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

Лабораторна робота № 6

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

Призначення патерну: Шаблон «Абстрактна фабрика» використовується для створення сімейств об'єктів без вказівки їх конкретних класів [6]. Для цього виноситься загальний інтерфейс фабрики (AbstractFactory) і створюються його реалізації для різних сімейств продуктів. Хорошим прикладом використання абстрактної фабрики є ADO.NET: існує загальний клас DbProviderFactory, здатний створювати об'єкти типів DbConnection, DbDataReader, DbAdapter та ін.; існують реалізації цих фабрик і об'єктів – SqlProviderFactory, SqlConnection,

SqlDataReader, SqlAdapter і так далі. Відповідно, якщо додатку необхідно працювати з різними базами даних (чи потрібна така можливість), то досить використати базові реалізації (Db.) і підставити відповідну фабрику у момент ініціалізації фабрики (Factory = new SqlProviderFactory()).

Цей шаблон передусім структурує знання про схожі об'єкти (що називаються сімействами, як класи для доступу до БД) і створює можливість взаємозаміни різних сімейств (робота з Oracle ведеться також, як і робота з SQL Server). Проте, при використанні такої схеми украй незручно розширювати фабрику – для додавання нового методу у фабрику необхідно додати його в усіх фабриках і створити відповідні класи, що створюються цим методом.

Проблема: Ви розробляєте для Roguelike гри модуль автоматичної генерації кімнат. Модуль генерації кімнат, в процесі генерації конкретної кімнати, створює стіни, двері, вікна, підлогу та меблі в кімнаті. Модуль повинен підтримувати генерацію кімнат у різних стилях, таких як стиль хайтек, модерн, класичний офіс. Також критично є щоб всі елементи в одній згенерованій кімнаті відносилися до одного стилю.

Рішення: Для вирішення поставленої задачі дуже добре підходить використання патерну «Абстрактна фабрика».

Для кожного типу продукту створюємо свій інтерфейс продукту який об'являє функції доступні для використання з цим продуктом. Наприклад, можна виділити інтерфейси для стін, вікон, дверей, підлоги, а також окремі інтерфейси для меблів, такі як стіл, шафа, крісло та інші. Від кожного інтерфейсу наслідуємо і реалізуємо продукти різних типів, наприклад, для стола: інтерфейс ITable та реалізації продуктів HighTechTable, ModernTable та інші.

Далі визначаємо інтерфейс для абстрактної фабрики, який буде містити методи для створення кожного типу продукту. Створюємо класи

конкретних фабрик під кожен стиль: HighTechFabric, ModernFabric та інші. Кожна конкретна фабрика буде створювати всі типи продуктів, але всі вони будуть відноситися до одного стилю.

Далі в алгоритм генерації кімнати будемо передавати конкретну фабрику і алгоритм при створенні продуктів тільки через фабрику завжди буде отримувати продукти одного стилю і працювати з продуктами тільки через інтерфейс продукта.

Таким чином ми вирішуємо проблему узгодженості стилів всіх елементів в кімнаті, а за рахунок використання різних конкретних фабрик ми зможемо генерувати кімнати різних стилів. Якщо нам потрібно буде добавити ще один стиль, то достатньо буде реалізувати нові дочірні класи для кожного елемента кімнати, а також нову конкретну фабрику під цей стиль.

Переваги та недоліки:

+ Спрощує створення об’єктів і код стає легшим для розуміння.

+ Об’єкти створені однією фабрикою добре узгоджуються один з одним і зменшується кількість помилок взаємодії між ними.
+ Відокремлення створення об’єктів від їх використання, за рахунок чого, код стає біль структурованим.
+ Додавання нових сімейств продуктів виконується без зміни існуючого коду.
- Збільшується складність коду, особливо для простих проєктів.

- Додавання нового типу продукту є складним і вимагає змін коду в багатьох місцях.

Шаблон «Factory Method»

Призначення: Шаблон «Фабричний метод» визначає інтерфейс для створення об'єктів певного базового типу [6]. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу. Тому шаблон «Фабричний метод» носить ще назву

«Віртуальний конструктор».

