# Зміст

[Зміст 1](#_Toc169546419)

[1. Теоретична частина 4](#_Toc169546420)

[1.1 Означення шаблонів проєктування 4](#_Toc169546421)

[1.2 Складові шаблонів проєктування 4](#_Toc169546422)

[1.3 Класифікація шаблонів проєктування 5](#_Toc169546423)

[1.4 Породжувальні шаблони 5](#_Toc169546424)

[1.4.1 Одинак 5](#_Toc169546425)

[1.4.2 Будівельник 6](#_Toc169546426)

[1.4.3 Прототип 8](#_Toc169546427)

[1.4.4 Фабричний метод 9](#_Toc169546428)

[1.4.5 Абстрактна фабрика 10](#_Toc169546429)

[1.5 Структурні шаблони 11](#_Toc169546430)

[1.5.1 Перехідник 11](#_Toc169546431)

[1.5.2 Міст 12](#_Toc169546432)

[1.5.3 Компонувальник 13](#_Toc169546433)

[1.5.4 Декоратор 14](#_Toc169546434)

[1.5.5 Фасад 15](#_Toc169546435)

[1.5.6 Легковаговик 16](#_Toc169546436)

[1.5.7 Замісник 17](#_Toc169546437)

[1.6 Поведінкові шаблони 18](#_Toc169546438)

[1.6.1 Ланцюг відповідальності 18](#_Toc169546439)

[1.6.2 Команда 20](#_Toc169546440)

[1.6.3 Ітератор 21](#_Toc169546441)

[1.6.4 Посередник 22](#_Toc169546442)

[1.6.5 Знімок 24](#_Toc169546443)

[1.6.6 Стан 25](#_Toc169546444)

[1.6.7 Стратегія 26](#_Toc169546445)

[1.6.8 Шаблонний метод 27](#_Toc169546446)

[1.6.9 Відвідувач 28](#_Toc169546447)

[1.6.10 Спостерігач 29](#_Toc169546448)

[1.7 Висновок 30](#_Toc169546449)

[2 Практична частина 31](#_Toc169546450)

[2.1 Аналіз та проєктування 31](#_Toc169546451)

[2.1.1 Постановка завдання 31](#_Toc169546452)

[2.1.2 Дорожня карта 31](#_Toc169546453)

[2.1.3 Структура даних 31](#_Toc169546454)

[2.1.4 База даних 31](#_Toc169546455)

[2.1.5 Архітектура 31](#_Toc169546456)

[2.1.5.1 Вступ 31](#_Toc169546457)

[2.1.5.2 Архітектурний (основний) Шаблон 31](#_Toc169546458)

[2.1.5.3 Авторизація 31](#_Toc169546459)

[2.1.5.4 Здійснення запитів 31](#_Toc169546460)

[2.3 Програмна реалізація 31](#_Toc169546461)

[2.3.1 Вибір технологій 31](#_Toc169546462)

[2.3.2 Інтерфейс користувача 31](#_Toc169546463)

[2.3.3 Бізнес-логіка Проєкту 31](#_Toc169546464)

[2.3.3.1 Шаблон Form Observer 31](#_Toc169546465)

[2.3.3.2 Авторизація користувача 31](#_Toc169546466)

[2.3.3.3 Здійснення запитів до бази даних 31](#_Toc169546467)

[2.4 Тестування 31](#_Toc169546468)

[2.4.1 Компіляція 31](#_Toc169546469)

[2.4.2 Запити 31](#_Toc169546470)

[Список використаних джерел 32](#_Toc169546471)

# 1. Теоретична частина

## Означення шаблонів проєктування

Шаблони проєктування (надалі шаблони /патерни) – **це рішення проблем, що часто виникають при проєктуванні** (надалі паралельно дизайні) **програмного забезпечення**. Немовби готові креслення, які можна налаштували для вирішення повторюваних проблем проєктування в коді.

Патерни не можна просто скопіювати в код програми як бібліотеки чи готові функції. Це не фрагменти коду, а **загальні** **концепції** вирішення конкретних проблем, яких можна дотримуватись для реалізації відповідного для конкретного програмного застосунку рішення.

Шаблони часто плутають з алгоритмами, оскільки вони обоє описують типові рішення відомих проблем. Однак алгоритми визначають **чіткий набір дії, що ведуть до певного результату**, а патерни містять більш **високорівневий опис рішення, код реалізації якого може сильно відрізнятись** у різних програмах.

[1]

## Складові шаблонів проєктування

Більшість патернів містять наступні секції:

* **Мета**: короткий опис проблеми та її рішення
* **Мотивація**: додаткове пояснення проблеми й рішення, яке робить можливим шаблон
* **Структура** класів, яка показує кожну частину патерну й зв’язок між ними.
* Приклад **коду** однією з популярних мов програмування полегшує розуміння ідеї, що лежить в основі патерну.

[1]

## Класифікація шаблонів проєктування

Патерни класифікують за **складністю**, **деталізацією** та **масштабом** застосування відносно всієї системи.

Найпростіші патерни найнижчого рівня часто називають **ідіомами**. Зазвичай їх можна застосовувати лише в одній мові програмування.

На противагу їм, найбільш складні та високорівневі шаблони називають **архітектурними**. Вони підходять для дизайну всього програмного забезпечення.

Крім того всі шаблони можна поділити на 3 категорією залежно від їх мети: на **породжувальний**, **структурні** та **поведінкові** патерни.

[2]

## Породжувальні шаблони

Цей тип патернів надає різноманітні **механізми створення об’єктів**, які підвищують гнучкість та можливість повторного використання коду.

До них належать: **Одинак**, **Будівельник**, **Прототип**, **Фабричний метод** та **Абстрактна фабрика**.

[2]

### Одинак

Одинак (Синглтон) – це породжувальний патерн, який гарантує, **наявність лише одного екземпляру класу**, забезпечуючи при цьому глобальну точку доступу до нього.

**Проблема**

**Гарантія єдиного екземпляру класу**. Найпоширенішою причиною потреби в єдиному екземплярі класу є контроль доступу до деяких спільних ресурсів, наприклад: бази даних або файлу.

**Глобальна точку доступу до цього екземпляра**. На відміну від звичайної глобальної (статичної) змінної, Синглтон забезпечує захист глобальної точки доступу від перезапису із зовні, при цьому вся логіка міститься всередині самого класу.

**Рішення**

Всі реалізації одинака містять **2** наступні **елементи**:

1. **Приватний конструктор** за замовчуванням, який не дає іншим об’єктам використовувати оператор **new** з класом.
2. **Статичний метод, що заміняє конструктор**. Цей метод викликає приватний конструктор для створення об'єкта і зберігає його в статичному полі. Усі наступні виклики цього методу повертають вже створений об’єкт.

[3]

Рис. 1.4.1.1 – Структура класів реалізації патерна Одинак

Зображення, що містить текст, знімок екрана, ряд, Шрифт

Автоматично згенерований опис

### Будівельник

**Будівельник** – це породжувальний патерн, який дозволяє **поділяти створення складних об’єктів на етапи**. Шаблон дозволяє створювати **різні типи й представлення об'єкта**, використовуючи **один і той самий код побудови**.

**Проблема**

Потрібно **створити складний об'єкт, який вимагає трудомісткої, покрокової ініціалізації багатьох полів і вкладених об'єктів**. Зазвичай такий **код ініціалізації** ховається **всередині монструозного конструктора** з великою кількістю параметрів. Або ще гірше: **розкиданий по клієнтському коду**.

**Рішення**

Шаблон Будівельник пропонує **витягти код побудови об'єкта** з його власного класу **в** окремі **об'єкти, які називаються будівельниками** або конструкторами.

Шаблон організовує побудову об'єкта у вигляді набору кроків. Важливо те, що **не обов’язково викликати всі кроки**, а лише необхідні для створення певної конфігурації об'єкта.

Деякі з кроків побудови можуть вимагати **різної реалізації**, коли потрібно створити різні представлення продукту. У цьому випадку можна створити **кілька різних класів-конструкторів**, які реалізують той самий набір кроків побудови, але у різний спосіб. Потім можна використовувати цих будівельників для створення різних типів об'єктів.

Кріт того можна зберегти **послідовність** виконання різних **кроків** побудови **в окремий клас – директор**. При цьому можна використовувати **різних будівельників для одного директора**.

[4]

Рис. 1.4.2.1 – Структура класів реалізації патерна Будівельник

Зображення, що містить текст, схема, знімок екрана, Шрифт

Автоматично згенерований опис

### Прототип

Прототип – це породжувальний патерн, який дозволяє **копіювати** існуючі **об'єкти**, **не роблячи** ваш **код** **залежним** від їхніх **класів**.

**Проблема**

Потрібно **створити точну копію існуючого об’єкта**. Найпростіший спосіб – створити новий об'єкт того ж класу, пройтись по всіх полях оригінального об'єкта і скопіювати їх значення до нового об'єкта.

Однак далеко **не всі поля можуть бути публічними**. Крім того такий **код залежний від класу** об’єкта. Який, своєю чергою далеко не завжди відомий, наприклад при роботі з інтерфейсом, який реалізує цей клас.

**Рішення**

Шаблон Прототип **делегує процес клонування об'єктам, які власне клонуються**. Патерн використовує спільний **інтерфейс** для всіх об'єктів, які підтримують клонування. Цей інтерфейс, який зазвичай містить лише один метод, **дозволяє клонувати об'єкт без зв'язування коду з їхнім класом**. Крім того, при такому клонуванні, **об’єкт, що копіюється має доступ до приватних полів копії**.

[5]

Рис. 1.4.3.1 – Структура класів реалізації патерна Прототип

Зображення, що містить текст, знімок екрана, Шрифт, схема

Автоматично згенерований опис

### Фабричний метод

Фабричний метод - це породжувальний патерн, який надає **інтерфейс для створення об'єктів у суперкласі**, при цьому дозволяючи підкласам **змінювати тип об'єктів**, які будуть. створені.

**Проблема**

Розглянемо приклад. Додаток для **управління логістикою**, який обробляє **вантажні перевезення**. часом виникає потреба **реалізувати й морські перевезення**.

Доведеться внести **значні зміни** до всього **коду** – у великій мірі його доведеться переписати. Ба більше, **додавання інших видів перевезень зідйсснення вимагатиме подібних дій**.

**Рішення**

Патерн Фабричний метод пропонує **замінити прямі виклики створення об'єктів** (тут вантажівок та кораблів) на **виклики спеціального фабричного методу**.

Тепер можна **перевизначити фабричний метод у підкласі** і змінити клас продуктів, що створюються цим методом.

**Клієнтський код не бачить різниці між продуктами**, що повертаються різними підкласами, **знаючи лише, що це транспорт**, який може здійснювати перевезення.

[6]

Рис. 1.4.4.1 – Структура класів реалізації патерна Фабричний метод

Зображення, що містить текст, знімок екрана, схема, ряд

Автоматично згенерований опис

### Абстрактна фабрика

Абстрактна фабрика - це породжувальний шаблон, який дозволяє створювати **сім'ї пов'язаних об'єктів**, не вказуючи їх конкретних класів.

**Проблема**

Розглянемо приклад. **Симулятор магазину меблів**. Код містить **набір меблів різного типу і стилі кожного набору**. Потрібно створювати різні **меблі, які будуть підходити** одне одному **за стилем**. Крім того слід максимально **спростити додавання** нових **стилів меблів**, адже це доволі частий процес.

**Рішення**

Перше, що пропонує Абстрактна фабрика – оголосити **інтерфейси для кожного окремого виду меблів** (стільця, дивана…). Всі конкретні **продукти повинні їх реалізовувати**, наприклад інтерфейс Chair для кожного стільця.

Наступним кроком буде **оголошення** AbstractFamily– **інтерфейсу зі списком методів створення всіх видів меблів** (CreateChair, CreateSofa…). Ці методи повинні повертати абстрактні типи продуктів, представлені інтерфейсами, створеними раніше.

