

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

Лабораторна робота № 7

з дисципліни «Технології розроблення
програмного забезпечення»

Тема: «Патерни проектування.»

«Менеджер завдань»

Виконав:
студент групи - ІА-32
Воробйов Кирило
Андрійович

Перевірів:
Мягкий Михайло
Юрійович

Київ 2025

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати їх в реалізації програмної системи.

Тема проєкту: Менеджер завдань (To Do Manager)

Патерни: Strategy, Command, Observer, Template Method, 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 (Composite), забезпечувати автоматичне оновлення візуальних компонентів при зміні даних (Observer), а також надавати функціонал генерації звітів та експорту даних, використовуючи послідовний алгоритм (Template Method). Зберігання та синхронізація завдань реалізується через клієнт-серверний зв'язок (Client-Server).

Зміст

Теоретичні відомості	3
Хід роботи	9
Діаграма класів патерну Template Method.....	9
Код програми.....	11
Висновки.....	20
Питання до лабораторної роботи	20

Теоретичні відомості

Шаблон «Mediator»

Призначення патерну: Шаблон «Mediator» (посередник) використовується для визначення взаємодії об'єктів за допомогою іншого об'єкта (замість зберігання посилань один на одного) [6]. Даний шаблон схожий на шаблон «команда», проте в даному випадку замість зберігання даних про конкретну дію, зберігаються дані про взаємодії між компонентами.

Даний шаблон зручно застосовувати у випадках, коли безліч об'єктів взаємодіє між собою деяким структурованим чином, однак складним для розуміння. У такому випадку вся логіка взаємодії виноситься в окремий об'єкт.

Кожен із взаємодіючих об'єктів зберігає посилання на об'єкт «медіатор». «Медіатор» нагадує диригента при управлінні оркестром. Диригент стежить за тим, щоб кожен інструмент грав в правильний час і в злагоді з іншими інструментами. Функції «медіатора» повністю це повторюють.

Проблема: Ви розробляєте систему зі складною візуальною частиною. Головна форма з якою працює користувач складається з великою кількості візуальних компонентів, які в свою чергу складаються з елементарних візуальних компонентів. Для такої складної форми, як правило, є багато логіки, коли дії в одному компоненті повинні впливати на інші компоненти.

Наприклад, коли ви вибрали чекбокс, то після цього відключається можливість редагування даних в деяких частинах форми, деякі блоки мають приховатися, а інші навпаки, показатися. В такій ситуації, коли на формі може бути сотня, а то і більше візуальних контролів, то зв'язки між ними можуть вимірюватися тисячами. Додавляти зміни в функціонал на таких формах дуже складно, а знайти та виправити помилку, взагалі, може бути дуже затратно по витраченому часу.

Рішення: В таких ситуаціях дуже добре підходить патерн «Mediator» (Посередник). При реалізації такого патерна, ми додаємо новий клас посередник, і всі взаємодії між компонентами повинні відбуватися вже через нього. За рахунок цього компоненти вже не знають один про одного, а лише знають про посередника і для відпрацювання логіки звертаються вже до нього. А вже посередник взаємодіє з іншими компонентами для відпрацювання цієї логіки.

Основною перевагою, яку ми отримуємо при такому підході є сильно зменшена зв'язність між компонентами. А за рахунок цього стає

набагато простіше розібратися з взаємодією між об'єктами коли потрібно додати на форму нові об'єкти або додати якийсь функціонал. Якщо компоненти будуть взаємодіяти не напряму з посередником, а через загальний інтерфейс посередника, то тоді можна буде підміняти посередника, наприклад, на тестового, щоб емулювати тестові випадки для перевірки.

Переваги та недоліки:

- + Організація взаємодії між об'єктами лише через посередника спрощує розуміння та супроводження такого коду.
- + Додавання нових посередників без зміни існуючих компонентів дозволяє розширювати систему без зміни існуючого коду.
- + Зменшення залежностей між об'єктами підвищує гнучкість системи.
- Посередник, з часом, може перетворитися на дуже складний об'єкт, який робить все («God Object»).

