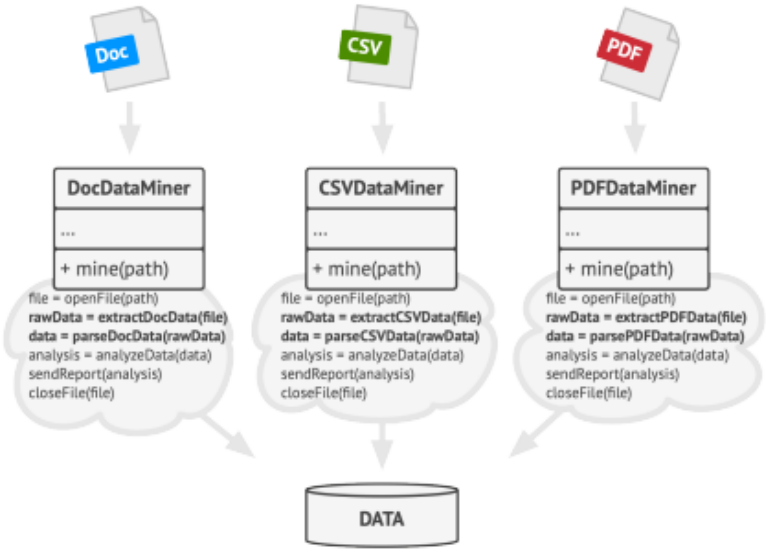

Рис.2.10 Класи дата-майнінгу містять багато дублювань.

В якийсь момент ви помітили, що код усіх трьох класів обробки документів хоч і відрізняється в частині роботи з файлами, але містить досить багато спільного в частині самого видобування даних. Було б добре позбутися від повторної реалізації алгоритму видобування даних у кожному з класів.
До того ж інший код, який працює з об'єктами цих класів, наповнений умовами, що перевіряють тип обробника перед початком роботи. Весь цей код можна спростити, якщо злити всі три класи в одне ціле або звести їх до загального інтерфейсу.

Рішення
Патерн Шаблонний метод пропонує розбити алгоритм на послідовність кроків, описати ці кроки в окремих методах і викликати їх в одному шаблонному методі один за одним.
Це дозволить підкласам перевизначити деякі кроки алгоритму, залишаючи без змін його структуру та інші кроки, які для цього підкласу не є важливими.
У нашому прикладі з дата-майнінгом ми можемо створити загальний базовий клас для всіх трьох алгоритмів. Цей клас складатиметься з шаблонного методу, який послідовно викликає кроки розбору документів.
Для початку кроки шаблонного методу можна зробити абстрактними. З цієї причини усі підкласи повинні будуть реалізувати кожен з кроків по-своєму. В нашому випадку всі підкласи вже містять реалізацію кожного з кроків, тому додатково нічого робити не потрібно.

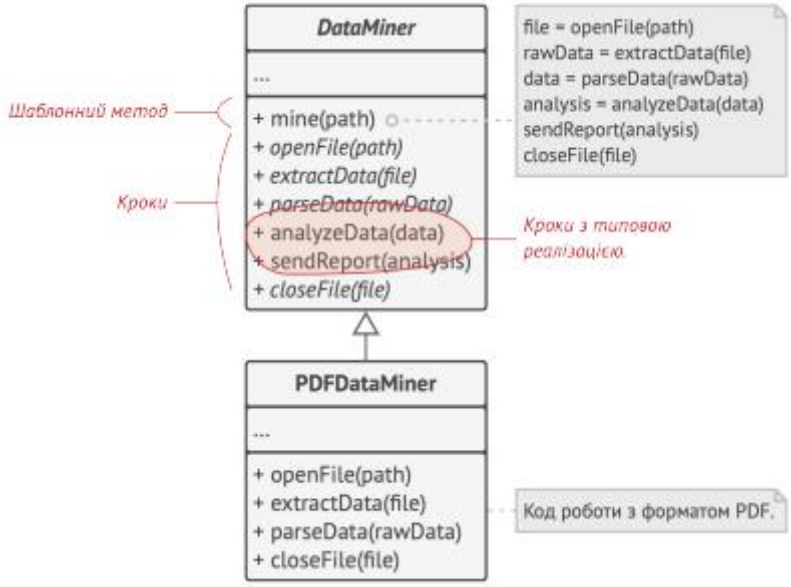

Рис. 2.11 Шаблонний метод розбиває алгоритм на кроки, дозволяючи підкласами перевизначити деякі з них.

