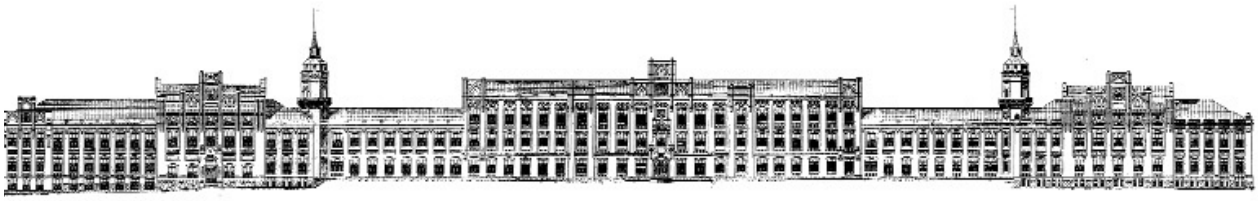

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Звіт

З дисципліни “Технології розроблення програмного забезпечення”

Варіант: 23

Виконала
студентка групи ІА-23:
Павленко Анастасія

Перевірив:
Мягкий Михайло
Юрійович

Київ 2024

Лабораторна робота №5

Шаблони «Adapter», «Builder», «Command», «Chain Of Responsibility», «Prototype»

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

«Анти-шаблони» проектування

Анти-патерни (anti-patterns), також відомі як пастки (pitfalls) - це класи найбільш часто впроваджуваних поганих рішень проблем. Вони вивчаються, як категорія, в разі коли їх хочуть уникнути в майбутньому, і деякі їхні окремі випадки можуть бути розпізнані при вивченні непрацюючих систем.

Шаблон «Adapter»

Адаптер — це структурний патерн проектування, що дає змогу об’єктам із несумісними інтерфейсами працювати разом.

Адаптер це об’єкт-перекладач, який трансформує інтерфейс або дані одного об’єкта таким чином, щоб він став зрозумілим іншому об’єкту.

Адаптер загортає один з об’єктів так, що інший об’єкт навіть не підозрює про існування першого.

Адаптери можуть допомагати об’єктам із різними інтерфейсами працювати разом. Це виглядає так:

1. Адаптер має інтерфейс, сумісний з одним із об’єктів.
2. Тому цей об’єкт може вільно викликати методи адаптера.
3. Адаптер отримує ці виклики та перенаправляє їх іншому об’єкту, але вже в тому форматі та послідовності, які є зрозумілими для цього об’єкта.

Іноді вдається створити навіть *двосторонній адаптер*, який може працювати в обох напрямках.

Застосування

- Якщо ви хочете використати сторонній клас, але його інтерфейс не відповідає решті кодів програми.
- Якщо вам потрібно використати декілька існуючих підкласів, але в них не вистачає якої-небудь спільної функціональності, а розширити суперклас ви не можете.

Шаблон «Builder»

Будівельник — це породжувальний патерн проектування, що дає змогу створювати складні об’єкти крок за кроком. Будівельник дає можливість використовувати один і той самий код будівництва для отримання різних відображень об’єктів.

Патерн Будівельник пропонує винести конструювання об’єкта за межі його власного класу, доручивши цю справу окремим об’єктам, які називаються *будівельниками*.

Патерн пропонує розбити процес конструювання об’єкта на окремі кроки. Щоб створити об’єкт, вам потрібно по черзі викликати методи будівельника. До того ж не потрібно викликати всі кроки, а лише ті, що необхідні для виробництва об’єкта певної конфігурації.

Зазвичай один і той самий крок будівництва може відрізнятися для різних варіацій виготовлених об’єктів. Наприклад, дерев’яний будинок потребує будівництва стін з дерева, а кам’яний — з каменю.

У цьому випадку ви можете створити кілька класів будівельників, які по-різному виконуватимуть ті ж самі кроки. Використовуючи цих будівельників в одному й тому самому будівельному процесі, ви зможете отримувати на виході різні об’єкти.

Застосування

- Коли ви хочете позбутися від «телескопічного конструктора».

