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

Лабораторна робота № 4

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Вступ до паттернів проектування.»

«Менеджер завдань»

Виконав:
студент групи - ІА-32
Воробйов Кирило Андрійович

Перевірив:
Мягкий Михайло Юрійович

Київ 2025

Тема: Вступ до паттернів проектування..

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи.

Тема проєкту: Менеджер завдань (To Do Manager)

Патерни: Strategy, Command, Observer, Mediator, Composite, Client-Server

Опис: Додаток повинен мати можливість створювати, редагувати та видаляти завдання, забезпечувати гнучку фільтрацію та сортування (Strategy), та обробляти дії користувача (Command). Система має підтримувати ієрархічну структуру (завдання/підзадачі або проєкт/завдання) (Composite), забезпечувати автоматичне оновлення візуальних компонентів при зміні даних (Observer) та керувати взаємодією елементів інтерфейсу (Mediator). Зберігання та синхронізація завдань реалізується через клієнт-серверний зв'язок (Client-Server).

Зміст

Теоретичні відомості

Поняття шаблону проєктування

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях [5]. Крім того, патерн проєктування обов'язково має загальновживане найменування.

Правильно сформульований патерн проєктування дозволяє, відшукавши одного разу вдале рішення, користуватися ним знову і знову. Варто підкреслити, що важливим початковим етапом при роботі з патернами є адекватне моделювання розглянутої предметної області. Це є необхідним як для отримання належним чином формалізованої постановки задачі, так і для вибору відповідних патернів проєктування.

Відповідне використання патернів проєктування дає розробнику ряд незаперечних переваг. Наведемо деякі з них. Модель системи, побудована в межах патернів проєктування, фактично є структурованим виокремленням тих елементів і зв'язків, які значимі при вирішенні поставленого завдання. Крім цього, модель, побудована з використанням патернів проєктування, більш проста і наочна у вивченні, ніж стандартна модель.

Проте, не дивлячись на простоту і наочність, вона дозволяє глибоко і всебічно опрацювати архітектуру розроблюваної системи з використанням спеціальної мови.

Застосування патернів проєктування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Крім того, важко переоцінити роль використання патернів при інтеграції інформаційних систем організації. Також слід зазначити, що сукупність патернів проєктування, по суті, являє собою єдиний словник проєктування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним.

Таким чином шаблони представляють собою, підтверджені роками розробок в різних компаніях і на різних проєктах, «ескізи» архітектурних рішень, які зручно застосовувати у відповідних обставинах.

Шаблон «Singleton»

Призначення патерну: «Singleton» (Одинак) являє собою клас в термінах ООП, який може мати не більше одного об'єкта (звідси і назва «одинак») [6]. Насправді, кількість об'єктів можна задати (тобто не можна створити більш n об'єктів даного класу). Даний об'єкт найчастіше зберігається як статичне поле в самому класі.

Проблема: Використання одинака виправдано для наступних випадків:

- може бути не більше N фізичних об'єктів, що відображаються в певних класах;
- необхідно жорстко контролювати всі операції, що проходять через даний клас.

Одинак вирішує відразу дві проблеми, порушуючи принцип єдиної відповідальності класу.

Рішення:

Перший випадок легко продемонструвати. У кожної програми є певний набір налаштувань, який як правило зберігається в окремий файл (для сучасних комп'ютерних ігор це може бути .ini файл, для .net додатків –.xml файл). Цей файл – єдиний, і тому використання безлічі об'єктів для завантаження/запису даних – нераціональне рішення.

Для демонстрації другого випадку, розглянемо наступний приклад. Припустимо, існує дві взаємодіючі системи, між якими встановлено сеанс зв'язку. Накладені обмеження, що по даному сеансу зв'язку дані можуть йти в один момент часу лише в одну сторону. Таким чином, на кожен надісланий запит необхідно дочекатися відповіді, перш ніж відсилати новий запит. Об'єкт «одинак» дозволить не тільки містити рівно один сеанс зв'язку, а й ще реалізувати відповідну логіку перевірок на основі bool операторів про можливість відправки запиту і, можливо, деяку чергу запитів.

