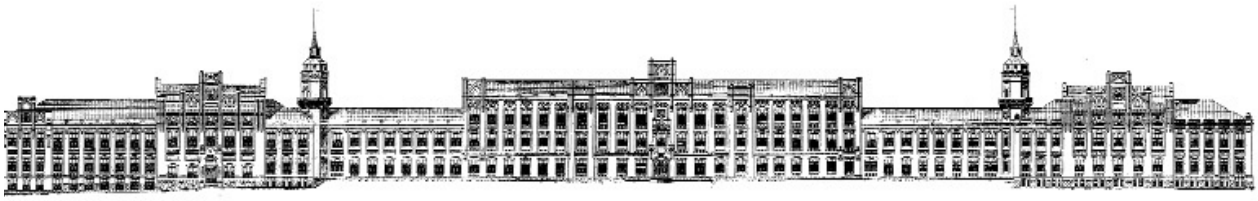

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Звіт

З дисципліни “Технології розроблення програмного забезпечення”

Варіант: 23

Виконала
студентка групи ІА-23:
Павленко Анастасія

Перевірив:
Мягкий Михайло
Юрійович

Київ 2024

Лабораторна робота №6

Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»

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 *4*

Шаблон «Abstract Factory» 4

Шаблон «Factory Method» 5

Шаблон «Memento» 7

Шаблон «Observer» 8

Шаблон «Decorator» 9

Хід виконання лабораторної роботи *11*

Основні компоненти патерну 11

Алгоритм реалізації патерну 11

Висновок *16*

Короткі теоретичні відомості

Шаблон «Abstract Factory»

Абстрактна фабрика — це породжувальний патерн проектування, що дає змогу створювати сімейства пов'язаних об'єктів, не прив'язуючись до конкретних класів створюваних об'єктів.

Для початку, патерн Абстрактна фабрика пропонує виділити загальні інтерфейси для окремих продуктів, що складають одне сімейство, і описати в них спільну для цих продуктів поведінку.

Далі ви створюєте *абстрактну фабрику* — загальний інтерфейс, який містить методи створення всіх продуктів сімейства. Ці операції повинні повертати **абстрактні** типи продуктів, представлені інтерфейсами, які ми виділили раніше.

Для кожної варіації сімейства продуктів ми повинні створити свою власну фабрику, реалізувавши абстрактний інтерфейс. Фабрики створюють продукти однієї варіації.

Клієнтський код повинен працювати як із фабриками, так і з продуктами тільки через їхні загальні інтерфейси. Це дозволить подавати у ваші класи будь-які типи фабрик і виробляти будь-які типи продуктів, без необхідності вносити зміни в існуючий код.

Застосування

- Коли бізнес-логіка програми повинна працювати з різними видами пов'язаних один з одним продуктів, незалежно від конкретних класів продуктів.

Абстрактна фабрика приховує від клієнтського коду подробиці того, як і які конкретно об'єкти будуть створені. Внаслідок цього, клієнтський код може працювати з усіма типами створюваних продуктів, так як їхній загальний інтерфейс був визначений заздалегідь.

- Коли в програмі вже використовується Фабричний метод, але чергові зміни передбачають введення нових типів продуктів.

У будь-якій добротній програмі кожен *клас має відповідати лише за одну річ*. Якщо клас має занадто багато фабричних методів, вони здатні затуманити його основну функцію. Тому є сенс у тому, щоб винести усю логіку створення продуктів в окрему ієрархію класів, застосувавши абстрактну фабрику.

Шаблон «Factory Method»

Фабричний метод — це породжувальний патерн проектування, який визначає загальний інтерфейс для створення об’єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об’єктів.

Патерн Фабричний метод пропонує відмовитись від безпосереднього створення об’єктів за допомогою оператора new, замінивши його викликом особливого *фабричного* методу. Об’єкти все одно будуть створюватися за допомогою new, але робити це буде фабричний метод.

