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

Технології розроблення програмного забезпечення

Лабораторна робота №7
«ШАБЛОНИ «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD».»

Виконала:
Студентка групи ІА-22
Микитенко Ірина

Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: шаблони «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD».

Мета: ознайомитися з шаблонами проектування «MEDIATOR», «FACADE», «BRIDGE», «TEMPLATE METHOD» та набути практичних навичок їх застосування. Реалізувати частину функціоналу програми за допомогою одного з розглянутих шаблонів для досягнення конкретних функціональних можливостей та забезпечення ефективної взаємодії між класами.

Хід роботи

..1 Музичний програвач (iterator, command, memento, facade, visitor, client-server)

Музичний програвач становить собою програму для програвання музичних файлів або відтворення потокової музики з можливістю створення, запам'ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіо-форматів, еквалайзер.

1. Короткі теоритичні відомості

Принципи проектування:

1. Принцип Don't Repeat Yourself (DRY) свідчить, що в початкових кодах програмного продукту не повинно бути повторень. Ці повторення можуть спостерігатися на різних рівнях - від повторень наборів рядків коду (абсолютно аналогічні рядки в різних місцях проекту) до різної реалізації функціональності, що повторюється. Повторень уникати слід з кількох причин: передусім, код без повторень значно менше і значно легше читається; по-друге, при знаходженні помилки в коді, що має повторення, є високий шанс не знати/забути виправити цю ж помилку у іншому місці; по-третє, при створенні ідентичного коду ви не лише копіюєте функціональність, але і помилки; по-четверте, при необхідності додавання нових можливостей скрізь де є повторення необхідно внести відповідні коректури. Повторення легко виправити застосувавши відповідну техніку рефакторинга - винесення метода/интерфейса/класса та ін.
2. Принцип Keep it simple, Stupid! (KISS) пропонує уникати зайвого ускладнення компонентів системи. Вважається, що система, яка складається з безлічі маленьких простих частин, працює значно надійніше, ніж одна велика складна система. Даною філософією слідує співтовариство Unix, де кожна програма виконує рівно одну функцію, але при цьому виконує її абсолютно добре (Do one thing right). Це також сприяє задоволенню відомого афоризму «The system obviously has no errors rather

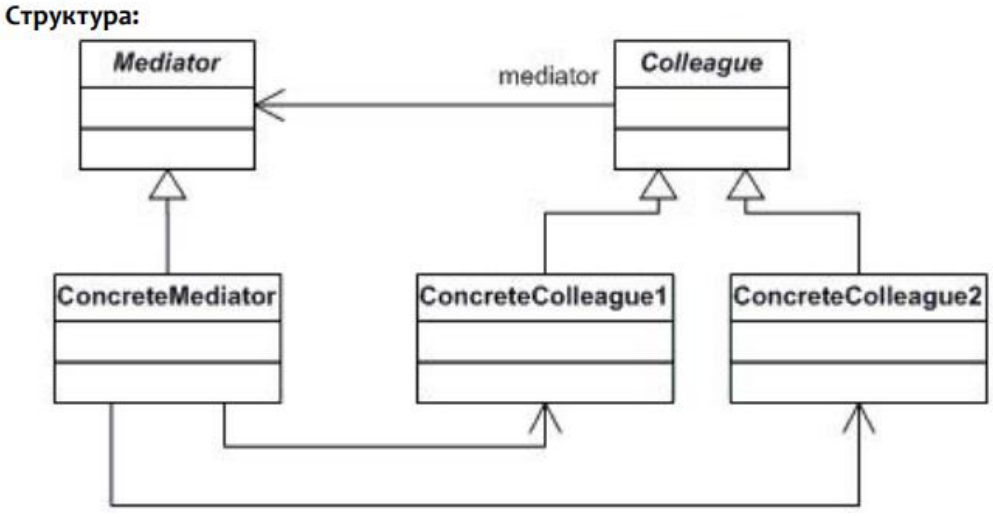
than system has no obvious errors» (очевидно, що в системі немає помилок; в системі немає очевидних помилок»). Використання простих конструкцій спрощує читаємість і сприйняття вихідних кодів.