Переваги та недоліки: Однак слід зазначити, що в даний час патерн «Одинак» багато хто вважає т.зв. «анти-шаблоном», тобто поганою практикою проєктування. Це пов'язано з тим, що «одинаки» представляють собою глобальні дані (як глобальна змінна), що мають стан. Стан глобальних об'єктів важко відслідковувати і підтримувати коректно; також глобальні об'єкти важко тестуються і вносять складність в програмний код (у всіх ділянках коду виклик в одне єдине місце з «одинаком»; при зміні підходу доведеться змінювати масу коду).

При цьому реалізація контролю доступу можлива за допомогою статичних змінних, замикань, мютексов та інших спеціальних структур.

+ Гарантує наявність єдиного екземпляра класу.

+ Надає до нього глобальну точку доступу.

- Порушує принцип єдиної відповідальності класу.

- Маскує поганий дизайн.

Шаблон «Iterator»

Призначення: «Iterator» (Ітератор) являє собою шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків:

колекція відповідає за зберігання даних, ітератор – за прохід по колекції [6].

При цьому алгоритм ітератора може змінюватися – при необхідності пройти в зворотньому порядку використовується інший ітератор. Можливо також написання такого ітератора, який проходить список спочатку по парних позиціях (2,4,6-й елементи і т.д.), потім по непарних. Тобто, шаблон ітератор дозволяє реалізовувати різноманітні способи проходження по колекції незалежно від виду і способу представлення даних в колекції.

Проблема: Більшість колекцій виглядають як звичайний список елементів. Але є й екзотичні колекції, побудовані на основі дерев, графів та інших складних структур даних.

Але як би не була структурована колекція, користувач повинен мати можливість послідовно обходити її елементи, щоб виробляти з ними якісь дії.

Але яким способом слід переміщатися по складній структурі даних? Наприклад, сьогодні може бути достатнім обхід дерева в глибину, але завтра буде потрібно можливість переміщатися по дереву в ширину. А на наступному тижні і того гірше – знадобиться обхід колекції у випадковому порядку.

Додаючи все нові алгоритми в код колекції, ви потроху розмиваєте її основне завдання, яке полягає в ефективному зберіганні даних.

Рішення: Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий клас.

Об'єкт-ітератор буде відстежувати стан обходу, поточну позицію в колекції і скільки елементів ще залишилося обійти. Одну і ту ж колекцію зможуть одночасно обходити різні ітератори, а сама колекція не буде навіть знати про це.

До того ж, якщо вам знадобиться додати новий спосіб обходу, ви зможете створити окремий клас ітератора, не змінюючи існуючий код колекції.

Шаблонний ітератор містить:

- First() – установка покажчика перебору на перший елемент колекції;
- Next() – установка покажчика перебору на наступний елемент колекції;
- IsDone – булевське поле, яке встановлюється як true коли покажчик перебору досяг кінця колекції;
- CurrentItem – поточний об'єкт колекції.

Переваги та недоліки: Цей шаблон дозволяє уніфікувати операції проходження по наборам об'єктів для всіх наборів. Тобто, незалежно від реалізації (масив, зв'язаний список, незв'язаний список, дерево та ін.), кожен з наборів може використовувати будь-який з реалізованих ітераторів.

+ Дозволяє реалізувати різні способи обходу структури даних.

+ Спрощує класи зберігання даних.

- Не виправданий, якщо можна обійтися простим циклом.

Шаблон «Proxy»

Призначення: «Proxy» (Проксі) – об'єкти є об'єктами-заглушками або двійниками/замінниками для об'єктів конкретного типу. Зазвичай, проксі об'єкти вносять додатковий функціонал або спрощують взаємодію з реальними об'єктами [5].

Проксі об'єкти використовувалися в більш ранніх версіях інтернетбраузерів, наприклад, для відображення картинки: поки

картинка завантажується, користувачеві відображається «заглушка» картинки.

Проблема: Ви супроводжуєте систему, одна із частин якої працює з зовнішнім сервісом підписання документів, нарпиклад, DocuSign. Періодично клієнти в вашій системі формують звіти и форматі pdf, далі ваша система відправляє їх на підпис в сервіс DocuSign і потім періодично перевіряєте чи документ вже підписаний. Якщо документ підписаний, ви викачуєте його з сервісу і поміщаєте в своїй базі.