Патерн Будівельник дозволяє збирати об’єкти покроково, викликаючи тільки ті кроки, які вам потрібні. Отже, більше не потрібно намагатися «запхати» до конструктора всі можливі опції продукту.

- Коли ваш код повинен створювати різні уявлення якогось об’єкта.

Будівельник можна застосувати, якщо створення кількох відображень об’єкта складається з однакових етапів, які відрізняються деталями.

Інтерфейс будівельників визначить всі можливі етапи конструювання. Кожному відображенню відповідатиме власний клас-будівельник. Порядок етапів будівництва визначатиме клас-директор.

Шаблон «Command»

Команда — це поведінковий патерн проектування, який перетворює запити на об’єкти, дозволяючи передавати їх як аргументи під час виклику методів, ставити запити в чергу, логувати їх, а також підтримувати скасування операцій.

Хороші програми зазвичай структурують у вигляді шарів. Найпоширеніший приклад — це шари користувацького інтерфейсу та бізнес-логіки. Перший лише малює гарне зображення для користувача, але коли потрібно зробити щось важливе, інтерфейс користувача «просить» шар бізнес-логіки зайнятися цим.

У дійсності це виглядає так: один з об’єктів інтерфейсу користувача викликає метод одного з об’єктів бізнес-логіки, передаючи до нього якісь параметри.

Патерн Команда пропонує більше не надсилати такі виклики безпосередньо. Замість цього кожен виклик, що відрізняється від інших, слід звернути у власний клас з єдиним методом, який і здійснюватиме виклик. Такий об’єкт зветься *командою*.

До об’єкта інтерфейсу можна буде прив’язати об’єкт команди, який знає, кому і в якому вигляді слід відправляти запити. Коли об’єкт інтерфейсу буде готовий передати запит, він викличе метод команди, а та — подбає про все інше.

Застосування

- Якщо ви хочете параметризувати об’єкти виконуваною дією.

Команда перетворює операції на об’єкти, а об’єкти, у свою чергу, можна передавати, зберігати та взаємозаміняти всередині інших об’єктів.

- Якщо ви хочете поставити операції в чергу, виконувати їх за розкладом або передавати мережею.
- Якщо вам потрібна операція скасування.

Головна річ, яка потрібна для того, щоб мати можливість скасовувати операції — це зберігання історії. Серед багатьох способів реалізації цієї можливості патерн Команда є, мабуть, найпопулярнішим.

Шаблон «Chain Of Responsibility»

Ланцюжок обов’язків — це поведінковий патерн проектування, що дає змогу передавати запити послідовно ланцюжком обробників. Кожен наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком.

Як і багато інших поведінкових патернів, ланцюжок обов’язків базується на тому, щоб перетворити окремі поведінки на об’єкти. У нашому випадку кожна перевірка переїде до окремого класу з одним методом виконання. Дані запиту, що перевіряється, передаватимуться до методу як аргументи.

Патерн пропонує зв’язати всі об’єкти обробників в один ланцюжок. Кожен обробник міститиме посилання на наступного обробника в ланцюзі. Таким чином, після отримання запиту обробник зможе не тільки опрацювати його самостійно, але й передати обробку наступному об’єкту в ланцюжку.

Передаючи запити до першого обробника ланцюжка, ви можете бути впевнені, що всі об’єкти в ланцюзі зможуть його обробити. При цьому довжина ланцюжка не має жодного значення.

І останній штрих. Обробник не обов’язково повинен передавати запит далі. Причому ця особливість може бути використана різними шляхами.

У прикладі з фільтрацією доступу обробники переривають подальші перевірки, якщо поточну перевірку не пройдено. Адже немає сенсу витрачати даремно ресурси, якщо і так зрозуміло, що із запитом щось не так.

Застосування

- Якщо програма має обробляти різноманітні запити багатьма способами, але заздалегідь невідомо, які конкретно запити надходитимуть і які обробники для них знадобляться.

За допомогою Ланцюжка обов'язків ви можете зв'язати потенційних обробників в один ланцюг і по отриманню запита по черзі питати кожного з них, чи не хоче він обробити даний запит.

- Якщо важливо, щоб обробники виконувалися один за іншим у суворому порядку.

