Національний технічний університет України «Київський
політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

ЛАБОРАТОРНА РОБОТА №7

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Виконав:

студент групи ІА-33

Грицай Андрій

Перевірив:

асистент кафедри ІСТ

Мягкий Михайло Юрійович

Київ - 2025

Тема: Патерни проектування
Мета: Вивчити структуру шаблонів «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати їх в реалізації програмної системи.
Тема лабораторного циклу:

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.

Теоретичні відомості

Посередник (Mediator)

Шаблон "Посередник" забезпечує централізоване управління взаємодією між об'єктами, зменшуючи кількість прямих залежностей між ними. Використовується, коли потрібно уникнути складності, що виникає через безпосередні зв'язки між об'єктами в системі. У Java цей шаблон зазвичай реалізується шляхом створення посередника, який координує обмін даними або подіями між об'єктами, що реєструються у ньому. Це спрощує модифікацію і підтримку компонентів, зменшуючи зв'язаність між ними.

Фасад (Facade)

Шаблон "Фасад" надає єдиний інтерфейс до групи взаємопов'язаних класів або підсистем, спрощуючи їх використання. Використовується для зменшення складності системи, приховуючи її деталі реалізації та надаючи зручний API. У Java цей шаблон зазвичай реалізується через клас, який містить методи для основних операцій, що викликають функціонал внутрішніх класів або підсистем. Це дозволяє зменшити залежності між клієнтським кодом і складними підсистемами, спрощуючи розробку і підтримку.

Міст (Bridge)

Шаблон "Міст" розділяє абстракцію та її реалізацію, дозволяючи змінювати їх незалежно одна від одної. Використовується для уникнення жорсткого зв'язку між абстракцією та її конкретними реалізаціями. У Java цей шаблон реалізується через створення інтерфейсу або абстрактного класу для абстракції, а також окремих класів для реалізацій, які пов'язуються через композицію. Це забезпечує більшу гнучкість у розширенні функціональності системи.

Шаблонний метод

(Template Method) Шаблон "Шаблонний метод" визначає загальну структуру алгоритму, делегуючи реалізацію деяких його кроків підкласам. Використовується для забезпечення гнучкості та уникнення дублювання коду, коли алгоритм має спільні етапи для різних реалізацій. У Java цей шаблон реалізується через абстрактний клас із визначеним методом-шаблоном, який

викликає конкретні реалізації абстрактних методів, що задаються підкласами. Це дозволяє стандартизувати алгоритм, зберігаючи варіативність його частин.

Хід роботи

У цій роботі для реалізації взаємодії клієнтського інтерфейсу зі складною системою сервісів та мережевою комунікацією було використано патерн "Фасад" (Facade). Цей патерн був обраний як архітектурне рішення для забезпечення простого та уніфікованого інтерфейсу доступу до функціоналу системи, приховуючи складність низькорівневих мережевих запитів та залежностей між сервісами.

Без використання патерну "Фасад", реалізація логіки у контролері (MusicPlayerController) вимагала б прямої взаємодії з класом NetworkClient, ручного формування об'єктів NetworkRequest, обробки NetworkResponse та самостійної координації роботи UserService, TrackService та PlaylistService.

Така сильна зв'язність (High Coupling) призвела б до перевантаження контролера деталями реалізації мережевого протоколу та бізнес-логіки. Більш того, будь-яка зміна у структурі запитів або логіці сервісів вимагала б модифікації коду в усіх контролерах, що використовують цей функціонал.