Розглянемо простий приклад. Нехай наш застосунок працює з мережевими драйвер-мі і використовує клас Packet для зберігання даних, що передаються в мережу. Залежно від використовуваного протоколу, існує два перевантаження – TcpPacket, UdpPacket. І відповідно два створюючі об'єкти (TcpCreator, UdpCreator) з фабричним методом (який створює відповідні реалізації).

Проте базова функціональність (передача пакету, прийом пакету, заповнення пакету даними) нічим не відрізняється один від одного, відповідно поміщається у базовий клас PacketCreator. Таким чином поведінка системи залишається тим же, проте з'являється можливість підстановки власних об'єктів в процес створення і роботи з пакетами.

Переваги та недоліки:

+ Позбавляє клас від прив’язки до конкретних класів продуктів.

+ Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
+ Спрощує додавання нових продуктів до програми.

- Може призвести до створення великих паралельних ієрархій класів.

Шаблон «Memento»

Призначення: Шаблон використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції [6]. Об'єкт «Memento» служить виключно для збереження змін над початковим об'єктом (Originator). Лише початковий об'єкт має можливість зберігати і отримувати стан об'єкту

«Memento» для власних цілей, цей об'єкт є «порожнім» для кого-небудь ще. Об'єкт «Caretaker» використовується для передачі і зберігання мементо об'єктів в системі.

Таким чином вдається досягти наступних цілей:

- зберігання стану повністю відділяється від початкових об'єктів, що полегшує їх реалізацію;
- передача об'єктів «Memento» лягає на плечі Caretaker об'єктів, що дозволяє гнучкіше управляти станами об'єктів і спростити дизайн класів початкових об'єктів;

- збереження і відновлення стану реалізовані у вигляді двох простих методів і є закритими для кого-небудь ще окрім початкових об'єктів, таким чином не порушуючи інкапсуляцію.

Шаблон «Мементо» дуже зручно використати разом з шаблоном «Команда» для реалізації «скасовних» дій – дані про дію зберігаються в мементо, а команда має можливість вважати і відновити початкове положення відповідних об'єктів.

Переваги та недоліки:

+ Не порушує інкапсуляцію вихідного об’єкта.

+ Спрощує структуру вихідного об’єкта. Не потрібно зберігати історію версій свого стану.

- Вимагає багато пам’яті, якщо клієнти дуже часто створюють знімки.

- Може спричинити додаткові витрати пам’яті, якщо об’єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.

Шаблон «Observer»

Призначення: Шаблон визначає залежність «один-до-багатьох» таким чином, що коли один об'єкт змінює власний стан, усі інші об'єкти отримують про це сповіщення і мають можливість змінити власний стан також [6].

Розглянемо цей шаблон на прикладі. Припустимо, є деяка банківська система і декілька користувачів переглядають баланс на рахунку пана И. У цей момент пан И. кладе на свій рахунок деяку суму, яка міняє загальний баланс. Кожен з користувачів, що переглядали баланс, отримує про це звістку (для користувачів ця звістка може бути прозорою – просто зміна цифр, або попередження про те, що баланс змінився). Раніше неможливі дії для користувачів (переведення до іншої категорії клієнтів і тому подібне) стають доступними.

Цей шаблон дуже широко поширений в шаблоні MVVM і механізмі «прив'язок» (bindings) в WPF і частково в WinForms. Інша назва шаблону – підписка/розсилка. Кожен з оглядачів власноручно підписується на зміни конкретного об'єкту, а об'єкти зобов'язані сповіщати своїх передплатників

про усі свої зміни (на даний момент конкретних механізмів автоматичного сповіщення про зміну стану в .NET мовах не існує).

Приклад з життя: Коли ви підписуєтеся на канал на YouTube і натискаєте на «дзвіночок» ви фактично оформлюєте підписку на отримання повідомлень про вихід нових відео. Вам не потрібно заходити на канал і перевіряти чи не вийшло нове відео. Ви будете отримувати повідомлення про вихід нових відео на каналах

Фактично на сервісі YouTube є список підписників на канал, хто має отримувати повідомлення про вихід нового відео.

