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

Лабораторна робота № 5

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування.»

«Менеджер завдань»

Виконав:
студент групи - ІА-32
Воробйов Кирило Андрійович

Перевірив:
Мягкий Михайло Юрійович

Київ 2025

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype» 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

Шаблон «Adapter»

Призначення патерну: Шаблон "Adapter" (Адаптер) використовується для адаптації інтерфейсу одного об'єкту до іншого [6]. Наприклад, існує декілька бібліотек для роботи з принтерами, проте кожна має різний інтерфейс (хоча однакові можливості і призначення). Має сенс розробити уніфікований інтерфейс (сканування, асинхронне сканування, двостороннє сканування, потокове сканування і тому подібне), і реалізувати відповідні адаптери для приведення бібліотек до уніфікованого інтерфейсу.

Це дозволить в програмі звертатися до загального інтерфейсу, а не приводити різні сценарії роботи залежно від способу реалізації бібліотеки. Адаптери також називаються "wrappers" (обгортками).

Проблема: Ви реалізовуєте аудіо-плеєр, який може програвати аудіо різних форматів. Ви почали аналізувати і бачите що для програвання аудіо із різних форматів краще використовувати різні компоненти.

Таким чином ви реалізуєте складну логіку роботи з різними компонентами. У вас у коді з'являється багато логіки з перевіркою, якщо формат один, то викликаємо такий метод у такого компонента, якщо інший, то викликаємо декілька методів у іншого компонента, і т.д. Логіка роботи з компонентами стає досить складною та заплутаною.

Коли потрібно добавити підтримку нового аудіо-формату, то потрібно добавити роботу з новим компонентом і при цьому внести зміну в уже існуючу логіку, що може привести до помилок там де все коректно працювало.

Рішення: Для вирішення цих проблем можна використати патерн Адаптер. Визначаємо загальний інтерфейс IPlayer для програвання музики через компоненти. Далі для кожного компонента робимо свій адаптер. Адаптер має посилання на об'єкт компонента та реалізує інтерфейс викликаючи методи з компонента, який він адаптує. Таким чином, адаптер ніби обгортає специфічний компонент і далі весь клієнтський код буде працювати з адаптерами через інтерфейс IPlayer. Класи, які будуть працювати з адаптером через цей інтерфейс не будуть знати інтерфейси кожного специфічного компонента.

При такій реалізації, якщо буде потрібно добавити підтримку нового формату, то просто створюється новий адаптер, який обгортає новий

компонент, але робота з цим адаптером буде виконуватися через існуючий інтерфейс. При цьому існуючий код не буде зазнавати змін, а значить і не "поламаємо" те що вже працює.

Переваги та недоліки:

+ Відокремлює інтерфейс або код перетворення даних від основної бізнеслогіки.

+ Можна добавляти нові адаптери не змінюючи код у класі Client.

- Умовним недоліком можна назвати збільшення кількості класів, але за рахунок використання патерна Адаптер програмний код, як привило, стає легше читати.

Шаблон «Builder»

Призначення патерну: Шаблон «Builder» (Будівельник) використовується для відділення процесу створення об'єкту від його представлення [6]. Це доречно у випадках, коли об'єкт має складний процес створення (наприклад, Web- сторінка як елемент повної відповіді web- сервера) або коли об'єкт повинен мати декілька різних форм створення (наприклад, при конвертації тексту з формату у формат).

Проблема: Візьмемо процес побудови відповіді на запит web-сервера. Побудова складається з наступних частин: додавання стандартних заголовків (дата/час, ім'я сервера, інш.), код статусу (після пошуку відповідної сторінки на сервері), заголовки відповіді (тип вмісту, інш.), утримуване, інше.

Рішення: Кожен з цих етапів може бути абстрагований в окремий метод будівельника. Це дасть наступні вигоди:

- Гнучкіший контроль над процесом створення сторінки;
- Незалежність від внутрішніх змін – наприклад, зміна назви сервера не сильно порушить процес побудови відповіді;

Переваги та недоліки:

+ Дозволяє використовувати один і той самий код для створення різноманітних продуктів.

- Клієнт буде прив’язаний до конкретних класів будівельників, тому що в інтерфейсі будівельника може не бути методу отримання результату.

