

Міністерство освіти і науки України
Запорізький національний технічний університет

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторних робіт
з дисципліни
“ПРОЕКТНИЙ ПРАКТИКУМ”
для студентів
напряму підготовки 6.050103 “Програмна інженерія”
всіх форм навчання

Методичні вказівки до лабораторних робіт з дисципліни
“Проектний практикум” для студентів напряму підготовки 6.050103
“Програмна інженерія” всіх форм навчання / Т.В. Юр, Н.О. Миронова.
– Запоріжжя: ЗНТУ, 2013. – 59 с.

Автори: Т. В. Юр, к.т.н., доцент
Н. О. Миронова, ст. викладач

Рецензент: А.В. Притула, к.т.н., проф.

Відповідальний
за випуск: Т. В. Юр, к.т.н., доцент

Затверджено
на засіданні кафедри
програмних засобів

Протокол №4
від “16” грудня 2013 р.

ЗМІСТ

Вступ.....	4
Лабораторна робота № 1. Твірні шаблони	6
Лабораторна робота № 2. Структурні шаблони проектування	15
Лабораторна робота № 3. Шаблони поведінки. Частина 1	30
Лабораторна робота № 4. Шаблони поведінки. Частина 2	43
Лабораторна робота № 5. Практичне використання шаблонів проектування.....	53
Література.....	55
Додаток А. Коротка довідка з мови UML	56

ВСТУП

Практична частина курсу «Проектний практикум» складається з п'яти лабораторних робіт і призначена для отримання практичних навичок з використання шаблонів проектування при розробці програмного забезпечення з метою підвищення його внутрішньої якості. Лабораторні роботи можуть бути виконані з використанням будь-якої об'єктно-орієнтованої мови програмування.

Матеріал до кожної лабораторної роботи містить мету, теоретичні довідки та рекомендації, загальне завдання, варіанти індивідуальних завдань, список питань для самоперевірки, зміст звіту про виконання лабораторних робіт, а також список рекомендованих інформаційних джерел для підготовки і виконання лабораторних робіт.

Шаблони проектування (паттерни) – це повторювана архітектурна конструкція, що представляє собою рішення проблеми проектування в рамках деякого часто виникаючого контексту. Іншими словами це ефективні способи вирішення задач проектування програмного забезпечення.

Шаблон не є закінченим зразком, який можна безпосередньо транслювати в програмний код. Об'єктно-орієнтований шаблон найчастіше є зразком вирішення проблеми проектування і відображає відношення між класами та об'єктами, без вказівки на те, як це відношення буде реалізоване. Розрізняють такі типи шаблонів: твірні шаблони, структурні шаблони, шаблони поведінки.

Твірні шаблони – це шаблони проектування, що абстрагують процес інстанціювання. Вони допоможуть зробити систему незалежною від способу створення, композиції та представлення її об'єктів. Шаблон, що створює об'єкти, делегує інстанціювання іншому об'єктові. Ці шаблони важливі, коли система більше залежить від композиції об'єктів, ніж від спадкування класів. Таким чином, замість прямого кодування фіксованого набору поведінок, визначається невеликий набір фундаментальних поведінок, за допомогою композиції яких можна отримувати складніші. Твірні шаблони інкапсулюють знання про конкретні класи, які застосовуються у системі та приховують деталі того, як ці класи створюються і стикаються між собою. Єдина інформація про об'єкти, що відома системі, – їхні інтерфейси.

Структурні шаблони – шаблони проектування, у яких розглядається питання про те, як із класів та об'єктів утворюються більші за розмірами структури. Структурні шаблони рівня класу використовують спадковість для утворення композицій із інтерфейсів та реалізацій. Структурні шаблони рівня об'єкта компонують об'єкти для отримання нової функціональності. Додаткова гнучкість у цьому разі пов'язана з можливістю змінювати композицію об'єктів під час виконання, що є неприпустимим для статичної композиції класів.

Шаблони поведінки – шаблони проектування, що пов'язані з алгоритмами та розподілом обов'язків поміж об'єктів. Мова в них йде не тільки про самі об'єкти та класи, але й про типові способи їхньої взаємодії. Шаблони поведінки характеризують складний потік керування, котрий досить важко прослідкувати під час виконання програми. Увага акцентована не на потоці керування, а на зв'язках між об'єктами. У шаблонах поведінки рівня класу використовується спадкування щоб розподілити поведінку поміж різними класами. У шаблонах поведінки рівня об'єкта використовується композиція. Деякі з них описують, як за допомогою кооперації багато рівноправних об'єктів пораяються із завданням, котре жодному з них не під силу. Тут є важливим те, як об'єкти отримують інформацію про існування один одного. Об'єкти-колеги можуть зберігати посилання один на одного, але це посилює ступінь зв'язаності системи. За максимального рівня зв'язаності кожному об'єкту довелось би мати інформацію про всі інші. Деякі з шаблонів поведінки вирішують цю проблему.