Ланцюжок обов'язків дозволяє запускати обробників один за одним у тій послідовності, в якій вони стоять в ланцюзі.

- Якщо набір об'єктів, здатних обробити запит, повинен задаватися динамічно.

У будь-який момент ви можете втрутитися в існуючий ланцюжок і перевизначити зв'язки так, щоби прибрати або додати нову ланку.

Шаблон «Prototype»

Прототип — це породжувальний патерн проектування, що дає змогу копіювати об'єкти, не вдаючись у подробиці їхньої реалізації.

Патерн Прототип доручає процес копіювання самим об'єктам, які треба скопіювати. Він вводить загальний інтерфейс для всіх об'єктів, що підтримують клонування. Це дозволяє копіювати об'єкти, не прив'язуючись до їхніх конкретних класів. Зазвичай такий інтерфейс має всього один метод — clone.

Реалізація цього методу в різних класах дуже схожа. Метод створює новий об’єкт поточного класу й копіює в нього значення всіх полів власного об’єкта. Таким чином можна скопіювати навіть приватні поля.

Об’єкт, який копіюють, називається *прототипом* (звідси і назва патерна). Коли об’єкти програми містять сотні полів і тисячі можливих конфігурацій, прототипи можуть слугувати своєрідною альтернативою створенню підкласів.

У цьому випадку всі можливі прототипи готуються і налаштовуються на етапі ініціалізації програми. Потім, коли програмі буде потрібний новий об’єкт, вона створить копію з попередньо заготовленого прототипа.

Застосування

- Коли ваш код не повинен залежати від класів об’єктів, призначених для копіювання.

Таке часто буває, якщо ваш код працює з об’єктами, поданими ззовні через який-небудь загальний інтерфейс. Ви не зможете прив’язатися до їхніх класів, навіть якби захотіли, тому що конкретні класи об’єктів невідомі.

Патерн Прототип надає клієнту загальний інтерфейс для роботи з усіма прототипами. Клієнту не потрібно залежати від усіх класів об’єктів, призначених для копіювання, а тільки від інтерфейсу клонування.

- Коли ви маєте безліч підкласів, які відрізняються початковими значеннями полів. Хтось міг створити усі ці класи для того, щоб мати легкий спосіб породжувати об’єкти певної конфігурації.

Патерн Прототип пропонує використовувати набір прототипів замість створення підкласів для опису популярних конфігурацій об’єктів.

Таким чином, замість породження об’єктів з підкласів ви копіюватимете існуючі об’єкти-прототипи, внутрішній стан яких вже налаштовано. Це дозволить уникнути вибухоподібного зростання кількості класів програми й зменшити її складність.

Хід виконання лабораторної роботи

Патерн *Ланцюжок обов'язків* дозволяє обробити запит послідовністю обробників, де кожен обробник відповідає за певний вид дій. Якщо обробник не може обробити запит, він передає його наступному обробнику в ланцюжку.

Цілі патерна

- Розділити відповідальність між різними обробниками.
- Зробити обробників взаємозамінними: у вас є можливість змінювати порядок обробників або їхню кількість без змін у коді клієнта.
- Зменшити залежність коду від конкретних обробників, дозволяючи додавати нові обробники без порушення існуючої логіки.

Кроки реалізації

Крок 1: Створення інтерфейсу IRequestHandler

Щоб реалізувати патерн, створюємо інтерфейс IRequestHandler, який визначає, що повинен робити кожен обробник у ланцюжку. Інтерфейс має метод HandleAsync, який обробляє запит.

```
public interface IRequestHandler
{
    4 usages  3 implementations
    Task HandleAsync(Request request);
    3 implementations
    IRequestHandler? Next { get; set; }
}
```

Рисунок 1. Інтерфейс IRequestHandler

- **Поле** Next: Вказує на наступного обробника в ланцюжку. Завдяки цьому ми можемо вибудувати ланцюг.
- **Метод** HandleAsync: Приймає запит (Request) і виконує певну логіку. Якщо поточний обробник не може повністю виконати дію, він передає запит наступному обробнику через Next.