Шаблон «Command»

Призначення патерну: Шаблон "command" (команда) перетворить звичайний виклик методу в клас [6]. Таким чином дії в системі стають повноправними об'єктами. Це зручно в наступних випадках:

- Коли потрібна розвинена система команд – відомо, що команди будуть добавлятися;
- Коли потрібна гнучка система команд – коли з'являється необхідність додавати командам можливість відміни, логування і інш.;
- Коли потрібна можливість складання ланцюжків команд або виклику команд в певний час.

Об'єкт команда сама по собі не виконує ніяких фактичних дій окрім перенаправлення запиту одержувачеві (тобто команди все ж виконуються одержувачем), однак ці об'єкти можуть зберігати дані для підтримки додаткових функцій відміни, логування і інш. Наприклад, команда вставки символу може запам'ятовувати символ, і при виклику відміни викликати відповідну функцію витирання символу. Можна також визначити параметр «застосовності» команди (наприклад, на картинці писати не можна) – і використати цей аттрибут для засвічування піктограми в меню.

Такий підхід до команд дозволяє побудувати дуже гнучку систему команд, що настроюється. У більшості додатків це буде зайвим (використовується спрощений варіант), проте життєво важливий в додатках з великою кількістю команд (редактори).

Проблема: Ви реалізуєте товстий клієнт який має багатий візуальний інтерфейс: має меню, кнопки і контекстне меню. Кожна дія, яку можна виконати, має три варіанти виконання – через меню, натисканням кнопки та через контекстне меню. Реалізацію кожної дії можна розмістити в обробнику візуального елементу, але тоді потрібно буде продублювати цей функціонал для меню, кнопки та контекстного меню. Таким чином ми будемо мати дублювання коду і при зміні функціоналу змінювати його в трьох місцях.

Додатковим викликом буде реалізувати автоматизоване тестування такої системи, тому що нам необхідно емулювати натискання кнопок користувачем.

Рішення: Виділення функціоналу який виконується по натисканню на кнопку в окремий клас дозволяє відвязати візуальну частину від логіки обробки. Таким чином ми будемо мати шар з UI елементами і шар з логікою обробки дій виконаних на UI. Це дозволяє один і той самий об'єкт з шару логіки обробки дій використати для реакцію на натискання кнопки і пункту меню і контекстного меню. Кнопки тепер не знають про конкретні класи реалізації команд, а взаємодіють з об'єктами команд через загальний інтерфейс. Ще одна перевага, що тепер ми можемо достатньо просто реалізувати механізм enable-disable для кнопок та контекстного меню через виклик у об'єкта команди метода IsEnabled() або через прив'язку (binding) на поле IsEnabled у об'єкта команди.

При такій реалізації також набагато простіше організувати тестування застосунку, тому що ми тепер не потрібно емулювати дії користувача, а достатньо протестувати шар логіки обробки використовуючи модульні тести (unit tests).

Переваги та недоліки:

+ Ініціатор виконання команди не знає деталей реалізації виконавця команди.

+ Підтримує операції скасування та повторення команд.

+ Послідовність команд можна логувати і при необхідності виконати цю послідовність ще раз.

+ Простота розширення за рахунок додавання нових команд без необхідності внесення змін в уже існуючий код (принцип відкритостізакритості).

Шаблон «Chain of Responsibility»

Призначення патерну: Шаблон «Chain of responsibility» (Ланцюжок відповідальності) частково можна спостерігати в житті, коли підписання відповідного документу проходить від його складання у одного із співробітників компанії через менеджера і начальника до головного начальника, який ставить свій підпис [5].

Проблема: Ви розробляєте систему зі складними UI формами, які містять багато вкладених один в один візуальних компонентів, по кліку

правою кнопкою миші вам потрібно сформувати контекстне меню. В залежності від того на якому компоненті був виконаний клік мишкою, вміст контекстного меню повинен відрізнятися, також в контекстне меню потрібно добавляти пункти, якщо компонент, в який входить поточний, теж має свої пункти. Один із очевидних підходів – ви робити метод, який викликається по кліку мишки. В методі ви робити перевірки і якщо в даний момент конкретний елемент, добавляєте в контекстне меню підходящі до нього пункти, потім перевіряєте інші елементи і перевіряєте чи вони не є такими, що містять в собі вибраний елемент. Якщо так, то добавляєте в контекстне меню ще пункти.

З часом алгоритм формування меню стає більш складним, тому що добавляються нові елементи. Також з часом в алгоритмі залишилися перевірки на компоненти, які з форми були вже видалені.

