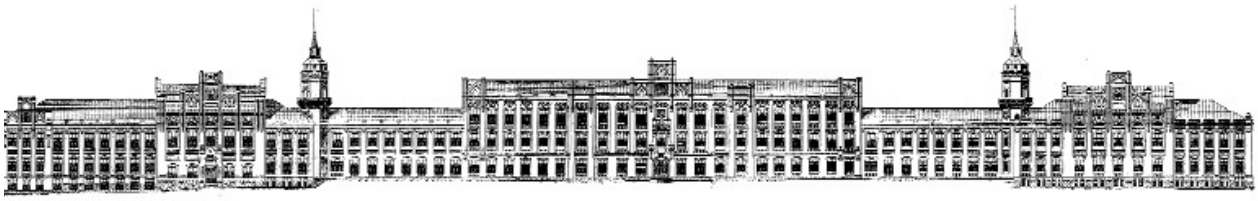

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Звіт

З дисципліни “Технології розроблення програмного забезпечення”

Варіант: 23

Виконала
студентка групи ІА-23:
Павленко Анастасія

Перевірив:
Мягкий Михайло
Юрійович

Київ 2024

Лабораторна робота №7

Шаблон «Mediator», «Facade», «Bridge», «Template Method»

Мета лабораторної роботи: Ознайомитися з основними шаблонами проектування програмного забезпечення, їх призначенням і застосуванням. Розглянути реалізацію на прикладах, вивчити їх переваги та недоліки, а також практично застосувати ці шаблони для вирішення конкретних задач програмування.

Варіант 23: Project Management software (proxy, chain of responsibility, abstract factory, bridge, flyweight, client-server)

Програмне забезпечення для управління проектами повинно мати наступні функції: супровід завдань/вимог/проектів, списків команд, поточних завдань, планування за методологіями agile/kanban/rup (включаючи дошку завдань, ітерації тощо), мати можливість прикріплювати вкладені файли до завдань та посилатися на конкретні версії програми, зберігати виконувані файли для кожної версії.

Завдання:

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їх взаємодій для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.

Зміст

Короткі теоретичні відомості

Шаблон «Mediator»

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

Патерн Посередник змушує об’єкти спілкуватися через окремий об’єкт-посередник, який знає, кому потрібно перенаправити той або інший запит. Завдяки цьому компоненти системи залежатимуть тільки від посередника, а не від десятків інших компонентів.

Придатність

- Коли складно змінювати деякі класи через те, що вони мають величезну кількість хаотичних зв’язків з іншими класами.
- Посередник дозволяє розмістити усі ці зв’язки в одному класі. Після цього буде легше їх відрефакторити, зробити більш зрозумілими й гнучкими.
- Коли ви не можете повторно використовувати клас, оскільки він залежить від безлічі інших класів.
- Після застосування патерна компоненти втрачають колишні зв’язки з іншими компонентами, а все їхнє спілкування відбувається опосередковано, через об’єкт посередника.
- Коли вам доводиться створювати багато підкласів компонентів, щоб використовувати одні й ті самі компоненти в різних контекстах.
- Якщо раніше зміна відносин в одному компоненті могла призвести до лавини змін в усіх інших компонентах, то тепер вам достатньо створити підклас посередника та змінити в ньому зв’язки між компонентами.

Структура

1. **Компоненти** — це різнорідні об’єкти, що містять бізнес-логіку програми. Кожен компонент має посилання на об’єкт посередника, але

працює з ним тільки через абстрактний інтерфейс посередників. Завдяки цьому компоненти можна повторно використовувати в інших програмах, зв’язавши їх з посередником іншого типу.

2. **Посередник** визначає інтерфейс для обміну інформацією з компонентами. Зазвичай достатньо одного методу, щоби повідомляти посередника про події, що відбулися в компонентах. У параметрах цього методу можна передавати деталі події: посилання на компонент, в якому вона відбулася, та будь-які інші дані.
3. **Конкретний посередник** містить код взаємодії кількох компонентів між собою. Найчастіше цей об’єкт не тільки зберігає посилання на всі свої компоненти, але й сам їх створює, керуючи подальшим життєвим циклом.
4. Компоненти не повинні спілкуватися один з одним безпосередньо. Якщо в компоненті відбувається важлива подія, він повинен повідомити свого посередника, а той сам вирішить, чи стосується подія інших компонентів, і чи треба їх сповістити. При цьому компонент-відправник не знає, хто обробить його запит, а компонент-одержувач не знає, хто його надіслав.