Справді важливим є наступний етап. Тепер ми можемо визначити спільну поведінку для всіх трьох класів і винести її до суперкласу. У нашому прикладі кроки відкривання та закривання документів відрізнятимуться для всіх підкласів, тому залишаться абстрактними. З іншого боку, код обробки даних, однаковий для всіх типів документів, переїде до базового класу.
Як бачите, у нас з'явилося два типа кроків: абстрактні, що кожен підклас обов'язково має реалізувати, а також кроки з типовою реалізацією, які можна перевизначити в підкласах, але це не обов'язково.
Але є ще й третій тип кроків — хуки. Це опціональні кроки, які виглядають як звичайні методи, але взагалі не містять коду. Шаблонний метод залишиться робочим, навіть якщо жоден підклас не перевизначить такий хук. Підсумовуючи сказане, хук дає підкласам додаткові точки «вклинювання» в хід шаблонного методу.

Структура

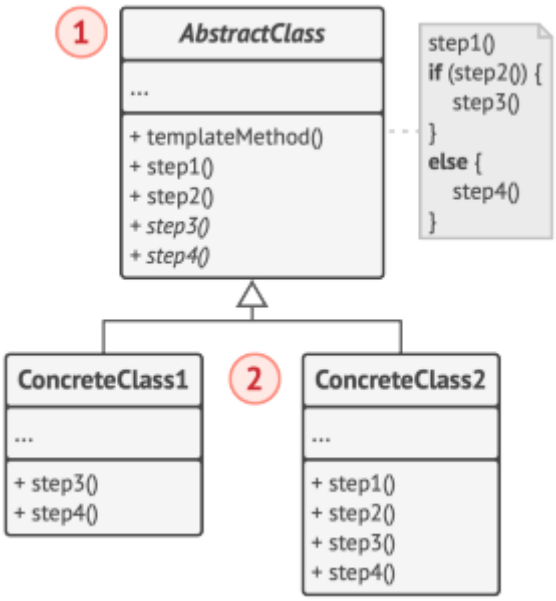

Рис.2.12 Адаптер об'єктів

1. Абстрактний клас визначає кроки алгоритму й містить шаблонний метод, що складається з викликів цих кроків. Кроки можуть бути як абстрактними, так і містити реалізацію за замовчуванням.
2. Конкретний клас перевизначає деякі або всі кроки алгоритму. Конкретні класи не перевизначають сам шаблонний метод

Застосування

Якщо підкласи повинні розширювати базовий алгоритм, не змінюючи його структури..
Шаблонний метод дозволяє підкласами розширювати певні кроки алгоритму через спадкування, не змінюючи при цьому структуру алгоритмів, оголошену в базовому класі.

Я Якщо у вас є кілька класів, які роблять одне й те саме з незначними відмінностями. Якщо ви редагуєте один клас, тоді доводиться вносити такі ж виправлення до інших класів.
Патерн шаблонний метод пропонує створити для схожих класів спільний суперклас та оформити в ньому головний алгоритм у вигляді кроків. Кроки, які відрізняються, можна перевизначити у підкласах. Це дозволить прибрати дублювання коду в кількох класах, які відрізняються деталями, але мають схожу поведінку

Кроки реалізації

1. Вивчіть алгоритм і подумайте, чи можна його розбити на кроки. Вирішіть, які кроки будуть стандартними для всіх варіацій алгоритму, а які можуть бути змінюваними.
2. Створіть абстрактний базовий клас. Визначте в ньому шаблонний метод. Цей метод повинен складатися з викликів кроків алгоритму. Є сенс у тому, щоб зробити шаблонний метод фінальним, аби підкласи не могли перевизначити його (якщо ваша мова програмування це дозволяє).
3. Додайте до абстрактного класу методи для кожного з кроків алгоритму. Ви можете зробити ці методи абстрактними або додати якусь типову реалізацію. У першому випадку всі підкласи повинні будуть реалізувати ці методи, а в другому — тільки якщо реалізація кроку в підкласі відрізняється від стандартної версії.
4. Подумайте про введення хуків в алгоритм. Найчастіше хуки розташовують між основними кроками алгоритму, а також до та після всіх кроків.