**Для кожного стилю** продуктів ми створюємо **окремий клас фабрики** на основі інтерфейсу AbstractFactory. **Фабрика - це клас, який повертає продукти певного типу**. Наприклад, ModernFurnitureFactory може створювати лише об'єкти ModernChair, ModernSofa…

**Клієнтський код працює** як з фабриками, так і з продуктами **через відповідні інтерфейси** тож конкретні їх **реалізації** ніяк **на** цей **код не впливають**.

Що стосується **створення самих фабрик** (тобто справжніх конкретних об’єктів), то зазвичай **застосунок створює** їх **на етапі ініціалізації** **залежно від конфігурації середовища** (наприклад обраного користувачем стилю меблів).

[7]

Рис. 1.4.5.1 – Структура класів реалізації патерна Абстрактна фабрика

Зображення, що містить текст, квитанція, схема, знімок екрана

Автоматично згенерований опис

## Структурні шаблони

Цей група патернів **допомагає збирати об’єкти в складні структури**, при цьому зберігаючипростоту та гнучкість робити з ними.

До них відносяться **Перехідник**, **Міст**, **Компонувальник**, **Декоратор**, **Фасад**, **Легковаговик** та **Посередник**.

[2]

### 1.5.1 Перехідник

Перехідник – це структурний шаблон проєктування, що **надає** двом **несумісним об’єктам інтерфейс** для **взаємодії**.

**Проблема**

Приклад. **Застосунок для моніторингу ринку**. Для своєї роботи він **бере дані** зі сторонніх ресурсів **у** форматі **JSON.** Для покращення роботи програми **було вирішено застосувати сторонню бібліотеку**, що відповідає за аналітику. Однак вона **підтримує лише** **XAML**. **Переписувати** **її** **код** складно, дорого, небезпечно й загалом **недоцільно**.

**Рішення**

**Потрібно створити перехідник** – спеціальний об’єкт, що **перетворює інтерфейс** одного **об’єкта** на той, що підходить для іншого. В нашому випадку **потрібно створити клас, що буде конвертувати JSON в XAML**. Всі використання сторонньої бібліотеки будуть здійснюватися через нього. При цьому **не доведеться** якось **редагувати наявний код**.

[8]

Рис. 1.5.1.1 – Структура класів реалізації патерна Перехідник

Зображення, що містить текст, знімок екрана, Шрифт, ряд

Автоматично згенерований опис

### Міст

Міст – це структурний шаблон проєктування, який **дозволяє** **розділити великий клас або набір тісно пов'язаних класів на дві окремі ієрархії** – абстракцію та реалізацію, незалежні одна від одної.

**Проблема**

Черговий приклад. Маємо **геометричні фігури**,які мають форму (куб, сфера, піраміда…) **та колір** (червоний, синій…). Перелічені **фігури є підкласами** суперкласу фігури. **Кожна комбінація** форми й кольору **вимагає** окремого **класу** (наприклад синя сфера, червона піраміда…). Таким чином **додавання** нової форми чи кольору **вимагає створення безлічі об’єктів** (фігур нового кольору всіх форм або фігур нової форми всіх кольорів).

**Рішення**

Патерн Міст пропонує **замінити успадковування на композицію** абстракції та реалізації. В нашому прикладі це означає, що потрібно **виділити з об’єкту фігури** (абстракції) **колір** (реалізацію). Створити **окремий клас колір**, що матиме набір дочірніх. Сама ж **фігура залишить** лише **набір підкласів, що визначатимуть її форму**.

Тепер **для створення** конкретної **фігури треба** лише **вибрати** її **тип** (форму) та **передати** їй **фігуру на 2 класи**. **Додавання** нової форми чи кольору **більше не вимагає створення набору нових об’єктів**.

[9]

Рис. 1.5.2.1 – Структура класів реалізації патерна Міст

Зображення, що містить текст, знімок екрана, Шрифт, число

Автоматично згенерований опис

### Компонувальник

Композит - це структурний патерн проєктування, який **дозволяє компонувати об'єкти у деревоподібні структури**, а потім **працювати** з цими структурами **так, ніби вони є** окремими **об'єктами**.

**Проблем**

Цей шаблон **має сенс** лиши при роботі **з моделями,** **які можна представити** **деревоподібно**. Наприклад маємо **продукти й коробки**, які можуть їх містити, їх або інші коробки, які можуть містити те саме й так далі.

Потрібно створити **систему для обрахунку вартості замовлень**. Для цього варто отримати всі продукти й додати їхню вартість. Найпростіший спосіб – **відкрити всі коробки**, підкоробки… Однак насправді це **досить складно**, як мінімум потрібно точно знати з яким об’єктом відбувається робота, що не завжди зручно реалізувати.

**Реалізація**

Патерн Компонувальник пропонує **працювати з усіма** цими **об’єктами через спільний інтерфейс**, що містить єдиний метод для отримання їх вартості.

**Продукти будуть** просто **повертати свою вартість**, а той час як **коробки** повертатимуть **сумарну вартість** всього **свого** **вмісту**. Таким чином не потрібно зосереджуватись на конкретних об’єкта при роботі з ними.

[10]

Рис. 1.5.3.1 – Структура класів реалізації патерна Компонувальник

Зображення, що містить текст, знімок екрана, схема, Шрифт

Автоматично згенерований опис

### Декоратор

Декоратор – це структурний шаблон проєктування, який **дозволяє додавати нові поведінки до об'єктів**, **розміщуючи ці об'єкти всередині** спеціальних **обгорткових об'єктів**, що містять ці поведінки.

**Проблема**

Приклад. **Бібліотека для** **надсилання сповіщень**. Перша версія – основою є **клас Notifier**, який **містить набір email-адрес** **та метод** через **який відправляє їм** вказане як параметр **текстове повідомлення**. Друга версія – **додано методи для надсилання сповіщень через SMS, Viber, Telegram, WhatsApp** тощо. Третя версія – потрібно створити **Notifier**, що надсилає сповіщення через **кілька сервісів одночасно** (при чому **можливе будь-яке поєднання**) – проблема.

**Реалізація**

Потрібно використати **агрегацію та композицію**. **Декоратор** – ключовий класвідповідного патерну – це **об’єкт (**1 чи кілька), **якому делегується робота певного класу** (який починає чимось нагадувати **інтерфейс**).

Як це працює? **Клас**, в нашому випадку Notifier, **містить набір – об’єктів декораторів**, **кожен** з яких має **таку саму** як ів Notifier’a **сигнатуру** (метод для надсилання сповіщень) однак зовсім різну реалізацію. **Виклик методів об’єкта** насправді **викликатиме** відповідний **метод у всіх його декораторів**. Ці **декоратори можна додавати й видаляти** під час виконання програми із клієнтського коду.

Наприклад: якщо **потрібно створити клас** **для надсилання сповіщень через SMS, Telegram, WhatsApp**. Потрібно створити **новий Notifier** **і** додати йому **3 відповідні декоратори**.

[11]

Рис. 1.5.4.1 – Структура класів реалізації патерна Декоратор

Зображення, що містить текст, знімок екрана, Шрифт, ряд

Автоматично згенерований опис

### Фасад

Фасад – це структурний шаблон проєктування, який **надає спрощений інтерфейс до** бібліотеки, фреймворку або будь-якого іншого **складного набору класів**.

**Проблема**

**Робота з набором складних бібліотек** часто **робить код** в значній мірі **залежним від їх** внутрішньої **реалізації**, що може сильно його ускладнити, адже ці **бібліотеки можуть** **містити** **багато прихованих залежностей**, які доведеться враховувати при кожному їх використанні.

**Реалізація**

Для вирішення цієї проблеми можна використати **фасад** – клас, що **надає простий доступ до комплексних підсистем** (як от бібліотеки), що містять багато рухомих частин. **Для** кожної **бібліотеки** можна **створити фасад**, що **містить лише** ті **методи**, які **безпосередньо потрібні для роботи** програми.

[12]

Рис. 1.5.5.1 – Структура класів реалізації патерна Фасад

Зображення, що містить текст, знімок екрана, Шрифт, ряд

Автоматично згенерований опис

### Легковаговик

Легковаговик – це структурний шаблон проєктування, який **дозволяє вмістити більше об'єктів у доступну оперативну пам'ять**, розділяючи спільні частини стану між кількома об'єктами, замість того, щоб зберігати всі дані в кожному з них.

**Проблема**

Приклад. **Відеогра, в якій є стрільба,** яка реалізується **через створення об’єктів** – **куль**, кожен з яких **містить набір параметрів**. Однак **гра** з часом **починає повільно працювати**. З часом – тобто після створення значної кількості куль.

**Реалізація**

Як вже вказувалось, кожна **куля містить набір параметрів**, однак **більшість з них спільні** для частини або взагалі всіх куль. Легковаговик пропонує **зберегти** ці **спільні параметри в** один **об’єкт**, щоб не дублювати їх. Всі **кулі** певного типу **будуть зберігати** **посилання на** цей **об’єкт**, яке займатиме куди менше пам’яті ніж його вміст.

[13]

Рис. 1.5.6.1 – Структура класів реалізації патерна Легковаговик

Зображення, що містить текст, знімок екрана, Шрифт, ряд

Автоматично згенерований опис

### Замісник

Замісник – це структурний шаблон проєктування, який **дозволяє створити замінник іншого об'єкта**. Проксі **контролює доступ до оригінального об'єкта**, дозволяючи виконати щось до або після того, як запит дійде до оригінального об'єкта.

**Проблема**

Маємо **масивний об’єкт**, **підключення до якого вимагає час**, часто значний. Наприклад **сервер**. Загалом можна **відкласти ініціалізацію** серверав **окремий потік** і чекати доки вона відбудеться.

**Реалізація**

Однак **не завжди є можливість редагувати оригінальний клас**. Тоді можна **поставити** цю логіку **в об’єкти, що викликають сервер** однак це спричинить **безліч** **дублікату коду**. Все що треба – **створити замісник**, об’єкт **з тим самим інтерфейсом взаємодії**, що й сервер, який **реалізовуватиме відкладений виклик оригінального об’єкту**.

[14]

Рис. 1.5.7.1 – Структура класів реалізації патерна Замісник

Зображення, що містить текст, знімок екрана, схема, Шрифт

Автоматично згенерований опис

## Поведінкові шаблони

Цей набір патернів стосується **алгоритмів** та **розподілу обов’язків** між об’єктами.

З них виділяють наступні: **Ланцюг відповідальності**, **Команда**, **Ітератор**, **Посередник**, **Знімок**, **Стан**, **Стратегія**, **Шаблонний метод** та **Відвідувач**.

[2]

### 1.6.1 Ланцюг відповідальності

Ланцюжок відповідальності - це поведінковий патерн проєктування, який дозволяє **передавати запити по ланцюжку обробників**. Отримавши запит, кожен обробник вирішує, **обробити** його, **чи передати наступному обробнику** в ланцюжку.

**Проблема**

Розглянемо приклад. **Система онлайн-замовлень**. **Доступ до системи обмежений** так, що лише авторизовані **користувачі могли створювати замовлення**. Крім того, **користувачі з правами адміністратора повинні мати повний доступ** до всіх замовлень.

Очевидно, що ці **перевірки повинні виконуватися послідовно**. Якщо не вдалося автентифікувати користувача, немає сенсу продовжувати будь-які інші перевірки.

Було впроваджено **більше перевірок**:

* перевірку, яка фільтрує повторні невдалі запити, що надходять з однієї IP-адреси.
* перевірка, яка пропускає запит до системи лише за відсутності відповідної кешованої відповіді.

В результаті код став **громіздким** і **незручним**. **При** **додаванні** нових **перевірок** все **ставатиме** лише **складніше**.

**Рішення**

Ланцюжок відповідальності покладається на **перетворення поведінки в окремі об'єкти**, які називаються обробниками. Кожна **перевірка повинна бути винесена** **в** **окремий клас** з єдиним методом, який виконує її. Цьому методу в якості аргументу передається запит разом з його даними.

Шаблон пропонує **зв'язати ці обробники в ланцюжок**. Кожен зв'язаний **обробник** має поле для зберігання посилання на наступний в ланцюжку й **може передати запит далі** **або** не робити цього й **зупити процес**.

**В наведеному прикладі** **кожен етап перевірки** (будь то перевірка автентичності або кешування) **винесений в окремий обробник** й всі вони знаходять в ланцюжку аж до обробника перевірки прав адміністратора,.