3. Принцип You only load it once! (YOLO) вказує на необхідність підвантаження ініціалізаційних і конфігураційних змінних один раз при запуску програми, щоб уникнути проблем зі швидкістю (повторні зчитування даних з вінчестера).

4. Принцип Парето застосовується до безлічі різних подій і визначає відсоткове співвідношення 80-20. Декілька прикладів застосування принципу Парето: - 80% навантаження на сервер створює 20% додатка; - 80% всього вихідного коду програми пишеться за 20% часу; - 80% помилок програмного забезпечення можна усунути закривши лише 20% багів. Таким чином, необхідно пам'ятати про те, що для вирішення більшості проблем необхідно докласти лише малу частину зусиль. Однак для доведення додатку до стану досконалості може коштувати величезну суму (80%).

5. Принцип You ain’t gonna need it пропонує відмовитися від деяких рішень в силу того, що вони просто не знадобляться. Досить відомий факт, що програмісти люблять узагальнювати та надавати універсальні рішення на всі випадки життя. Дуже часто досить простого і працюючого рішення замість універсального, оскільки інші випадки навряд чи будуть розглядатися в подальшому. Є необхідний набір функціональності, який необхідно реалізувати; додавання зайвої гнучкості не тільки захарастить код, але і зменшить хід процесу розробки.

Шаблон «Mediator» (Посередник):



Призначення:

Шаблон «Mediator» забезпечує взаємодію між об'єктами через окремий об'єкт-

посередник, замість прямого зв'язку між компонентами. Це дозволяє уникнути складних залежностей і робить код більш гнучким і повторно використовуваним.

Проблема:
При взаємодії елементів діалогу, таких як текстові поля, кнопки та чекбокси, складна логіка може ускладнити повторне використання компонентів у різних контекстах.

Рішення:
Посередник координує взаємодії між елементами, знижуючи їх залежність один від одного і дозволяючи використовувати компоненти в різних контекстах.

Приклад з життя:
Пілоти літаків не спілкуються безпосередньо між собою, а через диспетчера, який координує їх взаємодію.

Переваги:

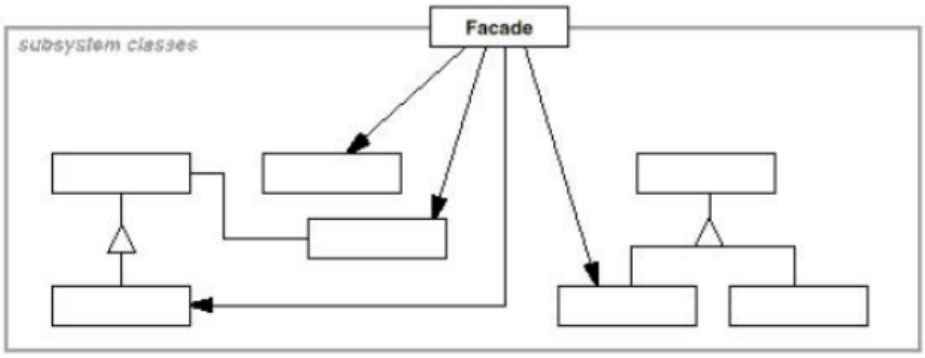
- Знижує залежності між компонентами.
- Спрощує взаємодію та централізує управління.

Недоліки:

- Посередник може стати занадто великим і складним.

Шаблон «Facade» (Фасад):

Структура:



Призначення:
Шаблон «Facade» створює єдиний, уніфікований спосіб доступу до складної підсистеми, приховуючи її внутрішні деталі. Це дозволяє спростити взаємодію з підсистемою, надаючи користувачу простий інтерфейс, при цьому приховуючи складність реалізації.

Проблема:
При роботі з великою кількістю об'єктів складної бібліотеки або фреймворка

бізнес-логіка може переплітатися з деталями реалізації сторонніх класів, що ускладнює підтримку коду.

Рішення:
Фасад надає простий інтерфейс для взаємодії з підсистемою, зберігаючи тільки необхідну функціональність для клієнта, приховуючи інші складності.