На перший погляд це може здатись безглуздим — ми просто перемістили виклик конструктора з одного кінця програми в інший. Проте тепер ви зможете перевизначити фабричний метод у підкласі, щоб змінити тип створюваного продукту.

Щоб ця система запрацювала, всі об’єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть виготовляти об’єкти різних класів, що відповідають одному і тому самому інтерфейсу.

Застосування

- Коли типи і залежності об’єктів, з якими повинен працювати ваш код, невідомі заздалегідь.

Фабричний метод відокремлює код виробництва продуктів від решти коду, який використовує ці продукти.

Завдяки цьому код виробництва можна розширювати, не зачіпаючи основний код. Щоб додати підтримку нового продукту, вам потрібно створити новий підклас та визначити в ньому фабричний метод, повертаючи звідти екземпляр нового продукту.

- Коли ви хочете надати користувачам можливість розширювати частини вашого фреймворку чи бібліотеки.

Користувачі можуть розширювати класи вашого фреймворку через успадкування. Але як же зробити так, аби фреймворк створював об’єкти цих класів, а не стандартних?

Рішення полягає у тому, щоб надати користувачам можливість розширювати не лише бажані компоненти, але й класи, які їх створюють. Тому ці класи повинні мати конкретні створюючі методи, які можна буде перевизначити.

- Коли ви хочете зекономити системні ресурси, повторно використовуючи вже створені об’єкти, замість породження нових.

Така проблема зазвичай виникає під час роботи з «важкими», вимогливими до ресурсів об’єктами, такими, як підключення до бази даних, файлової системи й подібними.

Уявіть, скільки дій вам потрібно зробити, аби повторно використовувати вже існуючі об’єкти:

1. Спочатку слід створити загальне сховище, щоб зберігати в ньому всі створювані об’єкти.
2. При запиті нового об’єкта потрібно буде подивитись у сховище та перевірити, чи є там невикористаний об’єкт.
3. Потім повернути його клієнтському коду.
4. Але якщо ж вільних об’єктів немає, створити новий, не забувши додати його до сховища.

Увесь цей код потрібно десь розмістити, щоб не засмічувати клієнтський код.

Найзручнішим місцем був би конструктор об’єкта, адже всі ці перевірки потрібні тільки під час створення об’єктів, але, на жаль, конструктор завжди створює **нові** об’єкти, тому він не може повернути існуючий екземпляр. Отже, має бути інший метод, який би віддавав як існуючі, так і нові об’єкти. Ним і стане фабричний метод.

Шаблон «Memento»

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

Патерн Знімок доручає створення копії стану об’єкта самому об’єкту, який цим станом володіє. Замість того, щоб робити знімок «ззовні», наш редактор сам зробить копію своїх полів, адже йому доступні всі поля, навіть приватні.

Патерн пропонує тримати копію стану в спеціальному об’єкті-*знімку* з обмеженим інтерфейсом, що дозволяє, наприклад, дізнатися дату виготовлення або назву знімка. Проте, знімок повинен бути відкритим для свого *творця* і дозволяти прочитати та відновити його внутрішній стан.

Така схема дозволяє творцям робити знімки та віддавати їх на зберігання іншим об’єктам, що називаються *опікунами*. Опікунам буде доступний тільки обмежений інтерфейс знімка, тому вони ніяк не зможуть вплинути на «нутрощі» самого знімку. У потрібний момент опікун може попросити творця відновити свій стан, передавши йому відповідний знімок.

Застосування

- Коли вам потрібно зберігати миттєві знімки стану об’єкта (або його частини) для того, щоб об’єкт можна було відновити в тому самому стані.

Патерн Знімок дозволяє створювати будь-яку кількість знімків об’єкта і зберігати їх незалежно від об’єкта, з якого роблять знімок. Знімки часто використовують не тільки для реалізації операції скасування, але й для транзакцій, коли стан об’єкта потрібно «відкотити», якщо операція не була вдалою.

- Коли пряме отримання стану об’єкта розкриває приватні деталі його реалізації, порушуючи інкапсуляцію.

