ЛАНЦЮЖОК ВІДПОВІДАЛЬНОСТІ

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.

ЛАНЦЮЖОК ВІДПОВІДАЛЬНОСТІ + КОМАНДА

Команди про які ми розмовляли раніше можуть бути як обробниками так і командою.

Тобто маємо дві взаємодії:

Один стан проходить через багато команд, або ж одна команда застосовується до багатьох станів.

УМОВА

Наприклад, Ви робите систему управління онлайн-замовленнями.

Хочете обмежити до неї доступ так, щоб тільки авторизовані користувачі могли створювати замовлення.

І тільки адміністратори повинні мати повний доступ до замовлень.

УМОВА

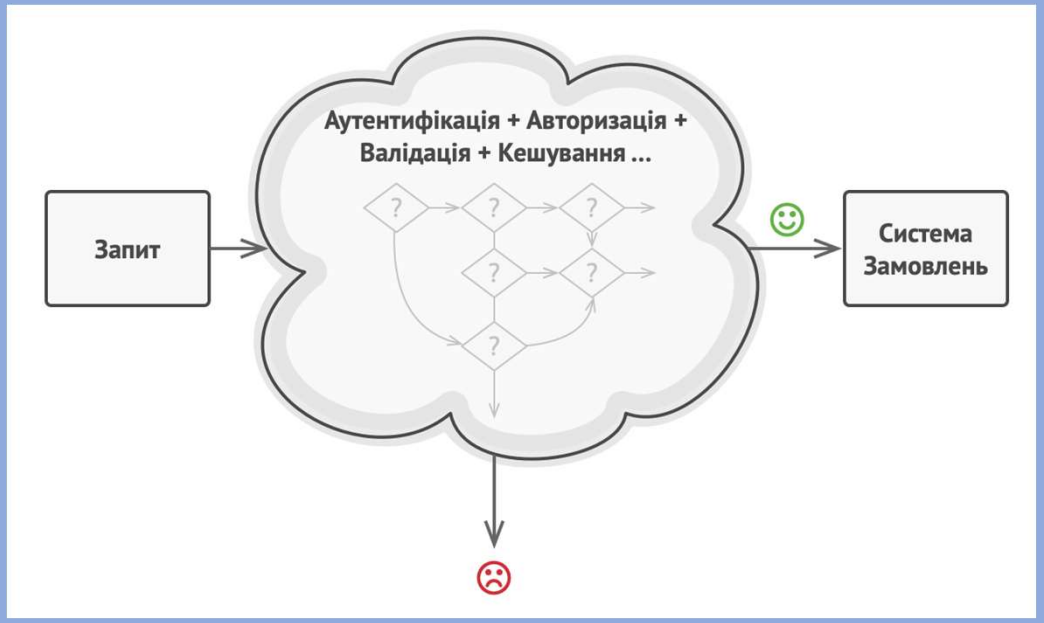
Протягом наступних кількох місяців Вам довелося додати ще декілька перевірок...

- Перевірка даних, що передаються в запиті, перед тим, як вносити їх до системи — раптом запит містить дані про покупку неіснуючих продуктів.
- Захист від DDoS атак – блокування масового надсилання форми з одним і тим самим логіном.
- Що користувач не намагається змінити замовлення після його відправки у опрацювання...