Приклад з життя:
Якщо ви телефонуєте в магазин для оформлення замовлення, співробітник служби підтримки є фасадом, який спрощує доступ до різних служб (замовлення, оплата, доставка).

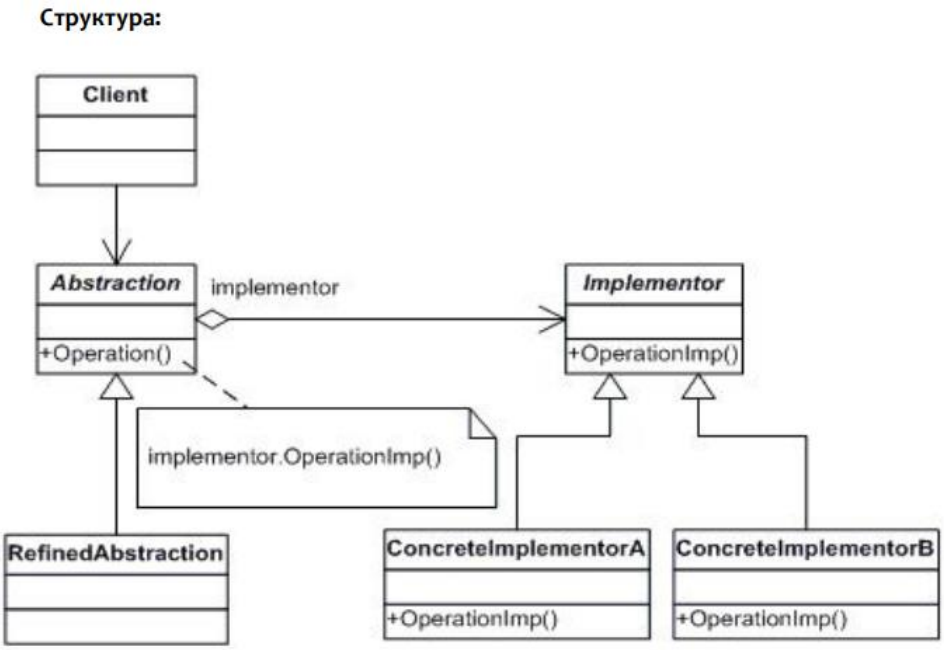
Переваги:

- Ізолює клієнта від складної підсистеми.
- Спрощує використання складних бібліотек або фреймворків.

Недоліки:

- Фасад може стати занадто великим об'єктом, що взаємодіє з усіма компонентами програми.

Шаблон «Bridge» (Міст):



Призначення:
Шаблон «Bridge» використовується для поділу абстракції та її реалізації. Це особливо корисно, коли існують різні абстракції, і їх можна реалізувати різними способами. Він дозволяє відокремити абстракцію від її реалізації та додавати нові абстракції незалежно від реалізацій.

Проблема:
Якщо ви намагаєтесь комбінувати кілька властивостей (наприклад, колір і форма), для кожної нової комбінації доведеться створювати нові класи, що призводить до різкого збільшення кількості класів та ускладнення підтримки. Наприклад, у разі додавання нових фігур або кольорів потрібно створювати нові підкласи для кожної можливості, що швидко стає неефективним.

Рішення:
Шаблон «Bridge» вирішує цю проблему, замінюючи спадкування агрегацією або композицією. Винятковим чином, можна створити окрему ієрархію для кожної «площини» (наприклад, колір), а потім використовувати ці об'єкти в класі абстракції (наприклад, фігури). Таким чином, при додаванні нових кольорів не потрібно змінювати класи фігур і навпаки.

Приклад з життя:
Уявіть два міста, розділені річкою. Для з'єднання між ними будують міст. Кожне місто розвивається за своїми законами, але міст залишається сталим, і міста підлаштовуються під міст. Міст є «мостом» між містами, дозволяючи їм взаємодіяти, не змінюючи їхню власну сутність.

