# Лекція 22
# Паттерни

Використання паттернів у розробці веб-застосунків є дуже корисним, оскільки допомагає створювати більш структурований, чистий та простий для розуміння код. Паттерни - це загально прийняті рішення для типових проблем, які зустрічаються при розробці програмного забезпечення, що дає змогу забезпечити більш високу якість програмного продукту та полегшити його розробку.

## 22.1 Класифікація паттернів

Існує класифікація паттернів проектування, яка зазвичай розподіляє їх на три групи: структурні паттерни, поведінкові паттерни та паттерни створення.

Структурні паттерни - це паттерни, які визначають, як об'єкти пов'язані між собою, щоб утворювати більш складні структури.

Поведінкові паттерни - це паттерни, які визначають, як об'єкти співпрацюють один з одним, щоб виконувати задачі та поводитись відповідно до змінних умов.

Паттерни створення - це паттерни, які визначають процес створення об'єктів та способи їх зміни під час роботи додатку.

До *структурних паттернів* проектування належать такі паттерни:

Адаптер (Adapter) - паттерн, який дозволяє об'єктам зі сумісними, але непов'язаними інтерфейсами співпрацювати.

https://refactoring.guru/uk/design-patterns/adapter

Легковаговик (Flyweight) — це структурний патерн проектування, що дає змогу вмістити більшу кількість об’єктів у відведеній оперативній пам’яті. Легковаговик заощаджує пам’ять, розподіляючи спільний стан об’єктів між собою, замість зберігання однакових даних у кожному об’єкті.

https://refactoring.guru/uk/design-patterns/flyweight

Міст (Bridge) - паттерн, який дозволяє розділити складну систему на дві окремі ієрархії класів, такі що зміни в одній з них не впливають на іншу.

https://refactoring.guru/uk/design-patterns/bridge

Компонувальник (Composite) - паттерн, який дозволяє об'єднувати об'єкти в деревоподібні структури для представлення ієрархії складних об'єктів.

https://refactoring.guru/uk/design-patterns/composite

Декоратор (Decorator) - паттерн, який дозволяє динамічно додавати нові функції до об'єкта, без зміни його коду.

https://refactoring.guru/uk/design-patterns/decorator

Фасад (Facade) - паттерн, який надає простий інтерфейс для складної системи класів.

https://refactoring.guru/uk/design-patterns/facade

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

https://refactoring.guru/uk/design-patterns/proxy

Ці паттерни допомагають розбити складну систему на більш прості компоненти, що полегшує її розробку та підтримку. Крім того, вони дозволяють змінювати структуру системи без зміни її основної функціональності.

До *поведінкових паттернів* проектування належать такі паттерни:

Ланцюжок відповідальності (Chain of Responsibility) - паттерн, який дозволяє передавати запити по ланцюжку обробників, кожен з яких може обробити запит або передати його далі по ланцюжку.

https://refactoring.guru/uk/design-patterns/chain-of-responsibility

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

https://refactoring.guru/uk/design-patterns/mediator

Команда (Command) - паттерн, який дозволяє інкапсулювати запит як об'єкт, тим самим дозволяючи зберігати та передавати його як аргумент, або зберігати його в черзі.

https://refactoring.guru/uk/design-patterns/command

Ітератор (Iterator) - паттерн, який дозволяє послідовно доступатися до елементів складних об'єктів без розкриття їх внутрішньої структури.

https://refactoring.guru/uk/design-patterns/iterator

Знімок (Memento) - це поведінковий патерн проектування, що дає змогу зберігати та відновлювати минулий стан об’єктів, не розкриваючи подробиць їхньої реалізації.

https://refactoring.guru/uk/design-patterns/memento

Спостерігач (Observer) - паттерн, який дозволяє відстежувати зміни стану об'єкту та автоматично повідомляти зацікавлених об'єктів про ці зміни.

https://refactoring.guru/uk/design-patterns/observer

Шаблонний метод (Template Method) - паттерн, який дозволяє визначити загальну структуру алгоритму, залишаючи деякі деталі реалізації для підкласів.

https://refactoring.guru/uk/design-patterns/template-method

Стан (State) - паттерн, який дозволяє об'єкту змінювати свій стан залежно від змінних умов, що забезпечує більш гнучку поведінку.

https://refactoring.guru/uk/design-patterns/state

Стратегія (Strategy) - паттерн, який дозволяє об'єкту вибирати поведінку з набору можливих варіантів в залежності від ситуації.

https://refactoring.guru/uk/design-patterns/strategy

Наведемо приклад застосування Strategy.

