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 №5**
із дисципліни «Технології Розробки Програмного Забезпечення»
**Тема: ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»**
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 Command 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 «ADAPTER»

Адаптер — це структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом.

***Проблема***

Уявіть, що ви пишете програму для торгівлі на біржі. Ваша програма спочатку завантажує біржові котирування з декількох джерел в XML, а потім малює гарні графіки

У якийсь момент ви вирішуєте покращити програму, застосувавши сторонню бібліотеку аналітики. Але от біда — бібліотека підтримує тільки формат даних JSON, несумісний із вашим додатком.

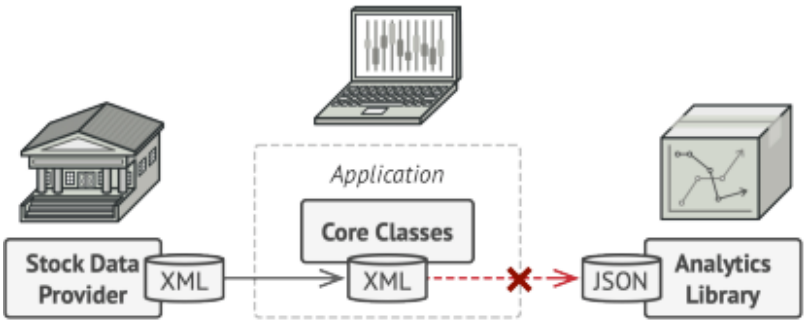


*Рис.2.1 Під’єднати сторонню бібліотеку неможливо через несумісність форматів даних*

Ви могли б переписати цю бібліотеку, щоб вона підтримувала формат XML, але, по-перше, це може порушити роботу наявного коду, який уже залежить від бібліотеки, по-друге, у вас може просто не бути доступу до її вихідного коду.

***Рішення***

Ви можете створити адаптер. Це об’єкт-перекладач, який трансформує інтерфейс або дані одного об’єкта таким чином, щоб він став зрозумілим іншому об’єкту.

Адаптер загортає один з об’єктів так, що інший об’єкт навіть не підозрює про існування першого. Наприклад, об’єкт, що працює в метричній системі вимірювання, можна «обгорнути» адаптером, який буде конвертувати дані у фути.

Адаптери можуть не тільки конвертувати дані з одного формату в інший, але й допомагати об’єктам із різними інтерфейсами працювати разом. Це виглядає так:

- Адаптер має інтерфейс, сумісний з одним із об’єктів.
- Тому цей об’єкт може вільно викликати методи адаптера.
- Адаптер отримує ці виклики та перенаправляє їх іншому об’єкту, але вже в тому форматі та послідовності, які є зрозумілими для цього об’єкта.

Іноді вдається створити навіть двосторонній адаптер, який може працювати в обох напрямках.

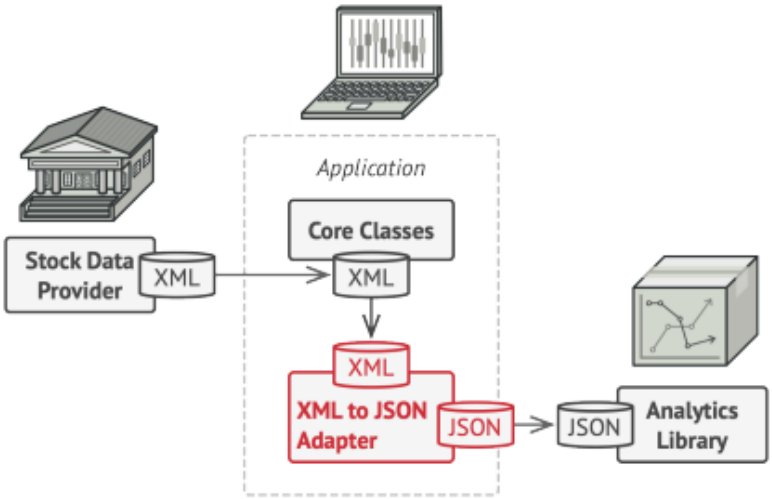


*Рис.2.2 Програма може працювати зі сторонньою бібліотекою через адаптер*

*Аналогія з життя*
Під час вашої першої подорожі за кордон спроба зарядити ноутбук може стати неприємним сюрпризом, тому що стандарти розеток у багатьох країнах різняться.
Ваша європейська зарядка стане непотрібом у США без спеціального адаптера, що дозволяє під'єднуватися до розетки іншого типу.

***Структура***

Адаптер об'єктів
Ця реалізація використовує агрегацію: об'єкт адаптера «загортає», тобто містить посилання на службовий об'єкт. Такий підхід працює в усіх мовах програмування.

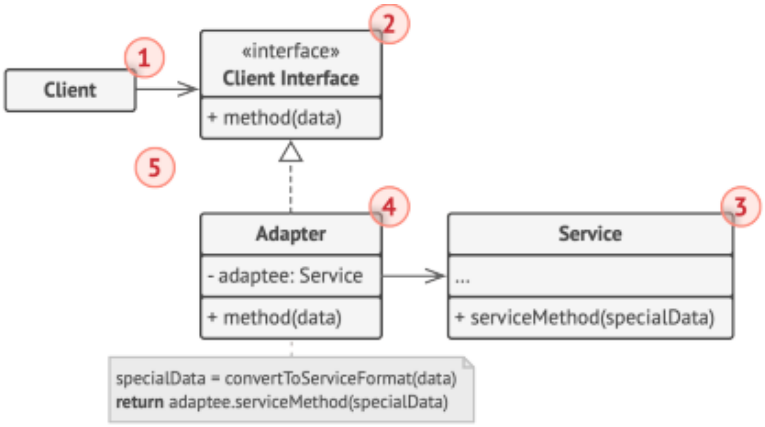


*Рис.2.3 Адаптер об'єктів*

1. Клієнт — це клас, який містить існуючу бізнес-логіку програми.
2. Клієнтський інтерфейс описує протокол, через який клієнт може працювати з іншими класами.
3. Сервіс — це який-небудь корисний клас, зазвичай сторонній. Клієнт не може використовувати цей клас безпосередньо, оскільки сервіс має незрозумілий йому інтерфейс.
4. Адаптер — це клас, який може одночасно працювати і з клієнтом, і з сервісом. Він реалізує клієнтський інтерфейс і містить посилання на об'єкт сервісу. Адаптер отримує виклики від клієнта через методи клієнтського інтерфейсу, а потім конвертує їх у виклики методів загорнутого об'єкта в потрібному форматі.
5. Працюючи з адаптером через інтерфейс, клієнт не прив'язується до конкретного класу адаптера. Завдяки цьому ви можете додавати до програми нові види адаптерів, незалежно від клієнтського коду. Це може стати в нагоді, якщо інтерфейс сервісу раптом зміниться, наприклад, після виходу нової версії сторонньої бібліотеки.

***Застосування***

*Якщо ви хочете використати сторонній клас, але його інтерфейс не відповідає решті кодів програми.*
Адаптер дозволяє створити об'єкт-прокладку, який перетворюватиме виклики програми у формат, зрозумілий сторонньому класу

*Якщо вам потрібно використати декілька існуючих підкласів, але в них не вистачає якої-небудь спільної функціональності, а розширити суперклас ви не можете.*
Ви могли б створити ще один рівень підкласів та додати до них забраклу функціональність. Але при цьому доведеться дублювати один і той самий код в обох гілках підкласів. Більш елегантним рішенням було б розмістити відсутню функціональність в адаптері й пристосувати його для роботи із суперкласом. Такий адаптер зможе працювати з усіма підкласами ієрархії. Це рішення сильно нагадуватиме патерн Декоратор.

***Кроки реалізації***

1. Переконайтеся, що у вас є два класи з незручними інтерфейсами:
    - корисний сервіс — службовий клас, який ви не можете змінювати (він або сторонній, або від нього залежить інший код);
    - один або декілька клієнтів — існуючих класів програми, які не можуть використовувати сервіс через несумісний із ним інтерфейс
2. Опишіть клієнтський інтерфейс, через який класи програм могли б використовувати клас сервісу.
3. Створіть клас адаптера, реалізувавши цей інтерфейс.
4. Розмістіть в адаптері поле, що міститиме посилання на об'єкт сервісу. Зазвичай це поле заповнюють об'єктом, переданим у конструктор адаптера. Але цей об'єкт можна передавати й безпосередньо до методів адаптера.
5. Реалізуйте всі методи клієнтського інтерфейсу в адаптері. Адаптер повинен делегувати основну роботу сервісу.
6. Програма повинна використовувати адаптер тільки через клієнтський інтерфейс. Це дозволить легко змінювати та додавати адаптери в майбутньому.

***Переваги***

Відокремлює та приховує від клієнта подробиці перетворення різних інтерфейсів

***Недоліки***

Ускладнює код програми внаслідок введення додаткових класів.

## Шаблон «BUILDER»

Будівельник — це породжувальний патерн проектування, що дає змогу створювати складні об'єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об'єктів.

***Проблема***

Уявіть складний об'єкт, що вимагає кропіткої покрокової ініціалізації безлічі полів і вкладених об'єктів. Код ініціалізації таких об'єктів зазвичай захований всередині монстроподібного конструктора з десятком параметрів. Або ще гірше — розпорошений по всьому клієнтському коду.