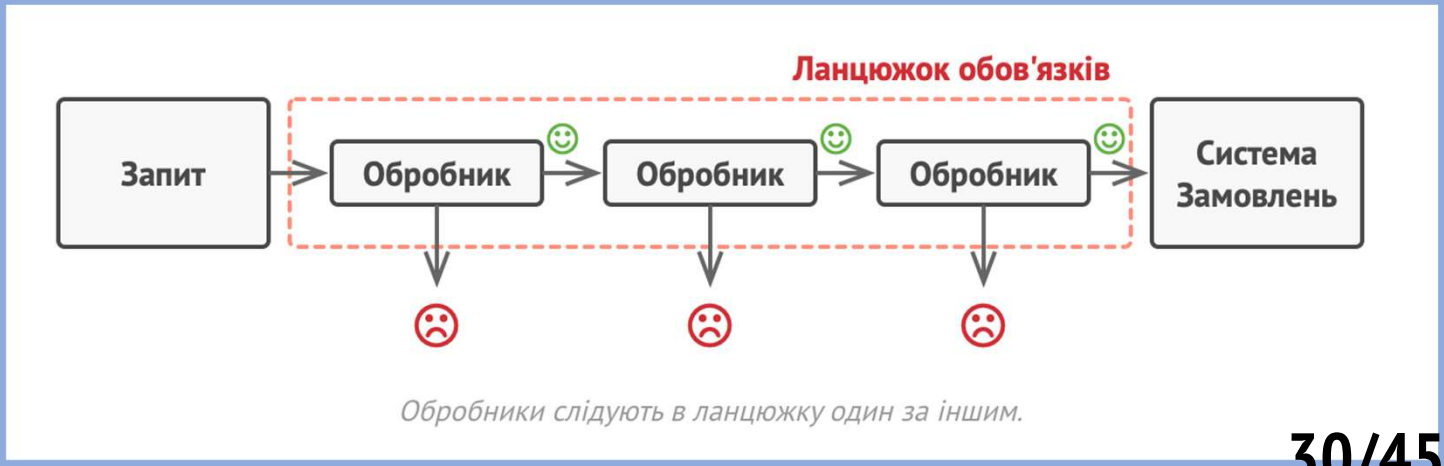
З кожною новою перевіркою, код, що виглядав як величезний клубок умовних операторів, все більше розрісся.

А щоб застосувати перевірки до інших даних, довелося також продублювати цей код в інших класах.

Будемо від цього...



Переходити до цього!



Обробники слідують в ланцюжку один за іншим.

АЛГОРИТМ РІШЕННЯ

- **Перетворюєʼмо операції на обʼєкти.**

У нашому випадку кожна перевірка стане окремим класом з одним методом виконання.

Дані запиту, що перевіряється, передаватимуться до методу як аргументи.

АЛГОРИТМ РІШЕННЯ

- **Зв’язати всі об’єкти обробників в один ланцюжок.**

Кожен обробник міститиме посилання на наступного обробника в ланцюзі.

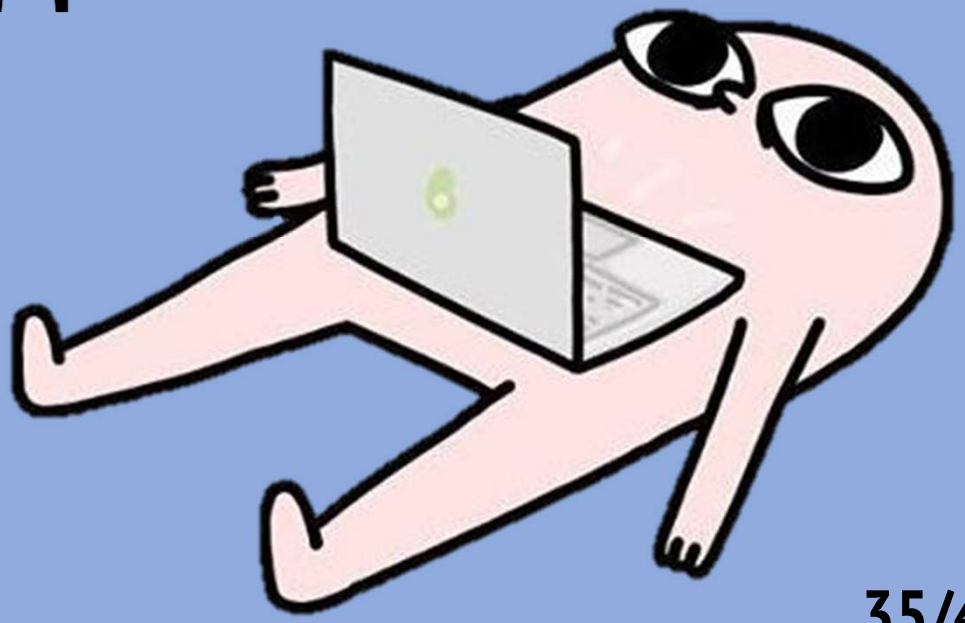
Таким чином, після отримання запиту обробник зможе не тільки опрацювати його самостійно, але й передати обробку наступному об’єкту в ланцюжку.

ВАЖЛИВО

Обробник не обов'язково повинен передавати запит далі.

У нашому прикладі 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.