Переваги та недоліки

- Усуває залежності між компонентами, дозволяючи використовувати їх повторно.
- Спрощує взаємодію між компонентами.
- Централізує керування в одному місці.
- Посередник може **сильно «роздутися»**.

Шаблон «Facade»

Фасад — це структурний патерн проектування, який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку.

Фасад — це простий інтерфейс для роботи зі складною підсистемою, яка містить безліч класів. Фасад може бути спрощеним відображенням системи, що не має 100% тієї функціональності, якої можна було б досягти, використовуючи складну підсистему безпосередньо. Разом з тим, він надає саме ті «фічі», які потрібні клієнтові, і приховує все інше.

Фасад корисний у тому випадку, якщо ви використовуєте якусь складну бібліотеку з безліччю рухомих частин, з яких вам потрібна тільки частина.

Структура

1. **Фасад** надає швидкий доступ до певної функціональності підсистеми. Він «знає», яким класам потрібно переадресувати запит, і які дані для цього потрібні.
2. **Додатковий фасад** можна ввести, щоб не захаращувати єдиний фасад різнорідною функціональністю. Він може використовуватися як клієнтом, так й іншими фасадами.
3. Складна підсистема має безліч різноманітних класів. Для того, щоб примусити усіх їх щось робити, потрібно знати подробиці влаштування підсистеми, порядок ініціалізації об’єктів та інші деталі.

Класи підсистеми не знають про існування фасаду і працюють один з одним безпосередньо.

4. **Клієнт** використовує фасад замість безпосередньої роботи з об’єктами складної підсистеми.

Застосування

- Якщо вам потрібно надати простий або урізаний інтерфейс до складної підсистеми.

Часто підсистеми ускладнюються в міру розвитку програми. Застосування більшості патернів призводить до появи менших класів, але у великій кількості. Фасад пропонує певний вид системи за замовчуванням, який влаштовує більшість клієнтів.

- Якщо ви хочете розкласти підсистему на окремі рівні.

Використовуйте фасади для визначення точок входу на кожен рівень підсистеми. Якщо підсистеми залежать одна від одної, тоді залежність можна спростити, дозволивши підсистемам обмінюватися інформацією тільки через фасади.

Шаблон «Bridge»

Міст — це структурний патерн проектування, який розділяє один або кілька класів на дві окремі ієрархії — абстракцію та реалізацію, дозволяючи змінювати код в одній гілці класів, незалежно від іншої.

Патерн Міст пропонує замінити спадкування на делегування. Для цього потрібно виділити одну з таких «площин» в окрему ієрархію і посилатися на об’єкт цієї ієрархії, замість зберігання його стану та поведінки всередині одного класу.

Отже, *абстракція* (або *інтерфейс*) — це уявний рівень керування чим-небудь, що не виконує роботу самостійно, а делегує її рівню *реалізації* (який зветься *платформою*).

Структура

1. **Абстракція** містить керуючу логіку. Код абстракції делегує реальну роботу пов’язаному об’єктові реалізації.
2. **Реалізація** описує загальний інтерфейс для всіх реалізацій. Всі методи, які тут описані, будуть доступні з класу абстракції та його підкласів.

   Інтерфейси абстракції та реалізації можуть або збігатися, або бути абсолютно різними. Проте, зазвичай в реалізації живуть базові операції, на яких будуються складні операції абстракції.
3. **Конкретні реалізації** містять платформо-залежний код.

4. **Розширені абстракції** містять різні варіації керуючої логіки. Як і батьківский клас, працює з реалізаціями тільки через загальний інтерфейс реалізацій.
5. **Клієнт** працює тільки з об’єктами абстракції. Не рахуючи початкового зв’язування абстракції з однією із реалізацій, клієнтський код не має прямого доступу до об’єктів реалізації.