Переваги:

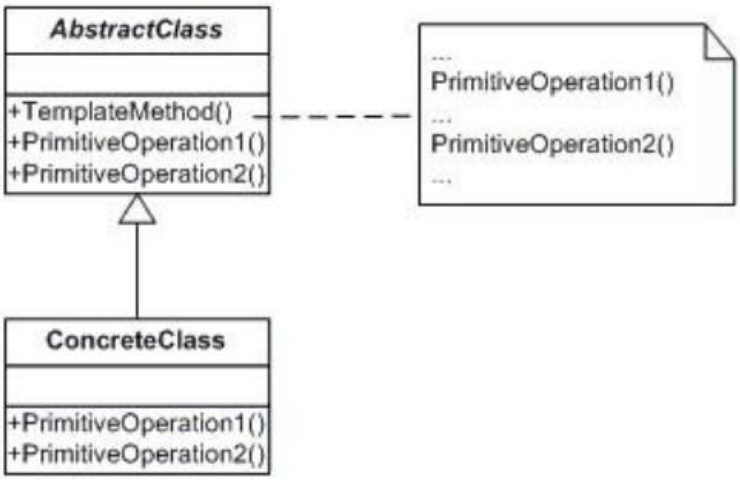
- Дозволяє створювати платформо-незалежні програми.
- Приховує зайві або небезпечні деталі реалізації від клієнтського коду.
- Реалізує принцип відкритості/закритості (легкість додавання нових функціональностей без зміни існуючого коду).

Недоліки:

- Ускладнює код через додавання нових класів і шарів абстракції.

Шаблон «Template Method»

Структура:



Призначення патерну:
Шаблон «Template Method» (шаблонний метод) дозволяє реалізувати покроковий алгоритм в абстрактному класі, але залишити специфіку реалізації для підкласів. Він дозволяє визначити загальний шаблон для алгоритму, а конкретні кроки цього алгоритму залишати на розсуд підкласів. Це зручно, коли потрібно забезпечити однакову структуру алгоритмів, але з можливістю змінювати конкретні кроки в залежності від потреб.

Приклад використання: Уявіть собі систему, яка формує веб-сторінки. Алгоритм формування сторінки включає кілька кроків:

- Формування заголовків,
- Формування контенту,
- Формування файлів, що додаються,
- Формування нижньої частини сторінки.

Ці кроки будуть однаковими для різних типів сторінок (наприклад, HTML, PHP, ASP.NET), але специфіка реалізації деяких кроків (наприклад, додавання вмісту) може варіюватися в залежності від типу сторінки.

Ось приклад на мові C # використання даного шаблону:

```
public class PageFormer
{
     void FormHeaders() { … }
     void FormFooters() { … }
     void FormAddedFiles() { … }
     abstract void FormContent();
     void FormPage()
     {
          FormHeaders();
          FormContent();
          FormFooters();
          FormAddedFiles();
}
}

public class AspNetCompiler : PageFormer
{
     override void FormContent() { … }
}
```

Проблема:
При розробці програми для дата-майнінгу документів, де потрібно обробляти різні формати (наприклад, PDF, DOC, CSV), спочатку обробляються лише DOC-файли. Пізніше додаються інші формати, і з'являється багато повторень в коді для кожного типу файлів, оскільки процес вилучення даних є схожим у всіх випадках.

Рішення:
Шаблон «Template Method» дозволяє описати загальний алгоритм обробки документів, при цьому залишити деталі реалізації (наприклад, як саме зчитувати DOC, CSV або PDF) для підкласів. Таким чином, спільний код з вилучення даних можна реалізувати в базовому класі, а конкретні кроки, які відрізняються для кожного типу файлів, залишити для перевизначення в підкласах.

Приклад з життя:
Будівельники можуть використовувати шаблонний метод при будівництві типових будинків. Зазначено, що будівництво включає стандартні етапи (фундамент, стіни, дах), але на кожному етапі можливі невеликі варіації, наприклад, типи матеріалів або техніка виконання. Саме ці варіації підклас будівельників може реалізувати, при цьому дотримуючись загального плану.

