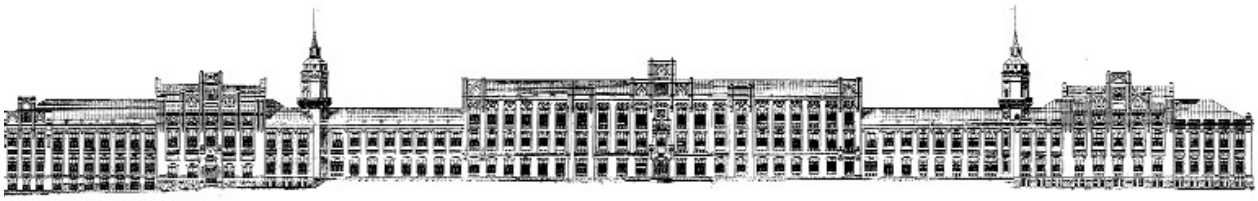

Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Звіт

З дисципліни “Технології розроблення програмного забезпечення”

Варіант: 23

Виконала
студентка групи ІА-23:
Павленко Анастасія

Перевірив:
Мягкий Михайло
Юрійович

Київ 2024

Лабораторна робота №8

Шаблони «Composite», «Flyweight», «Interpreter», «Visitor»

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 «Composite»

Компонувальник — це структурний патерн проектування, що дає змогу згрупувати декілька об’єктів у деревоподібну структуру, а потім працювати з нею так, ніби це одиничний об’єкт.

Патерн Компонувальник має сенс тільки в тих випадках, коли основна модель вашої програми може бути структурована у вигляді дерева.

Наприклад, є два об’єкти — Продукт і Коробка. Коробка може містити кілька Продуктів та інших Коробок меншого розміру. Останні, в свою чергу, також містять або Продукти, або Коробки і так далі. Тепер, припустімо, що ваші Продукти й Коробки можуть бути частиною замовлень. При цьому замовлення може містити як звичайні Продукт без пакування, так і наповнені змістом Коробки. Ваше завдання полягає в тому, щоб дізнатися вартість всього замовлення.

Компонувальник пропонує розглядати Продукт і Коробку через єдиний інтерфейс зі спільним методом отримання ціни. Продукт просто поверне свою вартість, а Коробка запитає про вартість кожного предмета всередині себе і поверне суму результатів. Якщо одним із внутрішніх предметів виявиться трохи менша коробка, вона теж буде перебирати власний вміст, і так далі, допоки не порахується вміст усіх складових частин.

Структура

1. **Компонент** описує загальний інтерфейс для простих і складових компонентів дерева.
2. **Лист** — це простий компонент дерева, який не має відгалужень. Класи листя міститимуть більшу частину корисного коду, тому що їм нікому передавати його виконання.

3. **Контейнер** (або *композит*) — це складовий компонент дерева. Він містить набір дочірніх компонентів, але нічого не знає про їхні типи. Це можуть бути як прості компоненти-листя, так і інші компоненти-контейнери. Проте, це не проблема, якщо усі дочірні компоненти дотримуються єдиного інтерфейсу.

Методи контейнера переадресовують основну роботу своїм дочірнім компонентам, хоча можуть додавати щось своє до результату.

4. **Клієнт** працює з деревом через загальний інтерфейс компонентів.

Завдяки цьому, клієнту не важливо, що перед ним знаходиться — простий чи складовий компонент дерева.

Застосування

- **Якщо вам потрібно представити деревоподібну структуру об’єктів.**

Патерн Компонувальник пропонує зберігати в складових об’єктах посилання на інші прості або складові об’єкти. Вони, у свою чергу, теж можуть зберігати свої вкладені об’єкти і так далі. У підсумку, ви можете будувати складну деревоподібну структуру даних, використовуючи всього два основних різновида об’єктів.

- **Якщо клієнти повинні однаково трактувати прості та складові об’єкти.**

Завдяки тому, що прості та складові об’єкти реалізують спільний інтерфейс, клієнту байдуже, з яким саме об’єктом він працюватиме.

Шаблон «Flyweight»

Легковаговик — це структурний патерн проектування, що дає змогу вмістити більшу кількість об’єктів у відведеній оперативній пам’яті. Легковаговик заощаджує пам’ять, розподіляючи спільний стан об’єктів між собою, замість зберігання однакових даних у кожному об’єкті.