Застосування

- Якщо ви хочете розділити монолітний клас, який містить кілька різних реалізацій якої-небудь функціональності (наприклад, якщо клас може працювати з різними системами баз даних).

Міст дозволяє розділити монолітний клас на кілька окремих ієрархій. Після цього ви можете змінювати код в одній гілці класів незалежно від іншої. Це спрощує роботу над кодом і зменшує ймовірність внесення помилок.

- Якщо клас потрібно розширювати в двох незалежних площинах.

Міст пропонує виділити одну з таких площин в окрему ієрархію класів, зберігаючи посилання на один з її об’єктів у початковому класі.

- Якщо ви хочете мати можливість змінювати реалізацію під час виконання програми.

Міст дозволяє замінювати реалізацію навіть під час виконання програми, оскільки конкретна реалізація не «зашита» в клас абстракції.

Шаблон «Template Method»

Шаблонний метод — це поведінковий патерн проектування, який визначає кістяк алгоритму, перекладаючи відповідальність за деякі його кроки на підкласи. Патерн дозволяє підкласам перевизначати кроки алгоритму, не змінюючи його загальної структури.

Патерн Шаблонний метод пропонує розбити алгоритм на послідовність кроків, описати ці кроки в окремих методах і викликати їх в одному *шаблонному* методі один за одним.

Це дозволить підкласам перевизначити деякі кроки алгоритму, залишаючи без змін його структуру та інші кроки, які для цього підкласу не є важливими.

Структура

1. **Абстрактний клас** визначає кроки алгоритму й містить шаблонний метод, що складається з викликів цих кроків. Кроки можуть бути як абстрактними, так і містити реалізацію за замовчуванням.
2. Конкретний клас перевизначає деякі або всі кроки алгоритму. Конкретні класи не перевизначають сам шаблонний метод.

Застосування

- Якщо підкласи повинні розширювати базовий алгоритм, не змінюючи його структури.

Шаблонний метод дозволяє підкласами розширювати певні кроки алгоритму через спадкування, не змінюючи при цьому структуру алгоритмів, оголошену в базовому класі.

- Якщо у вас є кілька класів, які роблять одне й те саме з незначними відмінностями. Якщо ви редагуєте один клас, тоді доводиться вносити такі ж виправлення до інших класів.

Патерн шаблонний метод пропонує створити для схожих класів спільний суперклас та оформити в ньому головний алгоритм у вигляді кроків. Кроки, які відрізняються, можна перевизначити у підкласах.

Хід виконання лабораторної роботи

Патерн "Міст" застосовується, коли є необхідність розділити абстракцію (логіку роботи системи) та реалізацію (способи виконання цієї логіки) для досягнення таких цілей:

1. **Управління складністю системи**:
   - Якщо система має різні аспекти, наприклад, роботу з типами файлів і платформами зберігання, їх розділення спрощує підтримку та розширення.
2. **Гнучкість і масштабованість**:
   - Нові типи файлів або нові платформи для зберігання можна додавати, не змінюючи існуючу логіку.
3. **Зниження залежностей**:
   - Замість прямої залежності між обробкою файлів і способами їх зберігання використовується спільний інтерфейс. Це полегшує заміну компонентів без масштабного переписування коду.
4. **Повторне використання коду**:
   - Абстракція і реалізація стають незалежними, дозволяючи використовувати одну реалізацію з багатьма варіантами абстракцій.

Отже, використаємо патерн "Міст" для роботи з файлами, такими як Звіти, Фото профілю користувачів та Файли застосовані до конкретних завдань.

Алгоритм реалізації патерна "Міст"

Крок 1. Визначення двох вимірів системи

1. **Абстракція**:
   - Типи файлів (BaseFile, ReportFile, TaskFile, UserProfileImage) і їх логіка.
2. **Реалізація**:
   - Робота з платформами для зберігання (наприклад, Azure Blob Storage) та базою даних.