[15]

Рис. 1.6.1.1 – Структура класів реалізації патерна Ланцюг відповідальності

Зображення, що містить текст, знімок екрана, Шрифт, схема

Автоматично згенерований опис

### 1.6.2 Команда

Команда – це поведінковий патерн проєктування, який **перетворює запит на окремий об'єкт**, що містить всю інформацію про запит. Таке перетворення дозволяє **передавати запити як аргументи методу**, затримувати або ставити в чергу виконання запиту, а також підтримувати операції, які можна скасувати.

**Проблема**

Черговий приклад. **Текстовий редактор**, який містить різноманітні функції. Наприклад збереження файлу. Її можна виконати **як через кнопку** на в панелі меню, **так і** **через** кнопку в діалоговому вікні чи **комбінацію клавіш**.

Фактично ці 3 об’єкти не мають **нічого спільного крім функції збереження файлу**. ЇЇ **код** фактично **доводиться тричі переписувати** в різних об’єктах. **Редагувати його** так само доведеться в них усіх. Що вже казати про **додавання нового об’єкта** для збереження файлу, наприклад пункту контекстного меню.

**Рішення**

Хороший дизайн ПЗ вимагає **відділення користувацького інтерфейсу**(надалі UI) **від бізнес-логіки проєкту**. Тобто кнопки, пункти меню, комбінації клавіш тощо **повинні викликати** вже **існуючу функцію** з бізнес-логіки.

Шаблон Команда пропонує **помістити** цей **виклик в** окремий об’єкт – **команду**. Всі **команди реалізують один інтерфейс**, через який з ними взаємодіє UI, при цьому **не маючи доступу до властивостей** конкретної **команди**.

Так само можна задати **інтерфейс взаємодії для елементів UI.** Через нього **з** цими **елементами може взаємодіяти клієнтський код**. Таким чином **можна** зручно **встановляти** конкретні **команди** **для елементів** **UI** **не зосереджуючись на внутрішній будові** ні команд ні власне елементів інтерфейсу.

[16]

Рис. 1.6.2.1 – Структура класів реалізації патерна Команда

Зображення, що містить текст, знімок екрана, Шрифт, схема

Автоматично згенерований опис

### 1.6.3 Ітератор

Ітератор – це поведінковий патер проєктування, який здійснювати **обхід** елементів будь-якої **колекції** **незалежно від** її **структури**.

**Проблема**

**Колекції** (структури даних) відіграють надзвичайно важливу роль в програмуванні, однак **це всього лиш набір об’єктів**. Вони **зберігають дані по-різному**, в списки, стеки, дерева… Незалежно від цього **потрібно** якось **отримати доступ** до елементів колекції із зовні, й **це** далеко **не** завжди **просто зробити.**

**Додавання** складних **алгоритмів обходу**, особливо, якщо їх декілька в **код** самої **колекції** **негативно вплине** на оптимізацію. Та й **не завжди можливо модифікувати наявний код** (наприклад, якщо він міститься в закритій бібліотеці).

**Рішення**

Суть патерна Ітератор полягає в **перенесенні поведінки, що відповідає за обхід** колекцій **в** окремий об’єкт – **ітератор**. Цей об’єкт **інкапсулює** всі **подробиці** пов’язані з реалізацією алгоритму, надаючи **простий і зручний інтерфейс** взаємодії **для клієнтського коду**.

[17]

Рис. 1.6.3.1 – Структура класів реалізації патерна Ітератор

Зображення, що містить текст, знімок екрана, схема, Шрифт

Автоматично згенерований опис

### 1.6.4 Посередник

Посередник – це поведінковий шаблон, який **дозволяє зменшити хаотичні залежності між об'єктами**. Патерн змушує їх співпрацювати лише через об'єкт-посередник.

**Проблема**

Розглянемо приклад. **Діалогове вікно редагування профілю користувача**. Деякі **елементи форми можуть взаємодіяти з іншими**. Наприклад: вибір прапорця може відкрити приховане текстове поле для. Кнопка "Надіслати", має перевірити значення всіх полів перед збереженням даних. Тощо.

Реалізація цієї логіки в коді елементів форми, значно **ускладнює повторне використання класів цих елементів в інших формах програми**. Наприклад, не вдасться використати клас прапорця в іншій формі, тому що він пов'язаний з текстовим полем.

**Рішення**

Патерн Посередник передбачає **припинення прямої взаємодії між компонентами, що надає їм незалежність одне від одного**. Ці **компоненти повинні співпрацювати опосередковано**, викликаючи спеціальний об'єкт-медіатор, який перенаправляє виклики до відповідних компонентів. В результаті вони залежні лише від цього медіатора.

У прикладі з діалоговим вікном в **ролі посередника може виступати сам клас вікна**, особливо зручно те, що скоріш за все він вже містить всі свої елементи.

[18]

Рис. 1.6.4.1 – Структура класів реалізації патерна Посередник

Зображення, що містить текст, знімок екрана, схема, Шрифт

Автоматично згенерований опис

### 1.6.5 Знімок

Знімок – це поведінковий патерн проєктування, який дозволяє **зберігати та відновлювати** попередній **стан об'єкта**, **не розкриваючи** деталей його **реалізації**.

**Проблема**

Знову приклад. **Текстовий редактор**. До нього вирішено додати функцію **скасування дій.** Для її реалізація **програма зберігає стан всіх об’єктів** перед кожною дію. При її скасування **об’єкти набувають попереднього стану**.

Однак виникли проблеми. По-перше **неможливо отримати доступ до приватних полів** об’єктів **ззовні**, а **робити їх публічними небезпечно**.

**Рішення**

Вся **проблема в порушенні інкапсуляції**. Шаблон Знімок вирішує її **передаючи функцію копіювання об’єкта йому самому**. Самі **копії зберігаються в об’єктах** – **знімках**, які є **вбудованими класами** відносно тих, чий стан зберігають. Ззовні **доступ** до цих об’єктів **обмежений інтерфейсом**, що містить лише їх **метадані**.

Ці знімки **зберігаються в** окремих об’єктах – **доглядачах**. В нашому випадку такий **доглядач міститиме стек зі знімками всіх об’єктів**. Для відновлення стану об’єкту потрібно лише **вибрати знімок** і **передати його** самому **об’єкту**.

[19]

Рис. 1.6.5.1 – Структура класів реалізації патерна Знімок

Зображення, що містить текст, знімок екрана, Шрифт, число

Автоматично згенерований опис

### Стан

Стан - це поведінковий патерн проєктування, який **дозволяє об'єкту змінювати свою поведінку, при зміні внутрішнього стану**. Немовби об'єкт змінив свій клас.

**Проблема**

Ідея шаблону пов’язана із концепцією **кінцевого автомата**.

В будь-який момент часу існує **скінченна кількість станів**, в яких може перебувати програма. У кожному з них **програма поводиться по-різному**, і її можна **миттєво переключити з одного стану** в інший. Однак існують правила зміни станів – **переходи**, вони також скінченні й наперед визначені.

**Рішення**

Патерн Стан передбачає **створення нових класів для всіх можливих станів об'єкта** з усією специфічною для цих станів поведінкою.

**Вихідний об’єкт**, який називається контекстом, зберігає **посилання на один з об'єктів стану**, який представляє його поточний стан, і делегує йому всю, пов'язану зі станом, цьому об'єкту роботу. **Для переведення контексту в інший стан достатньо замінити активний об'єкт стану іншим**.

[20]

Рис. 1.6.6.1 – Структура класів реалізації патерна Стан

Зображення, що містить текст, знімок екрана, схема, ряд

Автоматично згенерований опис

### Стратегія

Стратегія – це поведінковий патерн, який дозволяє помістити **сімейство алгоритмів, в окремі класи** і **зробити** їх **об'єкти взаємозамінними**.

**Проблема**

Приклад. **Навігатор**, який прокладає маршрут між 2-ма точками через дороги (**для автомобілів**). До програми з часом додали **прокладання маршруту для велосипедистів, пішоходів, туристів** (через визначні місця). Кожна **нова функція істотно ускладнювала код**, спричиняючи численні помилки й істотно здорожуючи подальшу підтримку й розширення застосунку.

**Рішення**

Патерн Стратегія **перетворює клас**, що виконує певну дію в різний спосіб **на набір об’єктів – стратегій**, що виконують цю дію кожен своїм шляхом. Оригінальний клас – **контекст містить посилання на** певну, вибрану ззовні **стратегію**. Тобто контекст не обирає стратегію, не містить інформації про те, які є стратегії тощо.

В наведеному прикладі кожна **стратегія повинна містити** один **метод прокладання маршруту**. **Клієнтський код повинен обрати стратегію** й **передати її контексту** – навігатору для побудови потрібного йому шляху.

[21]

Рис. 1.6.7.1 – Структура класів реалізації патерна Стратегія

Зображення, що містить текст, знімок екрана, Шрифт, схема

Автоматично згенерований опис

### Шаблонний метод

Шаблонний метод - це поведінковий патерн проєктування, який **визначає скелет алгоритму** в суперкласі, але **дозволяє** підкласам **перевизначати окремі кроки алгоритму без зміни його структури**.

**Проблема**

Черговий приклад. **Додаток для інтелектуального аналізу даних в документах**. Користувачі надають програмі **документи в різних форматах** (PDF, DOC, CSV), витягти з них значущі **дані в єдиному форматі**.

**Класи** для роботи з всіма типами даних **мають код, який відрізняється лише** тією **частиною**, яка безпосередньо стосується особливостей роботи з певним форматом файлу, що оброляється. Тож було б зручно **звести** **їх до** **підкласів єдиного суперкласу**.

**Рішення**

Шаблонний метод пропонує **поділити алгоритм на кроки**, представлені окремими методами, які викликаються в **шаблонному методі**. Таким чином можна створювати **дочірні** для поточного **класи**, в яких **будуть** **перевизначитися лише певні кроки**.

В нашому прикладі ми можемо створити **клас. що містить всі етапи аналізу даних, крім специфічних** для різних форматів файлів. В той час як ці специфічні кроки можна помістити в **абстрактні методи, які буде перевизначено** в конкретних дочірніх класах.

[22]

Рис. 1.6.8.1 – Структура класів реалізації патерна Шаблонний метод

Зображення, що містить текст, знімок екрана, Шрифт, ряд

Автоматично згенерований опис

### 1.6.9 Відвідувач

Відвідувач – це поведінковий шаблон проєктування, який дозволяє **відокремити алгоритми від об'єктів, на яких вони працюють**.

**Проблема**

Черговий приклад. **Застосунок, що містить дані про карту у вигляді величезного графа**. Вершина графа може містити як щось комплексне, наприклад, місто, так і щось більше конкретне, наприклад будівлю.

Необхідно **експортувати граф в XML**. Для цього вирішено **додати класу** вершини графа **метод для експорту**. Реалізувавши його для конкретних класів через поліморфізм й рекурсивно обійти всі об’єкти графа.

Однак тім лід відмовився дозволяти редагувати існуючий код через **невиправданий ризик** **поломки існуючого коду**, та й загалом код експорту в XML у класі, що відповідає за геодані був сприйнятий якось недоречно. Крім того, якщо б треба було **реалізувати експорт в інший формат,** то б знову **довелось редагувати код всіх класів**.

**Рішення**

Патерн Відвідувач **поміщає нову поведінку в окремий клас** – відвідувач. **Об’єкти** над якими здійснюються дії **передаються як аргументи для методів** відвідувача. При цьому слід використати механізм **перевантаження методів** й створити їх окремі версії під кожен тип об’єкта (Export(city), Export(building)…).

[23]

Рис. 1.6.9.1 – Структура класів реалізації патерна Відвідувач

Зображення, що містить текст, знімок екрана, схема, Шрифт

Автоматично згенерований опис

### Спостерігач

Спостерігач - це поведінковий патерн проєктування, який **дозволяє визначити механізм підписки для сповіщення** декількох **об'єктів про** будь-які **події**, **що відбуваються з об'єктом**, за яким ведеться **спостереження**.