В якості прикладу також можна згадати підписку «Повідомити про появу товару» на сторінці товару в магазині «Розетка».

Переваги та недоліки:

+ Можливість паралельної та асинхронної обробки повідомлень про оновлення.
+ Спостерігачів можна добавляти та видаляти в будь-який момент часу.

+ Спостерігач і суб’єкт можуть працювати в різних потоках.

+ Реалізує принцип слабкого зв’язку між об’єктами.

- Послідовність розсилки повідомлень підписникам не підтримується.

Шаблон «Decorator»

Призначення: Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми [6]. Декоратор деяким чином «обертає» (за рахунок агрегації) початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому. Простим прикладом є накладення смуги прокрутки до усіх візуальних елементів. Кожен об'єкт, який може прокручуватися, обертається в «прокручуваному» елементі, і при необхідності з'являється полоса прокрутки. Початкові функції елементу (наприклад, рядки статусу) залишаються незмінними.

Переваги та недоліки:

+ Дозволяє мати кілька дрібних об’єктів, замість одного об’єкта «на всі випадки життя».
+ Дозволяє додавати обов’язки «на льоту».

+ Більша гнучкість, ніж у спадкування.

- Велика кількість крихітних класів.

Важко конфігурувати об’єкти, які загорнуто в декілька обгорток одночасно.

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 Observer

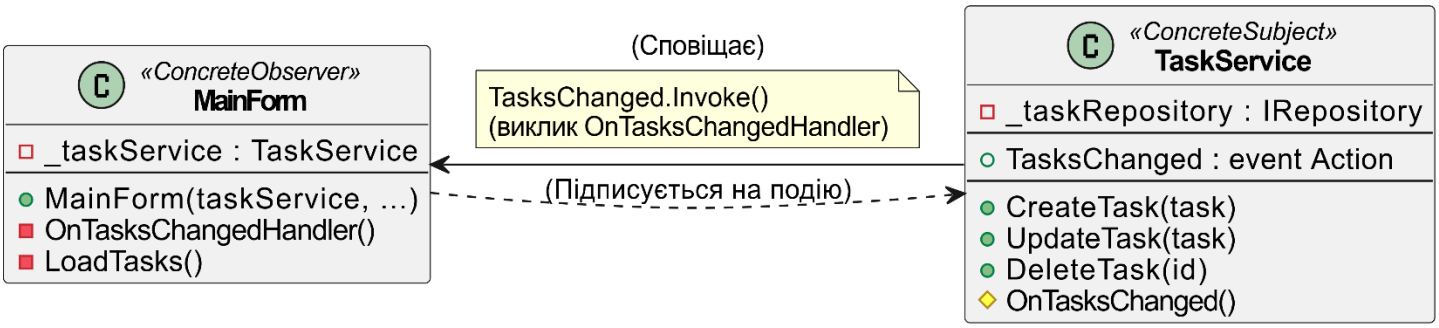


Рис. 1. Діаграма класів патерну Observer

Діаграма демонструє реалізацію патерна **Observer** за допомогою стандартних механізмів **подій (events)** C#, що забезпечує **слабку зв'язаність** між компонентами.

1. **Subject (Видавець) — TaskService:** Виступає джерелом даних. Він містить **подію** TasksChanged (еквівалент класичного методу Notify()). Після виконання операцій зі збереженням даних (CreateTask, UpdateTask), **TaskService** сповіщає всіх підписників про зміну.
2. **Observer (Спостерігач) — MainForm:** Це клас, який зацікавлений в оновленні даних. Він є **ConcreteObserver**, який **підписується** на подію TasksChanged у конструкторі.
3. **Автоматичне оновлення:** Коли відбувається зміна даних, **TaskService** викликає подію, яка автоматично активує метод-обробник у

MainForm, що, у свою чергу, викликає **LoadTasks()** для оновлення візуального інтерфейсу.

Ця архітектура дозволяє будь-якому іншому компоненту стати "Спостерігачем" без зміни коду в **TaskService**.