Крок 2. Створення базової абстракції

BaseFile (Базовий клас для файлів) - це абстрактний клас є основою для всіх типів файлів. Він визначає спільні властивості та методи, які потрібні для роботи з будь-яким файлом, незалежно від його специфіки (звіти, файли завдань, зображення профілю користувача тощо).

```
public class BaseFile
{
    [Key]
    public Guid FileId { get; set; }
    [NotMapped]
    1 usage
    public IFormFile File { get; set; }
    [Required]
    5 usages
    public string FileName { get; set; }
    [Required]
    4 usages
    public string FileExtension { get; set; }
    [Required]
    4 usages
    public long FileSizeInBytes { get; set; }
    [Required]
    4 usages
    public string FileUrl { get; set; }
    4 usages
    public Guid UploadedByUserId { get; set; }
    public AppUser UploadedByUser { get; set; }
    3 usages
    public DateTime UploadedAt { get; set; }
    1 usage
    public string Description { get; set; }
    3 usages
    public bool IsPublic { get; set; }
}
```

Рисунок 1. Клас BaseFile

Крок 3. Розширення абстракції

Для кожного типу файлів створюються спадкоємці BaseFile із додатковими специфічними властивостями.

UserProfileImage - представляє зображення профілю користувача, в икористовується у модулях аутентифікації або профілю користувача.

```
public class UserProfileImage : BaseFile
{
    4 usages
    public Guid UserId { get; set; }
    public AppUser User { get; set; }
}
```

Рисунок 2. Клас UserProfileImage

TaskFile - представляє файл, прив'язаний до завдання, забезпечує управління файлами завдань у проекті, особливо при роботі з версіями:

```
public class TaskFile : BaseFile
{
    3 usages
    public Guid TaskId { get; set; }
    public ProjectTask Task { get; set; }

    1 usage
    public Guid ProjectId { get; set; }
    public Project Project { get; set; }

    3 usages
    public string VersionNumber { get; set; }
    1 usage
    public Guid? PreviousVersionId { get; set; }
    public TaskFile PreviousVersion { get; set; }
}
```

Рисунок 3. Клас TaskFile

ReportFile - представляє файл, пов'язаний зі звітом, забезпечує контекст для зберігання та управління файлами звітів:

```
public class ReportFile : BaseFile
{
    [Required]
    1 usage
    public Guid ReportId { get; set; }

    2 usages
    public Guid ProjectId { get; set; }
    public Project Project { get; set; }
}
```

Рисунок 4. Клас ReportFile

Крок 4. Створення інтерфейсу реалізації

IStorageRepository (Інтерфейс сховища) - визначає набір операцій, які повинні підтримувати всі реалізації сховищ для файлів. Дозволяє працювати з різними платформами (Azure Blob Storage, AWS S3, локальна файлову система) через спільний інтерфейс, не змінюючи логіку використання файлів.

```
public interface IStorageRepository
{
    1 implementation
    Task<string> UploadFileAsync(string containerName, string filePath,
        Stream fileStream, bool isPublic = false);
    1 implementation
    Task DeleteFileAsync(string containerName, string filePath);
    1 implementation
    Task<Stream> DownloadFileAsync(string containerName, string filePath);
    1 implementation
    Task<List<string>> ListFilesAsync(string containerName, string directoryPath);
    1 implementation
    Task<bool> FileExistsAsync(string containerName, string filePath);
    1 implementation
    Task CreateDirectoryAsync(string containerName, string directoryPath);
    1 implementation
    Task DeleteDirectoryAsync(string containerName, string directoryPath);

}
```

Рисунок 5. Інтерфейс сховища IstorageRepository

Крок 5. Реалізація інтерфейсу

Реалізує інтерфейс IStorageRepository і надає реальні механізми завантаження та видалення файлів.

Роль у проекті: Реалізація зберігання файлів у Azure Blob Storage.

```
public class BlobStorageRepository : IStorageRepository
{
    private readonly BlobServiceClient _blobServiceClient;

    public BlobStorageRepository(IConfiguration configuration)
    {
        var connectionString = configuration["AzureBlobStorage:ConnectionString"];
        _blobServiceClient = new BlobServiceClient(connectionString);
    }
```