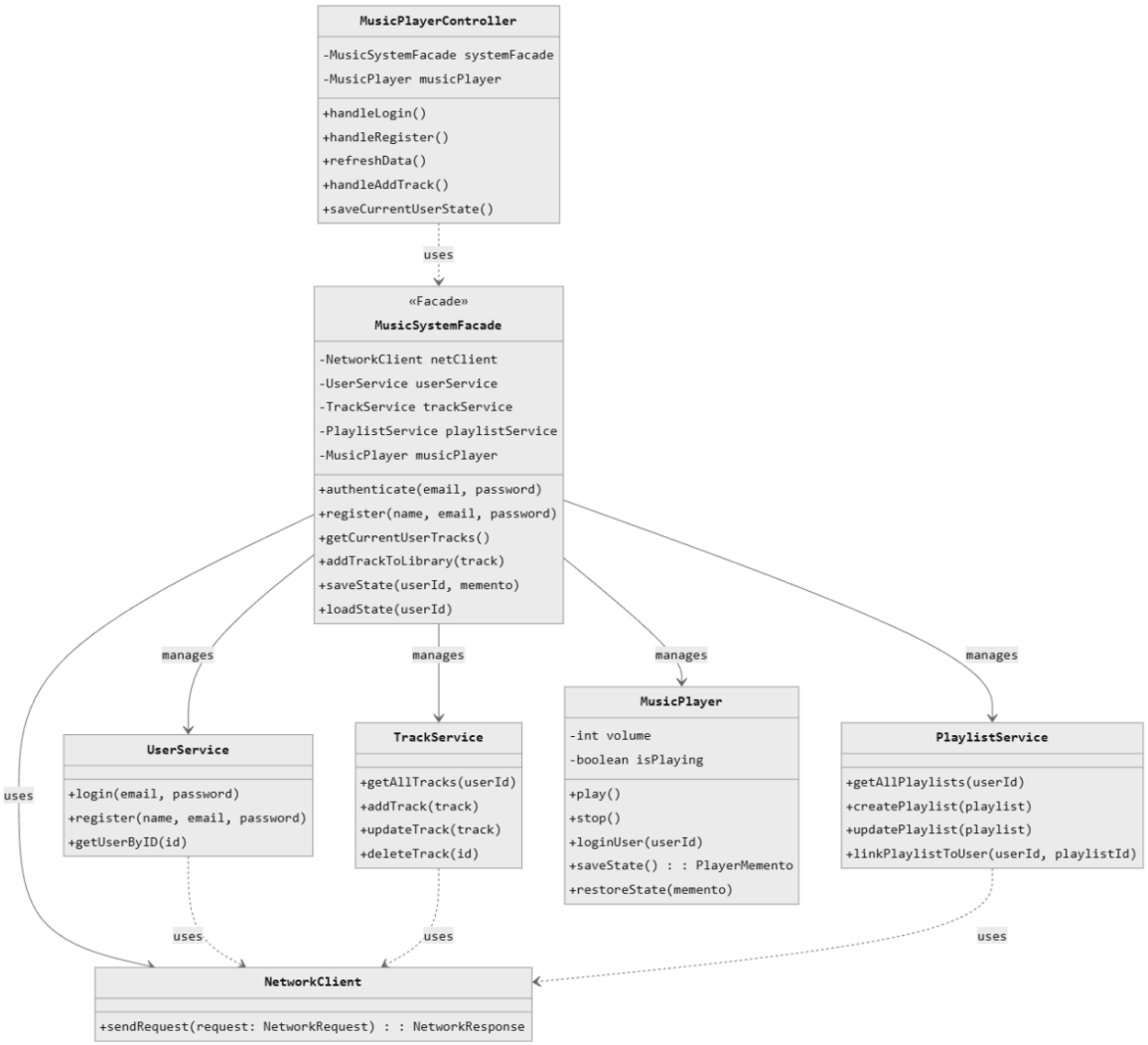


Рисунок 1 – Діаграма класів, яка представляє використання патерну Facade

Надана діаграма класів ілюструє структуру:

- Фасад (Facade): MusicSystemFacade. Він надає спрощений інтерфейс для клієнтського коду та делегує виклики відповідним об'єктам підсистеми.
- Підсистема (Subsystem Classes): Класи NetworkClient, UserService, TrackService, PlaylistService. Вони реалізують складну функціональність, працюючи з мережею та даними, але приховані від прямого доступу з UI.