Шаблон «Facade»

Призначення патерну: Шаблон «Facade» (фасад) передбачає створення єдиного уніфікованого способу доступу до підсистеми без розкриття внутрішніх деталей підсистеми [6]. Оскільки підсистема може складатися з безлічі класів, а кількість її функцій – не більше десяти, то щоб уникнути створення «спагетікоду» (коли все тісно пов'язано між собою) виділяють один загальний інтерфейс доступу, здатний правильним чином звертатися до внутрішніх деталей.

Це також відволікає користувачів від змін в підсистемі (внутрішня реалізація може змінюватися, а наданої послуги немає), що також скоротить кількість змін в використовуваних фасад класах (без фасаду довелося б змінювати вихідні коди в безлічі точок).

Звичайно, твердої умови повного закриття внутрішніх класів підсистеми не стоїть – при необхідності можна звертатися до окремих класів безпосередньо, минаючи об'єкт фасад.

Проблема: Ви розробляєте компонент, який дозволяє відправляти запити на різні типи endpoint, а також працює з протоколами HTTP та TCP/IP.

Прототип компонента вже працює, але структура класів вийшла досить складна, а при налаштуванні на різні протоколи мають використовуватися різні класи.

Інструкція для використання також виходить досить складна та заплутана. Слід додати, так як інші системи будуть знати про внутрішню будову вашого компонента, а тому при спробі змінити внутрішню структуру в наступних версіях, вам прийде повідомляти про це всіх користувачів вашого компонента і для них перехід на наступну версію вашого компонента буде достатньо складним.

Рішення: Тут краще використати патерн фасад. А саме, в даному випадку, створити один клас, наприклад, `InternetClient`, та набір методів у нього, які будуть

використовуватися для налаштування цього підключення, та його подальшого використання. Цей клас єдиний буде позначено як `public`, і тільки його будуть бачити ваші клієнти. Таким чином, з точки зору зовнішнього коду вони будуть працювати з набагато простішим інтерфейсом, а значить і інструкція використання буде значно простіша. Крім того, так як внутрішня структура повністю закрита, то в наступних версіях ви можете її змінювати, як вам буде зручно, а з точки зору користувачів компонента, перехід на нову версію буде просто переключенням на нову версію бібліотеки.

Переваги та недоліки:

- + Інкапсуляція внутрішньої структури від клієнтського коду.
- + Спрощується інтерфейс для роботи з модулем закритим фасадом.
- Зниження гнучкості в налаштуванні та використанні програмного коду закритого фасадом.

Шаблон «Bridge»

Призначення патерну: Шаблон «Bridge» (міст) використовується для поділу інтерфейсу і його реалізації. Це необхідно у випадках, коли може існувати кілька різних абстракцій, над якими можна проводити дії різними способами.

Проблема: Ви реалізовуєте графічний векторний редактор, який дозволяє рисувати кола, прямокутники, прямі та довільні лінії.

Ви маєте реалізувати функціонал відображення отриманого рисунку на екрані та друкувати на принтер.

Спростимо ситуацію: ваш редактор дозволяє рисувати лише лінії та кола.

При такому підході, нам потрібно будувати ієрархію фігур з різною реалізацією дочірніх класів: LinePrint, LineDraw, CirclePrint, CircleDraw.

якщо додати прямі, то добавиться ще два підкласи, і т.д. А як бути, коли нам потрібно буде ще і реалізувати збереження в bitmap форматі? добавляємо ще LineBinery, CircleBinery? При такому підході ми отримуємо дуже складну ієрархію класів.

Рішення: В даному випадку ми можемо використати патерн «Міст» (Bridge): робимо дві ієрархії – фігур (Shape) та рисування (DrawApi).

При такому підході DrawApi – це інтерфейс імплементації відображення (графічного драйвера), а Shape – інтерфейс абстракції фігур, яка має агрегацію з об'єктом DrawApi. При такому підході фігури будуть делегувати рисування об'єкту DrawApi.

Лінія, коло, та інші будуть дочірніми класами до Shape, а WindowDrawApi та PrinterDrawApi – дочірні класи до DrawApi, які представляють графічні драйвери для відображення на екрані та принтері відповідно. Якщо нам потрібно буде додати ще і збереження в bitmap форматі, то ми добавимо ще один підклас реалізації графічного драйвера BitmapDrawApi. Таким чином ми маємо дві різні ієрархії об'єктів і вони в нас не перетинаються і не збільшуються в геометричній прогресії при додаванні нових драйверів або фігур.