**Проблема**

Останній приклад. **Магазин**. **Покупців цікавить наявність** конкретних **товарів**. Є кілька методів того, **як їм про неї дізнатись**: **відвідувати магазин щодня**, що доволі незручно. **Отримувати повідомлення** від магазину кожен раз, **коли** якийсь **товар з’являється в наявності** – фактично величезну кількість спаму.

**Рішення**

Однак для цієї проблеми є **зручне рішення**. Потрібно **інформувати лише зацікавлених покупців у наявності цікавих їм товарів**. Кожен **користувач може підписатись** або відписатись від **отримування повідомлень** **про** **наявність конкретного товару** чи товарів.

У випадку із кодом застосунку **можна виділити 2 основні сутності**: клас **видавець** **та** інтерфейс **підписник**. **Видавець повинен містити список всіх підписників** та можливість додавати чи видаляти їх. При публікації **видавець викликає метод для оновлення у всіх підписниках**, передаючи їх контекст в якості параметру.

[24]

Рис. 1.6.10.1 – Структура класів реалізації патерна Спостерігач

Зображення, що містить текст, знімок екрана, Шрифт, схема

Автоматично згенерований опис

## Висновок

В цьому розділі було досить детально **розкрито тему шаблонів проєктування**. Пояснено їхню суть та **класифікацію**. Наведено **приклади** 22-х найпоширеніших патернів дизайну, описано їхнє **призначення** **й** **застосування**, **а також** до кожного із них додано діаграму класів, що описує їхню **структуру**. Розуміння даних, наведених в цьому розділі є необхідною умовою коректного сприйняття інформації, наведеної в практичній частині Роботи, особливо розділу, що стосується проєктування архітектури розроблюваної АІС.

# 2. Практична частина

## 2.1 Аналіз та проєктування

### 2.1.1 Постановка завдання

Структурно підприємство складається з цехів, які в свою чергу поділяються на дільниці. Вироби, що випускаються підприємства: вантажні, легкові автомобілі, автобуси, сільськогосподарські, дорожньо-будівельні машини, мотоцикли та інші вироби. Кожна категорія виробів має специфічні, притаманні лише їй атрибути. Наприклад, для автобусів це місткість, для сільськогосподарських і дорожньо-будівельних машин – продуктивність тощо. По кожній категорії виробів може збиратися кілька видів виробів. Кожній категорії інженерно-технічного персоналу (інженери, технологи, техніки) і робітників (збирачі, токарі, слюсарі, зварювальники тощо) також характерні атрибути, властиві тільки для цієї групи. Робочі об'єднується в бригади, якими керують бригадири. Бригадири обираються з числа робітників; майстри, начальники ділянок і цехів призначаються з числа інженерно-технічного персоналу.

Робітники: інженерно-технічний персонал (інженери, технологи, техніки), робітники (збирачі, токарі, слюсарі, зварювальники…) також характерні атрибути, властиві тільки для цієї групи. Робочі об'єднується в бригади, якими керують бригадири. Бригадири обираються з числа робітників; майстри, начальники ділянок і цехів призначаються з числа інженерно-технічного персоналу. Кожен виріб збирається у своєму цеху (в цеху може збиратися кілька видів виробів) та в процесі виготовлення проходить певний цикл робіт, переміщаючись з однієї ділянки на іншу. Всі роботи по збірці конкретного виробу на певній ділянці виконує одна бригада робітників, при цьому на ділянці може працювати декілька бригад. Очолює роботу на ділянці начальник ділянки, у підпорядкуванні якого знаходиться кілька майстрів. Різні вироби можуть проходити одні й ті самі цикли робіт на одних і тих же ділянках цеху. Зібраний виріб проходить серію випробувань у випробувальних лабораторіях. Випробувальні лабораторії можуть обслуговувати декілька цехів, у свою чергу цехи можуть користуватися кількома лабораторіями. Випробування проводяться фахівцями на обладнанні випробувальної лабораторії, при цьому при випробуванні конкретного виробу в лабораторії можуть бути задіяні різні види обладнання.

Ведеться облік руху кадрів та облік продукції, що випускається.

Види запитів в інформаційній системі:

1. Отримати перелік видів виробів окремої категорії і в цілому, що збираються зазначеним цехом, підприємством.

2. Отримати число і перелік виробів окремої категорії і в цілому, зібраних зазначеним цехом, ділянкою, підприємством в цілому за певний відрізок часу.

3. Отримати дані про кадровий склад цеху, підприємства в цілому і по зазначеним категоріям інженерно-технічного персоналу і робітників.

4. Отримати число і перелік ділянок зазначеного цеху, підприємства в цілому та їх начальників.

5. Отримати перелік робіт, які проходить вказаний виріб.

6. Отримати склад бригад зазначеної ділянки, цеху.

7. Отримати перелік майстрів вказаної ділянки, цеху.

8. Отримати перелік виробів окремої категорії і в цілому, що збираються зараз зазначеним ділянкою, цехом, підприємством.

9. Отримати складу бригад, що беруть участь в складанні зазначеного виробу.

10. Отримати перелік випробувальних лабораторій, що беруть участь у випробуваннях деякого конкретного виробу.

11. Отримати перелік виробів окремої категорії і в цілому, що проходили випробування у зазначеній лабораторії за певний період.

12. Отримати перелік випробувачів, що беруть участь у випробуваннях зазначеного виробу, виробів окремої категорії і в цілому у вказаній лабораторії за певний період.

13. Отримати склад обладнання, що використовувалося при випробуванні зазначеного виробу, виробів окремої категорії і в цілому у вказаній лабораторії за певний період.

14. Отримати число і перелік виробів окремої категорії і в цілому, що збираються зазначеним цехом, ділянкою, підприємством в даний час

### 2.1.2 Дорожня карта

Рис 2.1.2.1 – Дорожня карта Проєкту. URL – <http://surl.li/uagfz>

Зображення, що містить текст, Стікер, почерк, знімок екрана

Автоматично згенерований опис

Зважаючи на загальну складність завдання поточної курсової Роботи та помилки здійснені при виконанні роботи з курсу «Об’єктно орієнтоване програмування». **Було вирішено приділити увагу детальному аналізу завдання** перед його безпосереднім виконанням.

**Мету роботи можна коротко підсумувати наступним чином**. Створити АІС, що надає користувачам можливість робити запити до бази даних, яка містить дані з вказаної предметної області (автомобілебудівного підприємства). Фактично розроблювана програма лише надає зручний (той, що не потребує знання SQL) інтерфейс роботи з певною базою даних

Першим етапом цього аналізу стало **створення дорожньої карти Проєкту**. Робота було поділена на окремі завдання, які було класифіковано за 4-ма категоріями:

1. **Теорія**: містить завдання, що стосуються аналізу предметної області (тобто само постановки завдання та теми патернів проєктування)
2. **Проєктування**: містить завдання, що стосуються проєктування архітектури програми, реалізації в ній шаблонів проєктування
3. **База даних**: містить завдання, що стосуються підготовки бази даних, необхідної для роботи застосунку
4. **Програмування**: містить завдання, безпосередньо пов’язані із програмною реалізацією завдання Роботи

Завдання на карті з’єднані стрілками, які позначають порядок їх виконання. Ж**одна задача не може бути розпочата доки не будуть завершені всі ті, що йдуть перед нею**. Штриховані лінії, що ведуть до блоків тестування означають, що після виконня відповідного завдання слід провести його тестування, а потім полагодити помилки (фактчино повернутись назад до цього завдання). В усьому іншому порядок виконання завдань довільний.

**Передумовою всіх практичних завдань є аналіз теорії**. Поверхневого аналізу предметної області (тобто змісту Завдання та даних про роботу автомобілебудівних заводів) достатньо для проєктування структури класів Проєкту, що є основою для створення бази даних. Однак для її заповнення правдоподібними тестовими даними потрібно більш глибоке заглиблення в матеріал.

Аналіз теми курсу, тобто шаблонів проєктування необхідний для проєктування структури класів Проєкту – найважливішої частини роботи. Адже саме **інтеграція патернів у архітектуру проєкту визначає ключову відмінність від роботи з курсу «ООП».**

**Після завершення проєктування структури Проєкту та створення бази даних можна перейти до програмної реалізації Роботи**. Однак попередні підготування до цього етапу, зокрема вибір конкретних технологій та створення на їх основі інтерфейсу користувача можливе вже при поверховому аналізі теорії.

Про всі наведені етапи розробки Проєкту **детальніше розповідається в наступних розділах**.

### 2.1.3 Структура даних

Рис 2.1.3.1 UML-діаграма структури класів даних Проєкту. URL – <http://surl.li/uagha>

Зображення, що містить текст, схема, План, Креслення

Автоматично згенерований опис

Дані проєкту (тобто все, що стосується інформації про підприємство, а не бізнес-логіки програми як такої) можна поділити на **3 частини**:

1. Структура підприємства
2. Персонал підприємства
3. Вироби підприємства

**Структурно** підприємство поділяється на цехи, частинами яких ї ділянки (агрегація), а також дослідницькі лабораторії, пов’язані з цехами (композиція).

**Персонал** підприємства поділяється на робітників та інженерно технічний персонал, тобто інженери та тестувальники. Всі 3 підгрупи мають окремі категорії спеціалістів. На цій основі можна встановити численні зв’язки **успадкування**. Оскільки весь персонал розподілений між цехами, їх ділянками та лабораторія, то між ними наявна також агрегація.

**Вироби**, що виробляє підприємство ділиться на категорії, кожна з яких містить конкретні найменування (черговий ланцюг **успадковування**). Кожен виріб збирається на певних ділянках певного цеху та тестується в певних лабораторіях, а всі ці процеси здійснюють відповідні члени персоналу. Тож можна встановити між ними **композицію**.

### 2.1.4 База даних

Рис 2.1.4.1 – Діаграма структури бази даних Проєкту. URL – <http://surl.li/uaghe>

Зображення, що містить текст, схема, План, знімок екрана

Автоматично згенерований опис

**Попередній розділ став основою для проєктування бази даних** – фундаменту, зручним інтерфейсом доступу до якої, виступає розроблювана АІС. Основою БД ПЗ є 2 таблиці: **персонал** (staff) та **виробництво** (production).

Рис 2.1.4.2 – Структура таблиці staff

Зображення, що містить текст, схема, План, знімок екрана

Автоматично згенерований опис

**Таблиця staff містить дані про всіх працівників підприємтсва**. Від неї успадковуються (тобто зв’язуються відношенням один до одного) всі таблиці, що містить всі конкретні категорії працівників, інженерів, тестувальників. **Дочірні таблиці містять специфічні атрибути** кожної категорії й підкатегорії персоналу:

1. **Інформація про ефективність роботи** різнихвидів **працівників**. Наприклад: швидкість роботи монтажників, ефективність управління менеджерів
2. **Принадлежність особи до** певної **структруної частини підприємства**. Кожен **робітник є членом однієї з бригад** (можливо пригадиром). Кожен **інженер є, або майстром**, що працює на певній дільниці певного цеху, **або начальником** якогось цеху чи його ділянки. Кожен **тестувальник займається своєю діяльністю в** певній **лабораторії**. Ці **принаджлежності реалізуються через зв’язок з відповідними таблицями** (workshops, sites, brigades, laboratories).

Рис 2.1.4.3 – Структура таблиці production

Зображення, що містить текст, схема, План, знімок екрана

Автоматично згенерований опис

**В таблиці production знаходяться всі вироби підприємства**, дата їх виголовлення та структруні частини підприємства на яких вони вироблялись, Якщо дата є звичайнимполем, то решта з переліченого реалізоване набагато складніше:

1. **Виріб. Тобто тип**, а не конкретний транспортний засіб. Всі ці типи **знаходяться в таблиці products** (з якою таблиця production з’єднана у відношенні багато до одного). **Products містить категорії виробів та посилання** (зв’язок один до одного) **на їх характеристики**, які містяться **в таблиці vehicles**. Ця таблиця **містить загальні** для всіх транспортних засобів **атрибути**, а від **неї успадковуються таблиці зі специфічними** для кожної категорії виробу атрибути.
2. **Структурні частини підприємства**, тобто цехи, ділянки й бригади задіяні у виготовленні вибору **представлені зв’язками багато до одного з відповідними таблицями**. Тобто **кожен виріб у таблиці production містить кількість записів рівну кількості бригад, що брали участь в його виробництві**, оскільки бригади є найменшою структурною одиницею заводу.