*Завдання*. Потрібно написати програму-навігатор для подорожуючих, яка повинна показувати зручну карту, що дозволяє з легкістю орієнтуватися в незнайомому місті.

Однією з найбільш очікуваних функцій - пошук та прокладання маршрутів: перебуваючи в невідомому йому місті, користувач повинен

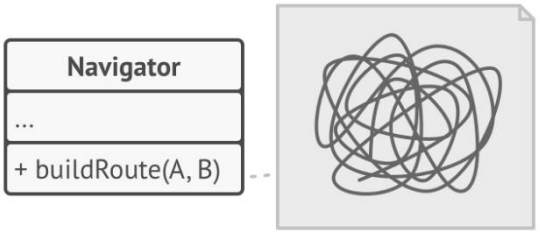
мати можливість вказати початкову точку та пункт призначення, а навігатор, в свою чергу, прокладе оптимальний шлях.

*Рішення.* Перша версія навігатора могла прокладати маршрут лише автомобільними шляхами, тому чудово підходила для подорожей автомобілем. Але, вочевидь, не всі їздять автомобілями, тому наступним кроком є додавання до навігатора можливості прокладання піших маршрутів.

Через деякий час з'ясувалося, що частина туристів під час пересування містом віддають перевагу громадському транспорту. Це вимагає додавання ще й такої опції як прокладання шляху.

У найближчій перспективі потрібно додати прокладку маршрутів велодоріжками, а у віддаленому майбутньому – маршрути, пов'язані з відвідуванням цікавих та визначних місць.

*Недоліки рішення*. З кожним новим алгоритмом код основного класу навігатора збільшувався вдвічі. В такому великому класі стало важкувато орієнтуватися.




Будь-яка зміна алгоритмів пошуку, чи то виправлення багів, чи додавання нового алгоритму, зачіпала основний клас. Це підвищувало ризик створення помилки шляхом випадкового внесення змін до робочого коду.

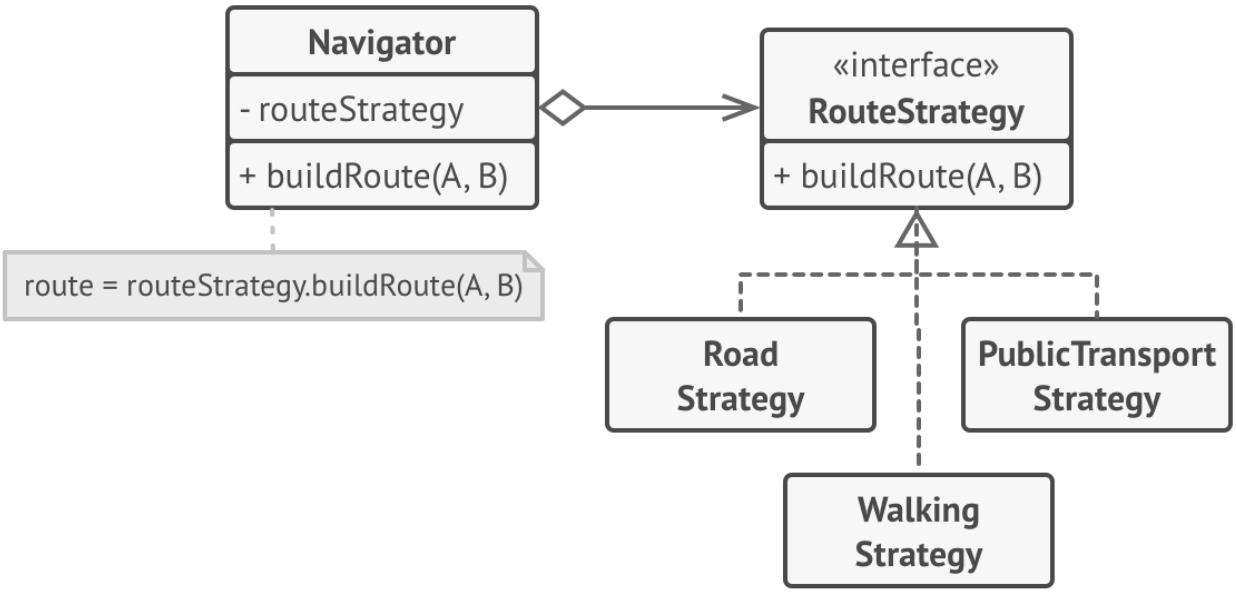
Крім того, ускладнювалася командна робота з іншими програмістами, яких ви найняли після успішного релізу навігатора. Ваші зміни нерідко торкалися одного і того самого коду, створюючи конфлікти, які вимагали додаткового часу на їхнє вирішення.