Центральною фігурою патерну є клас MusicSystemFacade, який визначає методи високого рівня для роботи з додатком: Цей клас інкапсулює складність взаємодії компонентів, гарантуючи, що клієнтський код викликає прості методи, а не керує потоком даних вручну:

1. authenticate(email, password): Виконує вхід користувача, приховуючи формування мережевого запиту LOGIN.
2. getCurrentUserTracks(): Отримує список треків, звертаючись до відповідного сервісу.
3. addTrackToLibrary(track): Делегує додавання треку через мережевий клієнт.
4. saveState() / loadState(): Керує збереженням стану плеєра, взаємодіючи з репозиторієм через мережу.

Без використання патерну "Фасад", реалізація взаємодії з сервером вимагала б дублювання коду для створення з'єднання та обробки запитів у кожному методі контролера. Наприклад, логіка створення NetworkRequest, серіалізації даних та обробки помилок з'єднання, якби вона була "розмазана" по MusicPlayerController, ускладнювала б підтримку та тестування: зміна формату передачі даних змусила б переписувати весь UI-код.

Патерн "Фасад" вирішує це, ізолюючи складну підсистему від клієнта:

1. Клас Фасаду (MusicSystemFacade): Приймає прості виклики від контролера і перетворює їх на послідовність дій всередині підсистеми (наприклад, надсилає запит через NetworkClient).
2. Класи Підсистеми: Виконують реальну роботу (автентифікація, CRUD-операції з треками), не знаючи про існування фасаду.

Завдяки цьому, основна логіка контролера працює лише з класом MusicSystemFacade, викликаючи зрозумілі методи бізнес-логіки, а не оперуючи мережевими сокетами та потоками даних.

Код класів:

MusicSystemFacade:

```
public class MusicSystemFacade {

    private final MusicPlayer musicPlayer;

    private final UserService userService;

    private final TrackService trackService;

    private final PlaylistService playlistService;

    private final NetworkClient netClient;


    public MusicSystemFacade() {

        this.netClient = new NetworkClient();


        this.userService = new UserService(null) {

            @Override

            public User login(String email, String password) {

                NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("LOGIN", new String[]{email, password}));

                return resp.isSuccess() ? (User) resp.getData() : null;

            }


            @Override

            public User register(String name, String email, String password) {

                NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("REGISTER", new String[]{name, email, password}));

                return resp.isSuccess() ? (User) resp.getData() : null;

            }


            @Override

            public User getUserByID(int id) {

                NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("GET_USER", id));

                return resp.isSuccess() ? (User) resp.getData() : null;

            }
```

```
        };


        this.trackService = new TrackService(null) {

            @Override

            public List<Track> getAllTracks(int userId) {

                NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("GET_USER_TRACKS", userId));

                return resp.isSuccess() ? (List<Track>) resp.getData() :
List.of();

            }


            @Override

            public void addTrack(Track track) {

                netClient.sendRequest(new NetworkRequest("ADD_TRACK", track));

            }


            @Override

            public void updateTrack(Track track) {

                netClient.sendRequest(new NetworkRequest("UPDATE_TRACK", track));

            }


            @Override

            public void deleteTrack(int id) {

                netClient.sendRequest(new NetworkRequest("DELETE_TRACK", id));

            }


            @Override

            public Track getTrackByID(int id) {

                NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("GET_TRACK", id));

                return resp.isSuccess() ? (Track) resp.getData() : null;

            }
```

```
            @Override
            public void linkTrackToUser(int userId, int trackId) {
                netClient.sendRequest(new NetworkRequest("LINK_TRACK_USER", new
int[]{userId, trackId}));
            }
        };

        this.playlistService = new PlaylistService(null) {
            @Override
            public List<Playlist> getAllPlaylists(int userId) {
                NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("GET_USER_PLAYLISTS", userId));
                return resp.isSuccess() ? (List<Playlist>) resp.getData() :
List.of();
            }

            @Override
            public Playlist createPlaylist(Playlist playlist) {
                NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("CREATE_PLAYLIST", playlist));
                return resp.isSuccess() ? (Playlist) resp.getData() : null;
            }

            @Override
            public void updatePlaylist(Playlist playlist) {
                netClient.sendRequest(new NetworkRequest("UPDATE_PLAYLIST",
playlist));
            }

            @Override
            public void deletePlaylist(int id) {
                netClient.sendRequest(new NetworkRequest("DELETE_PLAYLIST", id));
```

```
            }

            @Override
            public void linkPlaylistToUser(int userId, int playlistId) {
                netClient.sendRequest(new NetworkRequest("LINK_PLAYLIST_USER", new int[]{userId, playlistId}));
            }

            @Override
            public Playlist getPlaylistByID(int id) {
                NetworkResponse resp = netClient.sendRequest(new NetworkRequest("GET_PLAYLIST", id));
                return resp.isSuccess() ? (Playlist) resp.getData() : null;
            }
        };

        this.musicPlayer = new MusicPlayer(userService, trackService, playlistService);
    }

    public MusicPlayer getPlayer() { return musicPlayer; }
    public UserService getUserService() { return userService; }
    public TrackService getTrackService() { return trackService; }
    public PlaylistService getPlaylistService() { return playlistService; }

    public User authenticate(String email, String password) {
        User user = userService.login(email, password);
        if (user != null) musicPlayer.loginUser(user.getUserID());
        return user;
    }

    public User register(String name, String email, String password) {
```

```
        User user = userService.register(name, email, password);
        if (user != null) musicPlayer.loginUser(user.getUserID());
        return user;
    }

    public List<Track> getCurrentUserTracks() {
        if (musicPlayer.getCurrentUser() == null) return List.of();
        return
trackService.getAllTracks(musicPlayer.getCurrentUser().getUserID());
    }

    public void addTrackToLibrary(Track track) {
        if (musicPlayer.getCurrentUser() != null) {
            int userId = musicPlayer.getCurrentUser().getUserID();
            netClient.sendRequest(new NetworkRequest("ADD_TRACK", new
Object[]{userId, track}));
        }
    }

    public void saveState(int userId, PlayerMemento memento) {
        netClient.sendRequest(new NetworkRequest("SAVE_STATE", new
Object[]{userId, memento}));
    }

    public PlayerMemento loadState(int userId) {
        NetworkResponse resp = netClient.sendRequest(new
NetworkRequest("LOAD_STATE", userId));
        return resp.isSuccess() ? (PlayerMemento) resp.getData() : null;
    }
}
```