5. Створіть конкретні класи, успадкувавши їх від абстрактного класу. Реалізуйте в них всі кроки та хуки, яких не вистачає.

Переваги

Полегшує повторне використання коду.

Недоліки

- Ви жорстко обмежені скелетом існуючого алгоритму.
- Ви можете порушити принцип підстановки Барбари Лісков, змінюючи базову поведінку одного з кроків алгоритму через підклас.
- У міру зростання кількості кроків шаблонний метод стає занадто складно підтримувати.

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:

У процесі форматування JSON часто потрібно виконувати однакові операції, як-от парсинг JSON, додавання відступів, видалення зайвих символів чи рекурсивну обробку вузлів. Якщо ці операції реалізовувати окремо в кожному форматувальнику, це призводить до дублювання коду, ускладнює його підтримку та збільшує ризик помилок. Водночас необхідно надати можливість легко змінювати чи розширювати специфічні алгоритми форматування.

Рішення

Для вирішення цієї проблеми використано патерн Шаблонний метод, який дозволяє визначити загальний алгоритм роботи в абстрактному класі та реалізовувати специфічну логіку в його підкласах.

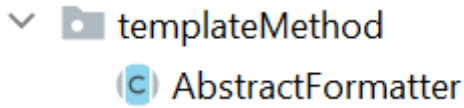

Рис. 1. Ієрархія класів. Реалізація патерну Template Method.

```java
public abstract class AbstractFormatter {
    1 usage
    protected final ObjectMapper objectMapper = new ObjectMapper();

    2 usages
    protected JsonNode parseJson(RawJsonDto rawJsonDto) throws Exception {
        return objectMapper.readTree(rawJsonDto.getRawData());
    }

    4 usages
    protected void appendIndent(StringBuilder builder, int level) { builder.append("  ".repeat(level)); }

    2 usages
    protected 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. Абстрактний клас AbstractFormatter для реалізації патерну Template Method.

У класі AbstractFormatter зібрано спільні методи, як-от parseJson, appendIndent і removeTrailingComma, які використовуються у конкретних класах форматування FlatFormatterStrategy і PrettyFormatterStrategy. Цей підхід дозволяє зосередити загальну логіку в одному місці, що спрощує її підтримку та зменшує дублювання. Це зменшує ризик помилок, підвищує масштабованість та забезпечує кращу структуризацію коду.

```
public class PrettyFormatterStrategy extends AbstractFormatter implements FormatterStrategy {

    @Override
    public String format(RawJsonDto rawJsonDto) {
        try {
            JsonNode rootNode = parseJson(rawJsonDto);
            StringBuilder builder = new StringBuilder();
            formatNodeRecursive(rootNode, level: 0, builder);
            return builder.toString().trim();
        } catch (Exception e) {
            throw new InvalidJsonException("Error during formatting", e);
        }
    }

    3 usages
    private void formatNodeRecursive(JsonNode node, int level, StringBuilder builder) {
        if (node.isObject()) {
            builder.append("{\n");
            node.fields().forEachRemaining(entry -> {
                appendIndent(builder, level: level + 1);
                builder.append("\"").append(entry.getKey()).append("\": ");
                formatNodeRecursive(entry.getValue(), level: level + 1, builder);
                builder.append(",\n");
            });
            removeTrailingComma(builder);
            builder.append("\n");
            appendIndent(builder, level);
```

Рис. 2.1.1 Використання методів класу AbstractFormatter у конкретних класах форматування

```
public class FlatFormatterStrategy extends AbstractFormatter implements FormatterStrategy {

    @Override
    public String format(RawJsonDto rawJsonDto) {
        try {
            JsonNode rootNode = parseJson(rawJsonDto);
            StringBuilder builder = new StringBuilder();
            formatNodeRecursive(rootNode, currentPath: "", builder);
            return builder.toString().trim();
        } catch (Exception e) {
            throw new InvalidJsonException("Error during formatting", e);
        }
    }
```

Рис. 2.1.2 Використання методів класу AbstractFormatter у конкретних класах форматування

ВИСНОВОК

Реалізація патерну Template method у межах цього застосунку дозволила зібрати спільні методи які використовуються у конкретних класах форматування. Цей підхід дозволяє зосередити загальну логіку в одному місці, що спрощує її підтримку та зменшує дублювання.