*Рішення за використання паттерну Strategy.* Патерн Стратегія пропонує визначити сімейство схожих алгоритмів, які часто змінюються або розширюються, та винести їх до власних класів, які називають стратегіями.

Замість того, щоб початковий клас сам виконував той чи інший алгоритм, він відіграватиме роль контексту, посилаючись на одну зі

стратегій та делегуючи їй виконання роботи. Щоб змінити алгоритм, буде достатньо підставити в контекст інший об’єкт-стратегію.

Важливо, щоб всі стратегії мали єдиний інтерфейс. Використовуючи цей інтерфейс, контекст буде незалежним від конкретних класів стратегій. З іншого боку, ви зможете змінювати та додавати нові види алгоритмів, не чіпаючи код контексту.

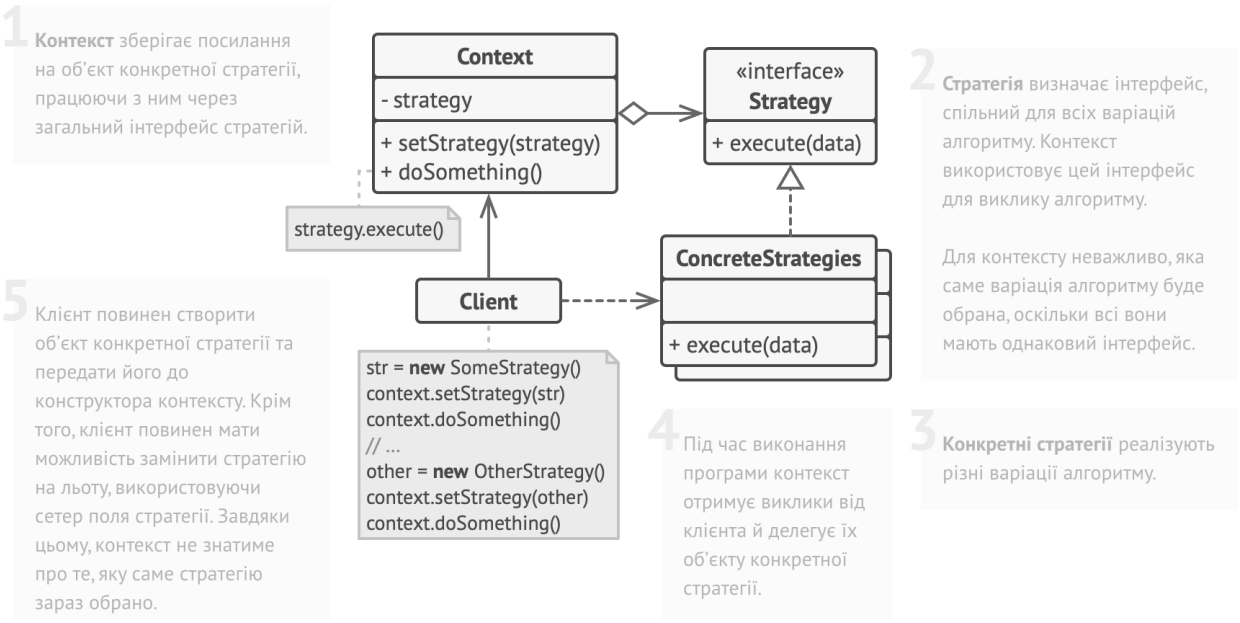



Для наведеного приклада кожен алгоритм пошуку шляху переїде до свого власного класу. В цих класах буде визначено лише один метод, що приймає в параметрах координати початку та кінця маршруту, а повертає масив всіх точок маршруту.