MusicPlayerController:

```
public class MusicPlayerController {
```

```
private final MusicSystemFacade systemFacade;
private MusicPlayer musicPlayer;

public MusicPlayerController(MusicPlayerView view) {
    this.systemFacade = new MusicSystemFacade();
    this.musicPlayer = systemFacade.getPlayer();
}

private void handleLogin() {
    // another code

    User user = systemFacade.authenticate(email, password);

    if (user != null) {
        refreshData();
        loadLastState();
    } else {
        view.showAlert("Помилка", "Невірний логін або пароль");
    }
}

private void saveTrackToDB(Track track) {
    systemFacade.addTrackToLibrary(track);
    refreshData();
}

private void loadLastState() {
    if (musicPlayer.getCurrentUser() == null) return;
```

```
        PlayerMemento savedState =
systemFacade.loadState(musicPlayer.getCurrentUser().getUserID());



        if (savedState != null) {

            musicPlayer.restoreState(savedState);

        }

    }

}
```

NetworkClient:

```
public class NetworkClient {

    private static final String SERVER_HOST = "localhost";

    private static final int SERVER_PORT = 8888;



    public NetworkResponse sendRequest(NetworkRequest request) {

        try (Socket socket = new Socket(SERVER_HOST, SERVER_PORT);

             ObjectOutputStream out = new
ObjectOutputStream(socket.getOutputStream());

             ObjectInputStream in = new
ObjectInputStream(socket.getInputStream())) {



            out.writeObject(request);

            return (NetworkResponse) in.readObject();



        } catch (Exception e) {

            e.printStackTrace();

            return new NetworkResponse(false, "Помилка з'єднання: " +
e.getMessage());

        }

    }

}
```

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з принципами та структурою ключових патернів проєктування. Особливу увагу я приділив

реалізації шаблону "Фасад" (Facade), детально описавши його логіку та функціональність у контексті взаємодії клієнтської частини музичного плеєра зі складною серверною підсистемою. Цей патерн було обрано як архітектурне рішення для забезпечення простого та уніфікованого інтерфейсу доступу до функціоналу, приховуючи складність низькорівневих мережевих операцій та управління сервісами.

