* 1. 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]

* 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]

* 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]

* 1. 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**.

[1]

* + 1. 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, ряд, знімок екрана, Шрифт

Автоматично згенерований опис

* + 1. 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, Паралель, знімок екрана

Автоматично згенерований опис

* + 1. 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, квитанція, схема, ряд

Автоматично згенерований опис

* + 1. 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, квитанція, схема, знімок екрана

Автоматично згенерований опис

* + 1. 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, Паралель

Автоматично згенерований опис

* 1. 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**.

* + 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, ряд, Шрифт

Автоматично згенерований опис**

* + 1. 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).

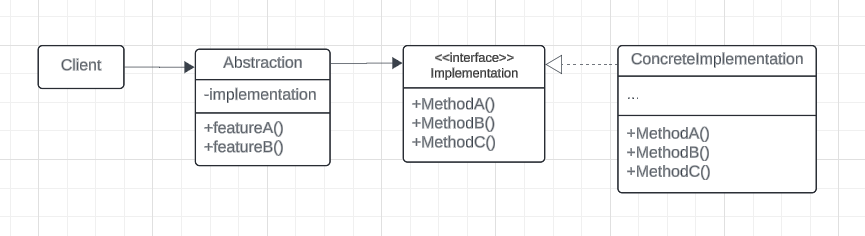
**Рішення**

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



* + 1. 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, схема, Шрифт, знімок екрана

Автоматично згенерований опис

* + 1. 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, Шрифт, знімок екрана, ряд

Автоматично згенерований опис

* + 1. 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, ряд, знімок екрана, Шрифт

Автоматично згенерований опис

* + 1. 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, ряд, схема, Прямокутник

Автоматично згенерований опис

* + 1. 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

* 1. 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. 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. 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**

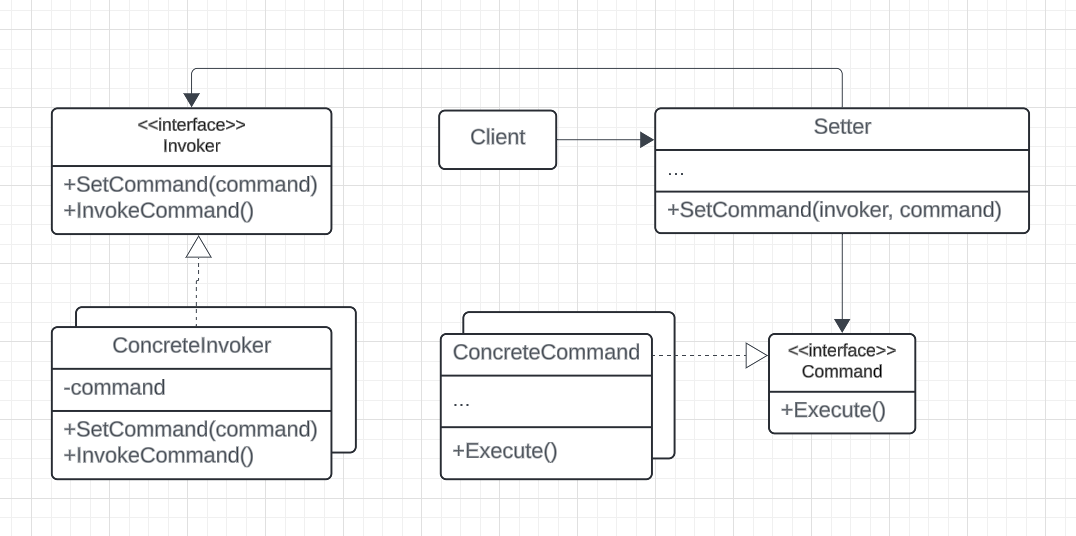
Хороший дизайн ПЗ вимагає **відділення користувацького інтерфейсу**(надалі UI) **від бізнес-логіки проєкту**. Тобто кнопки, пункти меню, комбінації клавіш тощо **повинні викликати** вже **існуючу функцію** з бізнес-логіки.

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



* + 1. 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.**

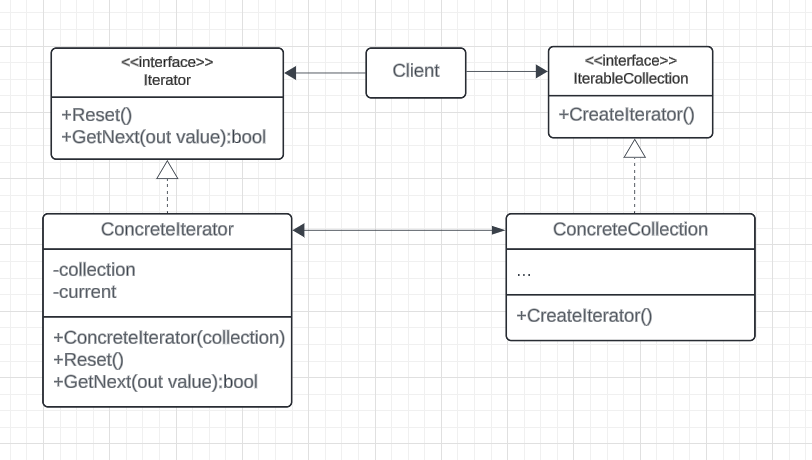
**Додавання** складних **алгоритмів обходу**, особливо, якщо їх декілька в **код** самої **колекції** **негативно вплине** на оптимізацію. Та й **не завжди можливо модифікувати наявний код** (наприклад, якщо він міститься в закритій бібліотеці).

**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



* + 1. 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. 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, схема, ряд, знімок екрана

Автоматично згенерований опис

* + 1. 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, схема, ряд, знімок екрана

Автоматично згенерований опис

* + 1. 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, схема, ряд

Автоматично згенерований опис

* + 1. 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, схема, Шрифт

Автоматично згенерований опис

* + 1. 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

* + 1. 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, схема, ряд

Автоматично згенерований опис

* 1. Висновок

В цьому розділі було 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**.

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)