Хоча кожен клас прокладатиме маршрут на свій розсуд, для навігатора це не буде мати жодного значення, оскільки його робота полягає тільки у зображенні маршруту. Навігатору достатньо подати до стратегії дані про початок та кінець маршруту, щоб отримати масив точок маршруту в обумовленому форматі.

Клас навігатора буде мати метод для встановлення стратегії, що дозволить змінювати стратегію пошуку шляху «на льоту». Цей метод стане у нагоді клієнтському коду навігатора, наприклад, кнопкам-перемикачам типів маршрутів в інтерфейсі користувача.

## Структура




Поведінкові паттерни дозволяють забезпечити більшу гнучкість та простоту розробки, оскільки вони дозволяють об'єктам взаємодіяти зі своїм середовищем та поводитися відповідно до змінних умов

До *патернів створення* (Creational patterns) проектування належать такі паттерни:

Абстрактна фабрика (Abstract Factory) - паттерн, який дозволяє створювати фабрики, які виробляють сімейства пов'язаних об'єктів, не вказуючи їх конкретних класів.

https://refactoring.guru/uk/design-patterns/abstract-factory

Будівельник (Builder) - паттерн, який дозволяє розділити процес створення об'єкту на окремі етапи, що дозволяє створювати об'єкти з різними конфігураціями.

https://refactoring.guru/uk/design-patterns/builder

Фабричний метод (Factory Method) - паттерн, який дозволяє створювати об'єкти без прив'язки до їх конкретних класів.

https://refactoring.guru/uk/design-patterns/factory-method

Прототип (Prototype) - паттерн, який дозволяє створювати нові об'єкти шляхом копіювання вже існуючих об'єктів.

https://refactoring.guru/uk/design-patterns/prototype

Одинак (Singleton) - паттерн, який дозволяє створювати тільки один об'єкт конкретного класу та забезпечує глобальний доступ до нього.

https://refactoring.guru/uk/design-patterns/singleton

Ці паттерни дозволяють створювати об'єкти відповідно до специфічних вимог проекту та дозволяють забезпечити більш гнучкий та ефективний процес створення об'єктів. Вони також допомагають забезпечити кращу підтримку коду, полегшуючи процес додавання нових об'єктів та змін у існуючих об'єктах.

Крім того, паттерни можна також поділити на класові та об'єктні паттерни.

*Класові паттерни* описують способи використання наслідування для вирішення проблем проектування, тоді як *об'єктні паттерни* описують способи використання композиції об'єктів для досягнення тих самих цілей.

Наприклад, у веб-розробці часто використовують паттерни проектування, такі як MVC (Model-View-Controller) або MVP (Model-View-Presenter), для організації логіки додатка та його інтерфейсу. Ці паттерни дозволяють розділити додаток на логічні компоненти, які можна змінювати незалежно один від одного та спрощують тестування.

Також, існує велика кількість паттернів, які допомагають розробникам вирішувати специфічні задачі, такі як кешування даних, керування сесіями, автентифікація та авторизація користувачів, та інші.

Незважаючи на те, що використання паттернів є корисним для розробників, потрібно розуміти, що вони не є універсальними рішеннями для всіх проблем, тому потрібно знати їх переваги та недоліки, а також розуміти, який паттерн вибрати для конкретної задачі.

### 22.2 Паттерн Model-View-Controller

MVC (Model-View-Controller) – це шаблон проектування програмного забезпечення, який дозволяє розділити додаток на три

основних компоненти: модель (Model), представлення (View) та контролер (Controller). Кожен з цих компонентів виконує свої відповідні функції, що дозволяє забезпечити більшу структурованість додатка та полегшити його розробку.

Модель (Model) – це компонент, який представляє дані та логіку додатку. Він забезпечує доступ до даних та дозволяє виконувати операції над ними. Наприклад, модель може представляти базу даних, файлову систему або будь-який інший джерело даних, яке використовується у додатку. Модель також відповідає за збереження та оновлення даних, які відображаються у представленні.