За останній рік кількість користувачів вашої системи виросло суттєво і почали приходити великі рахунки від DocuSign. Після аналізу ви розумієте, що рахунки зросли через велику кількість запитів з відправкою документів на підписання та запитів на перевірку чи документ вже підписаний. Після обговорення з бізнес-аналітиками та користувачам зрозуміли, що критичний інтервал доставки документів на підписання – 2 години і такий самий час критичності перевірки що документ підписаний і можна було б групувати всі запит на відправку та на отримання і відправляти пакетом раз на годину. На даний момент клієнтський код працює з класом DocSignManager через інтерфейс IDocSignManager.

Рішення: Для вирішення проблеми можна застосувати патерн "Замісник". Ви реалізовуєте клас замісник, який також реалізовує інтерфейс IDocSignManager, але він накопичує запити на відправку файлів на підписання і відправляє їх раз на годину, також він приблизно раз на годину отримує підписані документи, а на запити від клієнтів відповідає на основі інформації взятої з бази. Таким чином старий клас DocSignManager так само використовується для роботи з DocuSign сервісом, але вже набагато рідже, а клієнтський код взаємодіє з додатковим проміжним рівнем

DocSignManagerProxy, хоча з точки зору клієнського коду нічого не змінилося і він працює з тим самим об'єктом IDocSignManager.

Реалізувавши такий підхід ви тепер економите 40% від попередньої вартості використання DocuSign сервіса і тепер основний вплив на вартість робить розмір файлів, що передаються на підпис, а не кількість запитів до служби.

Переваги та недоліки:

+ Легкість впровадження проміжного рівня без переробки клієнтського коду.

+ Додаткові можливості по керуванню життєвим циклом об’єкту.

- Існує ризик падіння швидкості роботи через впровадження додаткових операцій.
- Існує ризик неадекватної заміни відповіді клієнтському коду

Шаблон «State»

Призначення: Шаблон «State» (Стан) дозволяє змінювати логіку роботи об'єктів у випадку зміни їх внутрішнього стану [6]. Наприклад, відсоток нарахованих на картковий рахунок грошей залежить від стану картки: Visa Electron, Classic, Platinum і т.д. Або обсяг послуг, які надані хостинг компанією, змінюється в залежності від обраного тарифного плану (стану членства – бронзовий, срібний або золотий клієнт). Реалізація даного шаблону полягає в наступному: пов'язані зі станом поля, властивості, методи і дії виносяться в окремий загальний інтерфейс (State); кожен стан являє собою окремий клас

(ConcreteStateA, ConcreteStateB), які реалізують загальний інтерфейс [6, 8].

Об’єкти, що мають стан (Context), при зміні стану просто записують новий об'єкт в поле state, що призводить до повної зміни поведінки об'єкта.

Це дозволяє легко додавати в майбутньому і обробляти нові стани, відокремлювати залежні від стану елементи об'єкта в інших об'єктах, і відкрито проводити заміну стану (що має сенс у багатьох випадках).

Проблема: Ви розробляєте систему, яка складається з мобільних клієнтів та центрального сервера.

Для отримання запитів від клієнтів ви створюєте модуль Listener. Але вам потрібно щоб під час старту, поки сервер не запустився Listener не приймав запити від клієнтів, а після команди shutdown, поки сервер зупиняється, Listener вже не приймав запити від клієнтів, але відповіді, якщо вони будуть готові, відправив.

Рішення: Тут явно видно три стани для Listener з різною поведінкою:

initializing, open, closing. Тому для вирішення краще за все підходить патерн

State.

Визначаємо загальний інтерфейс IListenerState з методами, які по різному працюють в різних станах. Під кожний стан створюємо клас з

реалізацією цього інтерфейсе. Контекст Listener при цьому всі виклики буде перенаправляти на об'єкт стану простим делегуванням. При старті він (Listener) буде посилатися на об'єкт стану InitializingState, знаходячись в якому Listener, фактично, буде ігнорувати всі вхідні запити. Після того як система запуститься і завантажить всі базові дані для роботи, Listener буде переключено в робочий стан, наприклад, викликом методу Open(). Після цього Listener буде посилатися на об'єкт OpenState і буде відпрацьовувати всі вхідні повідомлення в звичайному режимі. При запуску виключення системи, Listener буде переключено в стан ClosingState, викликом методу Close().

Переваги та недоліки:

+ Код специфічний для окремого стану реалізується в класі стану.

+ Класи та об’єкти станів можна використовувати з різними контекстами, за рахунок чого збільшується гнучкість системи.

+ Код контексту простіше читати, тому що вся залежна від станів логіка винесена в інші класи.

+ Відносно легко додавати нові стани, головне правильно змінити переходи між станами.