Рішення: Для вирішення проблеми зі зростаючою складністю формування контестного меню підходить патерн «Ланцюжок відповідальності».

Основна ідея в тому, що нам не потрібно мати загальний метод формування контекстного меню. Ми можемо зробити загальний інтерфейс з методом UpdateContextMenu() і реалізувати цей метод в усіх візуальних компонентах. Додатково для візуальних компонентів добавляємо поле для переходу на "батьківський" елемент, якій в собі містить поточний візуальний компонент, а в реалізації UpdateContextMenu в кінці додаємо виклик цього метода у "батьківського" елемента. Якщо елемент не потребує додавання пунктів в контекстне меню, він просто викликає цей метод у наступного елемента в ланцюжку. Тепер обробка правого кліку мишки буде виглядати наступним чином: По кліку правою кнопкою миші викликаємо UpdateContextMenu на тексті, він, наприклад, додає пункт меню «Копіювати текст», далі викликає аналогічний метод у компонента «текст-блок» в якому цей текст знаходиться, але текст-блок нічого в контекстне меню не добавляє, а викликає UpdateContextMenu в елемента Grid, в якому знаходиться текст-блок, а Grid при відпрацюванні метода добавляє пункт меню «Контестна допомога» і так далі.

Коли ми змінюємо структуру форми, то між компонентами змінюються зв'язки в ланцюжку, але самі алгоритми вже переробляти не потрібно. якщо видаляється з форми якийсь компонент, то разом з ним видаляється і логіка пов'язана з цим компонентом. Таким чином складність роботи з контекстним меню залишається контрольованою.

Слід добавити що елемент, що обробляє виклик, не обов'язково повинен передавати виклик далі по ланцюжку. Це залежить від того, яку поведінку ви хочете отримати.

Переваги та недоліки:

+ Зменшує залежність між клієнтом та обробниками: клієнт не знає хто обробить запит, а обробники не знають структуру ланцюжка.
+ Реалізовує додаткову гнучкість в обробці запиту: легко добавити або вилучити з ланцюжка нові обробники.
- Запит може залишитися ніким не опрацьованим: запит не має вказаного обробника, тому може бути не опрацьованим.

Шаблон «Prototype»

Призначення патерну: Шаблон «Prototype» (Прототип) використовується для створення об'єктів за «шаблоном» (чи «кресленням», «ескізом») шляхом копіювання шаблонного об'єкту, який називається прототипом. Для цього визначається метод «клонувати» в об'єктах цього класу [6].

Цей шаблон зручно використати, коли заздалегідь відомо як виглядатиме кінцевий об'єкт (мінімізується кількість змін до об'єкту шляхом створення шаблону), а також для видалення необхідності створення об'єкту – створення відбувається за рахунок клонування, і самій програмі немає необхідності знати, як створювати об'єкт.

Також, це дозволяє маніпулювати об'єктами під час виконання програми шляхом настроювання відповідних прототипів; значно зменшується ієрархія наслідування (оскільки в іншому випадку це були б не прототипи, а вкладені класи, що наслідують).

Проблема: Ви розробляєте редактор рівнів для 2D гри на основі спрайтів. В панелі інструментів ви маєте багато кнопок для різних елементів, які можна розташовувати на екрані, такі як сходи, стіни, підлога, оздоблення та інші. Ці елементи у вас об'єднані в ієрархію з базовим класом GameObject. Під кожен елемент можна зробити свій тип кнопки, але тоді ми отримаємо паралельну ієрархію кнопок і при додаванні нового типу ігрового об’єкту потрібно буде додавати і новий тип кнопки.

Рішення: Використовуючи патерн прототип, додаємо до базового об'єкта GameObject метод Clone(), а кнопки будуть зберігати посилання на об'єкт базового типу GameObject. При натисканні на

кнопку об'єкт який потрібно добавити на ігровому полі отримуємо не створенням нового, а клонуванням прототипу, який прив'язаний до кнопки. Таким чином, коли ми будемо додавати нові типи ігрових об'єктів, то логіка роботи з кнопками не буде змінюватися, тому що не має прив'язки до конкретних типів.

Також слід відзначити, що при копіюванні об'єкту, навіть приватні поля будуть скопійовані, тому що реалізація методу копіювання знаходиться в цьому класі, що копіюється і таким чином є доступ для копіювання і до відкритих і до закритих полів.