Під час реалізації шаблону "Фасад" я зрозумів, наскільки елегантно він вирішує проблему сильної зв'язності (High Coupling) між шаром представлення та бізнес-логікою. Клас MusicSystemFacade інкапсулює роботу з класами підсистеми (NetworkClient, UserService, TrackService), перетворюючи складні послідовності дій на прості виклики. Це дозволяє "Клієнту" (MusicPlayerController) викликати зрозумілі методи, такі як authenticate() або addTrackToLibrary(), не турбуючись про формування об'єктів NetworkRequest, серіалізацію даних чи обробку відповідей від сервера.

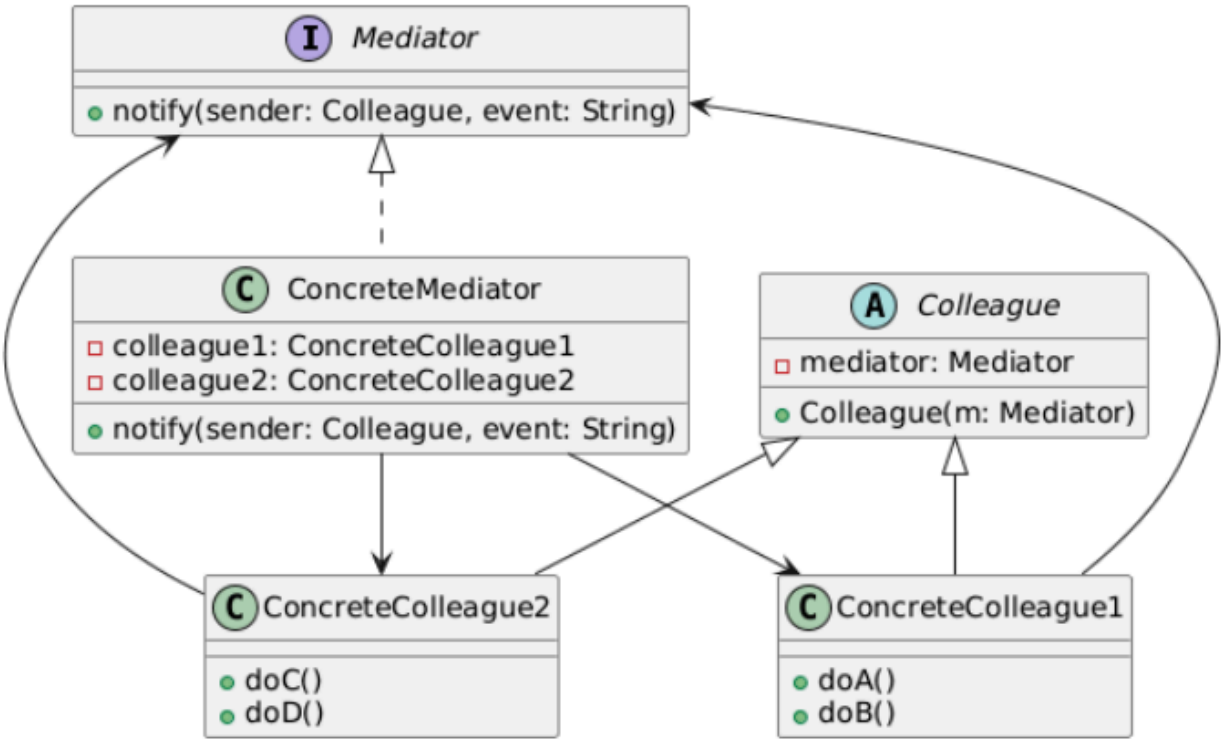
Цей досвід роботи з патерном "Фасад" дозволив мені краще зрозуміти його переваги у контексті побудови розподілених додатків. Я усвідомив його ключову роль у підвищенні читабельності коду та спрощенні підтримки системи: будь-які зміни у протоколі мережевої взаємодії або логіці сервісів тепер локалізовані всередині фасаду і не вимагають модифікації коду контролерів візуального інтерфейсу.