Рисунок 6. Реалізація сховища BlobStorageRepository

```
    public async Task<string> UploadFileAsync(string containerName, string filePath,
        Stream fileStream, bool isPublic = false)
    {
        var container = _blobServiceClient.GetBlobContainerClient(containerName);
        var blob = container.GetBlobClient(filePath);

        await blob.UploadAsync(fileStream, overwrite: true);

        if (isPublic)
        {
            await container.SetAccessPolicyAsync(PublicAccessType.Blob);
        }

        return blob.Uri.ToString();
    }

    public async Task DeleteFileAsync(string containerName, string filePath)
    {
        var container = _blobServiceClient.GetBlobContainerClient(containerName);
        var blob = container.GetBlobClient(filePath);
        await blob.DeleteIfExistsAsync();
    }

    public async Task<Stream> DownloadFileAsync(string containerName, string filePath)
    {
        var container = _blobServiceClient.GetBlobContainerClient(containerName);
        var blob = container.GetBlobClient(filePath);

        var stream = new MemoryStream();
        await blob.DownloadToAsync(stream);
        stream.Position = 0;
        return stream;
    }
```

```
public async Task<List<string>> ListFilesAsync(string containerName, string directoryPath)
{
    var container = _blobServiceClient.GetBlobContainerClient(containerName);
    var blobs :AsyncPageable<BlobItem>? = container.GetBlobsAsync(prefix: directoryPath);

    var fileList = new List<string>();
    await foreach (var blob in blobs)
    {
        fileList.Add(blob.Name);
    }

    return fileList;
}
```

Рисунок 6. Реалізація методів для роботи з файлами

Крок 6. Розширені абстракції

Якщо є кілька варіантів керуючої логіки, створюються підкласи, наприклад, різні репозиторії для різних типів файлів (TaskFileRepository, UserProfileImageRepository).

```
public class FileRepository<TFile> : IFileRepository<TFile> where TFile : BaseFile
{
    protected readonly AppDbContext _context;

    3 usages
    public FileRepository(AppDbContext context)
    {
        _context = context;
    }

    0+2 usages
    public async Task<TFile> AddAsync(TFile file)
    {
        await _context.Set<TFile>().AddAsync(file);
        await _context.SaveChangesAsync();
        return file;
    }

    public async Task<TFile> GetByIdAsync(Guid fileId)
    {
        return await _context.Set<TFile?>().FindAsync(fileId);
    }

    public async Task<IEnumerable<TFile>> GetAllByUserIdAsync(Guid userId)
    {
        return await _context.Set<TFile>() // DbSet<TFile>
            .Where(f :TFile => f.UploadedByUserId == userId) // IQueryable<TFile>
            .ToListAsync(); // Task<List<...>>
    }

    public async Task DeleteAsync(Guid fileId)
    {
        var file = await _context.Set<TFile>().FindAsync(fileId);
        if (file != null)
        {
            _context.Set<TFile>().Remove(file);
            await _context.SaveChangesAsync();
        }
    }
}
```

Рисунок 8. Базовий репозиторій для роботи з файлами

```
public class UserProfileImageRepository : FileRepository<UserProfileImage>, IUserProfileImageRepository
{
    public UserProfileImageRepository(AppDbContext context) : base(context) { }

    1+1 usages
    public async Task<UserProfileImage> GetProfileImageByUserIdAsync(Guid userId)
    {
        return await _context.UserProfileImages.FirstOrDefaultAsync(img :UserProfileImage => img.UserId == userId);
    }
```

Рисунок 9. Конкретна реалізація UserProfileImageRepository

```
public class ReportFileRepository : FileRepository<ReportFile>, IReportFileRepository
{
    public ReportFileRepository(AppDbContext context) : base(context) { }

    public async Task<IEnumerable<ReportFile>> GetAllByProjectIdAsync(Guid projectId)
    {
        return await _context.Set<ReportFile>() // DbSet<ReportFile>
            .Where(rf => rf.ProjectId == projectId) // IQueryable<ReportFile>
            .ToListAsync(); // Task<List<...>>
    }
}
```

Рисунок 10. Конкретна реалізація ReportFileRepository

```
public class TaskFileRepository : FileRepository<TaskFile>, ITaskFileRepository
{
    public TaskFileRepository(AppDbContext context) : base(context) { }

    public async Task<TaskFile?> GetLatestVersionAsync(Guid taskId)
    {
        return await _context.Set<TaskFile>() // DbSet<TaskFile>
            .Where(tf => tf.TaskId == taskId) // IQueryable<TaskFile>
            .OrderByDescending(tf => tf.VersionNumber) // IOrderedQueryable<TaskFile>
            .FirstOrDefaultAsync(); // Task<TaskFile?>
    }
```