Також слід відмітити, що DrawApi нічого не знає про фігури (абстракцію), а дочірні класи абстракції не залежать від реалізації графічного драйвера.

Переваги та недоліки:

- + Дозволяє змінювати ієрархії абстракції та реалізації незалежно одна від одної.
- + Розділивши абстракцію від реалізації отримуємо більшу гнучкість та простіший супровід такого коду.
- Підвищена гнучкість при використанні патерну отримується за рахунок більшої складності, введення додаткових проміжних рівнів.

Шаблон «Template Method»

Призначення патерну: Шаблон «Template Method» (шаблонний метод) дозволяє реалізувати покроково алгоритм в абстрактному класі,

але залишити специфіку реалізації підкласам [6]. Можна привести в приклад формування вебсторінки: необхідно додати заголовки, вміст сторінки, файли, що додаються, і нижню частину сторінки. Код для додавання вмісту сторінки може бути абстрактним і реалізовуватися в різних класах – `AspNetCompiler`, `HtmlCompiler`,

`PhpCompiler` і т.п. Додавання всіх інших елементів виконується за допомогою вихідного абстрактного класу з алгоритмом.

Даний шаблон дещо нагадує шаблон «Фабричний метод», однак область його використання абсолютно інша – для покрокового визначення конкретного алгоритму; більш того, даний шаблон не обов'язково створює нові об'єкти – лише визначає послідовність дій.

Проблема: Ви працюєте в команді, що займається розробкою застосунку для редагування відео-файлів. Застосунок вже працює з форматом відео MPEG4, а саме дозволяє читати такі файли, виконувати попередню обробку даних для відображення в відео-редакторі.

Ви отримуєте нову задачу на реалізацію можливості роботи з більш старим форматом MPEG-2. Ви бачите два варіанта: зробити копію існуючого класу, що працює з MPEG-4, або вносити зміни в уже існуючий клас. Щоб прийняти рішення ви більш детально розбираєтеся з існуючим алгоритмом і бачите, що близько 70 відсотків коду має бути таким самим. Тому ви вирішуєте змінити вже існуючий клас для роботи з MPEG-4 додаючи в місцях де це потрібно умови з перевіркою, що якщо формат MPEG-2 то відпрацьовувати новий код, який ви добавили. Через деякий час, на запити від користувачів, вам на реалізацію приходить задача добавити підтримку ще більш старого формату MPEG-1. Ви вносите зміни так само в існуючий клас, тільки умови стали більш складними, тому що розгалуження логіки йде на три гілки.

Ще через деякий час приходить аналогічна задача на додавання читання даних з файлів формату H.262. Ви починаєте працювати над задачею і бачите, що код, який до цього був ще більш-менш зрозумілим стає зовсім важким для читання та внесення змін.

Рішення: Патерн «Шаблонний метод» (Template Method) пропонує

загальний алгоритм винести в базовий клас, а частини алгоритма, які для різних задач виконуються по різному, виділити в окремі методи. Ці методи будуть викликатися в алгоритмі, що реалізований в базовому класі. В дочірніх класах ці виділені методи будуть перевизначатися.

Таким чином загальна логіка залишається в базовому класі, а специфічна частина реалізується в дочірніх класах.

Якщо подивитися на задачу з відео-редактором, то застосування «Шаблонного методу» наведе лад в коді і спростить його зміни.

Як це зробити: По перше, в алгоритмі всі блоки коду де є вибір гілки на основі типу формату виділяються в окремі методи. У випадку з відеоредактором, це скоріш за все будуть блоки коду пов'язані з читанням даних та розпакування їх в кадри, а також читання звукових доріжок. Далі створюється загальний базовий клас в який переноситься загальний алгоритм, а також об'являються віртуальні методи (фактично беремо сигнатуру тих методів, що виділили на попередньому кроці). Далі створюємо дочірні класи під кожен формат файлу і перевизначаємо віртуальні методи. Фактично при цьому в кожному такому методі в дочірньому класі із реалізації цих методів, що була виділена на першому кроці, залишається код гілки який відповідав вибраному формату.