Код програми

TaskService

```
using System;
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 event Action? TasksChanged;

      public TaskService(IRepository<Task> taskRepository)
      {
         _taskRepository = taskRepository;
      }
      public IEnumerable<Task> GetAllTasks()
      {
         return _taskRepository.GetAll();
      }
      public void CreateTask(Task task)
      {
         _taskRepository.Add(task);
         // Сповіщаємо всіх підписників, що дані змінилися
         OnTasksChanged();
      }

      public void UpdateTask(Task task)
      {
```

```
            _taskRepository.Update(task);
            OnTasksChanged(); // Сповіщаємо
        }

        public void DeleteTask(int taskId)
        {
            _taskRepository.Delete(taskId);
            OnTasksChanged(); // Сповіщаємо
        }
        protected virtual void OnTasksChanged()
        {
            // Викликаємо подію, якщо на неї є хоча б один підписник
            TasksChanged?.Invoke();
        }
    }
}
```

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;
        private readonly IRepository<Project> _projectRepository;
        private readonly IRepository<User> _userRepository;
        private readonly IEnumerable<ITaskSortStrategy> _sortStrategies;
```

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

        _taskService = taskService;
        _projectRepository = projectRepository;
        _userRepository = userRepository;
        _sortStrategies = sortStrategies;
        _taskService.TasksChanged += OnTasksChangedHandler;

        InitializeSortComboBox();
        LoadTasks();
    }

    private void OnTasksChangedHandler()
    {
        if (this.InvokeRequired)
        {
            this.Invoke(new Action(LoadTasks));
        }
        else
        {
            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!;
        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);
        }
```