Таким чином спроєктовану базу даних можна назвати в значній мірі **нормалізованою**. Вона непогано оптимізована для надсилання до неї запитів, однак заповнення такої бази даними займе значний час.

### 2.1.5 Архітектура

#### 2.1.5.1 Вступ

Рис 2.1.4.1 – UML Use Case діаграма Проєкту

Зображення, що містить схема, коло, текст, дизайн

Автоматично згенерований опис

Для проєктування архітектури АІС потрібно визначити її **функціонал**. Виділити основні **сценарії застосування**. Єдиним завданням Застосунку є здійснення **запитів до бази даних**. Фактично це можна назвати основним сценарієм. Однак слід виділити й сценарії, що випливають із цього. По-перше потрібно реалізувати **вхід користувача в систему**. Тобто підключення до бази даних. По-друге доцільно реалізувати можливість **збереження результату** будь-якого **запиту** в текстовий файл (фактично цей сценарій розширенням здійснення запиту).

Обидва **сценарії**: авторизація та здійснення **можна звести до** наступного узагальненого **формулювання**:

1. Користувач обирає Дію через певний інтерфейс (надалі Вікно)
2. Система відкриває діалогове вікно, що містить поля, в які користувач вводить дані щодо цієї Дії
3. Користувач заповнює поля
4. Система перевіряє правильність заповнення (якщо воно неправильне, користувач повторює крок 3)
5. Система використовує введені дані, щоб виконати Дію (в разі помилки здійснюється повернення до кроку 2)
6. Система повертає користувача назад до можливості виконання кроку 1

Таким чином **за основу проєкту** було **взято роботу з діалоговими вікнами** (замість нього використано слово «form – форма», оскільки «dialog» є зарезервованим словом у багатьох бібліотеках та фреймворках відповідальних за роботу графічного інтерфейсу). **Їхній виклик, заповнення та подальша обробку.** Цей процес потрібно зробити архітектурним шаблоном (надалі Шаблон, Патерн) Проєкту.

Рис 2.1.5.1.2 – UML-діаграма класів архітектури Проєкту. URL – <http://surl.li/uaisr>Зображення, що містить текст, схема, знімок екрана, План

Автоматично згенерований опис

Перш ніж детально говорити про сам Патерн слід пояснити деякі **особливості діаграми класів**, наведеної в ілюстрації вище, оскільки її візуальне сприйняття сильно спростить розуміння наведеного матеріалу. Загалом **діаграма поділена на 3 блоки**: **Шаблон** (Form Observer), та реалізацію на його основі **Авторизації** (Connection) і **Здійснення запитів** (Queries). Кожен блок окрім самих класів та інтерфейсів містить **підблоки**. Вони **означають шаблони проєктування**, **які імплементують класи поміщені в них**. Зеленим показано породжувальні патерни, блакитним – структурні, червоним – поведінкові.

#### 2.1.5.2 Архітектурний (основний) Шаблон

Рис 2.1.5.2 – UML-діаграма класів архітектури Проєкту.

Зображення, що містить текст, схема, знімок екрана, План

Автоматично згенерований опис

Тепер власне Архітектурний Патерн (див. розділ 2.1.5.1). Він не дарма називається **Form Observer,** оскільки **є окремим випадком шаблону Observer** (Спостерігач). Як відомо суть цього шаблону полягає в тому, що певні об’єкти (publishers) можуть надсилати іншим об’єктам (subscribers) інформацію про зміну свого стану.

Те саме реалізовано тут. **Publisher – це діалогове вікно** (форма), **Subscriber – це об’єкт, який використовує дані з форми** для виконання обраної користувачем дії. **Підписку** subscriber’а publisher на publisher **здійснює елемент інтерфейсу, який користувач використовує для виклику форми** (наприклад: кнопка авторизації чи здійснення конкретного запиту)

Тепер детальніше. Інтерфейс користувача (надалі **UI** для уникнення плутанини з інтерфейсами класів (надалі CI)) **для виклику форми мусить реалізувати CI** **IFormUser (user)**. Він містить метод, який дозволяє повернутись до UI після завершення роботи.

Власне **форма поділена на 2 частини**: **клас Form та інтерфейс FormUi**. **FormUi містить** наступне. **Поля з даними**, які можна використати як для зчитування значень так і для їх заповнення ззовні (наприклад для показу помилок). **Методи для її відкриття, приховування, показу та закриття**. Цей CI реалізують конкретні діалогові вікна, які власне містять всі необхідні поля. **FormUi** лише **надає впорядкований доступ** до них, при цьому **роблячи клас Form незалежним від** реалізації **UI** в застосунку.

**Form містить** власне **екземпляр FormUi та набір методів для роботи з ним**. Це методи для відкриття форми, надсилання даних, їх попередньої перевірки та повторного відкриття форми в разі помилки.

Будь-який **об’єкт, що обробляє форму повинен реалізувати інтерфейс IFormHandler (handler)**. Він володіє методом, що приймає форму в якості параметру. **За надсилання форми до її обробника відповідає клас FormSender**. Цей клас **обробляє помилки, що можуть виникнути при роботі handler`а**. Відповідно він може повторно відкрити форму (із повідомленням щодо контексту помилки) або в разі успіху повернутись user`а.

Таким чином **робота бізнес-логіки**, що імплементує Form Observer **поділяється на** такі етапи:

1. **IFormUser** відкриває **Form** для певного **IFormHandler**
2. Form перевіряє дані введені в **IFormUi**
3. **FormSender** передає **Form** до **IFormHandler**
4. **IFormHandler** виконує дію, маючи дані з форми
5. **FormSender** обробляє помилки, пов’язані з роботою **IFormHandler**

##### 2.1.5.2.1 Структура форми

Рис 2.1.5.2.1.1 – UML-діаграма класів, що відповідають за поля діалогових вікон

Зображення, що містить текст, знімок екрана, схема, План

Автоматично згенерований опис

**Основою форми** (тобто графічного діалогового вікна) **є** її **поля**. Поля, як об’єкти, **розділені на абстракцію** Field **та реалізацію** FieldUi, що **відповідає Патерну Компонувальник**

**Такий підхід** має 2 переваги. По-перше, він **робить Form незалежним від графічного інтерфейсу**. По-друге **вирішує проблему успадковування** різних за типами полів.

Будь-яке **поле має тип**. І тип це **не просто змінна –** це **певний графічний інтерфейс**, необхідний для введення даних: текстове поле, список, календар для вибору дати тощо. Проблема в тому, що в **більшості бібліотек та фреймворків, що відповідають за графічний інтерфейс неможливо успадковувати користувацькі UI-елементи одне від одного**. Сама по собі це не проблема, однак, якщо зробити ці UI-класи єдиним складовими полів (тобто інтегрувати в них базову бізнес-логіку), то доведеться дублювати значні обсяги коду.

Для уникнення цього **клас бізнес-логіки Field та UI-клас FieldUi розділено**. Таким чином клас **Field отримує дані з FieldUi,** **а також** **перевіряє правильність та, за потреби, редагує його заповнення.** Наприклад UI-клас TextField може використовуватися різними класами логіки: StringField, IntField, FloatField… Ці класи будуть по-різному перевіряти правильність заповнення TextField.

Рис 2.1.5.2.1.2 – UML-діаграма класів, що відповідають за реалізацію деактивації окремих полів при їх непотрібності

Зображення, що містить текст, знімок екрана, схема, План

Автоматично згенерований опис

**Не в кожній формі потрібно заповнювати всі поля**. Наприклад запит на виготовлені автомобілі може стосуватися як підприємства в цілому так і окремого цеху, чи навіть ділянки. Тож **доцільно поділити діалогове вікно на вкладки, які помістити в загальний контейнер**. В першій будуть обов’язкові поля, а в інших поля, потребу в яких визначає користувач.

Конкретизуємо на нашому прикладі. **Потрібно створити поле – Switch**, що матиме список на кшталт «підприємство, цех, ділянка». Тобто **це поле визначатиме необхідні поля**. **Залежно від вибраного варіанту** **вкладки**, які стосуються відповідно цехів, ділянок, **будуть доступні або недоступні** для заповнення. Крім того **форма буде передавати лише дані з полів активних вкладок**.

**Реалізувати** цей задум **можна на основі патерну Посередник** (яким буде виступати сам об’єкт, що реалізує IFormUi, або просто форма), **щоб уникнути непотрібних залежностей між елементами інтерфейсу**. Форма відстежує подію вибору для поля Switch і надає контейнеру (через CI) інформацію про вибір. А він самостійно реагує на цей вибір, роблячи форму незалежною від конкретного контейнера.

Крім того **container виступає як посередник при передачі даних полів** до IFormUi. Тобто форма отримує лише поля з активних вкладок контейнера. Взагалі важливо розуміти, що **контейнер може не мати вкладок і бути просто обгорткою над всіма полями, однак для форми внутрішня реалізація контейнера не має значення**. Для отримання даних з контейнера викликається так званий Шаблонний метод. Іншими словами **метод отримання полів з контейнера є шаблонним**. Тобто він поділений на окремі частини, які можна довільним чином перевизначити в дочірніх класах. Цими частинами є отримання активних вкладок, отримання з них полів та перевірка цих полів на правильність.

##### 2.1.5.2.2 Обробка помилок

Рис 2.1.5.2.2.1 – UML-діаграма класів, що відповідають за обробку помилок, пов’язаних із некоректно введеними користувачем даними

Зображення, що містить текст, знімок екрана, схема, План

Автоматично згенерований опис

**Обробка помилок**, викликаних виконанням IFormHandler **здійснюється шляхом імплементації патерну Ланцюг Відповідальності**. Програма містить клас з єдини екземпляром – FormSender, (тобто він реалізує Singleton). Цей клас після надсилання даних handler`у може отримати від нього помилку.

Ця **помилка проходить через ланцюг FormErrorHandler** (надалі errorHandler) – обробників конкретних помилок, передаючи дані про помилку в якості параметру для їх єдиного методу, що **перев іряє відповідність помилки умовам** кожного конкретного errorHandler’a. Примітка: ланцюг errorHandler’ів знаходить в самому IFormHandler’і та отримується FormSender’ом через інтерфейс.

Говорячи точніше **FormErrorHandler отримує дані про помилку від IFormSender’а**. **Якщо помилка не відповідає** його **умовам** (наприклад типу чи коду помилки) **він** **передає її наступному** errorHandler’у. **Якщо ж помилкавідповідає** поточному обробнику, то **він передає FormSender інформацію** про те, яке повідомлення вивести та які поля вважати неправильно заповненими.

#### 2.1.5.3 Авторизація

Рис 2.1.5.3.1 – UML-діаграма класів, що відповідають за обробку помилок, пов’язаних із некоректно введеними користувачем даними

Зображення, що містить текст, схема, знімок екрана, План

Автоматично згенерований опис

**Блок авторизації користувача імплементує Архітектурний Патерн Form Observer**. Тож означимо реалізацію основних ролей:

1. IFormUser: Цю роль виконує вікно, що містить 2 кнопки:
2. для під’єднання до сервера із вибором користувача та бази даних,
3. для додавання нових користувачів до бази даних
4. IFormUi: Інтерфейс містить 2 діалогові вікна для кожного зі сценаріїв:
   1. Вікно авторизації: host, user, password, database
   2. Вікно додавання користувача: username, password
5. IFormHandler: В якості обробників виступають 2 singleton-класи
   1. ServerConnector відповідає за під’єднання до сервера
   2. UserCreator відповідає за створення користувача бази даних

#### 2.1.5.4 Здійснення запитів

Рис 2.1.5.4.1 – UML-діаграма класів, що відповідають за обробку помилок, пов’язаних із некоректно введеними користувачем даними

Зображення, що містить текст, схема, знімок екрана, План

Автоматично згенерований опис

**Блок здійснення запитів так само реалізує Form Observer**. IFormUser – це вікно, що містить кнопки для кожного типу запиту. Відповідно кожен запит має унікальне діалогове вікно, яке використовується специфічним IFormHandler’ом. Його реалізація доволі складна.