Переваги та недоліки:

Переваги:

- Полегшує повторне використання коду, оскільки спільні частини алгоритму реалізуються в базовому класі.
- Спрощує підтримку, оскільки можна змінювати конкретні кроки, не торкаючись загальної структури алгоритму.

Недоліки:

- Жорстко обмежує можливість зміни структури алгоритму через базовий клас.
- Може порушити принцип підстановки Барбари Лісков, якщо підклас змінює поведінку одного з кроків таким чином, що алгоритм більше не працює коректно.
- При великій кількості кроків шаблонний метод може стати занадто складним для підтримки.

Цей патерн дуже корисний, коли є необхідність забезпечити повторне використання загальних кроків алгоритму, при цьому дозволяючи підкласам змінювати або доповнювати лише частини алгоритму.

2. Реалізація не менше 3-х класів відповідно до обраної теми.

Структура проекту з реалізованими класами зображена на рисунку 1.

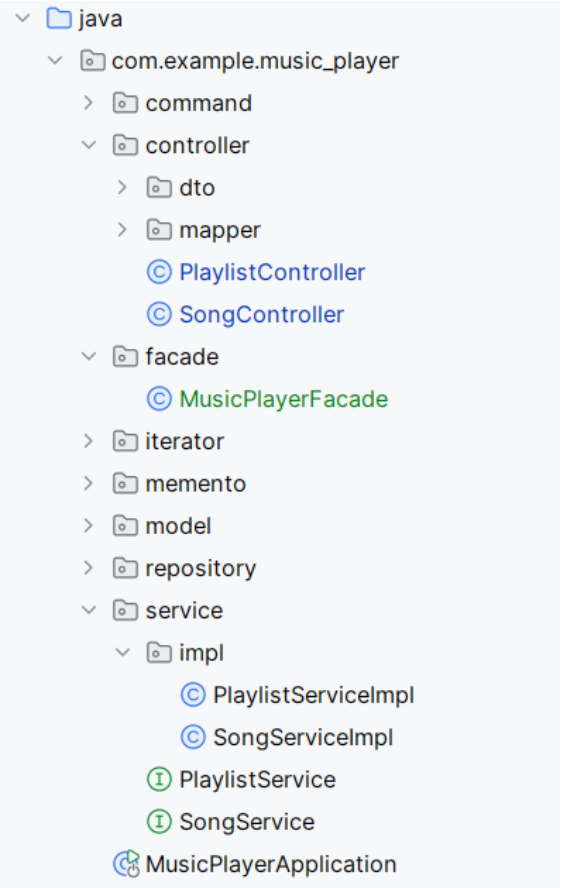


Рисунок 1 – структура проекту

У ході лабораторної роботи була реалізована система музичного плеєра, де основним завданням було використання патерна **Façade** для збереження та відновлення стану плейлистів. В результаті виконано такі кроки:

1. Сервіси

У системі є два основних сервіси, які виконують ключові операції:

- **SongService** — відповідає за роботу з піснями. Це сервіс, який надає можливості додавати пісні, знаходити їх за ідентифікатором, отримувати всі пісні з бази даних, а також видаляти пісні.
- **PlaylistService** — цей сервіс управляє плейлистами. Відповідає за створення нових плейлистів, оновлення вже існуючих, видалення плейлистів, а також додавання чи видалення пісень з конкретних плейлистів. Крім того, цей сервіс дозволяє отримувати всі пісні з плейлиста або зберігати/відновлювати стан плейлистів.

2. MusicPlayerFacade

MusicPlayerFacade — це фасад, який об'єднує роботу всіх сервісів в єдиний інтерфейс. Фасад дозволяє контролерам взаємодіяти з усіма сервісами через один об'єкт, приховуючи внутрішні деталі реалізації. Це означає, що контролери не повинні знати про конкретну реалізацію кожного сервісу, а лише

викликають методи фасаду для виконання операцій з піснями, плейлистами та програвачем.

Фасад надає такі можливості:

- Додавати, знаходити та видаляти пісні.
- Створювати, оновлювати, видаляти плейлисти та управляти піснями в плейлистах.
- Керувати відтворенням пісень (відтворення, пауза, зупинка).