Переваги та недоліки:

+ За рахунок клонування складних об’єктів замість їх створення, підвищується продуктивність.

+ Різні варіації об’єктів можна отримувати за рахунок клонування, а не розширення ієрархії класів.

+ Вища гнучкість, тому що клоновані об’єкти можна модифікувати незалежно, не впливаючи на об’єкт з якого була зроблена копія.

- Реалізація глибокого клонування досить проблематична, коли об’єкт що клонується містить складну внутрішню структуру та посилання на інші об’єкти.
- Надмірне використання патерну Прототип може привести до ускладнення коду та проблем із супроводом такого коду.

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 Command

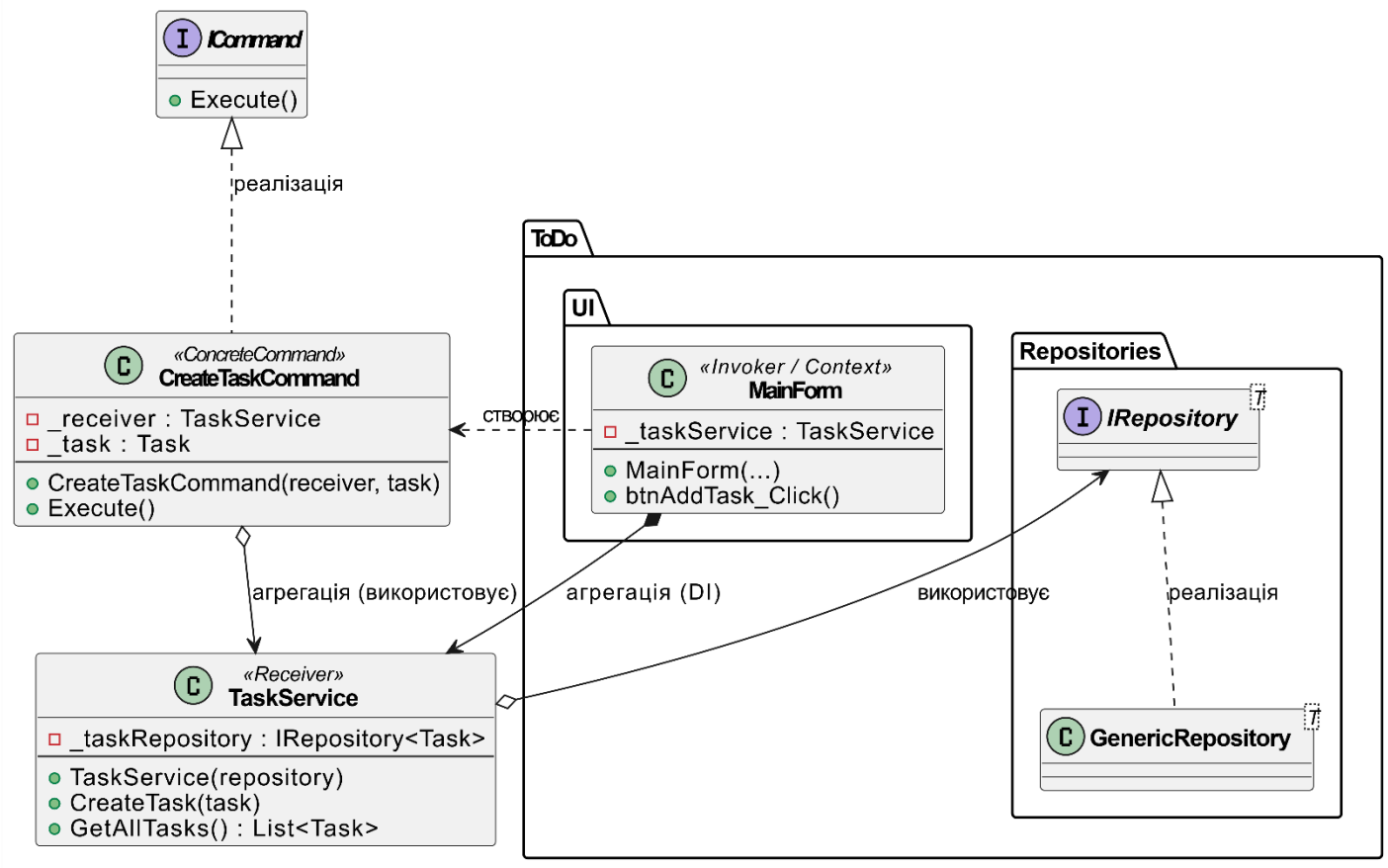


Рис. 1. Діаграма класів патерну Command

Діаграма класів ілюструє, як патерн **Command** впроваджено в систему **"Менеджер Завдань"** для керування діями користувача.