Інтерфейс IFormHandler реалізує клас Reqeust. Цей клас містить єдиний **шаблонний метод**. Він поділяється на 2 складові**: метод, що здійснює SQL запит та метод, що перетворює його в набір екземплярів класів, які потім можна вивести на екран**. Перший метод (пов’язаний з SQL) перевизначається в дочірніх класах суперкласу Request.

**Детальніше про класи для виводу**. Всі вони реалізують інтерфейс **IDisplayable**, що містить метод для їх перетворення на текст. Колекції цих об’єктів **виводяться класом Display** на інтерфейс користувача. Крім того **Display забезпечує функціонал для збереження** цих IDisplayable-об’єктів у текстовий файл.

## 2.3 Програмна реалізація

### 2.3.1 Вибір технологій

Для програмної реалізації АІС було обрано мову програмування **C#** версії 8.0 у поєднанні з технологією **WPF** для створення графічних інтерфейсів, яка використовує **XAML**- розмітку.

До переваг **C#** можна віднести її орієнтованість на об’єктно-орієнтований підхід до програмування, непогану оптимізацію в поєднанні з простотою використання тощо. Однак головною особливістю мови є те, що вона **створена** компанією Microsoft спеціально **для розробки застосунків під** ОС **Windows**. Тож для неї створено чимало технологій для розробки графічного інтерфейсу під цю операційну систему.

Багато в чому найкращою з них можна назвати WPF (**Windows Presentation Foundation**). Вона забезпечує максимальну гнучкість при розробці GUI. Розробник може тонко налаштувати будь-який елемент інтерфейсу, чи створити власні шляхом композиції чи успадковування вже існуючих.

При цьому використовується **мова розмітки XAML** (Extendable Application Markup Language), чий синтаксис багато в чому співпадає з таким мовами розмітки як HTML, тож її вивчення не потребує значних зусиль. **Верстання** за допомогою такого засобу **набагато швидший процес** ніж його створення за допомогою окремих програм з графічним інтерфейсом чи шляхом безпосереднього використання C#. Крім того використання окремої мови **дозволяє краще відділити GUI від бізнес-логіки проєкту.**

Оскільки метою Програми є спрощення роботи із **PostgreSQL**-базою даних, то **для роботи** **із цим діалектом** SQL **використано** бібліотеку **Npgsql**. Оскільки вона **має** значне поширення, а отже й **підтримку** (тобто наявність якісної офіційної документації та постійні оновлення для роботи із останніми версіями PostgreSQL). Крім того бібліотека має **простий синтаксис**, який здебільшого зводиться до передачі базі даних PostgreSQL-команд й отримання від неї зворотного зв’язку, тож її опанування, за умови знання відповідного SQL-діалекту, майже не викликає зусиль.

### 2.3.2 Інтерфейс користувача

Графічний інтерфейс містить **3 основні вікна**, не включаючи діалогові. Кожне з них є окремим файлом, що містить XAML-розмітку та C# файл. **Розмітка призначена для вказання елементів вікна** **та його параметрів**. **C#-код призначений** перш за все **для додавання до елементів вікна подій** (наприклад натискання кнопок), а також **створення зручного інтерфейсу для роботи** з вікном **через клієнтський код** (наприклад відкриття вікна чи отримання даних із елементів вікна, зокрема текстових полів). Розгляньмо детально кожне з цих вікон.

**Вікно авторизації користувача**, яке містить кнопки **Під’єднання до сервера**, **Створення користувачів** та **Переходу до вікна вибору запитів**. Вікно також містить код, що регулює активність кнопок Створення й Переходу залежно від того чи авторизацію було здійснено (в такому разі можна перейти до здійснення запитів або до створення нового користувача БД)

Розмітка: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/Authorization/UserCreatorWindow.xaml>

Код: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/Authorization/UserCreatorWindow.xaml.cs>

Рис 2.3.2.1 Вікно авторизації користувача (користувач не авторизований)

Зображення, що містить текст, знімок екрана, Шрифт, число

Автоматично згенерований опис

**Вікно вибору запитів** містить **набір з** чотирнадцяти **кнопок** для власне **здійснення** цих **запитів** та ще одну для повернення до вікна Авторизації. Кнопки мають короткі підписи, однак при наведенні показують підказки із детальним їх-записів описом. Ці **кнопки й підказки для них було згенеровано через C#** з метою автоматизації. Оскільки це фактично 14 однакових об’єктів, які відрізняються лише текстом й подіями, що відбуваються після їх натискання.

Розмітка: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/Queries/MainWindow.xaml>

Код: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/Queries/MainWindow.xaml.cs>

Рис 2.3.2.2 Вікно вибору запитів

Зображення, що містить текст, знімок екрана, Шрифт, число

Автоматично згенерований опис

**Вікно виводу результатів запитів** складається з **текстового поля**, що власне й виводить цей результат (до нього додано scrollbar на випадок, значного обсягу даних) та **кнопки** **для** його **збереження** **в** текстовий **файл**. Також вікно містить код для відображення відповідного тексту при відкритті його-вікна клієнтським кодом.

Розмітка: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/Output/Display.xaml>

Код: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/Output/Display.xaml.cs>

Рис 2.3.2.3 Вікно виводу результатів запитів

Зображення, що містить текст, знімок екрана, програмне забезпечення, комп’ютер

Автоматично згенерований опис

**Діалогові вікна** можна поділити на 2 **системні** й **користувацькі** (вони ж Форми (див. розділ 2.1.5.2). До системних можна віднести вікна для відображення помилок, вибору дати (див. рис. 2.3.2.4) чи шляху для збереження файлу. Всі вони вбудовані в систему й доступні до використання через відповідні функція для їх створення, відкриття, зчитування з них даних тощо.

Розмітка: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/General/Form/Field/UI/DateField.xaml>

Код: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/General/Form/Field/UI/DateField.xaml.cs>

Рис 2.3.2.4 Спливаюче вікно вибору дати

Зображення, що містить текст, знімок екрана, число, програмне забезпечення

Автоматично згенерований опис

**Форми** – це діалогові вікна **призначені для введення користувачем даних**, необхідної для здійснення певної дії, наприклад запиту. Фактично **форма** просто **надає** певному **класу, що її обробляє** **набір параметрів** та попередню перевірку коректності їхніх значень. Форми містять набір полів різних типів та кнопки Надсилання й Скасування надсилання даних. Всі поля є користувацькими елементами інтерфейсу, що складаються з набору вбудованих, тобто системних (наприклад текстове поле для вводу й напис для заголовку). Шляхом використання можливостей WPF **створення форм було** в значній мірі **автоматизоване**. Зведене до вказання набору полів та їх параметрів у XAML-розмітці.

Розмітка: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/Queries/Second/FormUi.xaml>

Код: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/Queries/Second/FormUi.xaml.cs>

Рис 2.3.2.4 Діалогове вікно для здійснення запиту про виробництво виробів протягом зазначеного періоду.

Зображення, що містить текст, знімок екрана, число, Шрифт

Автоматично згенерований опис

### 2.3.3 Бізнес-логіка Проєкту

#### 2.3.3.1 Шаблон Form Observer

**Основа проєкту – його Архітектурний патерн**. Розгляньмо ієрархію класів створених для його реалізації, що знаходяться у відповідних файлах теки Form. (див Рис. 2.3.3.1).

Рис. 2.3.3.1.1 – Тека з кодом, що відповідає за реалізацію Архітектурного Патерну проєкту

Зображення, що містить текст, знімок екрана, дизайн

Автоматично згенерований опис

Посилання: <https://github.com/Denchyk-chyk/cw-ad-2024-code/tree/master/CS/General/Form>

Почнімо з дочірньої директорії **Logic**. Вона містить **основні інтерфейси та класи, що відповідають за бізнес-логіку** форми. Зокрема інтерфейси для обробника форми (IFormHandler), її користувача (IFormUser) та графічного інтерфейсу (IFormUi). Їх реалізують відповідні класи в конкретних розділах проєкту: Авторизації та Здійснені запитів.

Крім того тут знаходяться **класи, що відповідають за використання форми** (FormSender (статичний)) **та обробку помилок** (FormErroHadnler), що можуть виникнути під час цього процесу. FormSender, як можна здогадатись із назви, здійснює «доставку» даних з форми до її обробника. В разі виникнення помилок, що відстежуються відповідними FormErroHadnler’ами, він інформує користувача про їх деталі й дозволяє повторно заповнити діалогове вікно.

Рис. 2.3.3.1.2 – Діаграма, що показує розділення абстракції та реалізації полів діалогових вікон

Зображення, що містить текст, схема, ряд, Паралель

Автоматично згенерований опис

Папка **Field** містить **класи, що відповідають за поля форми**. Вони **розділені на абстракцію** (бізнес-логіку – тека Logic) **та реалізацію** (графічний інтерфейс – тека Ui), що є реалізацією патерну Компонувальник (див. розділ 1.5.3). Таке розділення, схематично показане на Рис 2.3.3.2, дозволяє взаємодіяти з різними елементами інтерфейсу незалежно від їх реалізації. Різні типи об’єктів Field (логіки) взаємодіють з різними видами об’єктів інтерфейсу через їхній спільний інтерфейс IFieldUi. Таким чином **один об’єкт інтерфейсу може бути реалізацію кількох абстракцій,** тобто класів бізнес-логіки, **і навпаки**.

Зокрема, **TextField може** одночасно **використовуватись як** класом **IntegerField так** і класом **StringField** – вони здійснюють конвертацію даних із текстового рядка, та перевіряють правильність введених даних. При цьому сам TextField лише надає інтерфейс для взаємодії через клієнтський код, будучи повністю незалежним від того, хто його використовує.

Напротивагу **логічний клас StringField** **може використовувати** як текстове поле (**TextField**) так **і** поле з датою (**DateField**). Він не залежить від їх фактичного змісту, він просто отримує від них один інтерфейс взаємодії, який надає йому можливість зчитати або записати дані в форматі string (попри те, що ці дані можуть мати зовсім інший формат в цих класах).

Крім того в рамках створеного рішення (тобто реалізації шаблону Компонувальник) **допустима ієрархія успадковування класів логіки**. Наприклад EnumField (суперклас, що використовує дані із випадаючого списку) має дочірній клас OptionalEnumField, який відрізняється лише тим, що дозволяє користувачу не обирати жоден варіант зі списку.

Повернімось до змісту директорії Form. Вона також містить власне клас **Form** та клас **Container**, який регулює активність полів залежно від вибору користувача Наприклад користувач може обрати структурну частину підприємства, якої стосується його запит: цеху, ділянки чи бригади, й залежно від цього вибору активними будуть лише потрібні для його уточнення поля. В нашому прикладі лише одне з наступних: вибору цеху, вибору ділянки чи вибору бригади. **Обидва класи** (Field та Container) **беруть під контроль певний елемент інтерфейсу, не зосереджуючись на його реалізації**.

**Container використовує набір полів** (Field) **і регулює їх активність** залежно від умов, виставлених користувачем. Він не залежить від типу полів, їх розташування, елементів, які викликають події для зміни умови активності цих полів й тд. Значення має лише те, що всі ці поля можна ввімкнути або вимкнути.

Так само **Form працює з будь-яким вікном, якщо воно реалізує інтерфейс IFormUi**. Для форми важливо лише те, що з вікна обов’язково можна отримати значення полів або записати їх, а також те, що це власне вікно – тобто його можна відкривати, закривати, згортати тощо. Конкретну реалізацію всієї описаної тут логіки розглянемо в двох наступних розділах.

#### 2.3.3.2 Авторизація користувача

**Авторизація користувача** в основі своїй **є практичною реалізацією шаблону Form Observer**, описаного в попередньому розділі. Файли, що відповідають за цей розділ знаходяться в теці Authorization.