Патерн пропонує виготовити знімок саме вихідному об’єкту, тому що йому доступні всі поля, навіть приватні.

Шаблон «Observer»

Спостерігач — це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах.

Патерн Спостерігач пропонує зберігати всередині об’єкта видавця список посилань на об’єкти підписників. Причому видавець не повинен вести список підписки самостійно. Він повинен надати методи, за допомогою яких підписники могли б додавати або прибирати себе зі списку.

Коли у видавця відбуватиметься важлива подія, він буде проходитися за списком передплатників та сповіщувати їх про подію, викликаючи певний метод об’єктів-передплатників.

Видавцю байдуже, якого класу буде той чи інший підписник, бо всі вони повинні слідувати загальному інтерфейсу й мати єдиний метод оповіщення.

Побачивши, як добре все працює, ви можете виділити загальний інтерфейс і для всіх видавців, який буде складатися з методів підписки та відписки. Після цього підписники зможуть працювати з різними типами видавців, і отримувати від них сповіщення через єдиний метод.

Застосування

- Якщо після зміни стану одного об’єкта потрібно щось зробити в інших, але ви не знаєте наперед, які саме об’єкти мають відреагувати.

Описана проблема може виникнути при розробленні бібліотек користувацького інтерфейсу, якщо вам необхідно надати можливість стороннім класам реагувати на кліки по кнопках.

Патерн Спостерігач надає змогу будь-якому об’єкту з інтерфейсом підписника зареєструватися для отримання сповіщень про події, що трапляються в об’єктах-видавцях.

- Якщо одні об’єкти мають спостерігати за іншими, але тільки у визначених випадках.

Видавці ведуть динамічні списки. Усі спостерігачі можуть підписуватися або відписуватися від отримання сповіщень безпосередньо під час виконання програми.

Шаблон «Decorator»

Декоратор — це структурний патерн проектування, що дає змогу динамічно додавати об’єктам нову функціональність, загортаючи їх у корисні «обгортки».

Спадкування — це перше, що приходить в голову багатьом програмістам, коли потрібно розширити яку-небудь чинну поведінку. Проте механізм спадкування має кілька прикрих проблем.

- Він **статичний**. Ви не можете змінити поведінку об’єкта, який вже існує. Для цього необхідно створити новий об’єкт, вибравши інший підклас.
- Він **не дозволяє наслідувати поведінку декількох класів одночасно**. Тому доведеться створювати безліч підкласів-комбінацій, щоб досягти поєднання поведінки.

Одним зі способів, що дозволяє обійти ці проблеми, є заміна спадкування *агрегацією* або *композицією* . Це той випадок, коли один об’єкт *утримує* інший і делегує йому роботу, замість того, щоб самому *успадкувати* його поведінку. Саме на цьому принципі побудовано патерн Декоратор.

Декоратор має альтернативну назву — *обгортка*. Вона більш вдало описує суть патерна: ви розміщуєте цільовий об’єкт у іншому об’єкті-обгортці, який запускає базову поведінку об’єкта, а потім додає до результату щось своє.

Обидва об’єкти мають загальний інтерфейс, тому для користувача немає жодної різниці, з чим працювати — з чистим чи загорнутим об’єктом. Ви можете використовувати кілька різних обгорток одночасно — результат буде мати об’єднану поведінку всіх обгорток.

Застосування

- Якщо вам потрібно додавати об’єктам нові обов’язки «на льоту», непомітно для коду, який їх використовує.

Об’єкти вкладаються в обгортки, які мають додаткові поведінки. Обгортки і самі об’єкти мають однаковий інтерфейс, тому клієнтам не важливо, з чим працювати — зі звичайним об’єктом чи з загорнутим.

- Якщо не можна розширити обов’язки об’єкта за допомогою спадкування.

У багатьох мовах програмування є ключове слово final, яке може заблокувати спадкування класу. Розширити такі класи можна тільки за допомогою Декоратора.