Крок 2: Визначення класу запиту Request

Об’єкт Request містить дані для обробки запиту, такі як тип ресурсу, ідентифікатор користувача і назва дії.

```
public class Request
{
    2 usages
    public string UserId { get; set; }
    1 usage
    public string Action { get; set; }
    3 usages
    public string ResourceType { get; set; }
    3 usages
    public string ResourceName { get; set; }
    3 usages
    public DateTime? StartDate { get; set; }
    public Guid? ResourceId { get; set; }
}
```

Рисунок 2. Об’єкт Request

Цей клас визначає, які параметри оброблятиме кожен обробник у ланцюгу. У нашому випадку це дані про користувача і тип запиту.

Крок 3: Реалізація обробників, що успадковують IRequestHandler

Обробник AccessControlHandler

Цей обробник перевіряє, чи має користувач достатні права для виконання дії. Якщо користувач не має відповідної ролі, він викидає виняток; якщо має — передає запит далі.

```
public class AccessControlHandler : IRequestHandler
{
    private readonly UserManager<AppUser> _userManager;

    private readonly List<string> _allowedRolesForProject = new List<string> { "Admin", "PM" };
    private readonly List<string> _allowedRolesForTask = new List<string> { "Admin", "PM", "User" };

    1 usage
    public AccessControlHandler(UserManager<AppUser> userManager)
    {
        _userManager = userManager;
    }
```

Рисунок 3. Ініціалізація обробника AccessControlHandler

```
public async Task HandleAsync(Request request)
{
    var user = await _userManager.FindByIdAsync(request.UserId);

    if (user == null)
    {
        throw new UnauthorizedAccessException("User not found.");
    }

    var roles :IList<string> = await _userManager.GetRolesAsync(user);

    if (request.ResourceType == "Project" && !_allowedRolesForProject.Any(role :string => roles.Contains(role)))
    {
        throw new UnauthorizedAccessException
            ("You do not have the required role to perform this action on Project.");
    }

    if (request.ResourceType == "Task" && !_allowedRolesForTask.Any(role :string => roles.Contains(role)))
    {
        throw new UnauthorizedAccessException
            ("You do not have the required role to perform this action on Task.");
    }

    if (Next != null)
    {
        await Next.HandleAsync(request);
    }
}

3 usages
public IRequestHandler? Next { get; set; }
```

Рисунок 4. Методи обробника AccessControlHandler

Обробник UniqueProjectCheckHandler

Цей обробник перевіряє, чи вже існує проєкт з такою ж назвою. Якщо так, він викидає виняток. Інакше — передає запит наступному обробнику.

```
public class UniqueProjectCheckHandler : IRequestHandler
{
    private readonly IProjectRepository _projectRepository;

    1 usage
    public UniqueProjectCheckHandler(IProjectRepository projectRepository)
    {
        _projectRepository = projectRepository;
    }
```

Рисунок 5. Ініціалізація обробника UniqueProjectCheckHandler

```
public async Task HandleAsync(Request request)
{
    var existingProject = await _projectRepository.GetByNameAsync(request.ResourceName);
    if (existingProject != null)
    {
        throw new InvalidOperationException("A project with this name already exists.");
    }

    if (Next != null)
    {
        await Next.HandleAsync(request);
    }
}

3 usages
public IRequestHandler? Next { get; set; }
```

Рисунок 6. Методи обробника UniqueProjectCheckHandler

Обробник ProjectDataValidationHandler

Цей обробник перевіряє валідність дати початку проєкту. Якщо дата в минулому — викидає виняток, інакше — передає запит далі.

```
public class ProjectDataValidationHandler : IRequestHandler
{
    0+4 usages
    public async Task HandleAsync(Request request)
    {
        if (string.IsNullOrEmpty(request.ResourceName))
        {
            throw new ArgumentException("Project name cannot be empty.");
        }

        if (request.StartDate == null || request.StartDate < DateTime.Now)
        {
            throw new ArgumentException("Start date must be a valid date in the future.");
        }

        if (Next != null)
        {
            await Next.HandleAsync(request);
        }
    }

    2 usages
    public IRequestHandler? Next { get; set; }
}
```