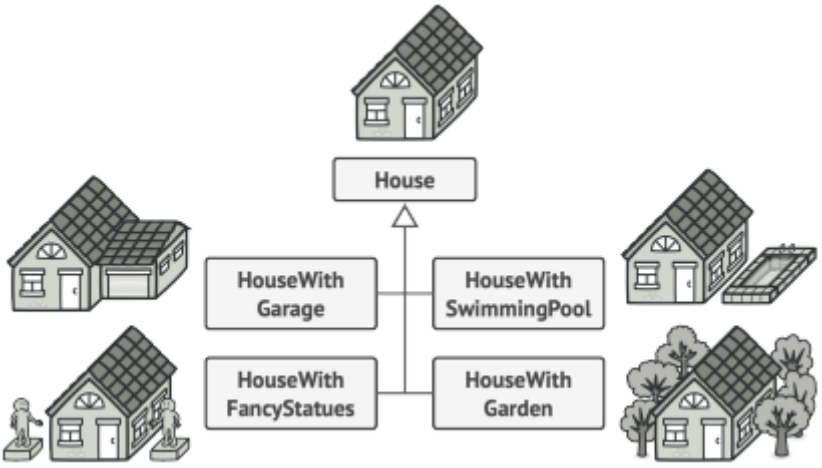


*Рис.2.4 Створивши купу підкласів для всіх конфігурацій об'єктів, ви можете надміру ускладнити програму*

Наприклад, подумаймо про те, як створити об'єкт Будинок . Щоб побудувати стандартний будинок, потрібно: звести 4 стіни, встановити двері, вставити пару вікон та постелити дах. Але що робити, якщо ви хочете більший та світліший будинок, що має басейн, сад та інше добро?

Найпростіше рішення — розширити клас Будинок , створивши підкласи для всіх комбінацій параметрів будинку. Проблема такого підходу — величезна кількість класів, які вам доведеться створити. Кожен новий параметр, на кшталт кольору шпалер чи матеріалу покрівлі, змусить вас створювати все більше й більше класів для перерахування усіх можливих варіантів.

Аби не плодити підкласи, можна підійти до вирішення питання з іншого боку. Ви можете створити гігантський конструктор Будинку , що приймає безліч параметрів для контролю над створюваним продуктом. Так, це позбавить вас від підкласів, але призведе до появи іншої проблеми.

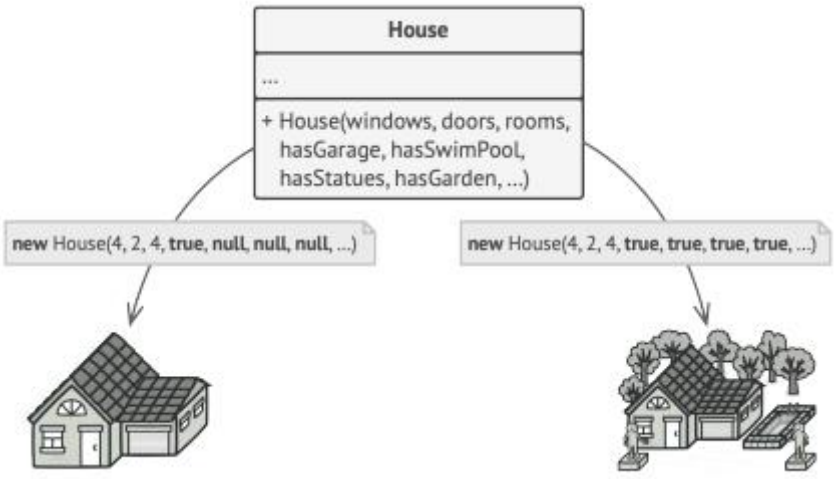


*Рис.2.5 Конструктор з безліччю параметрів має свій недолік: не всі параметри потрібні протягом більшої частини часу*

Більшість цих параметрів буде простоювати, а виклики конструктора будуть виглядати монстроподібно через довгий список параметрів. Наприклад, басейн є далеко не в кожному будинку, тому параметри, пов'язані з басейнами, даремно простоюватимуть у 99% випадків.

***Рішення***

Патерн Будівельник пропонує винести конструювання об'єкта за межі його власного класу, доручивши цю справу окремим об'єктам, які називаються будівельниками.

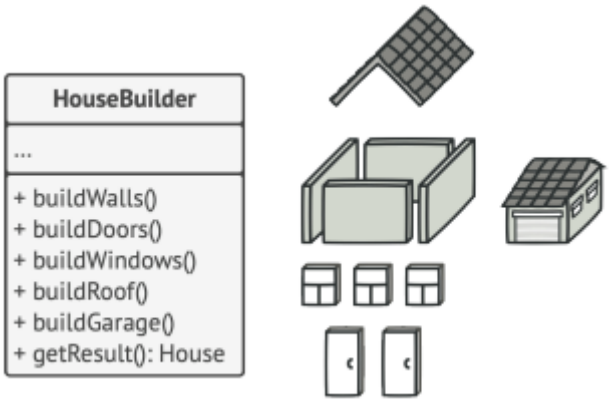


*Рис.2.6 Будівельник дозволяє створювати складні об'єкти покроково. Проміжний результат захищений від стороннього втручання.*

Патерн пропонує розбити процес конструювання об'єкта на окремі кроки (наприклад, побудуватиСтіни , встановитиДвері і т. д.) Щоб створити об'єкт, вам потрібно по черзі викликати методи будівельника. До того ж не потрібно викликати всі кроки, а лише ті, що необхідні для виробництва об'єкта певної конфігурації.

Зазвичай один і той самий крок будівництва може відрізнятися для різних варіацій виготовлених об'єктів. Наприклад, дерев'яний будинок потребує будівництва стін з дерева, а кам'яний — з каменю.

У цьому випадку ви можете створити кілька класів будівельників, які по-різному виконуватимуть ті ж самі кроки. Використовуючи цих будівельників в одному й тому самому будівельному процесі, ви зможете отримувати на виході різні об'єкти.


*Рис.2.7 Різні будівельники виконають одне і те саме завдання по-різному*

Наприклад, один будівельник робить стіни з дерева і скла, інший — з каменю і заліза, третій — із золота та діамантів. Викликавши одні й ті самі кроки будівництва, у першому випадку ви отримаєте звичайний житловий будинок, у другому — маленьку фортецю, а в третьому — розкішне житло. Зауважу, що код, який викликає кроки будівництва, повинен працювати з будівельниками через загальний інтерфейс, щоб їх можна було вільно замінювати один на інший.

*Директор*

Ви можете піти далі та виділити виклики методів будівельника в окремий клас, що називається «Директором». У цьому випадку директор задаватиме порядок кроків будівництва, а будівельник — виконуватиме їх.


*Рис.2.8. Директор знає, які кроки повинен виконати об'єкт-будівельник, щоб виготовити продукт.*

Окремий клас директора не є суворо обов'язковим. Ви можете викликати методи будівельника і безпосередньо з клієнтського коду. Тим не менш, директор корисний, якщо у вас є кілька способів конструювання продуктів, що відрізняються порядком і наявними кроками конструювання. У цьому випадку ви зможете об'єднати всю цю логіку в одному класі.

Така структура класів повністю приховає від клієнтського коду процес конструювання об'єктів. Клієнту залишиться лише прив'язати бажаного будівельника до директора, а потім отримати від будівельника готовий результат.

***Структура***

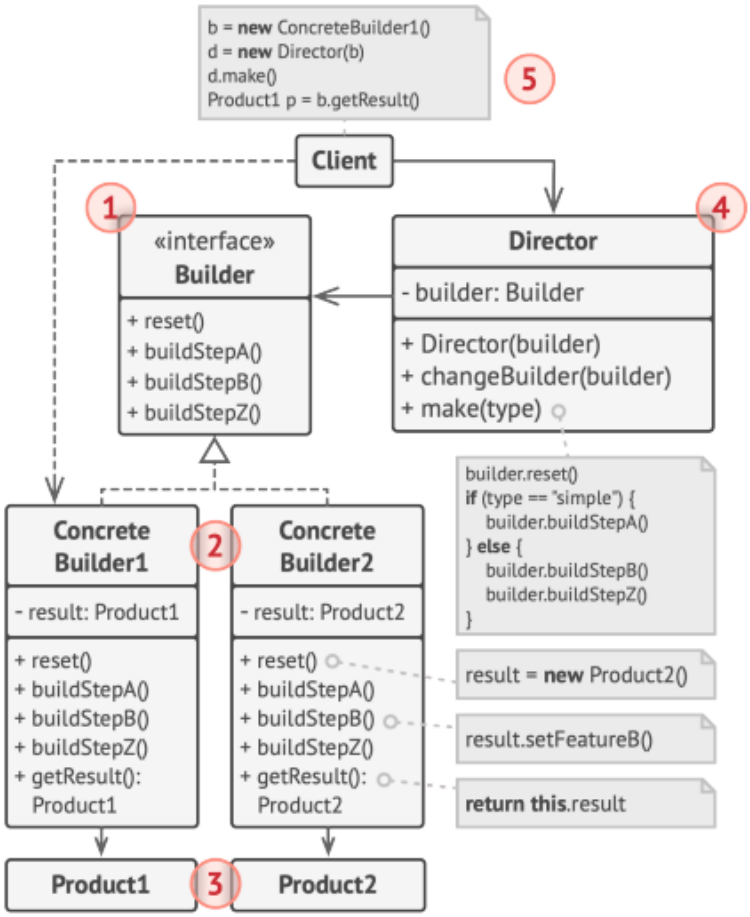


*Рис.2.10 Структура патерну Builder*

1. Інтерфейс будівельника оголошує кроки конструювання продуктів, спільні для всіх видів будівельників.
2. Конкретні будівельники реалізують кроки будівництва, кожен по-своєму. Конкретні будівельники можуть виготовляти різнорідні об'єкти, що не мають спільного інтерфейсу.