ЛАБОРАТОРНА РОБОТА № 1 ТВІРНІ ШАБЛОНИ

1.1 Мета роботи

Ознайомитися з видами шаблонів проектування ПЗ, вивчити творчі шаблони та отримати базові навички з застосування шаблонів Прототип (Prototype), Одинак (Singleton), Фабричний метод (Factory Method), Абстрактна фабрика (Abstract Factory) та Будівник (Builder).

1.2 Основні теоретичні відомості

1.2.1 Шаблон проектування Прототип (Prototype)

Призначення. Задає види об'єктів, що створюються, за допомогою екземпляру-прототипу, та створює нові об'єкти шляхом копіювання цього прототипу.

Застосування. Прототип слід використовувати коли:

- класи, що інстанціюються, визначаються під час виконання, наприклад, за допомогою динамічного завантаження;
- треба запобігти побудові ієрархій класів або фабрик, паралельних ієрархії класів продуктів;
- екземпляри класу можуть знаходитись у одному з декількох станів.

Може статися, що зручніше встановити відповідну кількість прототипів та клонувати їх, а не інстанціювати кожний раз клас вручну в необхідному стані.

На рис. 1.1. наведено **структуру** шаблону Прототип.

Учасниками шаблону є:

- **Prototype** – прототип:
 - визначає інтерфейс для клонування самого себе;
- **ConcretePrototype** – конкретний прототип:
 - реалізує операцію клонування самого себе;
- **Client** – клієнт:
 - створює новий об'єкт, звертаючись до прототипу із запитом клонувати себе.

Відносини. *Клієнт* звертається до *прототипу*, щоб той створив свого клона.

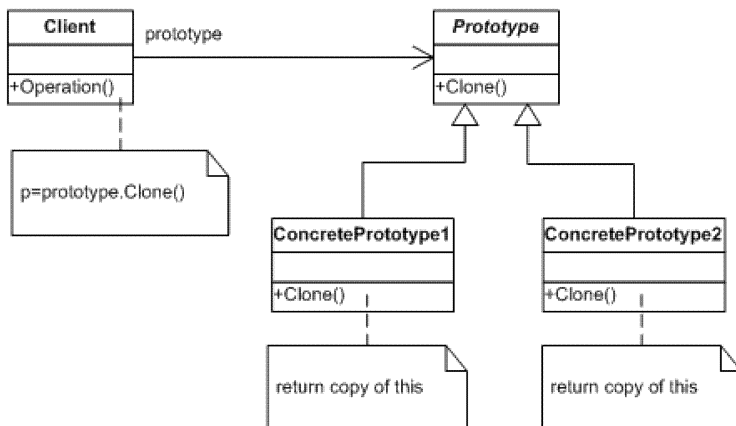


Рисунок 1.1 – UML діаграма, що описує структуру шаблону Prototype

1.2.2 Шаблон Одинак (Singleton)

Призначення. Для деяких класів важливо, щоб існував тільки один екземпляр. Глобальна змінна не вирішує такої проблеми, бо не забороняє створити інші екземпляри класу. Рішення полягає в тому, щоб сам клас контролював свою унікальність, забороняючи створення нових екземплярів, та сам забезпечував єдину точку доступу.

Застосування. Використовується коли:

- повинен бути тільки один єдиний екземпляр деякого класу, що легко доступний всім клієнтам;
- єдиний екземпляр повинен розширюватись шляхом успадкування, та клієнтам потрібно мати можливість працювати з розширеним екземпляром не змінюючи свій код.

Структура шаблону наведена на рис. 1.2.

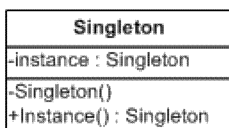


Рисунок 1.2 – Структура шаблону Singleton

Учасниками шаблону є:

- **Singleton** – одинак:
 - визначає операцію *Instance*, котра дозволяє клієнтам

отримувати доступ до єдиного екземпляру;

- може нести відповідальність за створення власного унікального екземпляру.

Відносини. Клієнти отримують доступ до єдиного об'єкта класу *Singleton* лише через його операцію *Instance*.