Рисунок 6. Обробник ProjectDataValidationHandler

Крок 4: Налаштування ланцюга обробників у ProjectService

У ProjectService ми створюємо обробники і налаштовуємо їх послідовно в ланцюг. Потім цей ланцюг використовується при обробці запиту.

```
public class ProjectService: IProjectService
{
    private readonly IProjectRepository _projectRepository;
    private readonly IRequestHandler _handlerChain;

    public ProjectService(UserManager<AppUser> userManager,
        IProjectRepository projectRepository)
    {
        var accessControlHandler = new AccessControlHandler(userManager);
        var uniqueProjectCheckHandler = new UniqueProjectCheckHandler(projectRepository);
        var projectDataValidationHandler = new ProjectDataValidationHandler();

        accessControlHandler.Next = uniqueProjectCheckHandler;
        uniqueProjectCheckHandler.Next = projectDataValidationHandler;

        _handlerChain = accessControlHandler;
        _projectRepository = projectRepository;
    }
```

```
    public async Task<Project?> CreateProjectAsync(string userId, CreateProjectDTO projectDto)
    {
        var request = new Request
        {
            UserId = userId,
            Action = "Create",
            ResourceType = "Project",
            ResourceName = projectDto.ProjectName,
            StartDate = projectDto.StartDate
        };

        await _handlerChain.HandleAsync(request);

        return await _projectRepository.CreateProjectAsync(projectDto);
    }
```

Рисунок 7. Налаштування ланцюга обробників у ProjectService

Алгоритм роботи ланцюга обробників:

1. Створення запиту. У CreateProjectAsync створюється запит Request з необхідними параметрами.

2. Обробка запиту через ланцюг. Метод HandleAsync першого обробника (AccessControlHandler) викликається з цим запитом.
3. Передача запиту по ланцюгу.
   - AccessControlHandler перевіряє, чи має користувач достатні права.
   - Якщо все добре, обробник передає запит наступному обробнику (UniqueProjectCheckHandler), і так далі.
4. Обробка і завершення. Якщо всі обробники пройшли успішно, повертається результат створення проєкту.

Таким чином, кожен обробник перевіряє лише свою частину логіки, і якщо щось не відповідає вимогам, викидає виняток, не допускаючи виконання запиту далі.

Висновок

У даній лабораторній роботі ми реалізували патерн *Ланцюжок обов'язків*, що дозволило значно підвищити гнучкість і масштабованість коду для обробки запитів при створенні проєктів. Завдяки розділенню функцій обробки запитів на окремі класи-обробники, ми змогли налаштувати послідовну обробку запиту з можливістю додавати або змінювати обробники без змін основного коду програми.

Реалізовані класи

Для реалізації патерна було створено 5 основних класів:

1. **Інтерфейс** IRequestHandler – інтерфейс, що визначає стандарт для всіх обробників у ланцюзі.
2. **Клас** Request – об'єкт запиту, який містить усі необхідні параметри для обробки.
3. **Клас** AccessControlHandler – обробник, що перевіряє права доступу користувача до певних ресурсів.

4. **Клас** UniqueProjectCheckHandler – обробник, що перевіряє унікальність назви проєкту.
5. **Клас** ProjectDataValidationHandler – обробник, що перевіряє коректність дати початку проєкту.

Досягнення від застосування патерна

Завдяки використанню патерна *Ланцюжок обов'язків* ми досягли таких результатів:

- **Гнучкість та масштабованість**: можна легко додавати або видаляти обробники в ланцюзі без зміни базової структури коду, що дозволяє адаптувати обробку запитів до нових вимог.
- **Зниження зв'язності компонентів**: кожен обробник відповідає за окремий аспект обробки, що спрощує підтримку і тестування коду.
- **Збільшення повторного використання коду**: обробники можуть використовуватись у різних ланцюгах обробки запитів, забезпечуючи єдиний підхід до валідації та контролю доступу в різних контекстах.

Отже, використання патерна *Ланцюжок обов'язків* значно покращило архітектуру програми, зробивши її більш структурованою та зручною для підтримки і розширення.

Вихідний код програми можна переглянути в папці до лабораторної роботи №5 на GitHub.