3. Продукт — об’єкт, що створюється. Продукти, зроблені різними будівельниками, не зобов’язані мати спільний інтерфейс.
4. Директор визначає порядок виклику кроків будівельників, необхідних для виробництва продуктів тієї чи іншої конфігурації.
5. Зазвичай Клієнт подає до конструктора директора вже готовий об’єкт-будівельник, а директор надалі використовує тільки його. Але можливим є також інший варіант, коли клієнт передає будівельника через параметр будівельного методу директора. У такому випадку можна щоразу використовувати різних будівельників для виробництва різноманітних відображень об’єктів.

***Застосування***

*Коли ви хочете позбутися від «телескопічного конструктора».*

```
class Pizza {
  Pizza(int size) { ... }
  Pizza(int size, boolean cheese) { ... }
  Pizza(int size, boolean cheese, boolean pepperoni) { ... }
  // ...
```

*Рис.2.11 «Телескопічний конструктор». Такого монстра можна створити тільки в мовах, що мають механізм перевантаження методів, наприклад, C# або Java*

Патерн Будівельник дозволяє збирати об’єкти покроково, викликаючи тільки ті кроки, які вам потрібні. Отже, більше не потрібно намагатися «запхати» до конструктора всі можливі опції продукту.

*Коли ваш код повинен створювати різні уявлення якогось об’єкта. Наприклад, дерев’яні та залізобетонні будинки.*

Будівельник можна застосувати, якщо створення кількох відображень об’єкта складається з однакових етапів, які відрізняються деталями. Інтерфейс будівельників визначить всі можливі етапи конструювання. Кожному відображенню відповідатиме власний клас-будівельник. Порядок етапів будівництва визначатиме клас-директор.

*Коли вам потрібно збирати складні об’єкти, наприклад, дерева Компонувальника*

Будівельник конструює об’єкти покроково, а не за один прохід. Більш того, кроки будівництва можна виконувати рекурсивно. А без цього не побудувати деревоподібну структуру на зразок Компонувальника.
Зауважте, що Будівельник не дозволяє стороннім об’єктам отримувати доступ до об’єкта, що конструюється, доки той не буде повністю готовий. Це захищає клієнтський код від отримання незавершених «битих» об’єктів.

***Кроки реалізації***

1. Переконайтеся в тому, що створення різних відображень об’єкта можна звести до загальних кроків.
2. Опишіть ці кроки в загальному інтерфейсі будівельників.
3. Для кожного з відображень об’єкта-продукту створіть по одному класу-будівельнику й реалізуйте їхні методи будівництва. Не забудьте про метод отримання результату. Зазвичай конкретні будівельники визначають власні методи отримання результату будівництва. Ви не можете описати ці методи в інтерфейсі будівельників, оскільки продукти не обов’язково повинні мати загальний базовий клас або інтерфейс. Але ви завжди можете додати метод отримання результату до загального інтерфейсу, якщо ваші будівельники виготовляють однорідні продукти, які мають спільного предка.
4. Подумайте про створення класу директора. Його методи створюватимуть різні конфігурації продуктів, викликаючи різні кроки одного і того самого будівельника.

5. Клієнтський код повинен буде створювати й об’єкти будівельників, й об’єкт директора. Перед початком будівництва клієнт повинен зв’язати певного будівельника з директором. Це можна зробити або через конструктор, або через сетер, або подавши будівельника безпосередньо до будівельного методу директора.
6. Результат будівництва можна повернути з директора, але тільки якщо метод повернення продукту вдалося розмістити в загальному інтерфейсі будівельників. Інакше ви жорстко прив’яжете директора до конкретних класів будівельників.

***Переваги***

- Дозволяє створювати продукти покроково.
- Дозволяє використовувати один і той самий код для створення різноманітних продуктів.
- Ізолює складний код конструювання продукту від його головної бізнес-логіки

***Недоліки***

- Ускладнює код програми за рахунок додаткових класів.
- Клієнт буде прив’язаний до конкретних класів будівельників, тому що в інтерфейсі будівельника може не бути методу отримання результату

## Шаблон «COMMAND»

Команда — це поведінковий патерн проектування, який перетворює запити на об’єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

***Проблема***

Уявіть, що ви працюєте над програмою текстового редактора. Якраз підійшов час розробки панелі керування. Ви створили клас гарних Кнопок і хочете використовувати його для всіх кнопок програми, починаючи з панелі керування та закінчуючи звичайними кнопками в діалогах.

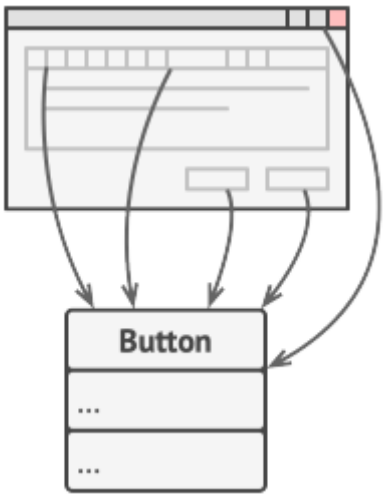


*Рис.2.12 Всі кнопки програми успадковані від одного класу.*

Усі ці кнопки, хоч і виглядають схоже, але виконують різні команди. Виникає запитання: куди розмістити код обробників кліків по цих кнопках? Найпростіше рішення — це створити підкласи для кожної кнопки та перевизначити в них методи дії для різних завдань.

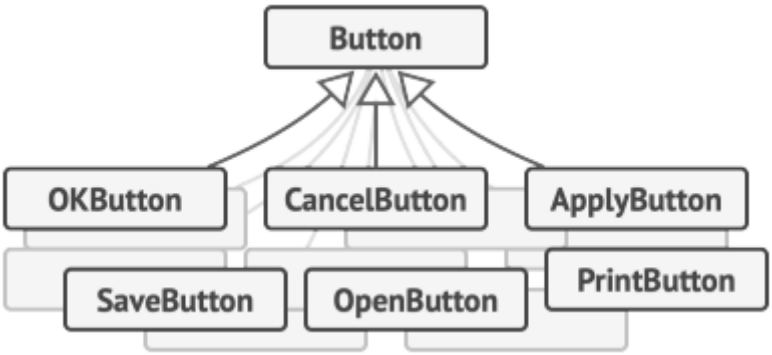


*Рис.2.12 Безліч підкласів кнопок*

Але скоро стало зрозуміло, що такий підхід нікуди не годиться. По-перше, з'являється дуже багато підкласів. По-друге, код кнопок, який відноситься до графічного інтерфейсу, починає залежати від класів бізнес-логіки, яка досить часто змінюється.


*Рис.2.13 Кілька класів дублюють одну і ту саму функціональність.*

Проте, найгірше ще попереду, адже деякі операції, на кшталт «зберегти», можна викликати з декількох місць: натиснувши кнопку на панелі керування, викликавши контекстне меню або натиснувши клавіші Ctrl+S . Коли в програмі були тільки кнопки, код збереження був тільки у підкласі SaveButton . Але тепер його доведеться продублювати ще в два класи.

***Рішення***

Хороші програми зазвичай структурують у вигляді шарів. Найпоширеніший приклад — це шари користувацького інтерфейсу та бізнес-логіки. Перший лише малює гарне зображення для користувача, але коли потрібно зробити щось важливе, інтерфейс користувача «просить» шар бізнес-логіки зайнятися цим.

У дійсності це виглядає так: один з об'єктів інтерфейсу користувача викликає метод одного з об'єктів бізнес-логіки, передаючи до нього якісь параметри.

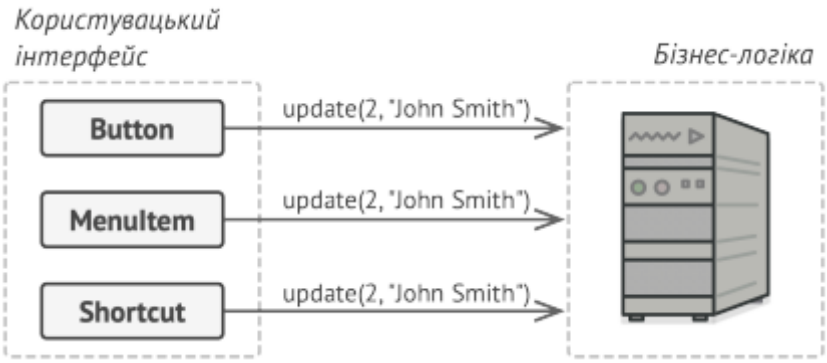


*Рис.2.14 Прямий доступ з UI до бізнес-логіки.*

Патерн Команда пропонує більше не надсилати такі виклики безпосередньо. Замість цього кожен виклик, що відрізняється від інших, слід звернути у власний клас з єдиним методом, який і здійснюватиме виклик. Такий об'єкт зветься командою.

До об'єкта інтерфейсу можна буде прив'язати об'єкт команди, який знає, кому і в якому вигляді слід відправляти запити. Коли об'єкт інтерфейсу буде готовий передати запит, він викличе метод команди, а та — подбає про все інше.