- Клас контекст стає складніше через ускладнений механізм переключення станів.

Шаблон «Strategy»

Призначення: Шаблон «Strategy» (Стратегія) дозволяє змінювати деякий алгоритм поведінки об'єкта іншим алгоритмом, що досягає ту ж мету іншим способом. Прикладом можуть служити алгоритми сортування: кожен алгоритм має власну реалізацію і визначений в окремому класі; вони можуть бути взаємозамінними в об'єкті, який їх використовує [6].

Даний шаблон дуже зручний у випадках, коли існують різні «політики» обробки даних. По суті, він дуже схожий на шаблон «State» (Стан), проте використовується в абсолютно інших цілях – незалежно від стану об'єкта відобразити різні можливі поведінки об'єкта (якими досягаються одні й ті самі або схожі цілі).

Рішення: Коли ви використовуєте патерн «Стратегія», то схожі алгоритми виносяться з класа контекста в конкретні стратегії, за рахунок чого клас контексту стає чистіше і його легше супроводжувати. Також одні і тіж самі стратегії можна використати з різними контекстами, що значно збільшує гнучкість вашої системи та зменшує кількість дублювань у коді.

Контекст при цьому містить посилання на конкретну стратегію, а коли стратегію потрібно замінити, то замінюється об’єкт стратегії в полі

Context.strategy.

Важливою умовою є відносна простота інтерфейсу для алгоритмів стратегій. Якщо алгоритмам стратегій прийдеться передавати десятки параметрів, то це, скоріш за все, приведе до ускладнення системи та заплутаності коду. Якщо стратегії на вході будуть приймати об’єкт контексту, щоб отримувати з нього всі необхідні дані, то такі стратегії будуть прив’язані до конкретного контексту і їх не можна буде використати з іншим типом контексту.

Приклад з життя: Ви їдете на роботу. Можна доїхати на автомобілі, на метро, або йти пішки. Тут алгоритм, як ви добираєтеся на роботу, є стратегією. В залежності від поточної ситуації ви вибираєте стратегію, що найбільше підходить в цій ситуації, наприклад, на дорогах великі пробки тоді ви їдете на метро, або метро тимчасово не ходить тоді ви їдете на таксі, або ви знаходитеся в 5 хвилинах ходьби від місця роботи і простіше добратися пішки.

Переваги та недоліки:

+ Використовувані алгоритми можна змінювати під час виконання.

+ Реалізація алгоритмів відокремлюється від коду, що його використовує.

+ Зменшує кількість умовних операторів типу switch та if в контексті.

- Надмірна складність, якщо у вас лише кілька невеликих алгоритмів.

- Під час виклику алгоритму, клієнтський код має враховувати різницю між стратегіями.

Хід роботи

Завдання:

- Ознайомитись з короткими теоретичними відомостями.
- Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
- Реалізувати один з розглянутих шаблонів за обраною темою.
- Реалізувати не менше 3-х класів відповідно до обраної теми.
- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