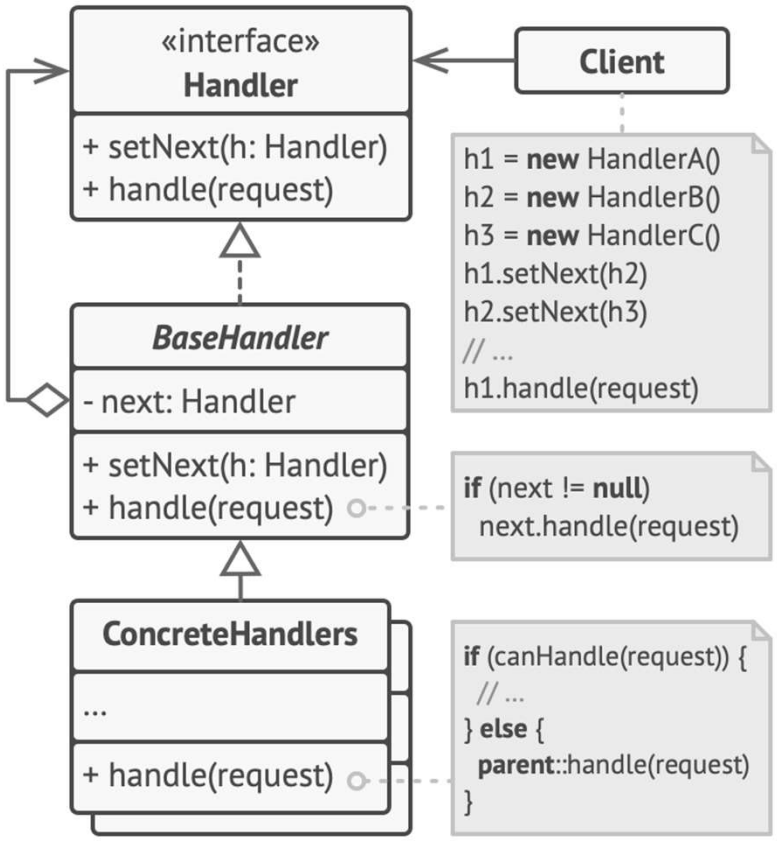
ПЕРЕЙДЕМО ДО КОДУ ЦЬОГО ПРИКЛАДУ



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.

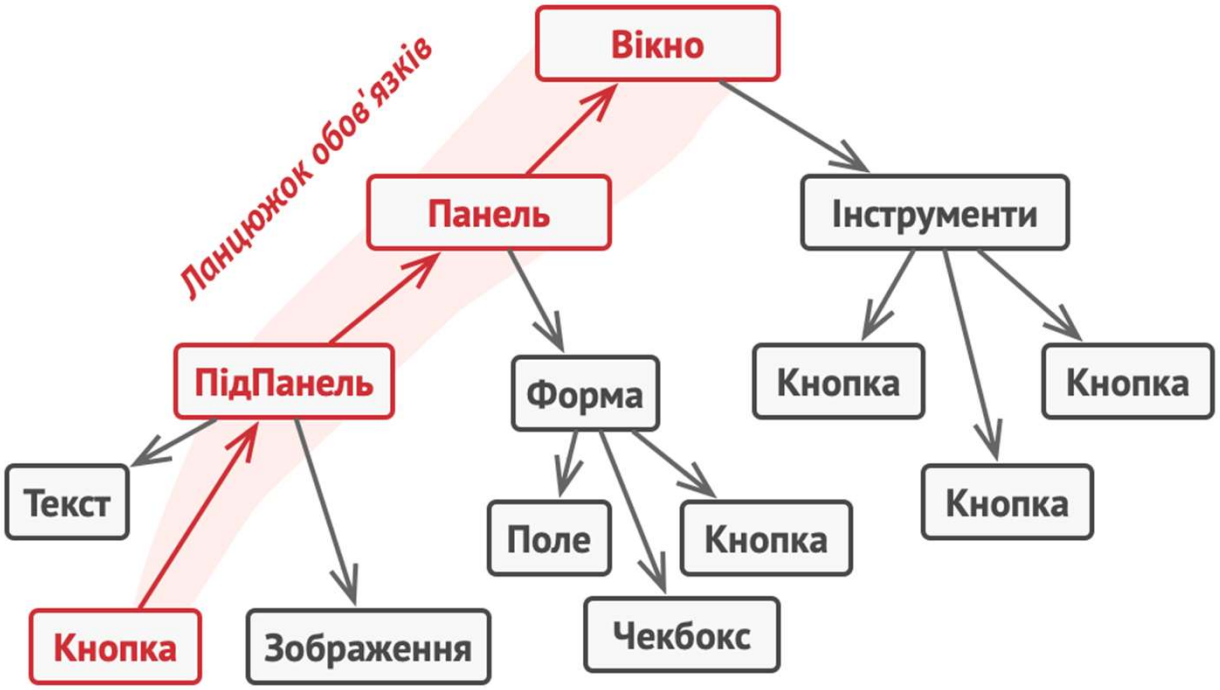
Зазвичай цей клас має поле для зберігання посилання на наступного обробника у ланцюжку. Клієнт зв'язує обробників у ланцюг, подаючи посилання на наступного обробника через конструктор або сетер поля. Також в цьому класі можна реалізувати базовий метод обробки, який би просто перенаправляв запити наступному обробнику, перевіривши його наявність.