Контрольні запитання

1. Яке призначення шаблону «Посередник»?

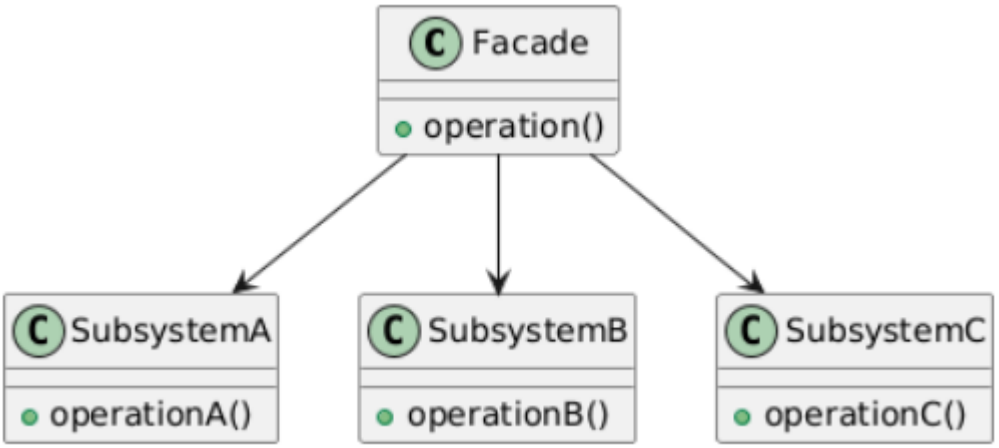
Шаблон «Посередник» (Mediator) призначений для спрощення взаємодії між об’єктами у системі. Замість того, щоб об’єкти напряму обмінювалися повідомленнями та створювали заплутані зв’язки, усі вони спілкуються через єдиний об’єкт-посередник. Посередник контролює обмін даними між компонентами, координує їхню роботу та зменшує кількість залежностей у програмі. Завдяки цьому система стає більш гнучкою та легкою в супроводі, оскільки зміни в одному компоненті не потребують змін в інших.

2. Нарисуйте структуру шаблону «Посередник».