Патерн Легковаговик пропонує не зберігати зовнішній стан у класі, а передавати його до тих чи інших методів через параметри. Таким чином, одні і ті самі об’єкти можна буде повторно використовувати в різних контекстах. Головна ж перевага в тому, що тепер знадобиться набагато менше об’єктів, адже вони тепер відрізнятимуться тільки внутрішнім станом, а він не має так багато варіацій.

Незмінність Легковаговиків

Оскільки об’єкти легковаговиків будуть використані в різних контекстах, ви повинні бути впевненими в тому, що їхній стан неможливо змінити після створення. Весь внутрішній стан легковаговик повинен отримувати через параметри конструктора. Він не повинен мати сеттерів і публічних полів.

Фабрика Легковаговиків

Для зручності роботи з легковаговиками і контекстами можна створити фабричний метод, що приймає в параметрах увесь внутрішній (іноді й зовнішній) стан бажаного об’єкта.

Найбільша користь цього методу в тому, щоб знаходити вже створених легковаговиків з таким самим внутрішнім станом, як потрібно. Якщо легковаговик знаходиться, його можна повторно використовувати. Якщо немає — просто створюємо новий.

Зазвичай цей метод додають до контейнера легковаговиків або створюють окремий клас-фабрику. Його навіть можна зробити статичним і розмістити в класі легковаговиків.

Структура

1. Легковаговик застосовується в програмі, яка має величезну кількість однакових об’єктів. Цих об’єктів повинно бути так багато, щоб вони не вміщалися в доступній оперативній пам’яті без додаткових хитрощів. Патерн розділяє дані цих об’єктів на дві частини — легковаговики та контексти.

2. **Легковаговик** містить стан, який повторювався в багатьох первинних об’єктах. Один і той самий легковаговик може використовуватись у зв’язці з безліччю контекстів. Стан, що зберігається тут, називається *внутрішнім*, а той, який він отримує ззовні, — *зовнішнім*.
3. **Контекст** містить «зовнішню» частину стану, унікальну для кожного об’єкта. Контекст пов’язаний з одним з об’єктів-легковаговиків, що зберігають стан, який залишився.
4. Поведінку оригінального об’єкта найчастіше залишають у легковаговику, передаючи значення контексту через параметри методів. Тим не менше, поведінку можна розмістити й в контексті, використовуючи легковаговик як об’єкт даних.
5. **Клієнт** обчислює або зберігає контекст, тобто зовнішній стан легковаговиків. Для клієнта легковаговики виглядають як шаблонні об’єкти, які можна налаштувати під час використання, передавши контекст через параметри.
6. **Фабрика легковаговиків** керує створенням і повторним використанням легковаговиків. Фабрика отримує запити, в яких зазначено бажаний стан легковаговика. Якщо легковаговик з таким станом вже створений, фабрика відразу його повертає, а якщо ні — створює новий об’єкт.

Застосування

- **Якщо не вистачає оперативної пам’яті для підтримки всіх потрібних об’єктів.**

Ефективність патерна **Легковаговик** багато в чому залежить від того, як і де він використовується. Застосовуйте цей патерн у випадках, коли виконано всі перераховані умови:

- у програмі використовується велика кількість об’єктів;
- через це високі витрати оперативної пам’яті;

- більшу частину стану об’єктів можна винести за межі їхніх класів;
- великі групи об’єктів можна замінити невеликою кількістю об’єктів, що розділяються, оскільки зовнішній стан винесено.

Шаблон «Interpreter»

Інтерпретатор — шаблон проєктування, належить до класу шаблонів поведінки.

Шаблон Інтерпретатор слід використовувати, коли є мова для інтерпретації, речення котрої можна подати у вигляді абстрактних синтаксичних дерев. Найкраще шаблон працює коли:

- граматика проста. Для складних граматик ієрархія класів стає занадто громіздкою та некерованою. У таких випадках краще застосовувати генератори синтаксичних аналізаторів, оскільки вони можуть інтерпретувати вирази, не будуючи абстрактних синтаксичних дерев, що заощаджує пам'ять, а можливо і час;
- ефективність не є головним критерієм. Найефективніші інтерпретатори зазвичай не працюють безпосередньо із деревами, а спочатку транслюють їх в іншу форму. Так, регулярний вираз часто перетворюють на скінченний автомат. Але навіть у цьому разі сам транслятор можна реалізувати за допомогою шаблону інтерпретатор.