Після всіх цих змін ми маємо реалізацію патерна «Шаблонний метод»: в базовому класі реалізовано базовий алгоритм (по суті більша частина алгоритму) і в дочірніх класах перевизначені методи зі специфічною логікою.

Після таких змін, додати підтримку нового формату стає легше, тому що достатньо буде додати лише новий дочірній клас і перевизначити в ньому необхідні методи.

Слід зауважити, що якщо у вас алгоритми співпадають більше ніж на 50 відсотків, то застосування шаблонного методу буде доцільним, але якщо у вас алгоритми співпадають лише відсотків на 10 або 20, то скоріш за все, краще буде використати патерн «Стратегія».

Переваги та недоліки:

- + Полегшує повторне використання коду.
- Ви жорстко обмежені скелетом існуючого алгоритму.
- Ви можете порушити принцип підстановки Барбара Лісков, змінюючи базову поведінку одного з кроків алгоритму через підклас.
- З ростом складності загального алгоритму шаблонний метод стає занадто складно підтримувати, особливо, коли є багато віртуальних методів для перевизначення в підкласах.

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 Template Method

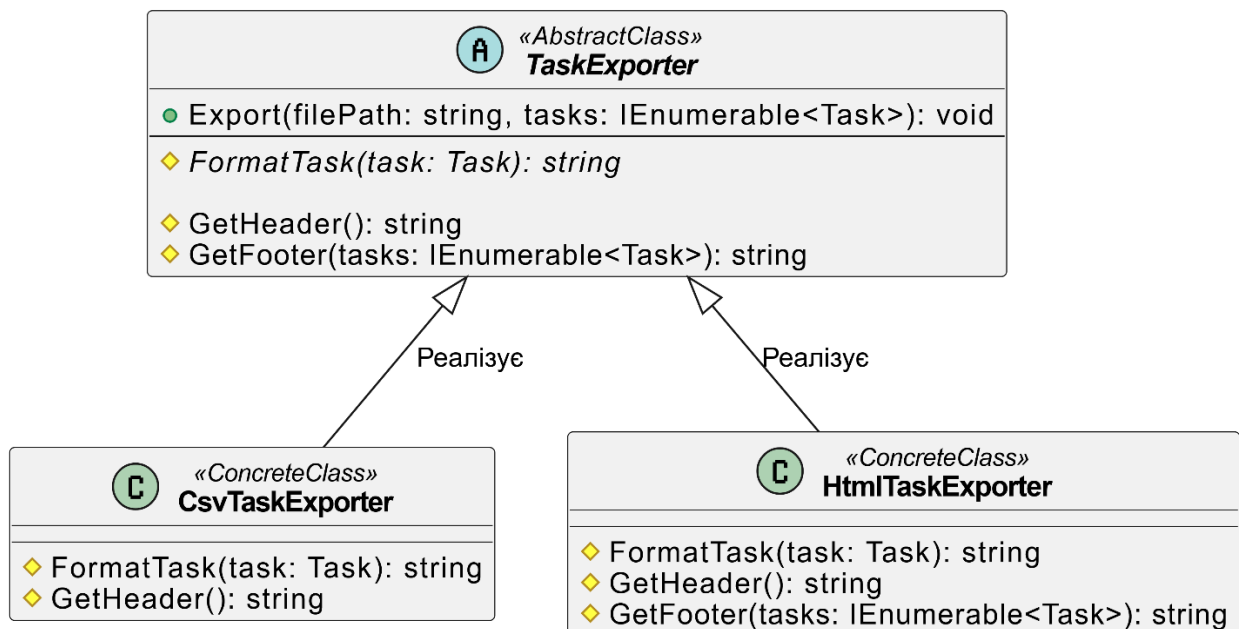


Рис. 1. Діаграма класів патерну Template Method

Діаграма демонструє реалізацію патерна Template Method для впровадження стандартизованого алгоритму експорту даних, забезпечуючи гнучкість у форматуванні.

1. Template Method (Шаблонний метод) — TaskExporter

- Роль: Виступає як абстрактний клас, що визначає незмінний скелет алгоритму експорту (порядок кроків). Метод Export() містить

послідовність дій (наприклад: відкрити файл → записати заголовок → форматувати кожен рядок → закрити файл).