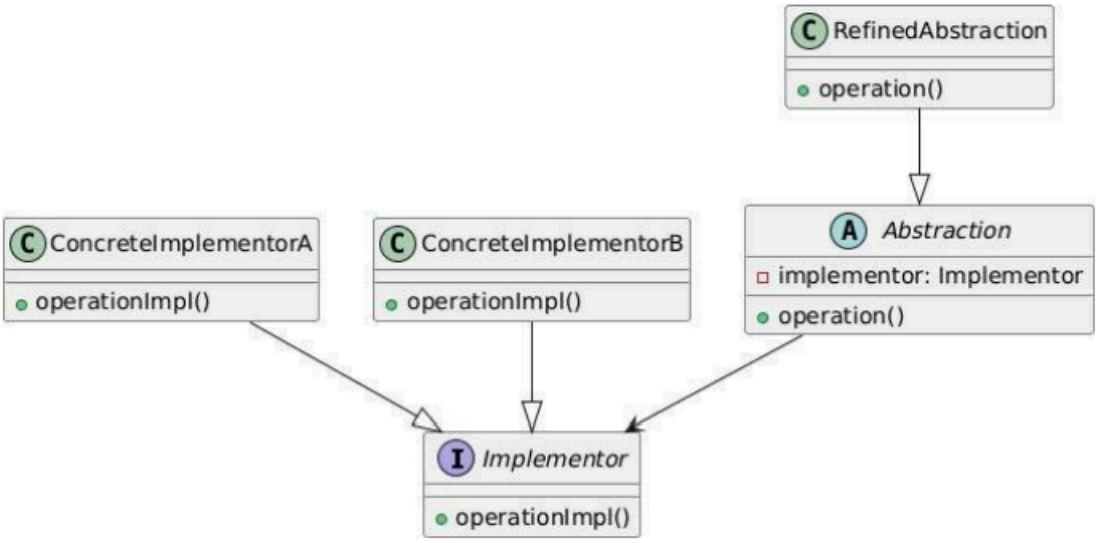


3. Які класи входять в шаблон «Посередник», та яка між ними взаємодія?
До шаблону «Посередник» (Mediator) входять такі основні класи:
- Mediator (Посередник) — це інтерфейс або абстрактний клас, який визначає методи для обміну інформацією між об’єктами.
- ConcreteMediator (Конкретний посередник) — реалізує інтерфейс посередника та координує взаємодію між конкретними об’єктами (колегами). Він знає, які об’єкти беруть участь у взаємодії, і спрямовує повідомлення між ними.
- Colleague (Колега) — це базовий клас або інтерфейс для об’єктів, які взаємодіють через посередника.
- ConcreteColleague (Конкретний колега) — об’єкти, які виконують певні дії, але не спілкуються напряму між собою. Вони надсилають повідомлення посереднику, а той уже вирішує, кому їх передати. Коли один із колег хоче повідомити інший об’єкт, він не викликає його метод безпосередньо, а звертається до посередника. Посередник приймає повідомлення та вирішує, який інший колега повинен отримати цю інформацію. Таким чином, усі зв’язки проходять через посередника, що робить систему більш керованою й менш залежною.
4. Яке призначення шаблону «Фасад»?
Шаблон «Фасад» (Facade) призначений для спрощення роботи зі складною системою. Він надає єдиний узагальнений інтерфейс для взаємодії з групою класів або підсистем, приховуючи їхню внутрішню складність. Завдяки фасаду клієнт може виконувати складні операції за допомогою одного простого

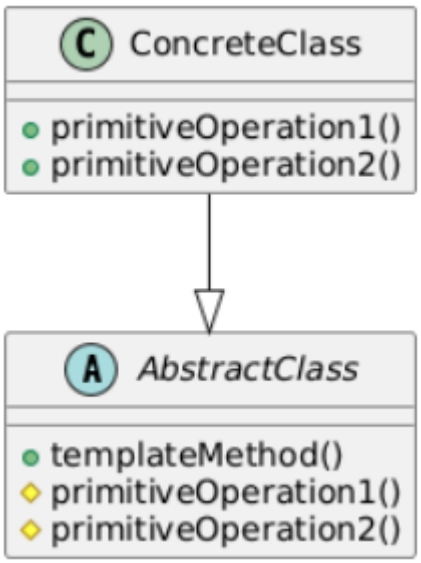
виклику методу, не знаючи, як саме усе реалізовано всередині. Це робить код зрозумілішим, зручнішим у використанні та легшим у супроводі.
5. Нарисуйте структуру шаблону «Фасад».



6. Які класи входять в шаблон «Фасад», та яка між ними взаємодія?
До шаблону «Фасад» (Facade) входять такі основні класи:
- Facade (Фасад) — головний клас, який надає спрощений інтерфейс для клієнта. Він містить посилання на класи підсистеми та викликає їхні методи у потрібному порядку.
- Subsystem classes (Класи підсистеми) — це реальні компоненти системи, які виконують основну роботу. Вони не знають про фасад і можуть використовуватися незалежно від нього.
- Client (Клієнт) — це код, який використовує фасад, щоб отримати доступ до функціональності підсистеми через простий інтерфейс. Клієнт звертається лише до фасаду, не взаємодіючи безпосередньо з класами підсистеми. Фасад приймає запит, викликає необхідні методи підсистеми та повертає результат. Таким чином, фасад приховує складність внутрішньої структури системи й спрощує роботу з нею.
7. Яке призначення шаблону «Міст»?
Шаблон «Міст» (Bridge) призначений для відокремлення абстракції від її реалізації, щоб вони могли розвиватися незалежно одна від одної. Замість того, щоб жорстко пов’язувати абстрактний клас із конкретною реалізацією, «Міст» створює між ними окремий зв’язок через інтерфейс. Це дозволяє легко змінювати або розширювати як абстракцію, так і реалізацію, не впливаючи одна на одну. У результаті код стає більш гнучким і масштабованим, особливо коли потрібно підтримувати різні варіанти реалізацій або платформ.
8. Нарисуйте структуру шаблону «Міст».