Рис. 2.3.3.2.1 – Тека з кодом, що відповідає за авторизацію користувача

Зображення, що містить текст, знімок екрана, Шрифт

Автоматично згенерований опис

Вона містить **головне вікно – клас MainWindow, який реалізує інтерфейс IFormUser**, для того, щоб відкривати відповідні діалогові вікна для введення даних, необхідних для виконання з’єднання з сервером та додавання нових користувачів до бази даних. (див. рис 2.3.3.2.3).

Рис. 2.3.3.2.2 – Вікно авторизації користувача (користувач авторизований)

Зображення, що містить текст, знімок екрана, Шрифт, число

Автоматично згенерований опис

**Ці сценарії** (авторизації (див. рис. 2.3.3.2.3) і додавання користувачів (див. рис. 2.3.3.2.4)) **мають по одній FormUi та FormHandler** відповідно. Щодо останніх, то це singleton-класи, що містять лише один метод та набір обробників помилок (FormErrorHandler), що можуть виникнути в процесі його-методу виконання.

**Для ServerConnector’а** такими **помилками можуть бути неправильний логін чи пароль користувача**. **Неможливість підключення до вказаного хоста чи некоректна назва бази даних.** В разі виникнення однієї з перелічених неполадок, користувач повертається до заповнення форми, при цьому отримуючи діалогове вікно з інформацією про помилку та позначення некоректно заповнених полів (їхній заголовок стає червоним, а значення змінюється, зазвичай на порожнє). На рис. 2.3.3.2.5 показано як це виглядає на прикладі неправильно введеного пароля.

Рис. 2.3.3.2.3 – Форма під’єднання до сервера (авторизації користувача)

Зображення, що містить текст, знімок екрана, число, Шрифт

Автоматично згенерований опис

Рис. 2.3.3.2.4 – Форма додавання нового користувача

Зображення, що містить текст, знімок екрана, монітор, Прямокутник

Автоматично згенерований опис

Рис. 2.3.3.2.5 – Повідомлення про помилку при під’єднанні до сервера

Зображення, що містить текст, знімок екрана, монітор, число

Автоматично згенерований опис

#### 2.3.3.3 Здійснення запитів до бази даних

Це **основна частина програми**, яка так само як і попередня **ґрунтується на її Архітектурному Шаблоні**. Відповідний **код розміщено в 2-х теках**: **Queries та Output** (див. рис. 2.3.3.3.4). Перша відповідає за власне надсилання запитів до бази даних та отримання від неї їхніх результатів. В той час, як друга забезпечує зчитування отриманих від БД й даних та їхній вивід у зручному для користувача форматі.

Рис. 2.3.3.3.1 – Теки з кодом, що відповідає за здійснення запитів до БД та вивід їхнього результату на екран користувача

Зображення, що містить текст, знімок екрана, дизайн

Автоматично згенерований описЗображення, що містить текст, знімок екрана, дизайн

Автоматично згенерований опис

Посилання:

Queries: <https://github.com/Denchyk-chyk/cw-ad-2024-code/tree/master/CS/Queries>

Output: <https://github.com/Denchyk-chyk/cw-ad-2024-code/tree/master/CS/Output>

Почнімо з **Queries**. **Її головний файл, MainWindow, містить IFormUser-вікно**, **з якого здійснюються всі запити**. Цих запитів 14. **Для кожного** з них **створено IFormUi-вікно та IFormHandler-клас**.

**Детальніше** про кожен із них. Принцип роботи інтерфейсу форми вже було детально розглянуто в попередніх розділах, однак **варто уточнити кілька моментів**. Зокрема **стосовно випадаючих списків** – **ListField** та їх логічної абстракції – EnumField. **Основною властивістю** будь-якого такого **списку є його варіанти вибору**. В контексті розроблюваної АІС можна видіти два аспекти, що їх стосуються.

По-перше, ці **варіанти часто повторюються** в різних формах (тобто діалогових вікнах). Це чудово видно з таблиці 2.3.3.3.1, в якій наведено всі поля, необхідні для введення параметрів для кожного запиту. Там аж **40 полів, з яких 36 – це випадаючі списки**. Однак з цих 36-и **унікальні лише 11**, решта – повтори.

Вводити їх вручну кожен раз доволі не раціонально. З одного боку вкрай працемістко. З іншого це може призвести до помилок. То ж **було створено окремий глобальний словник** (щось на кшталт Dictionary<enum, string[]>), що містить всі набори варіантів спадних списків, що зустрічаються в додатку.

Ключем в цьому словник є enum-клас, тож **помилка при виборі потрібного списку**, пов’язана з людським фактором (простіше неуважністю), **майже виключена**. Таким чином **все, що треба для заповнення** спадного списку – **обрати** в EnumField **відповідне значення ключа** з глобального словника.

Таблиця 2.3.3.3.1 – Поля, необхідні для здійснення кожного запиту

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № | Поле 1 | Поле 2 | Поле 3 | Поле 4 | Поле 5 | Поле 6 |
| 1 | Категорія | Цех |  |  |  |  |
| 2 | Категорія | Підрозділ | Цех | Ділянка | Початкова дата | Кінцева дата |
| 3 | Підрозділ | Цех | Ділянка | Персонал | Категорія інженерів | Категорія робітників |
| 4 | Цех |  |  |  |  |  |
| 5 | Виріб |  |  |  |  |  |
| 6 | Цех | Ділянка |  |  |  |  |
| 7 | Цех | Ділянка |  |  |  |  |
| 8 | Категорія | Підрозділ | Цех | Ділянка |  |  |
| 9 | Виріб |  |  |  |  |  |
| 10 | Виріб |  |  |  |  |  |
| 11 | Категорія | Лабораторія | Початкова дата | Кінцева дата |  |  |
| 12 | Вироби | Категорія | Тип | Лабораторія |  |  |
| 13 | Вироби | Категорія | Тип | Лабораторія |  |  |
| 14 | Категорія | Підрозділ | Цех | Ділянка |  |  |

Іншою проблемою є те, що **деякі спадні меню беруть свої набори значення з бази даних**. Наприклад: список виробів чи ділянок підприємства (див рис. 2.3.3.3.2). Можна ввести їх вручну, але це абсурд, оскільки треба буде переписувати програму при внесені змін в базу даних. **Рішенням для цієї проблеми стало отримання таких списків за допомогою запитів до бази даних**. Всі ці запити здійснюються в раніше згаданому ServerConenctor`і. Тобто при під’єднанні до сервера. Їхні результаті зберігаються в уже згаданому глобальному словнику під відповідними ключами й до них можна отримати доступ за допомогою наведеного у попередньому абзаці способу.

Розмітка: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/General/Form/Field/UI/ListField.xaml> ()

Код: <https://github.com/Denchyk-chyk/cw-ad-2024-code/blob/master/CS/General/Form/Field/UI/ListField.xaml.cs> ()

Рис. 2.3.3.3.2 – Випадаючий список

Зображення, що містить текст, електроніка, знімок екрана, монітор

Автоматично згенерований опис

Крім того варто розглянути **ще один аспект**, що стосується IFormUi. Для цього повернемось до таблиці 2.3.3.3.1, яка містить всі **поля**, необхідні для введення параметрів щодо виконання всіх чотирнадцяти запитів до БД. Як не складно помітити **частина їх збігається.** **Форми 5, 9, 10; 6, 7 та 12, 13, 14 – ідентичні**. Оскільки інтерфейс достатньо відділений від бізнес-логіки можна просто використати одну форму в кількох запитах. Таким чином достатньо створити всього 9 форм замість 14, оптимізувавши 36% роботи завдяки правильно підібраній архітектурі Проєкту.

**Що** **стосується Queries, то всі вони є класами дочірніми до супертипу Query**. Цей **клас містить один метод** для здійснення запиту, однак він **поділений на окремі підметоди, реалізуючи тим самим патерн Шаблонний метод**. Це зроблено для того, щоб в дочірніх класах можна було перевантажувати лише необхідну частину логіки, тим самим сильно спрощуючи процес успадковувавання.

Варто уточнити **принцип роботи** цього **єдиного методу класу Query**. Він поділяється на наступні етапи, виділені у відповідні підметоди:

1. **Створення текстового рядка** **із SQL-запитом** (select`ом) на основі даних, отриманих із форми. Це ключовий метод, який завжди перевантажується в дочірніх класах.
2. **Ініціалізація об’єкту-читача** (NpgsqlDataReader) на основі рядка із запитом. Цей читач проходиться кожним записом отриманого результату запиту (рядком таблиці). Знаходячись на певному рядку, читач може отримати значення будь-якого поля за його назвою.
3. **Прочитання кожного рядка**. Метод, який завжди перевантажується в дочірніх класах, однак **більшість логіки цього процесу передана стороннім об’єктам типу Displayable**. Кожен із них створює текстовий рядок на основі запису із таблиці. Тобто отримує в якості параметра конструктора NpgsqlDataReader, використовуючи який отримує потрібні дані з рядка, який ним читається у відповідний момент. Підкласу Query **треба лиш обрати** потрібний **тип** такого **Displayable-об’єкта**.
4. **Вивід даних**. В процесі прочитання таблиці із результатом select`у утворюється колекція Displayable-об’єктів. Вони передаються класу Display, який виводить їх на екран в окремому вікні.

Розгляньмо **особливості Displayable-об’єктів**. Їхні конструктори приймають в якості атрибуту запис select-таблиці, отримують з нього значення відповідних полів і зберігають їх в якості параметрів, з яких потім створюють рядок для виводу. Цей процес часто тривіальний. **Найпростіший такий об’єкт – Line. Він просто зчитує одне поле й виводить його значення**. Назва цього поля передається в якості параметра конструктора разом із записом (хоча зазвичай використовується значення за умовчуванням – «name»). Однак бувають набагато цікавіші випадки.

**Деякі із select-таблиць стосуються об’єктів зі складною ієрархією**. Наприклад персоналу підприємства. Кожен працівник входить одночасно до кількох таблиць. Скажімо персонал, робітники, зварювальники чи персонал, інженери, менеджмент. Оскільки в рамках реалізації запитів у Проєкті **можлива лише одна таблиця із результатами доводиться проводити join всіх таблиць ієрархії**. **На виході** отримуємо результуючу таблицю з десятками полів, **більшість записів**, яких **мають null-значення**. **Прочитання** такої таблиці **зводиться до того, що Displayable-об’єкт** (в цьому випадку Employee) **проходиться всіма полями таблиці й до додає їх значення в свій список параметрів, якщо воно не null.**

Бувають і цікавіші випадки. Наприклад **запит з відомостями про кількість та дату виробництва автомобілів**. Отримати інформацію про дати виробництва не складно. Інформацію про кількість виробів теж – для цього потрібно лиш згрупувати запит за відповідними полями. Однак для отримання обох відомостей **потрібні два select’и**, що, як і в минулому абзаці заперечує архітектуру Програми.

**Тож доводиться проводити «групування» на рівні C#.** Відповідні Displayable-об’єкти (Production) працюють наступними чином. В них приватні конструктори. Ззовні доступний лише метод розширення, що дозволяє додати об’єкт до списку List<Production>, передавши NpgsqlDataReader (тобто запис таблиці) в якості параметру. При цьому ідентифікатор рядка порівнюється з наявними у списку, якщо він відступній, то створюється новий Production, якщо ні, то дані з цього запису (дата й кількість виготовлених машин) додаються до існуючого об’єкта з відповідним id. Вся логіка інкапсульована в межах Displayable-об’єкта. Клієнтському коду лише треба створити список і додати до нього всі записи із select-таблиці.

## 2.4 Тестування

### 2.4.1 Компіляція

Після завершення розробки Застосунку потрібно скомпілювати його робочу (Release) версію й створити для неї .exe-архів завдяки якому Застосунок можна буде встановити на будь-який пристрій на базі Windows 7+. Крім того, враховуючи орієнтованість розроблюваної АІС на роботу з базою даних, доцільно експортувати її, для того, щоб дозволити подальший її імпорт на ПК, на яких встановлено Програму чи на сервер, до якого матимуть доступ ці комп’ютери.