3. Контролери

Контролери виступають як посередники між користувачем та системою. Вони приймають HTTP-запити від користувачів, взаємодіють з фасадом, щоб виконати необхідні операції, і повертають відповідні результати у вигляді HTTP-відповідей.

- **SongController** — цей контролер обробляє запити, які стосуються пісень. Він дозволяє додавати нові пісні, отримувати пісні за їхнім ідентифікатором або всі пісні в системі, а також видаляти пісні з бази.
- **PlaylistController** — цей контролер управляє запитами, які стосуються плейлистів. Він дозволяє створювати нові плейлисти, оновлювати їх, видаляти, а також додавати/видаляти пісні з плейлистів.

3. Реалізація шаблону за обраною темою

У ході виконання лабораторної роботи ми застосували патерн **Фасад** для спрощення взаємодії між контролерами, сервісами та обробкою бізнес-логіки в музичному плеєрі. Патерн **Фасад** дозволяє приховати складність внутрішніх операцій та надає єдиний інтерфейс для доступу до різних сервісів системи, спрощуючи роботу з ними.

Кроки реалізації патерну:

MusicPlayerFacade (Фасад):

- Клас **MusicPlayerFacade** є основною реалізацією патерну Фасад у нашому проекті. Його мета — об'єднати доступ до різних сервісів (таких як **PlaylistService** і **SongService**) через єдиний інтерфейс, щоб контролери могли взаємодіяти з ними без необхідності дізнаватися про деталі реалізації кожного окремого сервісу.
- **Атрибути:**
    - **PlaylistService playlistService**: служить для виконання операцій з плейлистами.

- **SongService songService**: служить для виконання операцій з піснями.

- **Методи:**
  - **addSong(Song song)**: додає нову пісню в систему.
  - **findSongById(Long id)**: знаходить пісню за її ідентифікатором.
  - **createPlaylist(Playlist playlist)**: створює новий плейлист.
  - **getAllPlaylists()**: отримує всі плейлисти.
  - **addSongToPlaylist(Long playlistId, Long songId)**: додає пісню в певний плейлист.
  - **removeSongFromPlaylist(Long playlistId, Long songId)**: видаляє пісню з певного плейлиста.
  - **savePlaylistState(Long playlistId)**: зберігає стан плейлиста (метод використовується у зв'язку з патерном Memento).
  - **restorePlaylistState(Long playlistId, int mementoIndex)**: відновлює попередній стан плейлиста.

- Даний клас представлено на рисунку 2 та рисунку 3.

```
@Component  4 usages  new *
public class MusicPlayerFacade {
    private final PlaylistService playlistService;  11 usages
    private final SongService songService;  5 usages

    public MusicPlayerFacade(PlaylistService playlistService, SongService songService) {  new *
        this.playlistService = playlistService;
        this.songService = songService;
    }

    // Методи для роботи з піснями
    public Song addSong(Song song) {  1 usage  new *
        return songService.add(song);
    }

    public Optional<Song> findSongById(Long id) {  1 usage  new *
        return songService.findById(id);
    }

    public List<Song> getAllSongs() {  1 usage  new *
        return songService.findAll();
    }

    public void deleteSongById(Long id) {  1 usage  new *
        songService.deleteById(id);
    }
```

Рисунок 2 – методи для роботи з піснями

```
    // Методи для роботи з плейлистами
    public Playlist createPlaylist(Playlist playlist) {  1 usage  new *
        return playlistService.createPlaylist(playlist);
    }

    public Playlist getPlaylistById(Long id) {  1 usage  new *
        return playlistService.getPlaylistById(id);
    }

    public List<Playlist> getAllPlaylists() {  1 usage  new *
        return playlistService.getAllPlaylists();
    }

    public Playlist updatePlaylist(Long id, Playlist playlist) {  1 usage  new *
        return playlistService.updatePlaylist(id, playlist);
    }

    public void deletePlaylist(Long id) {  1 usage  new *
        playlistService.deletePlaylist(id);
    }

    public void addSongToPlaylist(Long playlistId, Long songId) {  1 usage  new *
        playlistService.addSongToPlaylist(playlistId, songId);
    }

    public void removeSongFromPlaylist(Long playlistId, Long songId) {  1 usage  new *
        playlistService.removeSongFromPlaylist(playlistId, songId);
    }

    public List<Song> getSongsInPlaylist(Long playlistId) {  1 usage  new *
        return playlistService.getSongsInPlaylist(playlistId);
    }

    public void savePlaylistState(Long playlistId) {  1 usage  new *
        playlistService.savePlaylistState(playlistId);
    }

    public void restorePlaylistState(Long playlistId, int mementoIndex) {  1 usage  new *
        playlistService.restorePlaylistState(playlistId, mementoIndex);
    }
}
```