Представлення (View) – це компонент, який відповідає за відображення даних користувачу та інтерфейс взаємодії з додатком. Представлення отримує дані з моделі та відображає їх користувачу у зручний для сприйняття спосіб. Наприклад, представлення може бути веб-сторінкою, яка відображає список товарів або форму для заповнення даних користувача.

Контролер (Controller) – це компонент, який відповідає за обробку запитів користувача та взаємодію між моделлю та представленням. Контролер отримує запит від користувача та передає його моделі для обробки. Після того, як модель виконує запит, контролер передає результати до представлення для відображення.

Основний принцип MVC – розділення логіки додатку на три основні компоненти: модель (Model), представлення (View) та контролер (Controller) дозволяє забезпечити більшу структурованість додатка та полегшити його розробку.

Також, основний принцип MVC дозволяє забезпечити більшу переносимість коду, оскільки окремі компоненти можуть бути використані у різних проектах та додатках. Це дозволяє забезпечити більшу ефективність та продуктивність при розробці додатків.

Використання шаблону проектування MVC (Model-View-Controller) має декілька переваг, ось деякі з них:

1 Розділення логіки додатку: одна з основних переваг використання MVC полягає в тому, що він дозволяє розділити логіку додатку на три компоненти: модель, представлення та контролер. Це

полегшує розробку та тестування додатку, оскільки кожен компонент виконує свої відповідні функції.

2 Більша переносимість коду: компоненти MVC можуть бути використані у різних проектах та додатках. Це дозволяє забезпечити більшу ефективність та продуктивність при розробці додатків.

3 Легша та швидша розробка: розділення додатку на компоненти дозволяє більш ефективно працювати з окремими частинами додатку. Це полегшує розробку та забезпечує швидший процес розробки.

4 Більш просте тестування: розділення додатку на компоненти дозволяє здійснювати більш точне та ефективне тестування кожної частини додатку окремо.

5 Більша гнучкість: використання MVC дозволяє змінювати логіку додатку без впливу на інші компоненти. Наприклад, можна змінювати логіку додатку без впливу на представлення.

6 Більша безпека: розділення додатку на компоненти дозволяє забезпечити більшу безпеку додатку, оскільки можна забезпечити контроль доступу до кожного компонента.

Окрім перелічених переваг, використання шаблону проектування MVC також дозволяє забезпечити більшу структурованість та чистоту коду, що полегшує його розуміння та редагування.

Незважаючи на те, що використання шаблону проектування MVC має багато переваг, воно також має деякі недоліки:

1 Складність: використання MVC може бути складним для розуміння та використання для початківців, оскільки розробка додатку на основі MVC вимагає досить високого рівня знань та досвіду у розробці програмного забезпечення.

2 Додатковий код: використання шаблону проектування MVC може привести до створення додаткового коду, що може збільшити розмір додатку та ускладнити його розробку.

3 Не всі додатки підходять для MVC: не всі додатки підходять для розробки на основі шаблону проектування MVC. Деякі додатки можуть бути занадто прості, щоб використовувати такий шаблон проектування.

4 Недостатня функціональність: у деяких випадках шаблон проектування MVC може не надавати достатньої функціональності для

розробки додатку. У цьому випадку можуть знадобитися додаткові компоненти та шаблони проектування.

5 Велика кількість файлів: використання шаблону проектування MVC може призвести до створення великої кількості файлів, що може зробити розробку та управління додатком більш складним.

6 Вимоги до архітектури: використання шаблону проектування MVC вимагає правильної архітектури додатку, що може бути складним у випадку великих проектів або проектів зі складною логікою.

Наведемо приклад використання шаблону проектування MVC. Розглянемо веб-додаток для управління списком завдань (to-do list). У цьому додатку модель (Model) буде відповідати за збереження та оновлення списку завдань, представлення (View) буде відповідати за відображення списку завдань користувачу та інтерфейс взаємодії з додатком, а контролер (Controller) буде відповідати за обробку запитів користувача та взаємодію між моделлю та представленням.

Користувач може додавати, видаляти або редагувати завдання у списку. Якщо користувач хоче додати нове завдання:

1) контролер отримує запит від представлення та додає нове завдання до моделі;
2) модель зберігає це завдання та оновлює список завдань;
3) контролер передає оновлений список завдань до представлення;
4) представлення відображає список користувачеві.