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.

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.



Ланцюжок можна виділити навіть із дерева об'єктів.

Інший варіант ланцюга

Обробники переривають ланцюг, тільки якщо вони **можуть обробити запит**.

У цьому випадку запит рухається ланцюгом, поки не знайдеться обробник, який зможе його обробити.

ІНШІ ПРИКЛАДИ

- Звернення у технічну підтримку
- Нарахування відпустки новому співробітнику
- Підписання документів декількома особами

КОЛИ ПОТРІБЕН ЛАНЦЮЖОК

- Якщо програма має обробляти різноманітні запити багатьма способами, але заздалегідь невідомо, які конкретно запити надходитимуть і які обробники для них знадобляться.
- Якщо важливо, щоб обробники виконувалися один за іншим у суворому порядку.
- Якщо набір об'єктів, здатних обробити запит, повинен задаватися динамічно.

ПЛЮСИ ТА МІНУСИ

Зменшує залежність між клієнтом та обробниками.

Реалізує *принцип єдиного обов'язку*.

Реалізує *принцип відкритості/закритості*.

Запит може залишитися ніким не опрацьованим.

З ІНШИМИ ПАТТЕРНАМИ

Ланцюжок обов'язків та **Декоратор** мають дуже схожі структури.

Обидва патерни базуються на принципі рекурсивного виконання операції через серію пов'язаних об'єктів. Але є декілька важливих відмінностей.

Ланцюжок обов'язків часто використовують разом з **Компонувальником**. У цьому випадку запит передається від дочірніх компонентів до їхніх батьків.

ПОРІВНЯННЯ З ІНШИМИ

Ланцюжок обов'язків, **Команда** **Посередник** та **Спостерігач** показують різні способи роботи тих, хто надсилає запити, та тих, хто їх отримує:

- *Ланцюжок обов'язків* передає запит послідовно через ланцюжок потенційних отримувачів, очікуючи, що один з них обробить запит.
- *Команда* встановлює непрямий односторонній зв'язок від відправників до одержувачів.
- *Посередник* прибирає прямий зв'язок між відправниками та одержувачами, змушуючи їх спілкуватися опосередковано, через себе.
- *Спостерігач* передає запит одночасно всім зацікавленим одержувачам, але дозволяє їм динамічно підписуватися або відписуватися від таких повідомлень..

РІЗНИЦЯ МІЖ ЛАНЦЮЖКОМ ТА ІТЕРАЦІЄЮ ПО СПИСКУ КЛАСІВ

Якщо мова йде про список інтерфейсу обробника, то простіше додавати елементи посередині списку під час виконання програми, порівняно з ланцюгом.

Крім того, можна використовувати різні списки в залежності від запиту, тоді як у ланцюгу важливий порядок обробників і залежність один від одного (pre/post-processing).

ЛАНЦЮЖОК VS ДЕКОРАТОР

Для декоратора всі класи обробляють запит, тоді як для ланцюжка відповідальності запит обробляє точно один із класів у ланцюжку.

Ланюжок призначений для наявності кількох потенційних об’єктів, які можуть обробити запит, а об’єкт, який обробить запит, невідомий заздалегідь. Тоді як декоратор призначений для обгортання об’єкта та додавання деяких функцій під час виконання.