Структура

1. **AbstractExpression** — абстрактний вираз:
   - оголошує абстрактну операцію *Interpret*, загальну для усіх вузлів у абстрактному синтаксичному дереві;
2. **TerminalExpression** — термінальний вираз:
   - реалізує операцію *Interpret* для термінальних символів граматики;

- необхідний окремий екземпляр для кожного термінального символу у реченні;

3. **NonterminalExpression** — нетермінальний вираз:
   - по одному такому класу потребується для кожного граматичного правила;
   - зберігає змінні екземпляру типу *AbstractExpression* для кожного символу;
   - реалізує операцію *Interpret* для нетермінальних символів граматики. Ця операція рекурсивно викликає себе для змінних, зберігаючих символи;
4. **Context** — контекст:
   - містить інформацію, глобальну по відношенню до інтерпретатору;
5. **Client** — клієнт:
   - будує (або отримує у готовому вигляді) абстрактне синтаксичне дерево, репрезентуюче окреме речення мовою з даною граматикою. Дерево складено з екземплярів класів *NonterminalExpression* та *TerminalExpression*;
   - викликає операцію *Interpret*.

Шаблон «Visitor»

Відвідувач — це поведінковий патерн проектування, що дає змогу додавати до програми нові операції, не змінюючи класи об’єктів, над якими ці операції можуть виконуватися.

Патерн Відвідувач пропонує розмістити нову поведінку в окремому класі, замість того, щоб множити її відразу в декількох класах. Об’єкти, з якими повинна бути пов’язана поведінка, не виконуватимуть її самостійно. Замість цього ви будете передавати ці об’єкти до методів відвідувача.

Код поведінки, імовірно, повинен відрізнятися для об’єктів різних класів, тому й методів у відвідувача повинно бути декілька. Назви та принцип дії цих методів будуть подібними, а основна відмінність торкатиметься типу, що приймається в параметрах об’єкта.

Структура

1. **Відвідувач** описує спільний для всіх типів відвідувачів інтерфейс. Він оголошує набір методів, що відрізняються типом вхідного параметра. Кожному класу конкретних елементів повинен підходити свій метод. В мовах, які підтримують перевантаження методів, ці методи можуть мати однакові імена, але типи їхніх параметрів повинні відрізнятися.
2. **Конкретні відвідувачі** реалізують якусь особливу поведінку для всіх типів елементів, які можна подати через методи інтерфейсу відвідувача.
3. **Елемент** описує метод прийому відвідувача. Цей метод повинен мати лише один параметр, оголошений з типом загального інтерфейсу відвідувачів.
4. **Конкретні елементи** реалізують методи приймання відвідувача. Мета цього методу — викликати той метод відвідування, який відповідає типу цього елемента. Так відвідувач дізнається, з яким типом елементу він працює.
5. **Клієнтом** зазвичай виступає колекція або складний складовий об’єкт. Здебільшого, клієнт не прив’язаний до конкретних класів елементів, працюючи з ними через загальний інтерфейс елементів.

Застосування

- Якщо вам потрібно виконати якусь операцію над усіма елементами складної структури об’єктів, наприклад, деревом.

Відвідувач дозволяє застосовувати одну і ту саму операцію до об’єктів різних класів.

- Якщо над об’єктами складної структури об’єктів потрібно виконувати деякі не пов’язані між собою операції, але ви не хочете «засмічувати» класи такими операціями.

Відвідувач дозволяє витягти споріднені операції з класів, що складають структуру об’єктів, помістивши їх до одного класу-відвідувача. Якщо структура об’єктів використовується в декількох програмах, то патерн дозволить кожній програмі мати тільки потрібні в ній операції.

- Якщо нова поведінка має сенс тільки для деяких класів з існуючої ієрархії.

Відвідувач дозволяє визначити поведінку тільки для цих класів, залишивши її порожньою для всіх інших.

Хід виконання лабораторної роботи

Основною метою лабораторної роботи є дослідження та практичне застосування патерна проєктування Flyweight для оптимізації використання пам’яті та підвищення продуктивності системи управління командами в проєкті.