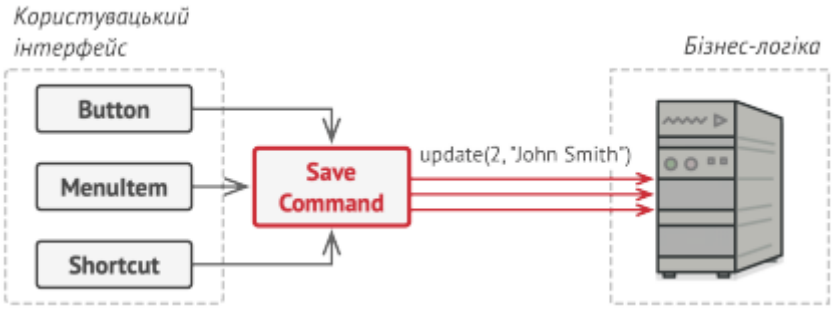


*Рис.2.15 Доступ з UI до бізнес-логіки через команду.*

Класи команд можна об'єднати під загальним інтерфейсом, що має єдиний метод запуску команди. Після цього одні й ті самі відправники зможуть працювати з різними командами, не прив'язуючись до їхніх класів. Навіть більше, команди можна буде взаємозаміняти «на льоту», змінюючи підсумкову поведінку відправників.

Параметри, з якими повинен бути викликаний метод об'єкта одержувача, можна заздалегідь зберегти в полях об'єктакоманди. Завдяки цьому, об'єкти, які надсилають запити, можуть не турбуватися про те, щоб зібрати необхідні дані для одержувача. Навіть більше, вони тепер взагалі не знають, хто буде одержувачем запиту. Вся ця інформація прихована всередині команди.

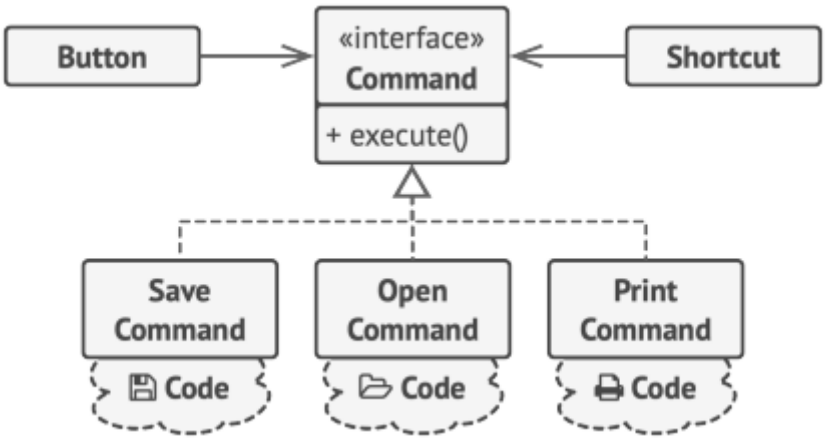


*Рис.2.16 Класи UI делегують роботу командам.*

Після застосування Команди в нашому прикладі з текстовим редактором вам більше не потрібно буде створювати безліч підкласів кнопок для різних дій. Буде достатньо одного класу з полем для зберігання об'єкта команди.

Використовуючи загальний інтерфейс команд, об'єкти кнопок посилатимуться на об'єкти команд різних типів. При натисканні кнопки делегуватимуть роботу командам, а команди — перенаправляти виклики тим чи іншим об'єктам бізнес-логіки.

Так само можна вчинити і з контекстним меню, і з гарячими клавішами. Вони будуть прив'язані до тих самих об'єктів команд, що і кнопки, позбавляючи класи від дублювання.

Таким чином, команди стануть гнучким прошарком між користувацьким інтерфейсом та бізнес-логікою. І це лише невелика частина тієї користі, яку може принести патерн Команда!

*Аналогія з життя*

Ви заходите в ресторан і сідаєте біля вікна. До вас підходить ввічливий офіціант і приймає замовлення, записуючи всі побажання в блокнот.

Закінчивши, він поспішає на кухню, вириває аркуш з блокнота та клеїть його на стіну. Далі лист опиняється в руках кухаря, який читає замовлення і готує описану страву. Далі лист опиняється в руках кухаря, який читає замовлення і готує описану страву.

*Рис. 2.17 Приклад замовлення в ресторані.*

У цьому прикладі ви є відправником, офіціант з блокнотом — командою, а кухар — отримувачем. Як і в самому патерні, ви не стикаєтесь з кухарем безпосередньо. Замість цього ви відправляєте замовлення офіціантом, який самостійно «налаштовує» кухаря на роботу. З іншого боку, кухар не знає, хто конкретно надіслав йому замовлення. Але йому це байдуже, бо вся необхідна інформація є в листі замовлення.

***Структура***

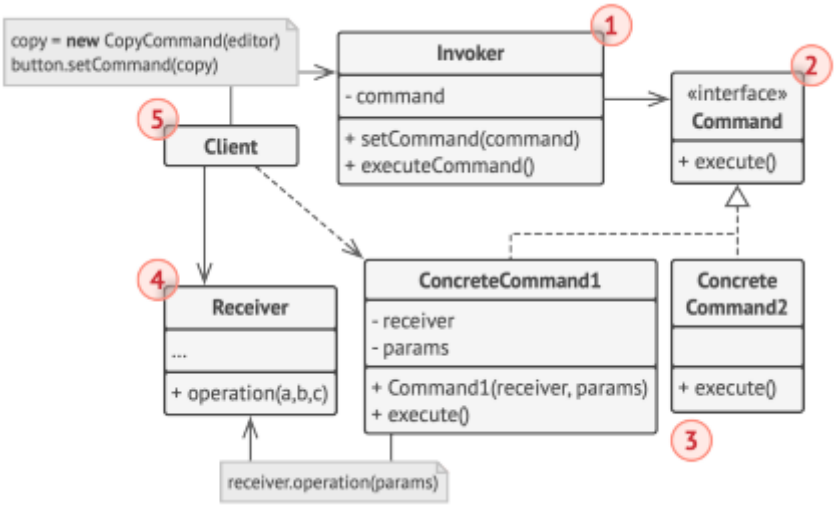


*Рис.2.18 Структура патерну Command*

1. Відправник зберігає посилання на об’єкт команди та звертається до нього, коли потрібно виконати якусь дію. Відправник працює з командами тільки через їхній загальний інтерфейс. Він не знає, яку конкретно команду використовує, оскільки отримує готовий об’єкт команди від клієнта.
2. Команда описує інтерфейс, спільний для всіх конкретних команд. Зазвичай тут описується лише один метод запуску команди.
3. Конкретні команди реалізують різні запити, дотримуючись загального інтерфейсу команд. Як правило, команда не робить всю роботу самостійно, а лише передає виклик одержувачу, яким виступає один з об’єктів бізнес-логіки. Параметри, з якими команда звертається до одержувача, необхідно зберігати у вигляді полів. У більшості випадків об’єкти команд можна зробити незмінними, передаючи у них всі необхідні параметри тільки через конструктор
4. Одержувач містить бізнес-логіку програми. У цій ролі може виступати практично будь-який об’єкт. Зазвичай, команди перенаправляють виклики одержувачам, але іноді, щоб спростити програму, ви можете позбутися від одержувачів, «зливши» їхній код у класи команд.
5. Клієнт створює об’єкти конкретних команд, передаючи до них усі необхідні параметри, серед яких можуть бути і посилання на об’єкти одержувачів. Після цього клієнт зв’язує об’єкти відправників зі створеними командами.

***Застосування***

*Якщо ви хочете параметризувати об'єкти виконуваною дією..*
Команда перетворює операції на об'єкти, а об'єкти, у свою чергу, можна передавати, зберігати та взаємозаміняти всередині інших об'єктів.
Скажімо, ви розробляєте бібліотеки графічного меню і хочете, щоб користувачі могли використовувати меню в різних програмах, не змінюючи кожного разу код ваших класів. Застосувавши патерн, користувачам не доведеться змінювати класи меню, замість цього вони будуть конфігурувати об'єкти меню різними командами.

*Якщо ви хочете поставити операції в чергу, виконувати їх за розкладом або передавати мережею.*
Як і будь-які інші об'єкти, команди можна серіалізувати, тобто перетворити на рядок, щоб потім зберегти у файл або базу даних. Потім в будь-який зручний момент його можна дістати назад, знову перетворити на об'єкт команди та виконати. Так само команди можна передавати мережею, логувати або виконувати на віддаленому сервері.

*Якщо вам потрібна операція скасування.*
Головна річ, яка потрібна для того, щоб мати можливість скасовувати операції — це зберігання історії. Серед багатьох способів реалізації цієї можливості патерн Команда є, мабуть, найпопулярнішим.
Історія команд виглядає як стек, до якого потрапляють усі виконані об'єкти команд. Кожна команда перед виконанням операції зберігає поточний стан об'єкта, з яким вона працюватиме. Після виконання операції копія команди потрапляє до стеку історії, продовжуючи нести у собі збережений стан об'єкта. Якщо знадобиться скасування, програма візьме останню команду з історії та відновить збережений у ній стан.
Цей спосіб має дві особливості. По-перше, точний стан об'єктів не дуже просто зберегти, адже його частина може бути приватною. Вирішити це можна за допомогою патерна Знімок.
По-друге, копії стану можуть займати досить багато оперативної пам'яті. Тому іноді можна вдатися до альтернативної реалізації, тобто замість відновлення старого стану, команда виконає зворотню дію. Недолік цього способу у складності (іноді неможливості) реалізації зворотньої дії.