Діаграма класів патерну Strategy

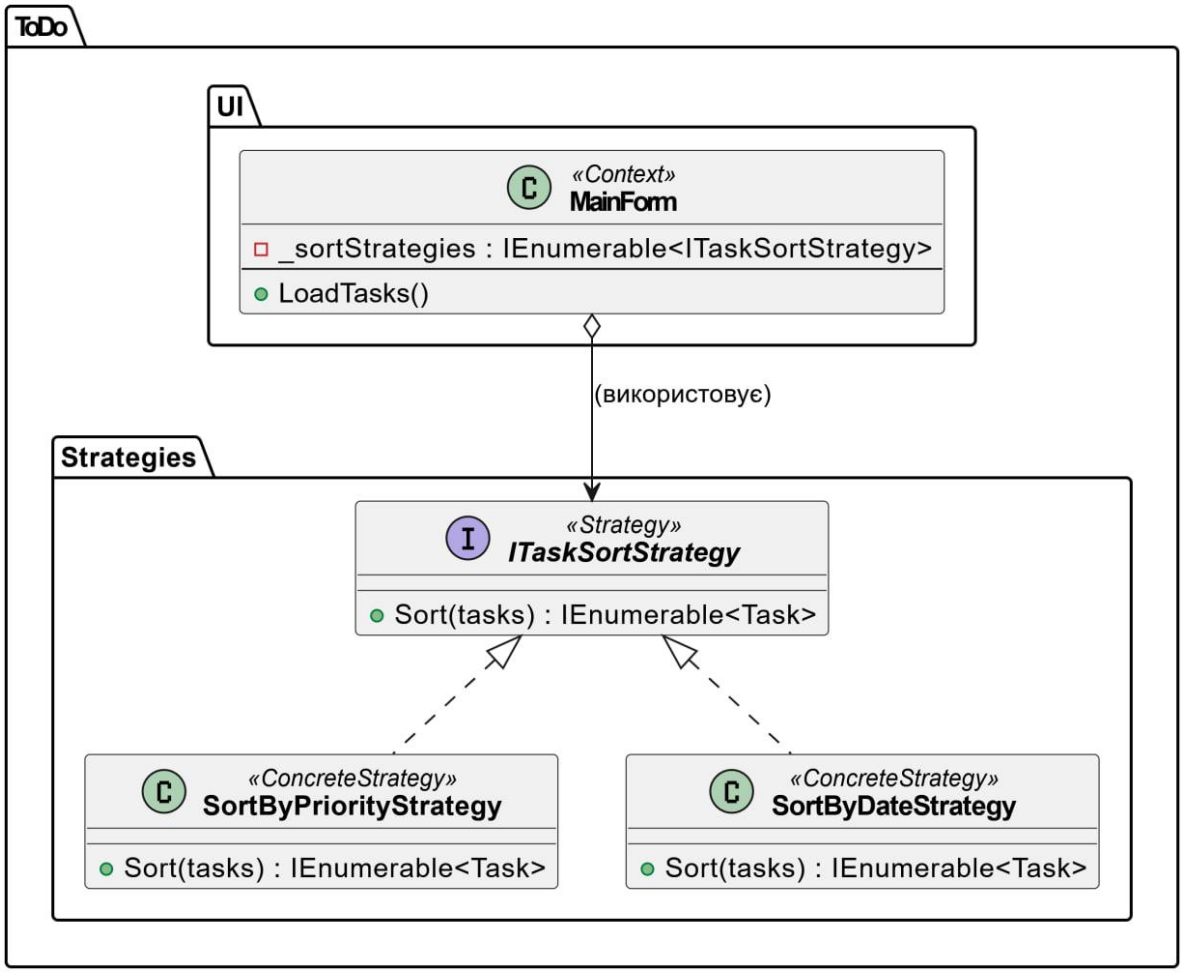


Рис. 1. Діаграма класів патерну Strategy

Код програми

ITaskSortStrategy.cs

```
using System.Collections.Generic;
using ToDo.Models;
using Task = ToDo.Models.Task;

namespace ToDo.Strategies
{
    // Це наш інтерфейс "Strategy"
    public interface ITaskSortStrategy
    {
        IEnumerable<Task> Sort(IEnumerable<Task> tasks);
    }
}
```

SortByPriorityStrategy.cs

```
using System.Collections.Generic;
using System.Linq;
using ToDo.Models;
using Task = ToDo.Models.Task;

namespace ToDo.Strategies
{
    public class SortByPriorityStrategy : ITaskSortStrategy
    {
        public IEnumerable<Task> Sort(IEnumerable<Task> tasks)
        {
            return tasks.OrderByDescending(t => t.Priority);
        }

        public override string ToString()
        {
            return "За пріоритетом";
        }
    }
}
```

SortByDateStrategy.cs

```
using System.Collections.Generic;
using System.Linq;
using ToDo.Models;
using Task = ToDo.Models.Task;

namespace ToDo.Strategies
{
    public class SortByDateStrategy : ITaskSortStrategy
    {
        public IEnumerable<Task> Sort(IEnumerable<Task> tasks)
        {
            return tasks.OrderBy(t => t.DueDate);
        }

        public override string ToString()
        {
            return "За датою";
        }
    }
}
```

MainForm

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using ToDo.Models;
using ToDo.Repositories;
using ToDo.Enums;
using ToDo.Strategies;
using Task = ToDo.Models.Task;

namespace ToDo.UI
{
    public partial class MainForm : Form
    {
        private readonly IRepository<Task> _taskRepository;
        private readonly IRepository<Project> _projectRepository;
        private readonly IRepository<User> _userRepository;
```