Таким чином, розділення додатку на модель, представлення та контролер дозволяє забезпечити більшу структурованість та чіткість додатку, що полегшує його розробку та тестування. Крім того, використання шаблону проектування MVC дозволяє змінювати логіку додатку без впливу на інші компоненти, що забезпечує більшу гнучкість та ефективність при розробці додатків.

## 22.3 Паттерн Model-View-Presenter

Model-View-Presenter (MVP) є шаблоном проектування, який використовується для розробки програмного забезпечення з графічним інтерфейсом користувача. MVP є вдосконаленням шаблону проектування Model-View-Controller (MVC) та відрізняється від нього тим, що презентер виконує функції, які раніше виконував контролер.

В MVP модель (Model) відповідає за обробку даних та логіку додатку, представлення (View) відповідає за відображення даних та інтерфейс взаємодії з користувачем, а презентер (Presenter) відповідає за обробку запитів користувача та взаємодію між моделлю та представленням.

Презентер є посередником між моделлю та представленням. Він отримує запити від представлення, обробляє їх та взаємодіє з моделлю, яка забезпечує необхідні дані. Після цього презентер передає дані до представлення, яке відображає їх користувачеві.

Однією з основних переваг MVP є те, що він дозволяє забезпечити більшу тестовуваність додатку, оскільки логіка додатку розділена на компоненти, що можуть бути тестовані окремо. Крім того, використання MVP дозволяє забезпечити більшу гнучкість та переносимість коду, оскільки окремі компоненти можуть бути використані у різних проектах та додатках.

Однак, використання MVP може бути складним для розуміння та використання для початківців. Також, MVP може привести до створення додаткового коду, що може збільшити розмір додатку та ускладнити його розробку.

Основні переваги використання Model-View-Presenter (MVP) включають:

1 Розділення логіки додатку: Розділення логіки додатку на модель, представлення та презентер дозволяє забезпечити більшу структурованість та чистоту коду. Це полегшує його розуміння та редагування.

2 Тестування: MVP забезпечує більшу тестуваність додатку, оскільки окремі компоненти можуть бути тестовані окремо. Це полегшує

виявлення та усунення помилок та допомагає зберегти час при розробці додатку.

3 Гнучкість: Використання MVP дозволяє змінювати логіку додатку без впливу на інші компоненти. Це забезпечує більшу гнучкість та ефективність при розробці додатків.

4 Переносимість коду: Компоненти MVP можуть бути використані у різних проектах та додатках. Це дозволяє забезпечити більшу ефективність та продуктивність при розробці додатків.

5 Легкість розуміння коду: Розділення логіки додатку на модель, представлення та презентер дозволяє забезпечити більшу читабельність та легкість розуміння коду, що полегшує його редагування та підтримку.

6 Розділення обов'язків: MVP дозволяє розділити обов'язки між різними командами розробників, що сприяє більш ефективному процесу розробки додатку.

Окрім перелічених переваг, використання MVP також дозволяє забезпечити більшу безпеку додатку, оскільки можна забезпечити контроль доступу до кожного компонента. Також, MVP забезпечує більшу стабільність додатку та більш ефективний процес його розроб

Існують деякі недоліки використання шаблону проектування MVP, включаючи:

1 Складність: Використання MVP може бути складним для розуміння та використання для початківців. Розробка додатку на основі MVP вимагає досить високого рівня знань та досвіду у розробці програмного забезпечення.

2 Додатковий код: Використання MVP може привести до створення додаткового коду, що може збільшити розмір додатку та ускладнити його розробку.

3 Не всі додатки підходять для MVP: Не всі додатки підходять для розробки на основі шаблону проектування MVP. Деякі додатки можуть бути занадто прості, щоб використовувати такий шаблон проектування.

4 Вимоги до архітектури: Використання шаблону проектування MVP вимагає правильної архітектури додатку, що може бути складним у випадку великих проектів або проектів зі складною логікою.

5 Велика кількість файлів: Використання шаблону проектування MVP може призвести до створення великої кількості файлів, що може зробити розробку та управління додатком більш складним.