- Абстрактні/Хук-методи: Вміст методу `Export()` викликає абстрактні (які потрібно реалізувати) або віртуальні (хук) методи, такі як `FormatTask()` або `GetHeader()`.

2. Primitive Operations (Примітивні операції) — `FormatTask()`

- Роль: Це абстрактний крок алгоритму, який обов'язково має бути реалізований у підкласах. Він дозволяє конкретним експортерам визначити специфічне форматування одного рядка даних (наприклад, CSV, HTML, або TXT).

3. Concrete Class (Конкретні класи) — `CsvTaskExporter`, `HtmlTaskExporter`

- Роль: Це конкретні реалізації, які успадковуються від `TaskExporter`. Вони реалізують примітивні операції та перевизначають хук-методи, надаючи логіку для специфічного формату (наприклад, додають HTML-теги або роздільники-коми).

4. Аргументація Гнучкості

- Ця архітектура дозволяє будь-якому новому формату звіту (наприклад, JSON) стати "Конкретним класом" без зміни основного алгоритму `Export()` у батьківському класі, дотримуючись принципу відкритості/закритості.

Код програми

TaskExporter.cs

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

namespace ToDo.Exporters
{
    public abstract class TaskExporter
    {
        // Template Method
        // послідовність кроків, яку не можна змінити
        public void Export(string filePath, IEnumerable<Task> tasks)
        {
            using (var writer = new StreamWriter(filePath, false, Encoding.UTF8))
            {
                writer.WriteLine(GetHeader());

                foreach (var task in tasks)
                {
                    writer.WriteLine(FormatTask(task));
                }

                writer.WriteLine(GetFooter(tasks));
            }
        }

        // Форматування одного завдання в рядок
        protected abstract string FormatTask(Task task);

        protected virtual string GetHeader()
        {
            return "";
        }

        protected virtual string GetFooter(IEnumerable<Task> tasks)
```

```

    {
        return "";
    }
}

```

CsvTaskExporter.cs

```

using System.Collections.Generic;
using System.IO;
using System.Text;
using ToDo.Models;
using Task = ToDo.Models.Task;

namespace ToDo.Exporters
{
    public abstract class TaskExporter
    {
        // Template Method
        // послідовність кроків, яку не можна змінити
        public void Export(string filePath, IEnumerable<Task> tasks)
        {
            using (var writer = new StreamWriter(filePath, false, Encoding.UTF8))
            {
                writer.WriteLine(GetHeader());

                foreach (var task in tasks)
                {
                    writer.WriteLine(FormatTask(task));
                }

                writer.WriteLine(GetFooter(tasks));
            }
        }

        // Форматування одного завдання в рядок
        protected abstract string FormatTask(Task task);

        protected virtual string GetHeader()
        {
            return "";
        }
    }
}

```

```

    }

    protected virtual string GetFooter(IEnumerable<Task> tasks)
    {
        return "";
    }
}

```

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 ToDo.Exporters;
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;

        // Поля для експортерів
        private readonly CsvTaskExporter _csvExporter;
        private readonly HtmlTaskExporter _htmlExporter;

        // Оновлений конструктор з новими залежностями
        public MainForm(
            TaskService taskService,

```

```

        IRepository<Project> projectRepository,
        IRepository<User> userRepository,
        IEnumerable<ITaskSortStrategy> sortStrategies,
        CsvTaskExporter csvExporter, // Омпумуємо CsvExporter через DI
        HtmlTaskExporter htmlExporter) // Омпумуємо HtmlExporter через
DI
    {
        InitializeComponent();

        _taskService = taskService;
        _projectRepository = projectRepository;
        _userRepository = userRepository;
        _sortStrategies = sortStrategies;

        // Зберігаємо експортери
        _csvExporter = csvExporter;
        _htmlExporter = htmlExporter;

        _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();

        // Перевірка на null для безпеки
        if (cmbSortStrategy.SelectedItem is not ITaskSortStrategy
selectedStrategy)
            return;

        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);
            }
        }
    }