1. **Invoker (Ініціатор) — MainForm:** Це UI-клас, який відповідає за обробку подій (натискання кнопки **"Додати завдання"**). Він створює об'єкт **ICommand** і викликає його метод **Execute()**, не знаючи деталей виконання.
2. **Command (Команда) — ICommand:** Інтерфейс, який визначає контракт. **CreateTaskCommand** реалізує цей контракт, зберігаючи посилання на **Одержувача** (TaskService) та параметри (Task).
3. **Receiver (Одержувач) — TaskService:** Клас, який містить необхідну бізнес-логіку. Він отримує запит від команди і викликає **IRepository<Task>** для фактичної роботи з даними.

4. **Слабкий зв'язок:** Завдяки цій структурі, UI залежить лише від абстракції **ICommand**, тоді як **TaskService** залишається чистим і використовує патерн **Repository** для взаємодії з базою даних.

Код програми

ICommand

```
namespace ToDo.Commands
{
   public interface ICommand
   {
      void Execute();
   }
}
```

TaskService

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

namespace ToDo.Services
{

   public class TaskService
   {
      private readonly IRepository<Task> _taskRepository;


      public TaskService(IRepository<Task> taskRepository)
      {
         _taskRepository = taskRepository;
      }

      public IEnumerable<Task> GetAllTasks()
      {
         return _taskRepository.GetAll();
      }
```

```
        public void CreateTask(Task task)
        {
            _taskRepository.Add(task);
        }

        public void UpdateTask(Task task)
        {
            _taskRepository.Update(task);
        }

        public void DeleteTask(int taskId)
        {
            _taskRepository.Delete(taskId);
        }
    }
}
```

CreateTaskCommand

```
using ToDo.Services;
using ToDo.Models;
using Task = ToDo.Models.Task;

namespace ToDo.Commands
{
    public class CreateTaskCommand : ICommand
    {
        private readonly TaskService _receiver; // Одержувач (хто виконує)
        private readonly Task _task; // Параметри (що створити)

        public CreateTaskCommand(TaskService receiver, Task task)
        {
            _receiver = receiver;
            _task = task;
        }

        // Реалізація інтерфейсу Command
```

```
        public void Execute()
        {
            _receiver.CreateTask(_task);
        }
    }
}
```

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 ToDo.Services;
using ToDo.Commands;
using Task = ToDo.Models.Task;

namespace ToDo.UI
{
    public partial class MainForm : Form
    {
        private readonly TaskService _taskService; // Наш "Receiver"

        // Залежності, що залишилися
        private readonly IRepository<Project> _projectRepository;
        private readonly IRepository<User> _userRepository;
        private readonly IEnumerable<ITaskSortStrategy> _sortStrategies;

        public MainForm(
            TaskService taskService, // Отримуємо Receiver
            IRepository<Project> projectRepository,
            IRepository<User> userRepository,
            IEnumerable<ITaskSortStrategy> sortStrategies)
        {
            InitializeComponent();
```

```
        // Зберігаємо залежності
        _taskService = taskService;
        _projectRepository = projectRepository;
        _userRepository = userRepository;
        _sortStrategies = sortStrategies;

        InitializeSortComboBox();
        LoadTasks();
    }

    private void InitializeSortComboBox()
    {
        foreach (var strategy in _sortStrategies)
        {
            cmbSortStrategy.Items.Add(strategy);
        }
        cmbSortStrategy.SelectedIndex = 0;
        cmbSortStrategy.SelectedIndexChanged += (sender, e) =>
LoadTasks();
    }

    private void LoadTasks()
    {
        tasksListBox.Items.Clear();

        var selectedStrategy =
(ITaskSortStrategy)cmbSortStrategy.SelectedItem;

        // Отримуємо завдання через наш Service
        var tasks = _taskService.GetAllTasks();

        var sortedTasks = selectedStrategy.Sort(tasks);

        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)
```

```
                };

                // Створюємо команду, передаючи їй Receiver (_taskService)
                //    та параметри (newTask).
                ICommand command = new CreateTaskCommand(_taskService,
newTask);

                // Виконуємо команду.
                command.Execute();

                // Оновлюємо UI
                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);
            }
        }
    }
  }
}
```