6 Необхідність великої кількості інтерфейсів: Використання MVP вимагає великої кількості інтерфейсів, що може призвести до збільшення кількості коду та збільшення часу розробки додатку.

7 Нестабільність: Використання MVP може призвести до неконтрольованої зміни стану додатку, що може призвести до непередбачуваних наслідків та зробити додаток нестабільним.

Розглянемо приклад використання шаблону проектування MVP для розробки додатку для відстеження фітнес-цілей. У цьому додатку модель (Model) буде відповідати за збереження та оновлення даних про цілі користувача, представлення (View) буде відповідати за відображення цілей користувача та інтерфейс взаємодії з додатком, а презентер (Presenter) буде відповідати за обробку запитів користувача та взаємодію між моделлю та представленням.

Наприклад, користувач може встановити фітнес-цілі, такі як кількість кроків, витрачений час на заняття спортом, кількість спожитих калорій тощо. Якщо користувач хоче встановити нову ціль:

1) презентер отримує запит від представлення та додає нову ціль до моделі;
2) модель зберігає цю ціль та оновлює список цілей;
3) презентер передає оновлений список цілей до представлення;
4) представлення відображає список цілей користувачеві.

Також, презентер може отримувати запити користувача для відображення прогресу досягнення цілей та передавати дані з моделі до представлення, щоб відображати прогрес користувачеві.

Шаблон проектування MVP (Model-View-Presenter) та шаблон проектування MVC (Model-View-Controller) мають схожу структуру та використовуються для розділення логіки додатку на компоненти. Однак, є деякі *відмінності* між ними:

1 Роль презентера та контролера: У шаблоні MVP презентер відповідає за обробку запитів користувача та взаємодію з моделлю, тоді як у шаблоні MVC контролер відповідає за обробку запитів користувача та взаємодію з моделлю.

2 Передача даних між компонентами: У шаблоні MVP передача даних між компонентами відбувається за допомогою інтерфейсів, тоді як у шаблоні MVC передача даних відбувається безпосередньо між компонентами.

3 Залежності між компонентами: У шаблоні MVP модель та представлення не мають залежностей одне від одного, тоді як у шаблоні MVC модель може мати залежності від контролера.

4 Тестування: У шаблоні MVP тестування може бути більш простим, оскільки презентер може бути легко підмінений для тестування, тоді як у шаблоні MVC контролер може мати залежності від візуального інтерфейсу, що може ускладнити тестування.

5 Взаємодія з візуальним інтерфейсом: У шаблоні MVP презентер відповідає за взаємодію з візуальним інтерфейсом, тоді як у шаблоні MVC контролер може мати пряму залежність від візуального інтерфейсу.

Загалом шаблон проектування MVP може забезпечити більшу гнучкість, простоту та тестуваність додатку, а шаблон проектування MVC може забезпечити більшу швидкість розробки в тому випадку, коли розробка вимагає більшої уваги до візуальномого інтерфейсу застосунку. Крім того, якщо додаток має просту логіку та невелику кількість взаємодій між компонентами, то розробка на базі шаблону MVC може бути менш складною, оскільки не потребує великої кількості інтерфейсів та інших деталей реалізації шаблону MVP.

У випадку складних додатків з багатьма взаємодіями між компонентами та складною логікою, може бути кращим варіантом використання шаблону проектування MVP для забезпечення більшої гнучкості та тестованості додатку.

Корисне посилання

https://refactoring.guru/uk/design-patterns/history
https://refactoring.guru/uk/design-patterns/what-is-pattern

### 22.4 Переваги паттернів

Розробка ПЗ без застосування паттернів цілком можлива, проте паттерни надають наступні переваги:

1 Перевірені рішення. Ви витрачаєте менше часу, використовуючи готові рішення, замість повторного винаходу велосипеда. До деяких рішень ви могли б дійти й самотужки, але багато які з них стануть для вас відкриттям.

2 Стандартизація коду. Ви робите менше прорахунків при проектуванні, використовуючи типові уніфіковані рішення, оскільки всі приховані в них проблеми вже давно знайдено.

3 Загальний словник програмістів. Ви вимовляєте назву патерна, замість того, щоб годину пояснювати іншим програмістам, який крутий дизайн ви придумали і які класи для цього потрібні.