*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. Створіть загальний інтерфейс команд і визначте в ньому метод запуску.
2. Один за одним створіть класи конкретних команд. У кожному класі має бути поле для зберігання посилання на один або декілька об'єктів-одержувачів, яким команда перенаправлятиме основну роботу. Крім цього, команда повинна мати поля для зберігання параметрів, потрібних під час виклику методів одержувача. Значення всіх цих полів команда повинна отримувати через конструктор. І, нарешті, реалізуйте основний метод команди, викликаючи в ньому ті чи інші методи одержувача.
3. Додайте до класів відправників поля для зберігання команд. Зазвичай об'єкти-відправники приймають готові об'єкти команд ззовні — через конструктор або через сетер поля команди
4. Змініть основний код відправників так, щоб вони делегували виконання дії команді.

5. Порядок ініціалізації об’єктів повинен виглядати так:
    - Створюємо об’єкти одержувачів.
    - Створюємо об’єкти команд, зв’язавши їх з одержувачами.
    - Створюємо об’єкти відправників, зв’язавши їх з командами.

***Переваги***

- Прибирає пряму залежність між об’єктами, що викликають операції, та об’єктами, які їх безпосередньо виконують.
- Дозволяє реалізувати просте скасування і повтор операцій.
- Дозволяє реалізувати відкладений запуск операцій.
- Дозволяє збирати складні команди з простих.
- Реалізує принцип відкритості/закритості.

***Недоліки***

Ускладнює код програми внаслідок введення великої кількості додаткових класів.

## Шаблон «CHAIN OF RESPONSIBILITY»

Ланцюжок обов’язків — це поведінковий патерн проектування, що дає змогу передавати запити послідовно ланцюжком обробників. Кожен наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком.

***Проблема***

Уявіть, що ви робите систему прийому онлайн-замовлень. Ви хочете обмежити до неї доступ так, щоб тільки авторизовані користувачі могли створювати замовлення. Крім того, певні користувачі, які володіють правами адміністратора, повинні мати повний доступ до замовлень.

Ви швидко збагнули, що ці перевірки потрібно виконувати послідовно. Адже користувача можна спробувати «залогувати» у систему, якщо його запит містить логін і пароль. Але, якщо така спроба не вдалась, то перевіряти розширені права доступу просто немає сенсу


*Рис.2.19 Запит проходить ряд перевірок перед доступом до системи замовлень..*

Протягом наступних кількох місяців вам довелося додати ще декілька таких послідовних перевірок.

- Хтось слушно зауважив, що непогано було б перевіряти дані, що передаються в запиті, перед тим, як вносити їх до системи — раптом запит містить дані про покупку неіснуючих продуктів.
- Хтось запропонував блокувати масові надсилання форми з одним і тим самим логіном, щоб запобігти підбору паролів ботами.
- Хтось зауважив, що непогано було б діставати форму замовлення з кешу, якщо вона вже була одного разу показана.

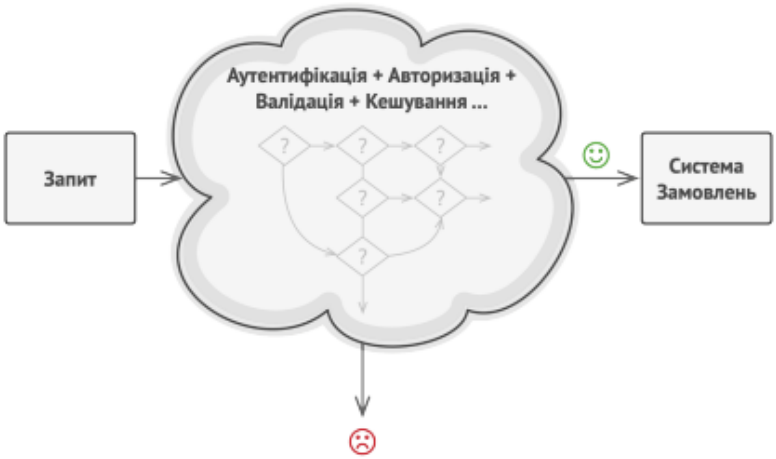


*Рис.2.20 З часом код перевірок стає все більш заплутаним.*

З кожною новою «фічою» код перевірок, що виглядав як величезний клубок умовних операторів, все більше і більше «розбухав». При зміні одного правила доводилося змінювати код усіх інших перевірок. А щоб застосувати перевірки до інших ресурсів, довелося також продублювати їхній код в інших класах.

Підтримувати такий код стало не тільки вкрай незручно, але й витратно. Аж ось одного прекрасного дня ви отримуєте завдання рефакторингу...

***Рішення***

Як і багато інших поведінкових патернів, ланцюжок обов'язків базується на тому, щоб перетворити окремі поведінки на об'єкти. У нашому випадку кожна перевірка переїде до окремого класу з одним методом виконання. Дані запиту, що перевіряється, передаватимуться до методу як аргументи.

А тепер справді важливий етап. Патерн пропонує зв'язати всі об'єкти обробників в один ланцюжок. Кожен обробник міститиме посилання на наступного обробника в ланцюзі. Таким чином, після отримання запиту обробник зможе не тільки опрацювати його самостійно, але й передати обробку наступному об'єкту в ланцюжку.

Передаючи запити до першого обробника ланцюжка, ви можете бути впевнені, що всі об'єкти в ланцюзі зможуть його обробити. При цьому довжина ланцюжка не має жодного значення.

І останній штрих. Обробник не обов'язково повинен передавати запит далі. Причому ця особливість може бути використана різними шляхами.

У прикладі з фільтрацією доступу обробники переривають подальші перевірки, якщо поточну перевірку не пройдено. Адже немає сенсу витрачати даремно ресурси, якщо і так зрозуміло, що із запитом щось не так.


*Рис.2.21 Обробники слідують в ланцюжку один за іншим*

Але є й інший підхід, коли обробники переривають ланцюг, тільки якщо вони можуть обробити запит. У цьому випадку запит рухається ланцюгом, поки не знайдеться обробник, який зможе його обробити. Дуже часто такий підхід використовується для передачі подій, що генеруються у класах графічного інтерфейсу внаслідок взаємодії з користувачем.

Наприклад, коли користувач клікає по кнопці, програма будує ланцюжок з об'єкта цієї кнопки, всіх її батьківських елементів і загального вікна програми на кінці. Подія кліку передається цим ланцюжком до тих пір, поки не знайдеться об'єкт, здатний її обробити. Цей приклад примітний ще й тим, що ланцюжок завжди можна виділити з деревоподібної структури об'єктів, в яку зазвичай і згорнуті елементи користувацького інтерфейсу.

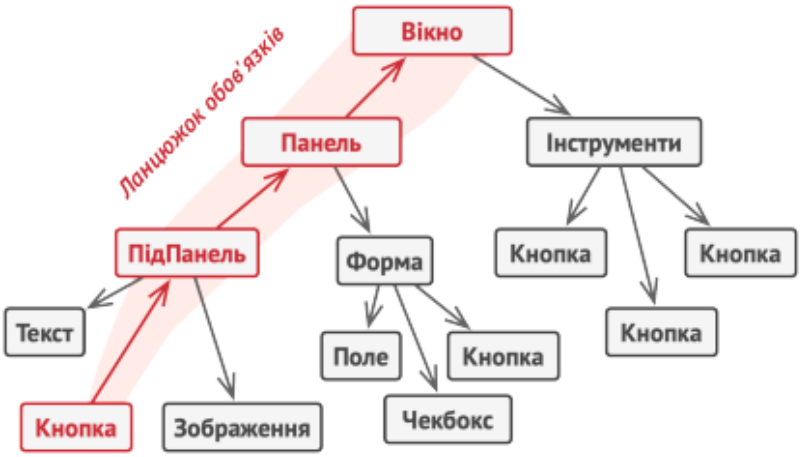


*Рис.2.22 Ланцюжок можна виділити навіть із дерева об'єктів*

Дуже важливо, щоб усі об'єкти ланцюжка мали спільний інтерфейс. Зазвичай кожному конкретному обробникові достатньо знати тільки те, що наступний об'єкт ланцюжка має метод виконати . Завдяки цьому зв'язки між об'єктами ланцюжка будуть більш гнучкими. Крім того, ви зможете формувати ланцюжки на льоту з різноманітних об'єктів, не прив'язуючись до конкретних класів.

*Аналогія з життя*

Ви купили нову відеокарту. Вона автоматично визначилася й почала працювати під Windows, але у вашій улюбленій Ubuntu «завести» її не вдалося. Ви телефонуєте до служби підтримки виробника, але без особливих сподівань на вирішення проблеми.

Спочатку ви чуєте голос автовідповідача, який пропонує вибір з десяти стандартних рішень. Жоден з варіантів не підходить, і робот з'єднує вас з живим оператором.

На жаль, звичайний оператор підтримки вміє спілкуватися тільки завченими фразами і давати тільки шаблонні відповіді. Після чергової пропозиції «вимкнути і ввімкнути комп'ютер» ви просите зв'язати вас зі справжніми інженерами.