Задачі, які необхідно виконати:

1. Оптимізація збереження ролей користувачів — уникнення дублювання об'єктів ролей шляхом збереження унікальних екземплярів ролей для всіх членів команди.
2. Ефективне управління командами — забезпечення механізмів створення, оновлення, видалення та роботи з командами та членами команд у контексті збереження та перевикористання спільних ролей.
3. Оптимізація пам’яті — за допомогою патерна Flyweight, система створює один об'єкт ролі для кожного унікального набору параметрів (назва ролі, опис і рівень доступу), замість створення окремих об'єктів для кожного члена команди.

Основні учасники патерна Flyweight:

1. **Легковаговик (Flyweight)** — об'єкт, що зберігає спільний стан і містить спільну поведінку.
2. **Контекст (Context)** — об'єкт, що зберігає зовнішній стан та використовує легковаговик.
3. **Фабрика легковаговиків (Flyweight Factory)** — контролює створення та повторне використання легковаговиків.
4. **Клієнт (Client)** — створює контексти та взаємодіє з легковаговиками.

Алгоритм реалізації патерна Flyweight

1. Створення легковаговика

Об'єкт Role зберігає стан, який є спільним для багатьох об'єктів.

```
public class Role
{
    [Key]
    public Guid RoleId { get; set; }
    [Required]
    [StringLength(50)]
    10 usages
    public string Name { get; set; }
    1 usage
    public string Description { get; set; }
    [Required]
    10 usages
    public AccessLevel AccessLevel { get; set; }
    private Role() { }
    3 usages
    public Role(string name, string description, AccessLevel accessLevel)
    {
        Name = name;
        Description = description;
        AccessLevel = accessLevel;
    }
}
```

Рисунок 1. Легковаговик – клас Role

2. Створення фабрики ролей

Клас RoleFlyweightFactory містить словник, де ключем є унікальна комбінація параметрів ролі (назва, опис, рівень доступу).

Якщо потрібної ролі ще немає у словнику, створюється новий екземпляр ролі та додається до словника.

Якщо роль уже існує (навіть якщо вона була додана з іншими регістрами літер, наприклад "team lead" і "Team Lead"), використовується вже існуючий екземпляр із Flyweight-словника.

```
public class RoleFlyweightFactory
{
    private readonly Dictionary<string, Role> _roles =
        new Dictionary<string, Role>(StringComparer.OrdinalIgnoreCase);

    9 usages
    public Role GetRole(string name, string description, AccessLevel accessLevel)
    {
        var key = $"{name}_{(int)accessLevel}";
        if (!_roles.TryGetValue(key, out var role))
        {
            role = new Role(name, description, accessLevel);
            _roles[key] = role;
        }
        return role;
    }
    1 usage
    public Role GetDefaultRole()
    {
        return GetRole( name: "Developer",  description: "Default role for team members",
            AccessLevel.TaskExecution);
    }
```

Рисунок 2. Клас RoleFlyweightFactory

3. Попереднє завантаження типових ролей

За допомогою методу PreloadRoles у фабриці заздалегідь створюються стандартні ролі (наприклад, Administrator, Scrum Master, Team Lead, Developer тощо).

Це дозволяє уникнути необхідності створювати ці ролі під час виконання програми, що підвищує швидкість створення команд.

```
public void PreloadRoles()
{
    GetRole( name: "Administrator", description: "Admin role with full permissions",
        AccessLevel.FullAccess);
    GetRole( name: "Scrum Master", description: "Manages iterations and planning",
        AccessLevel.IterationManagement);
    GetRole( name: "Team Lead", description: "Manages team tasks",
        AccessLevel.TaskManagement);
    GetRole( name: "Developer", description: "Executes tasks",
        AccessLevel.TaskExecution);
    GetRole( name: "Restricted", description: "Limited access role",
        AccessLevel.RestrictedAccess);
}
```

Рисунок 3. Метод PreloadRoles

4. Створення контексту

Об'єкт контексту зберігає унікальний для кожного члена команди стан, але спільну роль отримує з RoleFlyweightFactory.

```
public class TeamMember
{
    [Key]
    5 usages
    public Guid TeamMemberId { get; set; }
    [Required]
    22 usages
    public Guid UserId { get; set; }
    [ForeignKey( name: "UserId")]
    60 usages
    public AppUser User { get; set; }
    [ForeignKey( name: "RoleId")]
    public Guid RoleId { get; set; }
    23 usages
    public Role Role { get; set; }
```

Рисунок 4. Контекст - TeamMember

5. Інтеграція Flyweight у роботу з командами

При створенні команди або додаванні нового учасника до команди, через фабрику отримується роль для користувача на основі параметрів (назва, опис, рівень доступу).