Рисунок 11. Конкретна реалізація TaskFileRepository

Крок 7. Клієнтський код

Клас FileService (Сервіс роботи з файлами) агрегує функціональність сховища та забезпечує інтерфейс для клієнтського коду. Абстракція отримує доступ до методів реалізації через інтерфейси. Використовується для роботи з файлами через ін’єкцію залежності (Dependency Injection).

```
public class FileService : IFileService
{
    private readonly IStorageRepository _blobStorageRepository;
    private readonly IReportFileRepository _reportFileRepository;
    private readonly ITaskFileRepository _taskFileRepository;
    private readonly IUserProfileImageRepository _userProfileImageRepository;

    public FileService(
        IStorageRepository blobStorageRepository,
        IReportFileRepository reportFileRepository,
        ITaskFileRepository taskFileRepository,
        IUserProfileImageRepository userProfileImageRepository)
    {
        _blobStorageRepository = blobStorageRepository;
        _reportFileRepository = reportFileRepository;
        _taskFileRepository = taskFileRepository;
        _userProfileImageRepository = userProfileImageRepository;
    }

    3+2 usages
    public async Task<string> UploadFileAsync(Stream fileStream, string filePath, bool isPublic)
    {
        return await _blobStorageRepository.UploadFileAsync( containerName: "main-container", filePath, fileStream);
    }

    1 usage
    public async Task DeleteFileAsync(string filePath)
    {
        await _blobStorageRepository.DeleteFileAsync( containerName: "main-container", filePath);
    }

    public async Task<Stream> DownloadFileAsync(string filePath)
    {
        return await _blobStorageRepository.DownloadFileAsync( containerName: "main-container", filePath);
    }
```

Рисунок 7. Реалізація класу FileService

Результат реалізації патерна "Міст"

1. Модульність і розширюваність:
   - Легко додавати нові типи файлів, наприклад, InvoiceFile, не змінюючи існуючий код.
   - Легко інтегрувати інші платформи для зберігання файлів (наприклад, AWS S3) шляхом створення нової реалізації інтерфейсу IStorageRepository.
2. Незалежність компонентів:

- Логіка роботи з файлами (абстракція) відділена від платформи зберігання (реалізація).

3. Зниження залежності коду:
   - Завдяки використанню інтерфейсів клієнтський код (FileService) працює з файлами без прив'язки до конкретної реалізації.
4. Простота тестування:
   - Легко створювати мок-реалізації для тестування, наприклад, мок IFileRepository або IStorageRepository.

Висновок до лабораторної роботи

У ході виконання лабораторної роботи ми ознайомилися з низкою популярних патернів проектування, зокрема «Mediator», «Facade», «Bridge» та «Template Method», їх теоретичними засадами, принципами використання та застосуванням у реальних проектах. Особливу увагу було приділено реалізації патерна «Bridge».

Практична реалізація патерна: Реалізація «Bridge» у проекті показала його ефективність на прикладі роботи з файлами. Ми побудували систему, яка дозволяє працювати з різними типами файлів (звіти, файли завдань, зображення профілю) незалежно від способу їхнього зберігання (Azure Blob Storage, локальне сховище тощо). Реалізація інтерфейсу IStorageRepository дозволила нам абстрагуватися від конкретної реалізації сховища, що спрощує підтримку та тестування системи.

Знайомство з іншими патернами проектування: Ми також ознайомилися з іншими патернами:

- Mediator: для організації взаємодії між об’єктами через посередника.
- Facade: для спрощення доступу до складної підсистеми через єдиний інтерфейс.
- Template Method: для стандартизації послідовності виконання операцій із можливістю перевизначення окремих кроків у підкласах.

Знання, отримані під час вивчення інших патернів, допоможуть у майбутньому вибирати оптимальні архітектурні рішення для різноманітних задач у програмному забезпеченні.