```
    // Зберігаємо список доступних стратегій
    private readonly IEnumerable<ITaskSortStrategy> _sortStrategies;

    public MainForm(IRepository<Task> taskRepository,
                    IRepository<Project> projectRepository,
                    IRepository<User> userRepository,
                    IEnumerable<ITaskSortStrategy> sortStrategies) //
Отримуємо стратегії
    {
        InitializeComponent();

        _taskRepository = taskRepository;
        _projectRepository = projectRepository;
        _userRepository = userRepository;
        _sortStrategies = sortStrategies; // Зберігаємо стратегії

        // Завантажуємо дані при старті
        InitializeSortComboBox();
        LoadTasks();
    }

    private void InitializeSortComboBox()
    {
        // Заповнюємо ComboBox нашими стратегіями
        foreach (var strategy in _sortStrategies)
        {
            cmbSortStrategy.Items.Add(strategy);
        }
        cmbSortStrategy.SelectedIndex = 0; // Обираємо першу за
замовчуванням

        cmbSortStrategy.SelectedIndexChanged += (sender, e) =>
LoadTasks();
    }

    // (Використання Стратегії)
    private void LoadTasks()
    {
```

```
        tasksListBox.Items.Clear();

        // Отримуємо обрану стратегію з ComboBox
        var selectedStrategy =
(ITaskSortStrategy)cmbSortStrategy.SelectedItem;

        // Отримуємо всі завдання
        var tasks = _taskRepository.GetAll();

        // Застосовуємо обрану стратегію
        var sortedTasks = selectedStrategy.Sort(tasks);

        // Відображаємо відсортований список
        foreach (var task in sortedTasks)
        {
            tasksListBox.Items.Add($"[{task.Status}] {task.Title} (Пріоритет:
{task.Priority})");
        }
    }

    private void btnAddTask_Click(object sender, EventArgs e)
    {
        var defaultUser = _userRepository.GetAll().FirstOrDefault();
        if (defaultUser == null)
        {
            MessageBox.Show("Критична помилка: Не знайдено
користувача.", "Помилка", MessageBoxButtons.OK, MessageBoxIcon.Error);
            return;
        }

        var defaultProject = _projectRepository.GetAll().FirstOrDefault(p =>
p.UserId == defaultUser.UserId);
        if (defaultProject == null)
        {
            defaultProject = new Project
            {
                Title = "Загальний Проєкт",
                UserId = defaultUser.UserId,
                CreationDate = DateTime.Now
```

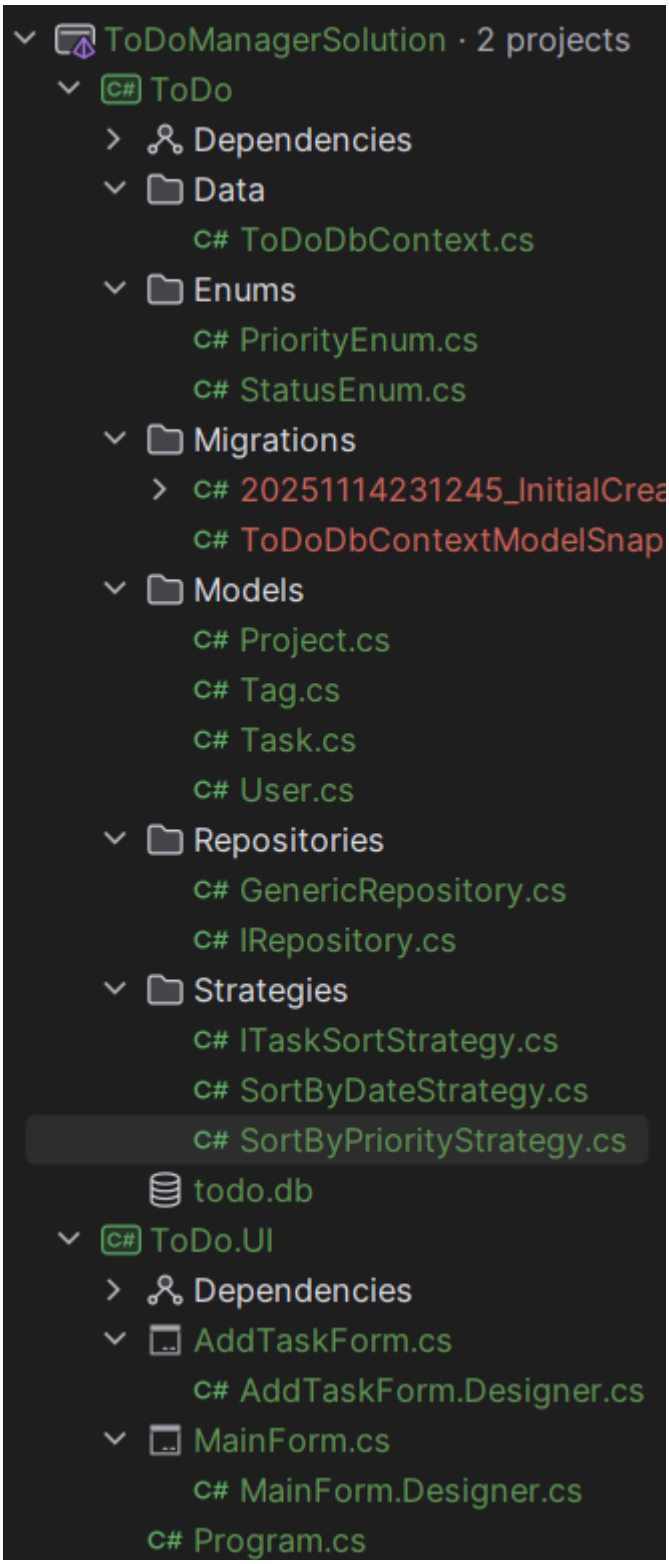
```
            };
            _projectRepository.Add(defaultProject);
        }