9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?
До шаблону «Міст» (Bridge) входять такі основні класи:
- Abstraction (Абстракція) — визначає базовий інтерфейс для користувачів і містить посилання на об’єкт реалізації. Вона делегує частину роботи об’єкту реалізації, замість того щоб виконувати її самостійно.
- RefinedAbstraction (Уточнена абстракція) — розширює функціональність базової абстракції, але все одно використовує реалізацію через міст.
- Implementor (Реалізатор) — це інтерфейс, який визначає методи, що мають бути реалізовані в конкретних реалізаторах.
- ConcreteImplementor (Конкретний реалізатор) — надає специфічну реалізацію методів, оголошених в інтерфейсі Implementor. Абстракція зберігає посилання на об’єкт типу Implementor і викликає його методи для виконання конкретних дій. Клієнт працює лише з абстракцією, не знаючи про деталі реалізації. Таким чином, абстракція та реалізація можуть змінюватися незалежно, що забезпечує гнучкість і розширюваність системи.
10. Яке призначення шаблону «Шаблоний метод»?
Шаблон «Шаблонний метод» (Template Method) призначений для визначення загальної послідовності дій алгоритму в базовому класі, залишаючи реалізацію окремих кроків підкласам. Це дозволяє створити спільний “каркас” алгоритму, який не потрібно переписувати в кожному класі, а лише перевизначати окремі частини, що відрізняються. Завдяки цьому шаблон забезпечує повторне використання коду, підтримує єдину структуру процесу та дозволяє легко змінювати поведінку алгоритму через наслідування.
11. Нарисуйте структуру шаблону «Шаблонний метод».



12. Які класи входять в шаблон «Шаблоний метод», та яка між ними взаємодія?
До шаблону «Шаблонний метод» (Template Method) входять такі основні класи:
- AbstractClass (Абстрактний клас) — визначає шаблонний метод, який описує загальну послідовність виконання алгоритму. Деякі кроки алгоритму реалізовані тут, а деякі оголошені як абстрактні методи, які мають бути реалізовані підкласами.
- ConcreteClass (Конкретний клас) — наслідує абстрактний клас і реалізує абстрактні методи, тобто визначає конкретну поведінку окремих кроків алгоритму. Клієнт викликає шаблонний метод абстрактного класу. Шаблонний метод послідовно викликає внутрішні методи: частина з них реалізована в абстрактному класі, частина — у підкласах. Підклас відповідає лише за деталі окремих кроків, не змінюючи загальну структуру алгоритму.
13. Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?
Шаблон «Шаблонний метод» використовується для визначення загальної структури алгоритму в базовому класі, залишаючи підкласам реалізацію окремих кроків. Він дозволяє повторно використовувати код і змінювати лише деталі алгоритму без зміни його загальної структури. Шаблон «Фабричний метод» застосовується для створення об’єктів без жорсткого зв’язку з конкретними класами. Підкласи вирішують, який конкретний об’єкт створювати, а клієнт користується лише базовим інтерфейсом або класом. Основна відмінність полягає в тому, що «Шаблонний метод» визначає послідовність дій, а «Фабричний метод» — спосіб створення об’єктів. 14. Яку функціональність додає шаблон «Міст»? Шаблон «Міст» (Bridge) додає функціональність для незалежного розділення абстракції та її реалізації. Він дозволяє змінювати або розширювати абстракцію і реалізацію окремо, не впливаючи одна на одну. Завдяки цьому система стає гнучкішою, масштабованішою та легшою для підтримки, особливо коли потрібно підтримувати кілька варіантів реалізації або платформ.