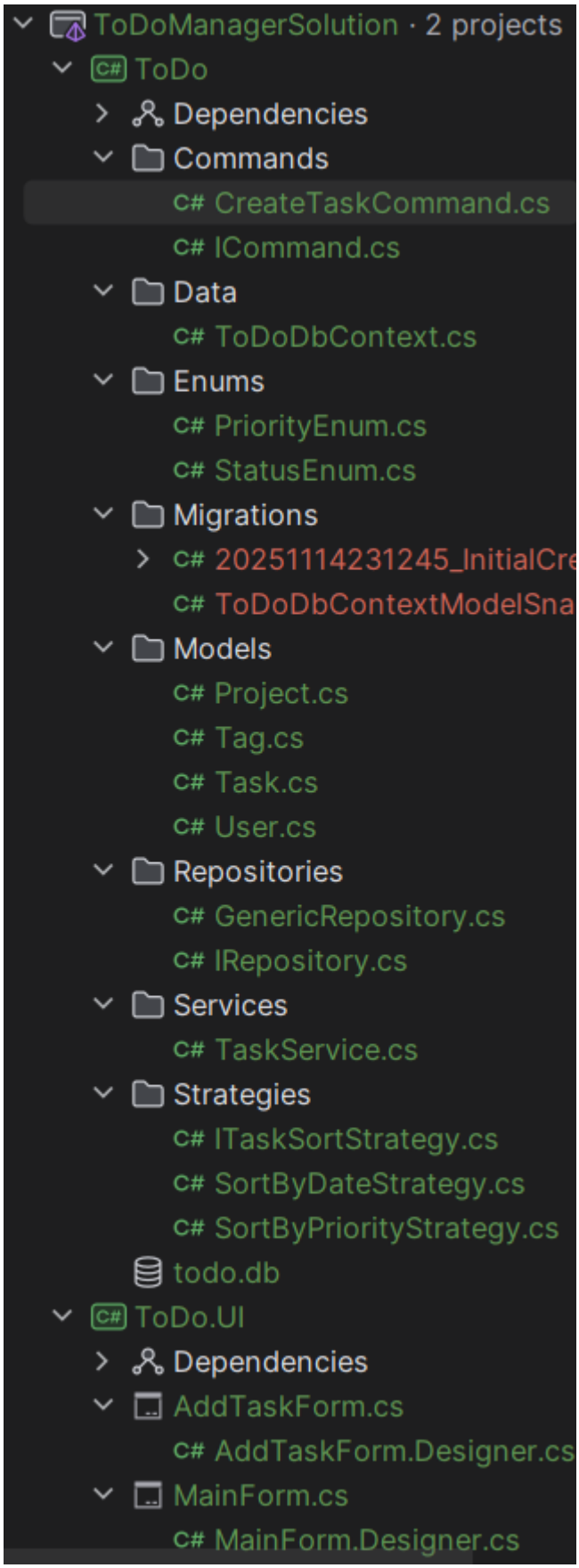
```
        using (var addTaskForm = new
AddTaskForm(defaultProject.ProjectId))
        {
            if (addTaskForm.ShowDialog() == DialogResult.OK)
            {
                try
                {
                    var newTask = new Task
                    {
                        Title = addTaskForm.TaskTitle!,
                        Description = string.Empty,
                        Priority = addTaskForm.TaskPriority,
                        ProjectId = defaultProject.ProjectId,
                        Status = StatusEnum.New,
                        DueDate = DateTime.Now.AddDays(1)
                    };

                    ICommand command = new CreateTaskCommand(_taskService,
newTask);
                    command.Execute();
                }
                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.AddSingleton<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 засвоїли принципи реалізації патерну Observer (Спостерігач) за допомогою механізму подій (events) C#. Ця реалізація дозволила усунути тісний зв'язок між шаром бізнес-логіки (TaskService) та візуальним інтерфейсом (MainForm). Тепер UI автоматично оновлюється за фактом зміни даних без ручного виклику функції оновлення, що забезпечує гнучкість та реактивність архітектури системи.

Питання до лабораторної роботи

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика» (Abstract Factory) використовується для створення сімейств об'єктів без вказівки їх конкретних класів. Це зручно для створення узгоджених наборів взаємопов'язаних об'єктів, наприклад, об'єктів для доступу до різних баз даних (ADO.NET).

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

Входять Client (Клієнт), AbstractFactory (Абстрактна фабрика), ConcreteFactory (Конкретна фабрика), AbstractProduct (Абстрактний продукт) та Product (Конкретний продукт) . Client взаємодіє з AbstractFactory для створення сімейств Product, а ConcreteFactory створює продукти конкретного типу.

3. Яке призначення шаблону «Фабричний метод»?

Шаблон «Фабричний метод» (Factory Method) визначає інтерфейс для створення об'єктів певного базового типу. Він служить "зачіпкою" для впровадження власного конструктора об'єктів у дочірніх класах.

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

Входять Product, ConcreteProduct, Creator та ConcreteCreator . Creator визначає фабричний метод, який перевизначається у ConcreteCreator для створення об'єктів ConcreteProduct.

5. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

«Абстрактна фабрика» використовується для покрокового визначення конкретного алгоритму, тоді як «Фабричний метод» використовується для створення об'єктів. (Також, Абстрактна фабрика створює сімейства продуктів, а Фабричний метод створює один продукт).

6. Яке призначення шаблону «Знімок»?

Шаблон «Знімок» (Memento) використовується для збереження та відновлення стану об'єктів без порушення інкапсуляції. Об'єкт «Memento» служить виключно для збереження змін над початковим об'єктом (Originator).

7. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

Входять Originator (Ініціатор), Memento (Знімок) та Caretaker (Опікун) . Originator має доступ до Memento для збереження/відновлення стану , а Caretaker використовується для зберігання та передачі об'єктів Memento.

8. Яке призначення шаблону «Декоратор»?

Шаблон «Декоратор» (Decorator) призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. Декоратор «обертає» початковий об'єкт, зберігаючи його функції, але додаючи додаткові дії.

9. Які класи входять в шаблон «Декоратор», та яка між ними взаємодія?

Входять Component (Інтерфейс), ConcreteComponent (Конкретний компонент), Decorator (Абстрактний декоратор) та ConcreteDecorator (Конкретний декоратор) . Decorator успадковує від Component і містить посилання на Component (через агрегацію), делегуючи йому виконання початкових операцій.

10. Які є обмеження використання шаблону «декоратор»?

Обмеження включають: велика кількість крихітних класів; труднощі у конфігурації об'єктів, які загорнуто в декілька обгорток одночасно.