Оператор перекидає дзвінок черговому інженерові, який знемагає від нудьги у своїй комірчині. От він вже точно знає, як вам допомогти! Інженер розповідає вам, де завантажити драйвери та як налаштувати їх під Ubuntu. Запит вирішено. Ви кладете слухавку

***Структура***

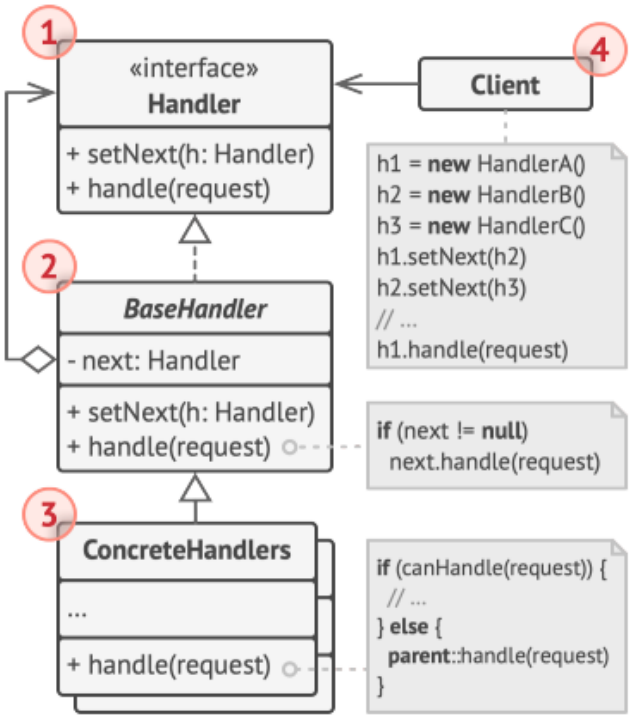


*Рис.2.23 Структура патерну Chain of responsibility*

1. Обробник визначає спільний для всіх конкретних обробників інтерфейс. Зазвичай достатньо описати один метод обробки запитів, але іноді тут може бути оголошений і метод встановлення наступного обробника.
2. Базовий обробник — опціональний клас, який дає змогу позбутися дублювання одного і того самого коду в усіх конкретних обробниках. Зазвичай цей клас має поле для зберігання посилання на наступного обробника у ланцюжку. Клієнт зв'язує обробників у ланцюг, подаючи посилання на наступного обробника через конструктор або сетер поля. Також в цьому класі можна реалізувати базовий метод обробки, який би просто перенаправляв запити наступному обробнику, перевіривши його наявність.
3. Конкретні обробники містять код обробки запитів. При отриманні запиту кожен обробник вирішує, чи може він обробити запит, а також чи варто передати його наступному об'єкту. У більшості випадків обробники можуть працювати самостійно і бути незмінними, отримавши всі необхідні деталі через параметри конструктора.
4. Клієнт може сформувати ланцюжок лише один раз і використовувати його протягом всього часу роботи програми, так і перебудовувати його динамічно, залежно від логіки програми. Клієнт може відправляти запити будь-якому об'єкту ланцюжка, не обов'язково першому з них.

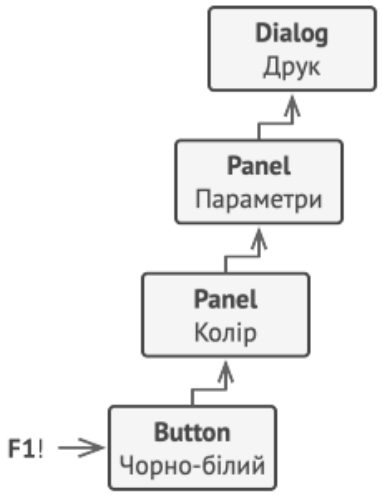


*Рис.2.24 Приклад виклику контекстної допомоги у ланцюжку об'єктів UI*

***Застосування***

*Якщо програма має обробляти різноманітні запити багатьма способами, але заздалегідь невідомо, які конкретно запити надходитимуть і які обробники для них знадобляться.*

За допомогою Ланцюжка обов'язків ви можете зв'язати потенційних обробників в один ланцюг і по отриманню запита по черзі питати кожного з них, чи не хоче він обробити даний запит.

*Якщо важливо, щоб обробники виконувалися один за іншим у суворому порядку.*

Ланцюжок обов'язків дозволяє запускати обробників один за одним у тій послідовності, в якій вони стоять в ланцюзі.

*Якщо набір об'єктів, здатних обробити запит, повинен задаватися динамічно.*

У будь-який момент ви можете втрутитися в існуючий ланцюжок і перевизначити зв'язки так, щоби прибрати або додати нову ланку

***Кроки реалізації***

1. Створіть інтерфейс обробника і опишіть в ньому основний метод обробки. Продумайте, в якому вигляді клієнт повинен передавати дані запиту до обробника. Найгнучкіший спосіб — це перетворити дані запиту на об'єкт і повністю передавати його через параметри методу обробника.
2. Є сенс у тому, щоб створити абстрактний базовий клас обробників, аби не дублювати реалізацію методу отримання наступного обробника в усіх конкретних обробниках. Додайте до базового обробника поле для збереження посилання на наступний елемент ланцюжка. Встановлюйте

початкове значення цього поля через конструктор. Це зробить об’єкти обробників незмінюваними. Але якщо програма передбачає динамічну перебудову ланцюжків, можете додати ще й сетер для поля. Реалізуйте базовий метод обробки так, щоб він перенаправляв запит наступному об’єкту, перевіривши його наявність. Це дозволить повністю приховати поле-посилання від підкласів, давши їм можливість передавати запити далі ланцюгом, звертаючись до батьківської реалізації методу.

3. Один за іншим створіть класи конкретних обробників та реалізуйте в них методи обробки запитів. При отриманні запиту кожен обробник повинен вирішити:
    - Чи може він обробити запит, чи ні?
    - Чи потрібно передавати запит наступному обробникові, чи ні?
4. Клієнт може збирати ланцюжок обробників самостійно, спираючись на свою бізнес-логіку, або отримувати вже готові ланцюжки ззовні. В останньому випадку ланцюжки збираються фабричними об’єктами, спираючись на конфігурацію програми або параметри оточення.
5. Клієнт може надсилати запити будь-якому обробникові ланцюга, а не лише першому. Запит передаватиметься ланцюжком, допоки який-небудь обробник не відмовиться передавати його далі або коли буде досягнуто кінець ланцюга.
6. Клієнт повинен знати про динамічну природу ланцюжка і бути готовим до таких випадків: Ланцюжок може складатися з одного об’єкта.
    - Запити можуть не досягати кінця ланцюга.
    - Запити можуть досягати кінця, залишаючись необробленими.

***Переваги***

- Зменшує залежність між клієнтом та обробниками.
- Реалізує принцип єдиного обов’язку.
- Реалізує принцип відкритості/закритості.

***Недоліки***

Запит може залишитися ніким не опрацьованим.

## Шаблон «PROTOTYPE»

Прототип — це породжувальний патерн проектування, що дає змогу копіювати об’єкти, не вдаючись у подробиці їхньої реалізації.

***Проблема***

У вас є об’єкт, який потрібно скопіювати. Як це зробити? Потрібно створити порожній об’єкт того самого класу, а потім по черзі копіювати значення всіх полів зі старого об’єкта до нового. Чудово! Проте є нюанс. Не кожен об’єкт вдасться скопіювати у такий спосіб, адже частина його стану може бути приватною, а значить — недоступною для решти коду програми.

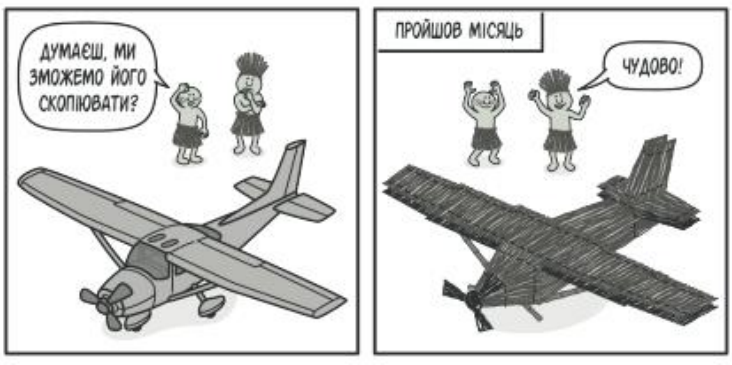


*Рис.2.25 Копіювання «ззовні» не завжди можливе на практиці.*

Є й інша проблема. Код, що копіює, стане залежним від класів об’єктів, які він копіює. Адже, щоб перебрати усі поля об’єкта, потрібно прив’язатися до його класу. Тому ви не зможете копіювати об’єкти, знаючи тільки їхні інтерфейси, але не їхні конкретні класи.

***Рішення***

Патерн Прототип доручає процес копіювання самим об’єктам, які треба скопіювати. Він вводить загальний інтерфейс для всіх об’єктів, що підтримують клонування. Це дозволяє копіювати об’єкти, не прив’язуючись до їхніх конкретних класів. Зазвичай такий інтерфейс має всього один метод — clone. Реалізація цього методу в різних класах дуже схожа. Метод створює новий об’єкт поточного класу й копіює в нього значення всіх полів власного об’єкта. Таким чином можна скопіювати навіть приватні поля, оскільки більшість мов програмування дозволяє отримати доступ до приватних полів будь-якого об’єкта поточного класу.

Об’єкт, який копіюють, називається прототипом (звідси і назва патерна). Коли об’єкти програми містять сотні полів і тисячі можливих конфігурацій, прототипи можуть слугувати своєрідною альтернативою створенню підкласів.

У цьому випадку всі можливі прототипи готуються і налаштовуються на етапі ініціалізації програми. Потім, коли програмі буде потрібний новий об’єкт, вона створить копію з попередньо заготовленого прототипа.


*Рис.2.26 Попередньо заготовлені прототипи можуть стати заміною підкласів.*

*Аналогія з життя*

У промисловому виробництві прототипи створюються перед виготовленням основної партії продуктів для проведення різноманітних випробувань. При цьому прототип не бере участі в подальшому виробництві, відіграючи пасивну роль.