```

```
}  
}  
}  
}
```

```
private void btnExport_Click(object sender, EventArgs e)  
{  
    using (var saveFileDialog = new SaveFileDialog())  
    {  
        // Налаштування діалогу збереження  
        saveFileDialog.Filter = "CSV файл (*.csv)|*.csv|HTML сторінка  
(* .html)|*.html";  
        saveFileDialog.Title = "Експорт завдань";  
        saveFileDialog.FileName = "tasks_export";  
  
        if (saveFileDialog.ShowDialog() == DialogResult.OK)  
        {  
            try  
            {  
                var tasks = _taskService.GetAllTasks();  
                string filePath = saveFileDialog.FileName;  
  
                // Вибір стратегії експорту  
                TaskExporter exporter;  
  
                if (filePath.EndsWith(".csv"))  
                {  
                    exporter = _csvExporter;  
                }  
                else  
                {  
                    exporter = _htmlExporter;  
                }  
  
                // Виклик шаблонного методу  
                exporter.Export(filePath, tasks);  
  
                MessageBox.Show("Дані успішно експортовано!", "Успіх",  
                MessageBoxButtons.OK, MessageBoxIcon.Information);  
            }  
            catch { }  
        }  
    }  
}
```

```

    }
    catch (Exception ex)
    {
        MessageBox.Show($"Помилка експорту: {ex.Message}",
"Помилка", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
}
}
}
}
}
}
}

```

HtmlTaskExporter.cs

```

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

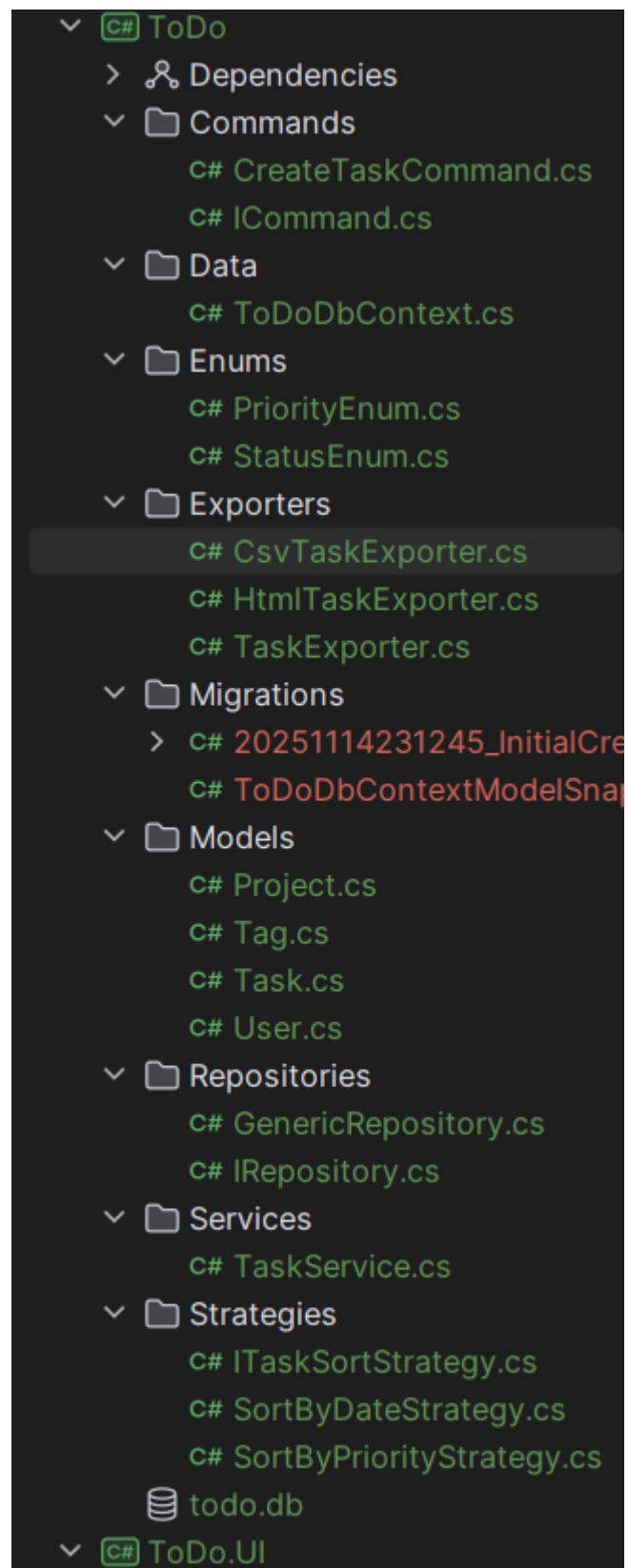
namespace ToDo.Exporters
{
    public class HtmlTaskExporter : TaskExporter
    {
        protected override string GetHeader()
        {
            return "<html><head><title>Список
завдань</title></head><body><h1>Мої Завдання</h1><ul>";
        }

        protected override string FormatTask(Task task)
        {
            return $"<li><b>{task.Title}</b> [{task.Priority}] -
<i>{task.Status}</i></li>";
        }

        protected override string GetFooter(IEnumerable<Task> tasks)
        {
            return $"</ul><p>Всього завдань:
{tasks.Count()}</p></body></html>";
        }
    }
}