        using (var addTaskForm = new
AddTaskForm(defaultProject.ProjectId))
        {
            if (addTaskForm.ShowDialog() == DialogResult.OK)
            {
                try
                {
                    var newTask = new Task
                    {
                        Title = addTaskForm.TaskTitle!,
                        Description = string.Empty,
                        Priority = addTaskForm.TaskPriority,
                        ProjectId = defaultProject.ProjectId,
                        Status = StatusEnum.New,
                        DueDate = DateTime.Now.AddDays(1)
                    };
                    _taskRepository.Add(newTask);
                    LoadTasks();
                }
                catch (Exception ex)
                {
                    MessageBox.Show($"Помилка збереження завдання:
{ex.Message}\n\n" +
                                    (ex.InnerException != null ?
ex.InnerException.Message : ""),
                                    "Помилка БД", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                }
            }
        }
    }
}
```

Program

```
using System;
using System.Windows.Forms;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using ToDo.Data;
using ToDo.Models;
using ToDo.Repositories;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using ToDo.Strategies;
using Task = ToDo.Models.Task;

namespace ToDo.UI
{
   static class Program
   {
      public static IServiceProvider? Services { get; private set; }

      [STAThread]
      static void Main()
      {
         Application.SetHighDpiMode(HighDpiMode.SystemAware);
         Application.EnableVisualStyles();
         Application.SetCompatibleTextRenderingDefault(false);

         var host = CreateHostBuilder().Build();
         Services = host.Services;

         using (var scope = Services.CreateScope())
         {
            var dbContext =
scope.ServiceProvider.GetRequiredService<ToDoDbContext>();
            try
            {
               dbContext.Database.Migrate();
               EnsureDataExists(dbContext);
            }
            catch (Exception ex)
```

```
            {
                MessageBox.Show($"Критична помилка ініціалізації бази
даних: {ex.Message}",
                                "Помилка БД", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                return;
            }
        }

        Application.Run(Services.GetRequiredService<MainForm>());
    }

    static IHostBuilder CreateHostBuilder() =>
        Host.CreateDefaultBuilder()
            .ConfigureServices((context, services) =>
            {
                // Реєструємо DbContext
                services.AddDbContext<ToDoDbContext>();

                // Реєструємо Репозиторії
                services.AddTransient<IRepository<Task>,
GenericRepository<Task>>();
                services.AddTransient<IRepository<Project>,
GenericRepository<Project>>();
                services.AddTransient<IRepository<User>,
GenericRepository<User>>();

                // Реєстрація Стратегій
                services.AddTransient<ITaskSortStrategy,
SortByPriorityStrategy>();
                services.AddTransient<ITaskSortStrategy, SortByDateStrategy>();

                // Реєструємо наші Форми
                services.AddTransient<MainForm>();
            });

    private static void EnsureDataExists(ToDoDbContext context)
    {
        if (!context.Users.Any())
```