1.2.3 Шаблон Фабричний метод (Factory Method)

Призначення. Визначає інтерфейс для створення об'єкта, але залишає підкласам рішення про те, який саме клас інстанціювати. Фабричний метод дозволяє класу делегувати інстанціювання підкласам.

Також відомий під назвою віртуальний конструктор (Virtual Constructor).

Застосування. Використовується коли:

- класу не відомо заздалегідь, об'єкти яких саме класів йому потрібно створювати;
- клас спроектовано так, щоб об'єкти, які він створює, специфікувалися підкласами;
- клас делегує свої обов'язки одному з декількох допоміжних підкласів, та потрібно локалізувати знання про те, який саме підклас приймає ці обов'язки на себе.

Структура шаблону наведена на рис. 1.3.

Учасниками шаблону є:

- **Product** – продукт (абстрактний клас):
 - визначає інтерфейс об'єктів, що створюються фабричним методом;
- **ConcreteProduct** – конкретний продукт:
 - реалізує інтерфейс *Product*;
- **Creator** – творець:
 - оголошує фабричний метод *FactoryMethod*, що повертає об'єкт класу *Product*;
 - може також визначати реалізацію за замовчанням фабричного методу, що повертає об'єкт *ConcreteProduct*;
 - може викликати фабричний метод для створення об'єкта *Product*;
- **ConcreteCreator** – конкретний творець:
 - заміщує фабричний метод, що повертає об'єкт

ConcreteProduct.

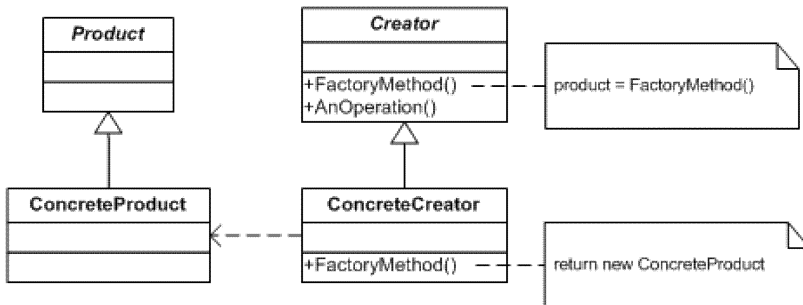


Рисунок 1.3 – Структура шаблону Factory Method

Відносини. Творець покладається на свої підкласи в означенні фабричного методу, котрий буде повертати екземпляр відповідного конкретного продукту.

Результати. Шаблон позбавляє проектувальника від необхідності вбудовувати в код залежні від програми класи. При цьому виникає додатковий рівень підкласів.

1.2.4 Шаблон Абстрактна фабрика (Abstract Factory)

Призначення. Подає інтерфейс для утворення родин взаємозв'язаних або взаємозалежних об'єктів, не специфікуючи їхніх конкретних класів.

Також відомий під назвою інструментарій (Kit).

Застосування. Використовується коли:

- система не повинна залежати від того, як утворюються, компонуються та представляються об'єкти, що входять до неї;
- взаємозв'язані об'єкти, що входять до родини, повинні використовуватися разом і необхідно забезпечити виконання цього обмеження;
- система повинна конфігуруватися однією з родин об'єктів, що її складають;
- треба подати бібліотеку об'єктів, розкриваючи тільки їхні інтерфейси, але не реалізацію.

Структура шаблону наведена на рис. 1.4.

Учасниками шаблону є:

- **AbstractFactory** – абстрактна фабрика:

- оголошує інтерфейс для операцій, що створюють абстрактні об'єкти-продукти;
- **ConcreteFactory** – конкретна фабрика:
 - реалізує операції, що створюють конкретні об'єкти-продукти;
- **AbstractProduct** – абстрактний продукт:
 - оголошує інтерфейс для типу об'єкта-продукту;
- **ConcreteProduct** – конкретний продукт:
 - визначає об'єкт-продукт, що створюється відповідною конкретною фабрикою;
 - реалізує інтерфейс *AbstractProduct*;
- **Client** – клієнт:
 - користується виключно інтерфейсами, котрі оголошені у класах *AbstractFactory* та *AbstractProduct*.