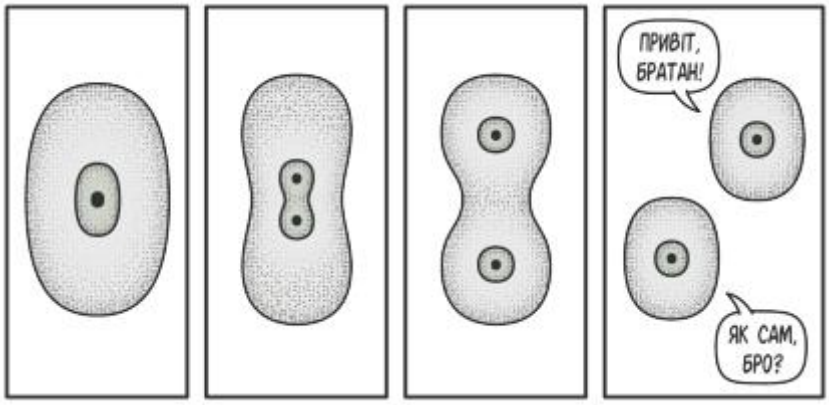


*Рис.2.27 Приклад поділу клітини*

Виробничий прототип не створює копію самого себе, тому більш наближений до патерна приклад — це поділ клітин. Після мітозного поділу клітин утворюються дві абсолютно ідентичні клітини. Материнська клітина відіграє роль прототипу, беручи активну участь у створенні нового об’єкта.

***Структура***

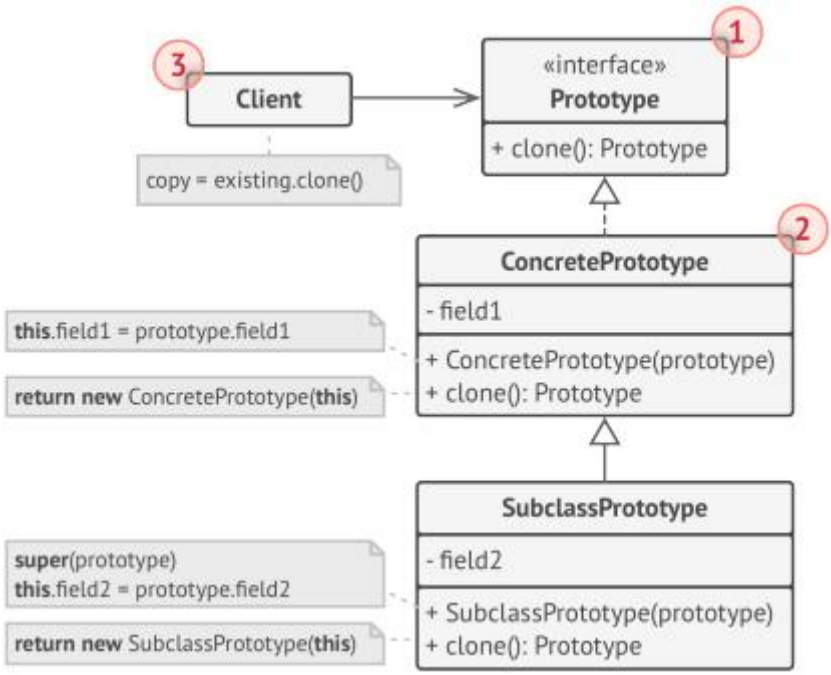


*Рис.2.28 Базова реалізація*

1. Інтерфейс прототипів описує операції клонування. Для більшості випадків — це єдиний метод clone
2. Конкретний прототип реалізує операцію клонування самого себе. Крім звичайного копіювання значень усіх полів, тут можуть бути приховані різноманітні складнощі, про які клієнту не потрібно знати. Наприклад, клонування пов’язаних об’єктів, розплутування рекурсивних залежностей та інше.
3. Клієнт створює копію об’єкта, звертаючись до нього через загальний інтерфейс прототипів.

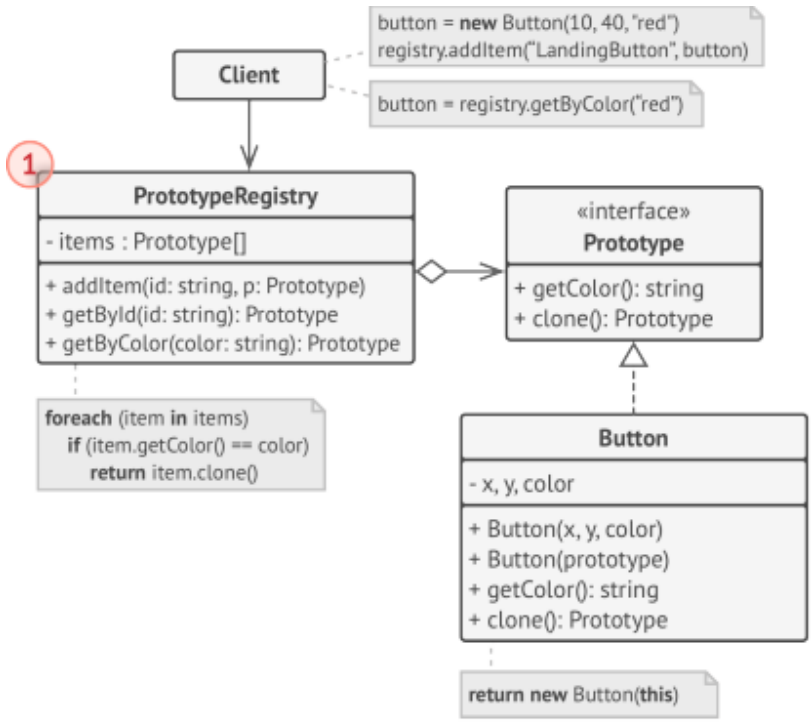


*Рис.2.28 Реалізація зі спільним сховищем прототипів*

Сховище прототипів полегшує доступ до часто використовуваних прототипів, зберігаючи попередньо створений набір еталонних, готових до копіювання об’єктів. Найпростіше сховище може бути побудовано за допомогою хеш-таблиці виду ім'я-прототипу → прототип . Для полегшення пошуку прототипи можна маркувати ще й за іншими критеріями, а не тільки за умовним іменем.

***Застосування***

*Коли ваш код не повинен залежати від класів об'єктів, призначених для копіювання.*
Таке часто буває, якщо ваш код працює з об'єктами, поданими ззовні через який-небудь загальний інтерфейс. Ви не зможете прив'язатися до їхніх класів, навіть якби захотіли, тому що конкретні класи об'єктів невідомі.
Патерн Прототип надає клієнту загальний інтерфейс для роботи з усіма прототипами. Клієнту не потрібно залежати від усіх класів об'єктів, призначених для копіювання, а тільки від інтерфейсу клонування.

*Коли ви маєте безліч підкласів, які відрізняються початковими значеннями полів. Хтось міг створити усі ці класи для того, щоб мати легкий спосіб породжувати об'єкти певної конфігурації.*
Патерн Прототип пропонує використовувати набір прототипів замість створення підкласів для опису популярних конфігурацій об'єктів. Таким чином, замість породження об'єктів з підкласів ви копіюватимете існуючі об'єкти-прототипи, внутрішній стан яких вже налаштовано. Це дозволить уникнути вибухоподібного зростання кількості класів програми й зменшити її складність

***Кроки реалізації***

1. Створіть інтерфейс прототипів з єдиним методом clone . Якщо у вас вже є ієрархія продуктів, метод клонування можна оголосити в кожному з її класів.
2. Додайте до класів майбутніх прототипів альтернативний конструктор, що приймає в якості аргументу об'єкт поточного класу. Спочатку цей конструктор повинен скопіювати значення всіх полів поданого об'єкта, оголошених в рамках поточного класу. Потім — передати виконання батьківському конструктору, щоб той потурбувався про поля, оголошені в суперкласі. Якщо мова програмування, яку ви використовуєте, не підтримує перевантаження методів, тоді вам не вдасться створити декілька версій конструктора. В цьому випадку копіювання значень можна проводити в іншому методі, спеціально створеному для цих цілей. Конструктор є зручнішим, тому що дозволяє клонувати об'єкт за один виклик.
3. Зазвичай метод клонування складається з одного рядка, а саме виклику оператора new з конструктором прототипу. Усі класи, що підтримують клонування, повинні явно визначити метод clone для того, щоб вказати власний клас з оператором new . Інакше результатом клонування стане об'єкт батьківського класу.
4. На додачу можете створити центральне сховище прототипів. У ньому зручно зберігати варіації об'єктів, можливо, навіть одного класу, але по-різному налаштованих. Ви можете розмістити це сховище або у новому фабричному класі, або у фабричному методі базового класу прототипів. Такий фабричний метод, керуючись вхідними аргументами, повинен шукати відповідний екземпляр у сховищі прототипів, а потім викликати його метод клонування і повертати отриманий об'єкт. Нарешті, потрібно позбутися прямих викликів конструкторів об'єктів, замінивши їх викликами фабричного методу сховища прототипів.

***Переваги***

- Дозволяє клонувати об'єкти без прив'язки до їхніх конкретних класів.
- Менша кількість повторювань коду ініціалізації об'єктів.
- Прискорює створення об'єктів.
- Альтернатива створенню підкласів під час конструювання складних об'єктів.