```
        {
            context.Users.Add(new User
            {
                Username = "default_user",
                PasswordHash = "hashedpassword"
            });
            context.SaveChanges();
        }

        var defaultUser = context.Users.First();
        if (!context.Projects.Any())
        {
            context.Projects.Add(new Project
            {
                Title = "Загальний Проєкт",
                CreationDate = DateTime.Now,
                UserId = defaultUser.UserId
            });
            context.SaveChanges();
        }
    }
  }
}
```


ToDoManagerSolution · 2 projects
ToDo
Dependencies
Data
ToDoDbContext.cs
Enums
PriorityEnum.cs
StatusEnum.cs
Migrations
20251114231245_InitialCrea
ToDoDbContextModelSnap
Models
Project.cs
Tag.cs
Task.cs
User.cs
Repositories
GenericRepository.cs
IRepository.cs
Strategies
ITaskSortStrategy.cs
SortByDateStrategy.cs
SortByPriorityStrategy.cs
todo.db
ToDo.UI
Dependencies
AddTaskForm.cs
AddTaskForm.Designer.cs
MainForm.cs
MainForm.Designer.cs
Program.cs

Висновки

Висновки: під час виконання лабораторної роботи ми засвоїли принципи застосування патернів проєктування в об’єктно-орієнтованих системах. Було успішно реалізовано патерн Strategy (Стратегія), що дозволило інкапсулювати різні алгоритми сортування завдань та зробити їх взаємозамінними, підтвердивши гнучкість та модульність спроєктованої архітектури.

Питання до лабораторної роботи

1. Що таке шаблон проєктування?

   Шаблон проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування , вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях.

2. Навіщо використовувати шаблони проєктування?

   Відповідне використання патернів проєктування підвищує стійкість системи до зміни вимог та спрощує подальше доопрацювання системи. Крім того, вони являють собою єдиний словник проєктування, що є уніфікованим засобом для спілкування розробників.

3. Яке призначення шаблону «Стратегія»?

   Шаблон «Strategy» (Стратегія) дозволяє змінювати деякий алгоритм поведінки об'єкта іншим алгоритмом , що досягає ту ж мету іншим способом. Він зручний у випадках, коли існують різні «політики» обробки даних.

4. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

   Входять класи Context , Strategy (інтерфейс) та ConcreteStrategy (конкретні реалізації). Context містить посилання на конкретну Strategy , а ConcreteStrategy реалізує алгоритм.

5. Яке призначення шаблону «Стан»?

Шаблон «State» (Стан) дозволяє змінювати логіку роботи об'єктів у випадку зміни їхнього внутрішнього стану. Це досягається шляхом винесення пов'язаних зі станом елементів в окремий загальний інтерфейс.

6. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

Входять класи Context , State (інтерфейс) та ConcreteState (конкретні стани). Об'єкти Context при зміні стану просто записують новий об'єкт в поле state, що призводить до повної зміни поведінки об'єкта.

7. Яке призначення шаблону «Ітератор»?

Шаблон «Iterator» (Ітератор) являє собою об'єкт доступу до набору (колекції) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції з самої колекції.

8. Які класи входять в шаблон «Ітератор», та яка між ними взаємодія?

Входять Client , Iterator , Aggregate та їхні конкретні реалізації. Колекція (Aggregate) відповідає за зберігання даних, а Ітератор — за прохід по колекції.

9. В чому полягає ідея шаблона «Одинак»?

Ідея полягає в тому, що клас «Singleton» (Одинак) може мати не більше одного об'єкта , який найчастіше зберігається як статичне поле в самому класі.

10. Чому шаблон «Одинак» вважають «анти-шаблоном»?

Його вважають «анти-шаблоном», оскільки одинаки представляють собою глобальні дані, що мають стан , який важко відслідковувати і підтримувати коректно. Також він порушує принцип єдиної відповідальності класу.

11. Яке призначення шаблону «Проксі»?

«Proxy» (Проксі) об'єкти є об'єктами-заглушками або двійниками/замінниками для об'єктів конкретного типу. Проксі об'єкти зазвичай вносять додатковий функціонал або спрощують взаємодію з реальними об'єктами.

12. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

Входять Client , Subject (інтерфейс) , RealSubject (реальний об'єкт) та Proxy (замісник). Proxy реалізує інтерфейс Subject і містить посилання на RealSubject, виконуючи додатковий функціонал перед викликом реального об'єкта.