Рисунок 3 – методи для роботи з плейлистами

4. Діаграма класів для паттерну Facade

Діаграма класів, які реалізовують паттерн Memento зображена на рисунку 4

Проблема, яку допоміг вирішити шаблон Facade

У нашій системі музичного плеєра виникла проблема складності взаємодії між численними компонентами та сервісами. Кожен сервіс (наприклад, **PlaylistService**, **SongService**) мав безліч методів для виконання операцій над піснями та плейлистами, що ускладнювало роботу контролерів. Контролери повинні були звертатися до кожного з цих сервісів безпосередньо, що збільшувало складність коду, погіршувало його підтримуваність і робило його важким для масштабування.

Патерн Facade допоміг вирішити цю проблему:

- Він об'єднав усі операції над піснями та плейлистами в єдиний інтерфейс.
- Це дозволило контролерам взаємодіяти з однією точкою доступу (**MusicPlayerFacade**) замість того, щоб звертатися до декількох сервісів.
- Патерн зменшив залежність контролерів від конкретної реалізації сервісів, що дозволило спростити логіку контролерів і зробило код більш структурованим і чистим.

Переваги застосування патерну Facade

1. **Інкапсуляція складності:**
    - Патерн Фасад приховує складність внутрішніх операцій між сервісами та підсистемами. Він надає простий інтерфейс, через

який можна виконувати операції без необхідності взаємодії з кожним сервісом окремо.

2. **Зменшення залежностей:**
    - Контролери не повинні взаємодіяти з кожним сервісом безпосередньо, що знижує їх залежність від внутрішньої реалізації сервісів. Це дозволяє легше змінювати або оновлювати частини системи без необхідності вносити зміни в контролери.
3. **Спрощення контролерів:**
    - Контролери більше не повинні містити складний код для роботи з кількома сервісами. Вони просто викликають методи фасаду, що робить код контролерів менш громіздким і легшим для підтримки.
4. **Легкість у тестуванні:**
    - Завдяки Фасаду можна створювати модульні тести, які будуть взаємодіяти лише з фасадом, а не з усіма сервісами окремо. Це спрощує тестування і дозволяє зосередитися на бізнес-логіці, не занурюючись у деталі реалізації кожного сервісу.
5. **Полегшення змін і розширень:**
    - Якщо потрібно змінити або додати нові сервіси, це можна зробити в межах фасаду, не торкаючись контролерів або інших частин програми. Це дозволяє безпечно і ефективно розширювати систему, не порушуючи існуючий код.

Загалом, патерн **Facade** значно підвищує зручність і підтримуваність коду, дозволяючи спростити структуру проекту і зробити систему більш гнучкою до змін та розширень.

Висновок: У даній лабораторній роботі я реалізувала патерн проектування Facade для музичного плеєра, що дозволило створити простий і єдиний інтерфейс для взаємодії з численними сервісами плейлистів та пісень. Завдяки цьому патерну, складність взаємодії між різними компонентами системи була прихована за одним фасадом, що значно спростило код контролерів та зробило його більш чистим і зрозумілим. Переваги застосування Facade включають інкапсуляцію складних операцій, зменшення залежностей між компонентами, спрощення тестування та підвищення гнучкості системи. Реалізація цього патерну зробила систему більш організованою, зручнішою для підтримки та масштабування, а також сприяла кращій структуризації коду та підвищенню ефективності подальшого розвитку.