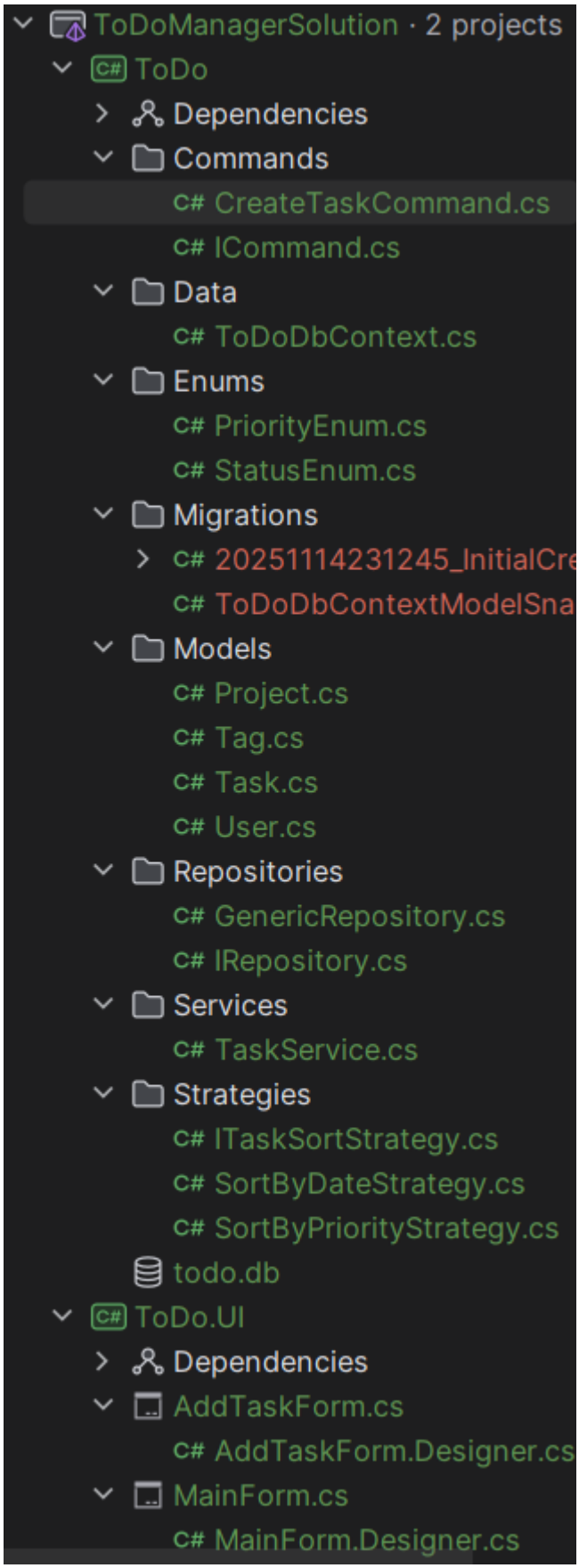
Program

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

```
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;

            // Виконуємо міграцію та сидінг даних
            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>());
```

```
    }

    // Налаштування "Хоста" та всіх наших сервісів
    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>>();

                // Реєструємо Стратегії
                services.AddTransient<ITaskSortStrategy,
SortByPriorityStrategy>();
                services.AddTransient<ITaskSortStrategy, SortByDateStrategy>();

                // Реєстрація Service
                services.AddTransient<TaskService>();

                // Реєструємо 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
Commands
CreateTaskCommand.cs
ICommand.cs
Data
ToDoDbContext.cs
Enums
PriorityEnum.cs
StatusEnum.cs
Migrations
20251114231245_InitialCre
ToDoDbContextModelSna
Models
Project.cs
Tag.cs
Task.cs
User.cs
Repositories
GenericRepository.cs
IRepository.cs
Services
TaskService.cs
Strategies
ITaskSortStrategy.cs
SortByDateStrategy.cs
SortByPriorityStrategy.cs
todo.db
ToDo.UI
Dependencies
AddTaskForm.cs
AddTaskForm.Designer.cs
MainForm.cs
MainForm.Designer.cs

Висновки

Висновки: під час виконання лабораторної роботи ми успішно реалізували патерн Command (Команда). Цей патерн дозволив інкапсулювати операцію створення завдання (CreateTaskCommand) та відокремити ініціатора (UI) від Одержувача (TaskService).

Питання до лабораторної роботи

1. Яке призначення шаблону «Адаптер»?

Шаблон "Adapter" (Адаптер) використовується для адаптації інтерфейсу одного об'єкту до інтерфейсу іншого. Адаптер (який також називається "wrapper" або "обгортка") дозволяє розробити уніфікований інтерфейс для роботи з різними бібліотеками, що мають однакові можливості, але різні інтерфейси.

2. Які класи входять в шаблон «Адаптер», та яка між ними взаємодія?

Входять Client (Клієнт), Target (Цільовий інтерфейс), Adaptee (Адаптований об'єкт) та Adapter (Адаптер) . Adapter реалізує Target і містить посилання на Adaptee, викликаючи його специфічні методи.

3. Яке призначення шаблону «Будівельник»?

Шаблон «Builder» (Будівельник) використовується для відділення процесу створення об'єкту від його представлення. Це доречно, коли об'єкт має складний, багатоетапний процес створення або повинен мати декілька різних форм представлення.

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

Входять Director (Директор), Builder (Інтерфейс Будівельника), ConcreteBuilder (Конкретний Будівельник) та Product (Створюваний

об'єкт) . Director керує процесом конструювання, викликаючи методи Builder для створення частин Product.

5. Яке призначення шаблону «Команда»?