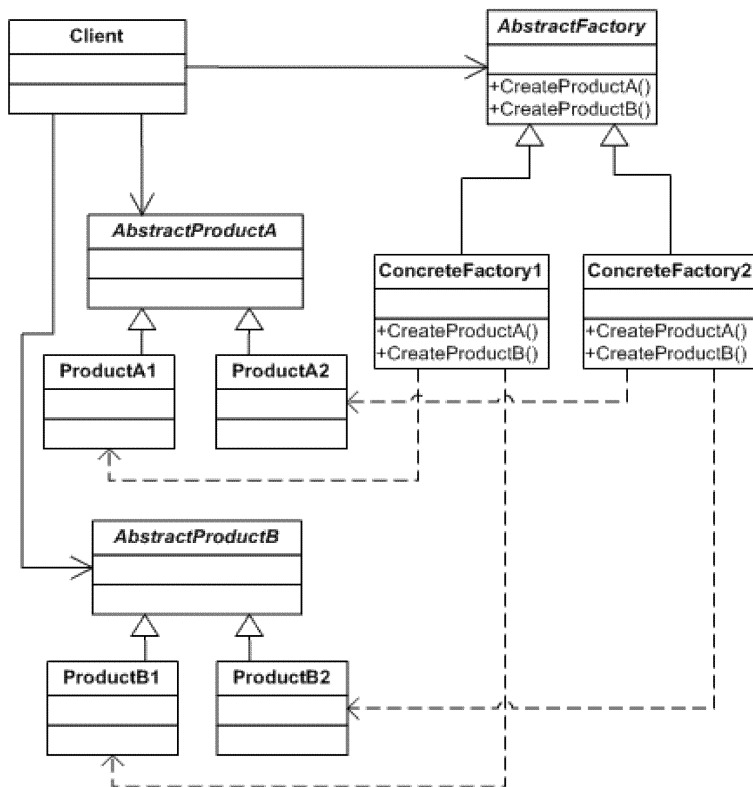


Рисунок 1.4 – Структура шаблону Abstract Factory

Відносини. Зазвичай під час виконання створюється єдиний екземпляр класу *ConcreteFactory*. Ця конкретна фабрика створює об'єкти-продукти, що мають досить визначену реалізацію. Для створення інших видів об'єктів клієнт повинен користуватися іншою конкретною фабрикою.

AbstractFactory передоручає створення об'єктів-продуктів своєму підкласу *ConcreteFactory*.

Результати. Шаблон ізолює конкретні класи. Оскільки *AbstractFactory* інкапсулює відповідальність за створення класів і сам процес їх створення, то вона ізолює клієнта від деталей реалізації класів.

Спрощує заміну сімейств продуктів, оскільки клас конкретної фабрики використовується в додатку тільки один раз при

інстанціюванні.

Слід зазначити, що інтерфейс *AbstractFactory* фіксує набір об'єктів, які можна створити. Це в певній мірі ускладнює розширення *AbstractFactory* для виготовлення нових об'єктів.

1.2.5 Шаблон Будівник (Builder)

Призначення. Відокремлює конструювання складного об'єкта від його подання таким чином, що у результаті одного й того ж процесу конструювання можуть бути отримані різні подання.

Застосування. Використовується коли:

- алгоритм створення складного об'єкта не повинен залежати від того, з яких частин складається об'єкт та як вони стикаються поміж собою;
- процес конструювання повинен забезпечити різні подання об'єкта, що конструюється.

Структура шаблону наведена на рис. 1.5.

Учасниками шаблону є:

- **Builder** – будівник:
 - визначає абстрактний інтерфейс для створення частин об'єкта *Product*;
- **ConcreteBuilder** – конкретний будівник:
 - конструює та збирає докупи частини продукту шляхом реалізації інтерфейсу *Builder*;
 - визначає подання, що створюється, та слідкує за ним;
 - надає інтерфейс для доступу до продукту;
- **Director** – розпорядник:
 - конструює об'єкт, користуючись інтерфейсом *Builder*;
- **Product** – продукт:
 - подає складний конструйований об'єкт, при чому *ConcreteBuilder* будує внутрішнє подання продукту та визначає процес його зборки;
 - вносить класи, що визначають складені частини, у тому числі інтерфейси для зборки кінцевого результату з частин.