Хід виконання лабораторної роботи

Патерн Абстрактна фабрика є структурним патерном проектування, який дозволяє створювати сімейства пов'язаних об'єктів без вказівки їх конкретних класів. У моєму випадку патерн використовується для створення об'єктів **методологій**, які мають спільний інтерфейс IMethodology, але реалізують свої специфічні поведінки.

Основні компоненти патерну:

1. *Абстракція продукту* — спільний інтерфейс для всіх методологій.
2. *Абстракція фабрики* — метод для створення методологій.
3. *Конкретні фабрики* — створюють конкретні методології.
4. *Продукти* — конкретні методології, що реалізують інтерфейс IMethodology.

Алгоритм реалізації патерну:

1. Інтерфейс IMethodology (Абстракція продукту)

Інтерфейс IMethodology визначає загальні методи для всіх методологій, такі як Start, Stop та GetMethodologyName.

```
public interface IMethodology
{
    1 usage  2 implementations
    void Start();
    2 implementations
    void Stop();
    2 implementations
    string GetMethodologyName();
}
```

Рисунок 1. Інтерфейс IMethodology

Цей інтерфейс гарантує, що всі методології, які створюються фабрикою, матимуть однакові методи для початку, зупинки та отримання імені методології. Це дозволяє абстрагуватися від конкретних типів методологій в інших частинах коду.

2. Конкретні реалізації методологій

Реалізації інтерфейсу IMethodology — це конкретні методології, які мають свою специфічну логіку. Наприклад, KanbanMethodology:

```
public class KanbanMethodology : IMethodology
{
    private readonly Guid _projectId;
    1 usage
    public KanbanMethodology(Guid projectId)
    {
        _projectId = projectId;
    }
    0+1 usages
    public void Start()
    {
        Console.WriteLine($"Kanban methodology started for project {_projectId}");
    }
    public void Stop()
    {
        Console.WriteLine($"Kanban methodology stopped for project {_projectId}");
    }
    public string GetMethodologyName() => "Kanban";
}
```

Рисунок 2. Реалізація KanbanMethodology

```
public class ScrumMethodology : IMethodology
{
    private readonly Guid _projectId;
    1 usage
    public ScrumMethodology(Guid projectId)
    {
        _projectId = projectId;
    }
    0+1 usages
    public void Start()
    {
        Console.WriteLine($"Scrum methodology started for project {_projectId}");
    }
    public void Stop()
    {
        Console.WriteLine($"Scrum methodology stopped for project {_projectId}");
    }
    public string GetMethodologyName() => "Scrum";
}
```

Рисунок 3. Реалізація ScrumMethodology

Ці класи реалізують специфічну логіку для кожної методології, але всі вони дотримуються одного інтерфейсу IMethodology.

3. Інтерфейс IMethodologyFactory (Абстракція фабрики)

Фабрика визначає метод для створення конкретних методологій.

```
public interface IMethodologyFactory
{
    1 usage  2 implementations
    IMethodology CreateMethodology(Guid projectId);
}
```

Рисунок 4. Інтерфейс IMethodologyFactory

Цей інтерфейс дає можливість створювати об'єкти методологій для конкретних проектів без прив'язки до їх конкретних класів.

4. Конкретні фабрики

Конкретні фабрики реалізують інтерфейс IMethodologyFactory і створюють конкретні методології, наприклад KanbanFactory та ScrumFactory:

```
public class KanbanFactory : IMethodologyFactory
{
    0+1 usages
    public IMethodology CreateMethodology(Guid projectId)
    {
        return new KanbanMethodology(projectId);
    }
}
```

Рисунок 5. Конкретна фабрика KanbanFactory

```
public class ScrumFactory : IMethodologyFactory
{
    0+1 usages
    public IMethodology CreateMethodology(Guid projectId)
    {
        return new ScrumMethodology(projectId);
    }
}
```

Рисунок 6. Конкретна фабрика ScrumFactory

Ці фабрики інкапсулюють логіку створення конкретних методологій. Завдяки цьому, основний код не має залежностей від конкретних реалізацій методологій, а працює лише з абстракцією фабрики.