Шаблон "Command" (Команда) перетворює звичайний виклик методу в клас, роблячи дії в системі повноправними об'єктами. Це дозволяє реалізувати гнучку систему команд з можливістю відміни (Undo), логування або складання ланцюжків команд.

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

Входять Client (Клієнт), Invoker (Ініціатор), Command (Інтерфейс команди), ConcreteCommand (Конкретна команда) та Receiver (Одержувач) . Invoker зберігає та викликає Command, а ConcreteCommand перенаправляє запит до Receiver, який виконує фактичну дію.

7. Яке призначення шаблону «Прототип»?

Шаблон «Prototype» (Прототип) використовується для створення об'єктів за «шаблоном» шляхом копіювання (клонування) шаблонного об'єкту, який називається прототипом. Це дозволяє уникнути необхідності створення об'єкту через конструктор.

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

Входять Client (Клієнт), Prototype (Інтерфейс прототипу) та ConcretePrototype (Конкретний прототип) . Client викликає метод Clone() у об'єкта-прототипу, щоб отримати його копію.

9. Яке призначення шаблону «Ланцюжок відповідальності»?

Шаблон «Chain of Responsibility» (Ланцюжок відповідальності) використовується для передачі запиту послідовно через ланцюжок обробників. Він дозволяє обробляти запити без жорсткої прив'язки до конкретного обробника.

10. Які класи входять в шаблон «Ланцюжок відповідальності», та яка між ними взаємодія?

Входять Client (Клієнт), Handler (Інтерфейс обробника) та ConcreteHandler (Конкретні обробники) . Handler містить посилання на successor (наступника) , і кожен обробник вирішує, чи обробляти запит самостійно, чи передавати його далі по ланцюжку.

11. Яка різниця між реалізацією «Адаптера» на рівні об'єктів та на рівні класів?

На рівні об'єктів (найпоширеніший): Adapter містить посилання (композицію) на об'єкт, що адаптується (Adaptee).

На рівні класів: Adapter успадковує інтерфейс Target та реалізацію Adaptee (вимагає множинного успадкування, недоступного в C#).

12. У яких випадках варто застосовувати шаблон «Будівельник»?

Шаблон «Будівельник» варто застосовувати, коли об'єкт має складний процес створення або коли об'єкт повинен мати декілька різних форм представлення.

13. Розкажіть, як працює шаблон «Команда».

Команда інкапсулює виклик методу (дію) в об'єкт. Клієнт створює ConcreteCommand із посиланням на Receiver (одержувач, який знає, як виконати дію) та необхідними параметрами. Invoker (Ініціатор) викликає Execute() у команді, а команда, у свою чергу, викликає відповідний метод у Receiver.

14. Які можна привести приклади використання шаблону «Ланцюжок відповідальності»?

Приклади використання шаблону «Ланцюжок відповідальності» включають: формування контекстного меню у складних UI-формах (де вкладені компоненти додають свої пункти) або підписання документу, що проходить послідовно через менеджерів до головного начальника.