Для створення .exe – файлу використано утиліту Inno Setup Compiler (Рис. 2.4.1.1). Вона дає можливість просто і швидко створити відповідний виконуваний файл, не заглиблюючись в особливості процесу. Програма дозволяє створити скрипт, як вручну так і через графічний інтерфейс. Виконання цього скрипта системою створює відповідний файл. Щодо експорту бази даних, то для цього використано інструменти створення Backup-файлів програми phAdmin4, призначеної для адміністрування баз даних на базі PostgreSQL.

Рис. 2.4.1.1 – Вікно програми Inno Setup Compiler Зображення, що містить текст, знімок екрана, програмне забезпечення, комп’ютер

Автоматично згенерований опис

.exe- та .sql-файли, що містять Застосунок та базу даних відповідно розміщено на GitHub за наступними посиланням. Там же ж знаходиться вихідний код Програми. Тобто - Visual Studio Solution File (.sln) та всі файли проєкту (зокрема C#-код і XAML-розмітка).

Застосунок: <https://github.com/Denchyk-chyk/cw-ad-2024-result/blob/master/Setup.exe>

База даних: <https://github.com/Denchyk-chyk/cw-ad-2024-result/blob/master/Factory.sql>

Вихідний код: <https://github.com/Denchyk-chyk/cw-ad-2024-code>

### 2.4.2 Запити

Для тестування Застосунку перевіримо роботу кожного із його чотирнадцяти запитів.

Перший запит сформульовано наступним чином: «Перелік видів виробів окремої категорії і в цілому, що збираються зазначеним цехом, підприємством».

Таким чином потрібно отримати перелік виробів із повною інформацією про них. Для фільтрування результатів потрібні 2 поля: категорія і цех. Кожне із них є необов’язковим.

Рис. 2.4.2.1 – Тестування першого запиту

Зображення, що містить текст, знімок екрана, дизайн, Прямокутник

Автоматично згенерований опис

Другий запит вказано наступним чином: «Число і перелік виробів окремої категорії і в цілому, зібраних зазначеним цехом, ділянкою, підприємством в цілому за певний відрізок часу.».

Отже слід отримати перелік дат, кількості та назв вироблених автомобілів. При цьому їх ще треба згрупувати, щоб дізнатися загальну кількість. Для відбору результатів потрібні наступні поля: початкова й кінцева дати, категорія виробу (необов’язково), а також блок, що відповідає за структурну одиницю підприємства: перемикач (підприємство, цех, ділянка) й поля для вибору власне цеху й ділянки.

Рис. 2.4.2.2 – Тестування другого запиту

Зображення, що містить текст, знімок екрана, дизайн

Автоматично згенерований опис

Третій запит заявлено так: «Дані про кадровий склад цеху, підприємства в цілому і по зазначеним категоріям інженерно-технічного персоналу і робітників».

Тобто потрібно отримати кількість персоналу відповідних категорій, що пройдуть відповідні умови. Для відбору результатів потрібні вже описаний блок структурного підрозділу підприємства та ще один блок – підрозділу персоналу: перемикач (персонал, інженери, робітники) та поля з категоріями відповідного типу працівників.

Рис. 2.4.2.3 – Тестування третього запиту

Зображення, що містить текст, знімок екрана, Прямокутник, дизайн

Автоматично згенерований опис

Четвертий запит подано таким чином: «Перелік ділянок зазначеного цеху, підприємства в цілому та їх начальників».

Тож треба лише дізнатись всі дані про ділянки цеху чи підприємства загалом, а також короткі відомості про їх начальників. Для фільтрації результатів достатньо необов’язково поля вибору цеху.

Рис. 2.4.2.4 – Тестування четвертого запиту

Зображення, що містить текст, знімок екрана, дизайн

Автоматично згенерований опис

П’ятий запит має такий вигляд: «Перелік робіт, які проходить вказаний виріб». Потрібен звичайний список назв робіт. Для фільтрації достатньо поля з типом виробу.

Рис. 2.4.2.5 – Тестування п’ятого запиту

Зображення, що містить текст, знімок екрана, програмне забезпечення, дизайн

Автоматично згенерований описШостий запит звучить так: «Склад бригад зазначеної ділянки, цеху Склад бригад зазначеної ділянки, цеху».

Отже потрібно отримати перелік робітників з повною інформацією про них. Для їх відбору потрібні 2 необов’язкові поля: цех і ділянка.

Рис. 2.4.2.6 – Тестування шостого запиту

Зображення, що містить текст, знімок екрана, число, дизайн

Автоматично згенерований опис

Сьомий запит подано таким чином: «Перелік майстрів вказаної ділянки, цеху». Він відрізняється від попереднього лише тим, що стосується інженерно-технічного персоналу, а не робітників.

Рис. 2.4.2.7 – Тестування сьомого запиту

Зображення, що містить текст, знімок екрана, дизайн

Автоматично згенерований опис

Восьмий запит представлений таким формулюванням: «Перелік виробів окремої категорії і в цілому, що збираються зараз зазначеним ділянкою, цехом, підприємством».

Достатньо лише отримати кількість всіх ще не зібраних автомобілів (тобто тих, чия дата випуску ще не настала) й короткі відомості про них. Для відбору потрібні необов’язкове поле категорії виробу й блок структурного підрозділу підприємства.

Рис. 2.4.2.8 – Тестування восьмого запиту

Зображення, що містить текст, знімок екрана, дизайн, Прямокутник

Автоматично згенерований опис

Дев’ятий запит: «Склад бригад, що беруть участь в складанні зазначеного виробу».

Слід вказати повну інформацію про робітників відповідних бригад. Для фільтрації потрібне лише поле вибору виробу.

Рис. 2.4.2.9 – Тестування дев’ятого запиту

Зображення, що містить текст, програмне забезпечення, число, знімок екрана

Автоматично згенерований опис

Десятий запит представлено так: «Перелік випробувальних лабораторій, що беруть участь у випробуваннях деякого конкретного виробу». Тобто треба просто вказати назви лабораторій, в яких тестується виріб вибраний у відповідному полі.

Рис. 2.4.2.10 – Тестування десятого запиту

Зображення, що містить текст, знімок екрана, Побутова техніка, дизайн

Автоматично згенерований опис

Одинадцятий запис сформульовано як: «Перелік виробів окремої категорії і в цілому, що проходили випробування у зазначеній лабораторії за певний період»

Запит подібний на другий, однак замість структурного підрозділу підприємства й блоку для його вказання лише поле для вибору лабораторії.

Рис. 2.4.2.11 – Тестування одинадцятого запиту

Зображення, що містить текст, знімок екрана, дизайн

Автоматично згенерований опис

Дванадцятий запис подано наступним чином: «Перелік випробувачів, що беруть участь у випробуваннях зазначеного виробу, виробів окремої категорії і в цілому у вказаній лабораторії за певний період».

Потрібно отримати повну інформацію про випробувачів, що працюють з вказаним виробом. Для фільтрації потрібні блок вибору виробу: перемикач (категорія, виріб) й 2 відповідні поля, а ще поле вибору лабораторії.

Рис. 2.4.2.12 – Тестування дванадцятого запиту

Зображення, що містить текст, знімок екрана, Прямокутник, дизайн

Автоматично згенерований опис

Тринадцятий запит заявлено так: «Склад обладнання, що використовувалося при випробуванні зазначеного виробу, виробів окремої категорії і в цілому у вказаній лабораторії за певний період». Черговий список назв. Поля для фільтрації такі ж, як і в минулому запиті.

Рис. 2.4.2.13 – Тестування тринадцятого запиту

Зображення, що містить прилад, Побутова техніка, кухонна техніка, текст

Автоматично згенерований опис

Чотирнадцятий запит вказано наступним чином: «Число і перелік виробів окремої категорії і в цілому, що збираються зазначеним цехом, ділянкою, підприємством в даний час». Єдина різниця між цим запитом і другим полягає у вказуванні повної інформації про вироби, а не лише їхньої назви.

Рис. 2.4.2.14 – Тестування чотирнадцятого запиту

Зображення, що містить текст, дизайн, знімок екрана

Автоматично згенерований опис

## 2.5 Висновок

В цьому розділі було детально **описано процес створення** АІС, що є завданням курсової роботи. Наведено основні етапи розробки. Аналіз проєкту. Створення дорожньої карти проєкту. Проєктування архітектури застосунку Створення бази даних. Верстання графічного інтерфейсу. Програмування основної частини. Посилання на всі дані наявні через GitHub.

# Список використаних джерел

1. Refactoring Guru. Design Patterns. What's a Design Pattern? URL: [https://refactoring.guru/design–patterns/what–is–pattern](https://refactoring.guru/design-patterns/what-is-pattern) (дата звернення: 01.06.2024)
2. Refactoring Guru. Design Patterns. The Catalog of Design Patterns. URL: [https://refactoring.guru/design–patterns/catalog](https://refactoring.guru/design-patterns/catalog) (дата звернення: 01.06.2024)
3. Refactoring Guru. Design Patterns. Singleton. URL: [https://refactoring.guru/design–patterns/singleton](https://refactoring.guru/design-patterns/singleton) (дата звернення: 01.06.2024)
4. Refactoring Guru. Design Patterns. Builder. URL: <https://refactoring.guru/design-patterns/builder> (дата звернення: 01.06.2024)
5. Refactoring Guru. Design Patterns. Prototype. URL: <https://refactoring.guru/design-patterns/prototype> (дата звернення: 01.06.2024)
6. Refactoring Guru. Design Patterns. Factory Method. URL: <https://refactoring.guru/design-patterns/factory-method> (дата звернення: 01.06.2024)
7. Refactoring Guru. Design Patterns. Abstract Factory. URL: <https://refactoring.guru/design-patterns/abstract-factory> (дата звернення: 01.06.2024)
8. Refactoring Guru. Design Patterns. Adapter. URL: <https://refactoring.guru/design-patterns/adapter> (дата звернення: 01.06.2024)
9. Refactoring Guru. Design Patterns. Bridge. URL: <https://refactoring.guru/design-patterns/bridge> (дата звернення: 01.06.2024)
10. Refactoring Guru. Design Patterns. Composite. URL: <https://refactoring.guru/design-patterns/composite> (дата звернення: 01.06.2024)
11. Refactoring Guru. Design Patterns. Decorator. URL: <https://refactoring.guru/design-patterns/decorator> (дата звернення: 01.06.2024)
12. Refactoring Guru. Design Patterns. Facade. URL: <https://refactoring.guru/design-patterns/facade> (дата звернення: 01.06.2024)
13. Refactoring Guru. Design Patterns. Flyweight. URL: <https://refactoring.guru/design-patterns/flyweight> (дата звернення: 01.06.2024)
14. Refactoring Guru. Design Patterns. Proxy. URL: <https://refactoring.guru/design-patterns/proxy> (дата звернення: 01.06.2024)
15. Refactoring Guru. Design Patterns. Chain of Responsibility. URL: <https://refactoring.guru/design-patterns/chain-of-responsibility> (дата звернення: 01.06.2024)
16. Refactoring Guru. Design Patterns. Command. URL: <https://refactoring.guru/design-patterns/command> (дата звернення: 01.06.2024)
17. Refactoring Guru. Design Patterns. Iterator. URL: <https://refactoring.guru/design-patterns/iterator> (дата звернення: 01.06.2024)
18. Refactoring Guru. Design Patterns. Mediator. URL: <https://refactoring.guru/design-patterns/mediator> (дата звернення: 01.06.2024)
19. Refactoring Guru. Design Patterns. Memento. URL: <https://refactoring.guru/design-patterns/memento> (дата звернення: 01.06.2024)
20. Refactoring Guru. Design Patterns. State. URL: <https://refactoring.guru/design-patterns/state> (дата звернення: 01.06.2024)
21. Refactoring Guru. Design Patterns. Strategy. URL: <https://refactoring.guru/design-patterns/strategy> (дата звернення: 01.06.2024)
22. Refactoring Guru. Design Patterns. Template Method. URL: <https://refactoring.guru/design-patterns/template-method> (дата звернення: 01.06.2024)
23. Refactoring Guru. Design Patterns. Visitor. URL: <https://refactoring.guru/design-patterns/visitor> (дата звернення: 01.06.2024)
24. Refactoring Guru. Design Patterns. Observer. URL: <https://refactoring.guru/design-patterns/observer> (дата звернення: 01.06.2024)