У разі, якщо користувач спробує створити роль із тією ж назвою та параметрами, Flyweight поверне вже існуючу роль зі словника, а не створить новий екземпляр, зберігаючи пам'ять.

```
public class CreateTeamService
{
    private readonly ITeamRepository _teamRepository;
    private readonly IUserRepository _userRepository;
    private readonly ITaskRepository _taskRepository;
    private readonly IProjectRepository _projectRepository;

    public CreateTeamService(ITeamRepository teamRepository, IUserRepository userRepository,
        ITaskRepository taskRepository, IProjectRepository projectRepository)
    {
        _teamRepository = teamRepository;
        _userRepository = userRepository;
        _taskRepository = taskRepository;
        _projectRepository = projectRepository;
    }
```

Рисунок 3. Ініціалізація класу CreateTeamService

```
    public async Task<ResultT<TeamResponse>> CreateTeamAsync(CreateTeamRequest request)
{
    var roleFactory = new RoleFlyweightFactory();
    roleFactory.PreloadRoles();

    var project = await ValidateAndGetProjectAsync(request.ProjectId);
    var teamLead :AppUser = await ValidateAndGetTeamLeadAsync(request.TeamLeadId);
    var users :List<AppUser> = await ValidateAndGetUsersAsync(request.MemberIds);
    ValidateRequest(request, users);

    var team = CreateTeam(request, teamLead, users, roleFactory);

    await _teamRepository.AddAsync(team);

    var response = MapToResponse(team);
    return ResultT<TeamResponse>.Success(response);
}
```

Рисунок 4. Метод CreateTeamAsync

6. Обробка запитів

Під час створення команди система отримує від користувача параметри ролей кожного члена команди.

Якщо користувач вводить роль із різними регістрами (наприклад, "Team Lead" та "team lead"), фабрика використовує нормалізацію ключа (перетворення до нижнього регістру) для пошуку відповідного екземпляра.

Результати використання патерна «Flyweight»

- **Економія пам'яті**: Завдяки патерну Flyweight у системі зберігається лише один екземпляр для кожної унікальної ролі, що дозволяє уникнути дублювання об'єктів для кожного члена команди.
- **Оптимізація роботи системи**: Час створення ролі скорочується, оскільки система перевіряє наявність ролі у фабриці Flyweight перед створенням нової ролі.
- **Уніфікованість роботи з ролями**: Усі учасники команди з однаковою роллю посилаються на один об'єкт ролі, що спрощує керування та зміну параметрів цієї ролі (наприклад, оновлення прав доступу для всіх членів команди).

Висновок

У ході виконання лабораторної роботи ми ознайомилися з теоретичними основами та практичним застосуванням шаблонів проєктування **Composite**, **Flyweight**, **Interpreter** та **Visitor**. Ці патерни належать до різних категорій та вирішують конкретні завдання у проєктуванні програмного забезпечення. Найбільшу увагу було приділено патерну **Flyweight**, який ми успішно реалізували у нашому проєкті з управління командами та ролями користувачів. Основною метою використання цього патерна є оптимізація пам’яті за рахунок розділення стану об'єктів на спільний (внутрішній) та унікальний (зовнішній).

Практичне застосування патерна Flyweight:

- Реалізовано клас **Role** (легковаговик), який зберігає спільний стан (назва ролі, рівень доступу та опис) для багатьох об'єктів.

- Реалізовано клас **RoleFlyweightFactory**, який відповідає за створення та повторне використання об'єктів ролей. Завдяки використанню словника (Dictionary), фабрика знаходить та повертає вже існуючий об'єкт або створює новий, якщо його ще немає в колекції.
- Створено клас **TeamMember** (контекст), який зберігає унікальний стан (ID члена команди, ID користувача, посилання на об'єкт ролі) та використовує спільний об'єкт ролі.
- У **сервісі управління командами** при створенні нових членів команди використовується фабрика Flyweight для повторного використання існуючих об'єктів ролей. Завдяки цьому оптимізується споживання пам’яті та підвищується ефективність роботи додатка.

Повний код можна переглянути у відповідній папці лабораторної роботи №8 на GitHub.

Застосування патернів дозволяє створювати більш ефективний, гнучкий та легко масштабований код, що підвищує якість та продуктивність програмного забезпечення.