```

}
}
}



Висновки

Висновки: під час виконання лабораторної роботи ми успішно засвоїли принципи реалізації патерну Template Method (Шаблонний метод). Цей патерн дозволив нам уніфікувати алгоритм експорту даних (наприклад, у формати CSV та HTML). Шляхом створення абстрактного класу TaskExporter та його конкретних підкласів ми забезпечили гнучкість та дотрималися принципу відкритості/закритості, що дозволяє легко додавати нові формати звітів без зміни основного коду програми.

Питання до лабораторної роботи

1. Яке призначення шаблону «Посередник»?

Шаблон «Mediator» (Посередник) використовується для визначення взаємодії об'єктів за допомогою іншого об'єкта (замість зберігання посилань один на одного). Він зручний, коли безліч об'єктів взаємодіє між собою складним чином, і дозволяє винести всю логіку взаємодії в окремий об'єкт.

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

Входять Mediator (Посередник), Colleague (Колега) та ConcreteColleague (Конкретний колега). Кожен із взаємодіючих об'єктів (Colleague) зберігає посилання на об'єкт Mediator, і всі взаємодії відбуваються лише через нього.

3. Яке призначення шаблону «Фасад»?

Шаблон «Facade» (Фасад) передбачає створення єдиного уніфікованого способу доступу до підсистеми без розкриття її внутрішніх деталей. Це допомагає уникнути створення «спагеті-коду» та відволікає користувачів від внутрішніх змін у підсистемі.

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

Головним елементом є клас Facade, який містить набір спрощених методів для доступу до складної Subsystem (Підсистема). Facade керує внутрішніми класами підсистеми і перенаправляє до них запити від клієнтського коду.

5. Яке призначення шаблону «Міст»?

Шаблон «Bridge» (Міст) використовується для поділу інтерфейсу і його реалізації. Це необхідно у випадках, коли може існувати кілька різних абстракцій, над якими можна проводити дії різними способами.

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

Входять Client (Клієнт), Abstraction (Абстракція), RefinedAbstraction (Уточнена Абстракція), Implementor (Інтерфейс реалізації) та ConcreteImplementor (Конкретна реалізація). Abstraction містить посилання на Implementor та делегує йому виконання операцій.

7. Яке призначення шаблону «Шаблонний метод»?

Шаблон «Template Method» (Шаблонний метод) дозволяє реалізувати покроково алгоритм в абстрактному класі, але залишити специфіку реалізації підкласам. Він визначає послідовність дій.

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

Входять AbstractClass (Абстрактний клас) та ConcreteClass (Конкретний клас). AbstractClass містить TemplateMethod (Шаблонний метод), який викликає послідовність PrimitiveOperation; ці примітивні операції перевизначаються у ConcreteClass.

9. Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?

«Шаблонний метод» використовується для покрокового визначення конкретного алгоритму і не обов'язково створює нові об'єкти.
«Фабричний метод» використовується для створення об'єктів.

10. Яку функціональність додає шаблон «Міст»?

Шаблон «Міст» додає функціональність розділення ієрархій абстракції та реалізації, дозволяючи змінювати їх незалежно одну від одної. Це дає більшу гнучкість та простіший супровід коду.