5. Методологічний провайдер (MethodologyProvider)

MethodologyProvider є компонентом, який надає фабрики для створення відповідних методологій. Він має словник, в якому зберігаються фабрики для різних типів методологій.

```
public class MethodologyProvider
{
    private readonly Dictionary<string, IMethodologyFactory> _factories;

    public MethodologyProvider()
    {
        _factories = new Dictionary<string, IMethodologyFactory>
        {
            { "Kanban", new KanbanFactory() },
            { "Scrum", new ScrumFactory() }
        };
    }
    1 usage
    public IMethodologyFactory GetFactory(string methodologyName)
    {
        if (!_factories.ContainsKey(methodologyName))
        {
            throw new ArgumentException($"Unsupported methodology: {methodologyName}");
        }

        return _factories[methodologyName];
    }
}
```

Рисунок 7. Методологічний провайдер MethodologyProvider

Метод GetFactory повертає фабрику для конкретної методології на основі її назви. Якщо методологія не підтримується, генерується виключення.

6. Інтерфейс репозиторія IMethodologyRepository

Цей інтерфейс визначає метод для отримання імені методології за її ідентифікатором. Він використовується для того, щоб дізнатися, яка методологія має бути застосована до проекту.

```
public interface IMethodologyRepository
{
    1 usage   1 implementation
    Task<string> GetMethodologyByIdAsync(Guid methodologyId);
}
```

Рисунок 8. Інтерфейс репозиторія IMethodologyRepository

Реалізація цього інтерфейсу MethodologyRepository використовує базу даних для отримання імені методології за її ID.

```
public class MethodologyRepository : IMethodologyRepository
{
    private readonly AppDbContext _context;

    public MethodologyRepository(AppDbContext context)
    {
        _context = context;
    }

    0+1 usages
    public async Task<string> GetMethodologyByIdAsync(Guid methodologyId)
    {
        var methodology =  await _context.Methodologies.FindAsync(methodologyId);
        return methodology.Name;
    }
}
```

Рисунок 9. Реалізація інтерфейсу MethodologyRepository

7. Як працює Абстрактна Фабрика в ProjectService:

```
public async Task<Project?> CreateProjectAsync(string userId, CreateProjectDTO projectDto)
{
    return await ExecuteWithTransactionAsync( operation: async () =>
    {
        var request = BuildRequest(userId, projectDto);

        await _handlerChain.HandleAsync(request);

        var project = await _projectRepository.CreateProjectAsync(projectDto);
        if (project == null)
        {
            throw new InvalidOperationException("Failed to create the project.");
        }

        var methodologyName = await _methodologyRepository.GetMethodologyByIdAsync(projectDto.MethodologyId);
        var factory = _methodologyProvider.GetFactory(methodologyName);
        var methodology = factory.CreateMethodology(project.ProjectId);

        methodology.Start();

        return project;
    });
}
```

Рисунок 9. Реалізація класу ProjectService

У класі ProjectService, коли створюється новий проект, спочатку отримується ім'я методології з репозиторія. Потім за допомогою провайдера

(MethodologyProvider) отримується відповідна фабрика. Далі за допомогою цієї фабрики створюється конкретна методологія. Після цього методологія ініціалізується і запускається.

Цей процес дозволяє вам вибирати і створювати різні типи методологій без прив'язки до їх конкретних класів у логіці створення проектів.

Висновок

В даній лабораторній роботі були розглянуті основні шаблони проектування програмного забезпечення, їх призначення і застосуванням. Також було практично реалізовано патерн Абстрактна фабрика таким чином, що:

- Ми абстрагували створення методологій через інтерфейси і фабрики.
- Замість того, щоб безпосередньо створювати об'єкти методологій у коді, ми отримуємо їх через фабрики, що дозволяє легко додавати нові типи методологій, не змінюючи основну бізнес-логіку.
- Це забезпечує гнучкість і підтримуваність коду, оскільки зміни в способі створення нових методологій потребують лише додавання нових фабрик і зміни провайдера, без необхідності змінювати інші частини програми.