***Недоліки***

Складно клонувати складові об'єкти, що мають посилання на інші об'єкти

## 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:

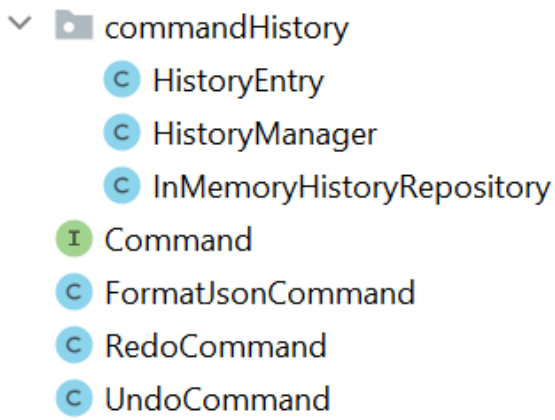


*Рис. 1. Ієрархія класів. Реалізація патерну Command.*

```
5 usages   3 implementations
public interface Command {
    3 usages   3 implementations
    String execute();
}
```

*Рис. 2. Інтерфейс Command для реалізації патерну Command.*

```java
@Override
public String execute() {
    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);
    }
    result = formatterContext.format(rawJsonDto);
    historyManager.addEntry(rawJsonDto.getSchemaData(), result, formatType);
    return result;
}
```

*Рис. 2.1.1 Метод execute класу FormatJsonCommand*

```java
@Override
public String execute() {
    historyManager.undoLastEntry();
    HistoryEntry currentEntry = historyManager.getCurrentEntry();
    return currentEntry != null ? currentEntry.getFormattedJson() : "{}";
}
```

*Рис. 2.1.2 Метод execute класу UndoCommand*

```java
@Override
public String execute() {
    historyManager.redoLastEntry();
    HistoryEntry currentEntry = historyManager.getCurrentEntry();
    return currentEntry != null ? currentEntry.getFormattedJson() : "{}";
}
```

*Рис. 2.1.3 Метод execute класу RedoCommand*

```
15 usages
public class HistoryEntry {
    1 usage
    private final String rawJson;
    2 usages
    private final String formattedJson;
    1 usage
    private final String formatType;

    1 usage
    public HistoryEntry(String rawJson, String formattedJson, String formatType) {
        this.rawJson = rawJson;
        this.formattedJson = formattedJson;
        this.formatType = formatType;
    }

    2 usages
    public String getFormattedJson() { return formattedJson; }

}
```

*Рис. 2.1.4 Клас HistoryEntry для роботи з історією змін*

```
public class HistoryManager {
    5 usages
    private final InMemoryHistoryRepository repository;

    no usages
    public HistoryManager(InMemoryHistoryRepository repository) { this.repository = repository; }

    1 usage
    public void addEntry(String rawJson, String formattedJson, String formatType) {
        repository.save(new HistoryEntry(rawJson, formattedJson, formatType));
    }

    1 usage
    public void undoLastEntry() { repository.removeLastEntry(); }

    1 usage
    public void redoLastEntry() { repository.redoLastEntry(); }

    2 usages
    public HistoryEntry getCurrentEntry() { return repository.getCurrentEntry(); }
}
```

*Рис. 2.1.5 Клас HistoryManager для роботи з історією змін*

```java
public class InMemoryHistoryRepository {
    6 usages
    private final Stack<HistoryEntry> undoStack = new Stack<>();
    4 usages
    private final Stack<HistoryEntry> redoStack = new Stack<>();

    1 usage
    public void save(HistoryEntry entry) {
        undoStack.push(entry);
        redoStack.clear();
    }
    1 usage
    public void removeLastEntry() {
        if (!undoStack.isEmpty()) {
            redoStack.push(undoStack.pop());
        }
    }

    1 usage
    public void redoLastEntry() {
        if (!redoStack.isEmpty()) {
            undoStack.push(redoStack.pop());
        }
    }
    1 usage
    public HistoryEntry getCurrentEntry() { return undoStack.isEmpty() ? null : undoStack.peek(); }
}
```

*Рис. 2.1.6 Клас InMemoryHistoryRepository для in-memory збереження змін та операцій над ними*

```java
@Override
public String formatJson(RawJsonDto rawJsonDto, String formatType) {
    try {
        FormatJsonCommand command = new FormatJsonCommand(rawJsonDto, formatType, formatterContext, historyManager);
        return command.execute();
    } catch (InvalidJsonException e) {
        System.err.println("Error: " + e.getMessage());
        return "{}";
    }
}

@Override
public String undoLastFormat() {
    try {
        UndoCommand command = new UndoCommand(historyManager);
        return command.execute();
    } catch (InvalidJsonException e) {
        System.err.println("Error: " + e.getMessage());
        return "{}";
    }
}

@Override
public String redoLastFormat() {
    try {
        RedoCommand command = new RedoCommand(historyManager);
        return command.execute();
    } catch (InvalidJsonException e) {
        System.err.println("Error: " + e.getMessage());
        return "{}";
    }
}
```

*Рис. 2.1.7 Виклики методів execute класів патерну Command у сервісі RawJsonServiceImplementation*

```java
@PostMapping("/format/{type}")
public ResponseEntity<String> formatJson(@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());
    }
}

no usages
@PostMapping("/undo")
public ResponseEntity<String> undoLastChange() {
    String json = rawJsonService.undoLastFormat();
    return ResponseEntity.ok(json);
}

no usages
@PostMapping("/redo")
public ResponseEntity<String> redoLastChange() {
    String json = rawJsonService.redoLastFormat();
    return ResponseEntity.ok(json);
}
```

*Рис. 2.1.8 Виклики методів сервісу RawJsonServiceImplementation у контролері*

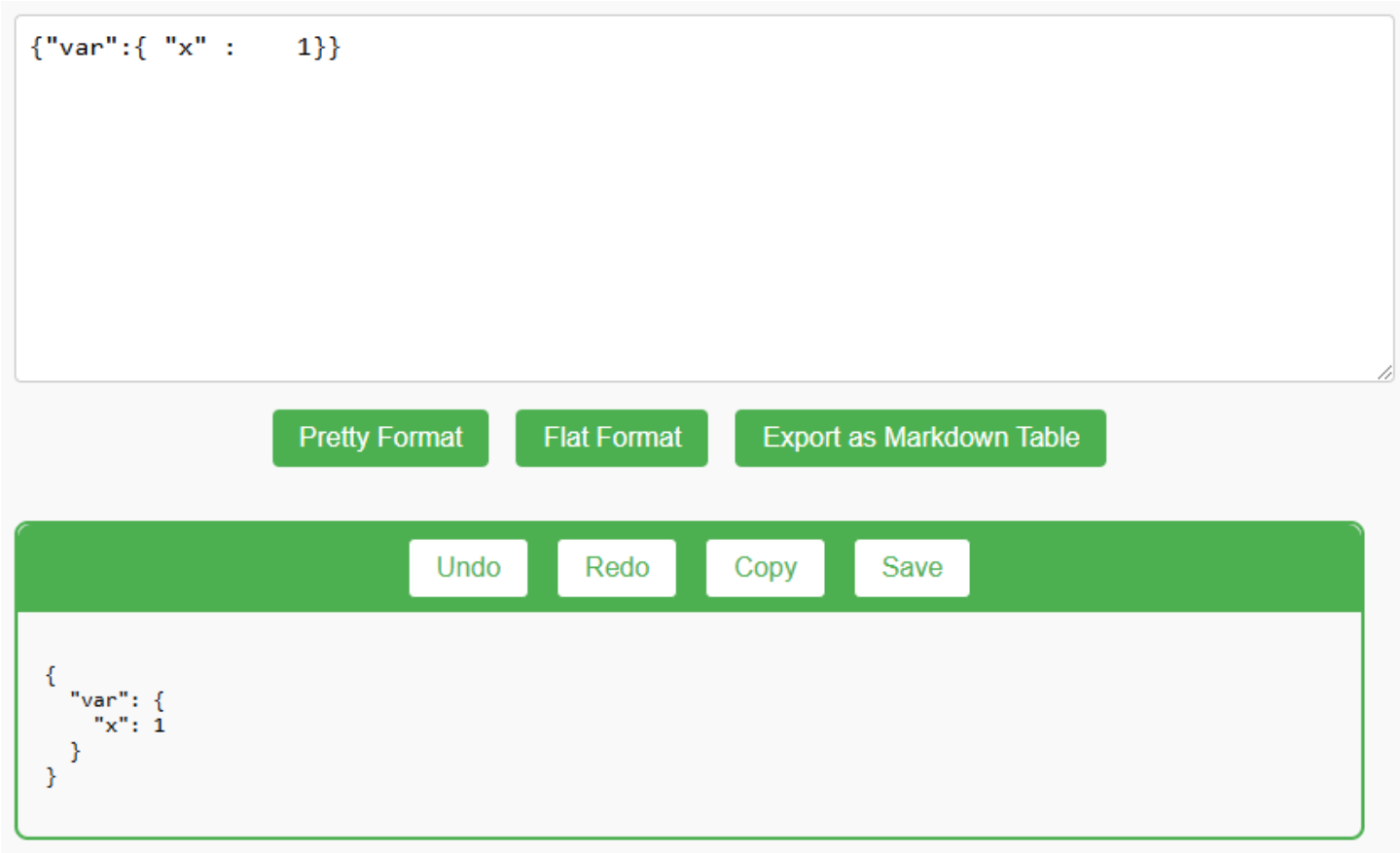


*Рис. 2.1.9 Результат роботи застосунку*

# ВИСНОВОК

Реалізація патерну Command у межах цього застосунку дозволила ефективно управляти історією змін відформатованих JSON-схем. Завдяки чіткому розподілу відповідальності між командами, система отримала можливість виконувати, відміняти та повторювати дії, забезпечуючи відповідність принципам SOLID. Такий підхід сприяв підвищенню зручності використання, спрощенню розширення функціональності та забезпечив гнучкість і надійність розробленої системи.