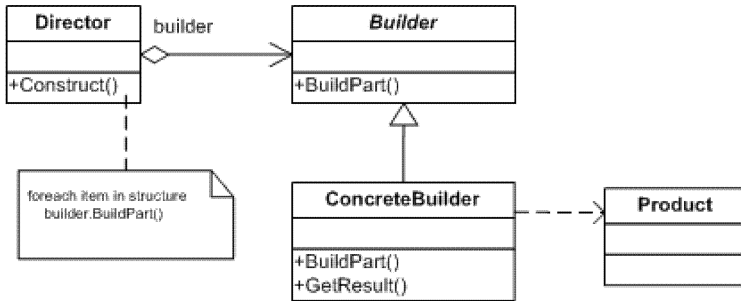


Рисунок 1.5 – Структура шаблону Builder

Відносини. *Client* створює об'єкт-розпорядник *Director* і конфігурує його об'єктом *Builder*. *Director* повідомляє *Builder* про те, що потрібно побудувати чергову частину *Product*. *Builder* обробляє запити *Director* і додає нові частини до *Product*, потім *Client* забирає *Product* у *Builder*.

Результати. Об'єкт *Builder* надає об'єкту *Director* абстрактний інтерфейс для конструювання *Product*, за яким може приховати подання та внутрішню структуру продукту, та, крім того, процес складання *Product*. Для зміни внутрішнього представлення *Product* досить визначити новий вид *Builder*. Даний шаблон ізолює код, що реалізує створення об'єкта та його подання.

1.3 Завдання на лабораторну роботу

1.3.1 Вивчити твірні шаблони проектування, їхню загальну характеристику та призначення. Для кожного з цих шаблонів:

- вивчити призначення шаблону, альтернативні назви, призначення, випадки коли його застосування є доцільним та результати такого застосування;
- знати особливості реалізації шаблону, споріднені шаблони, відомі випадки його застосування в програмних додатках;
- вільно володіти структурою шаблону, призначенням його класів та відносинами між ними.

1.3.2 За допомогою джерел зі списку рекомендованих для виконання лабораторної роботи, виконати програмну реалізацію твірних шаблонів.

1.4 Зміст звіту

- 1.4.1 Тема та мета роботи.
- 1.4.2 Короткі теоретичні відомості.
- 1.4.3 Результати роботи.
- 1.4.4 Висновки, що відображають результати виконання роботи та їх критичний аналіз.

1.5 Контрольні запитання

- 1.5.1 Класифікація шаблонів проектування ПЗ.
- 1.5.2 Призначення твірних шаблонів.
- 1.5.3 Коротка характеристика кожного твірного шаблону.
- 1.5.4 Назви, призначення та мотивація шаблону Prototype.
- 1.5.5 Структура шаблону Prototype та його учасники.
- 1.5.6 Особливості реалізації шаблону Prototype. Результат використання шаблону.
- 1.5.7 Назви, призначення та мотивація шаблону Singleton.
- 1.5.8 Структура шаблону Singleton та його учасники.
- 1.5.9 Особливості реалізації шаблону Singleton. Результат використання шаблону.
- 1.5.10 Назви, призначення та мотивація шаблону Factory Method.
- 1.5.11 Структура шаблону Factory Method та його учасники.
- 1.5.12 Особливості реалізації шаблону Factory Method. Результат використання шаблону.
- 1.5.13 Шаблони, які використовуються сумісно з Prototype, Singleton та Factory Method.
- 1.5.14 Глибоке та поверхнєве клонування.
- 1.5.15 Назви, призначення та мотивація шаблону Abstract Factory.
- 1.5.16 Структура шаблону Abstract Factory та його учасники.
- 1.5.17 Особливості реалізації шаблону Abstract Factory. Результат використання шаблону.
- 1.5.18 Назви, призначення та мотивація шаблону Builder.
- 1.5.19 Структура шаблону Builder та його учасники.
- 1.5.20 Особливості реалізації шаблону Builder. Результат використання шаблону.
- 1.5.21 Шаблони, які використовуються сумісно з Abstract Factory та Builder.

ЛАБОРАТОРНА РОБОТА № 2 СТРУКТУРНІ ШАБЛони ПРОЕКТУВАННЯ

2.1 Мета роботи

Вивчити структурні шаблони, отримати базові навички із застосування шаблонів Адаптер (Adapter), Декоратор (Decorator), Замісник (Proxy), Компонувальник (Composite), Міст (Bridge), Легковаговик (Flyweight), Фасад (Facade).

2.2 Основні теоретичні відомості

2.2.1 Шаблон проектування Адаптер (Adapter)

Призначення. Адаптує інтерфейс одного класу в інтерфейс іншого класу, очікуваного клієнтом, тобто адаптер забезпечує роботу класів з несумісними інтерфейсами.

Також відомий під назвою обгортка (wrapper).

Застосування. Використовується коли:

- необхідно використати існуючий клас, що надає необхідний функціонал, але його інтерфейс не відповідає потребі.

Адаптер передбачає створення класу-оболонки з необхідним інтерфейсом.

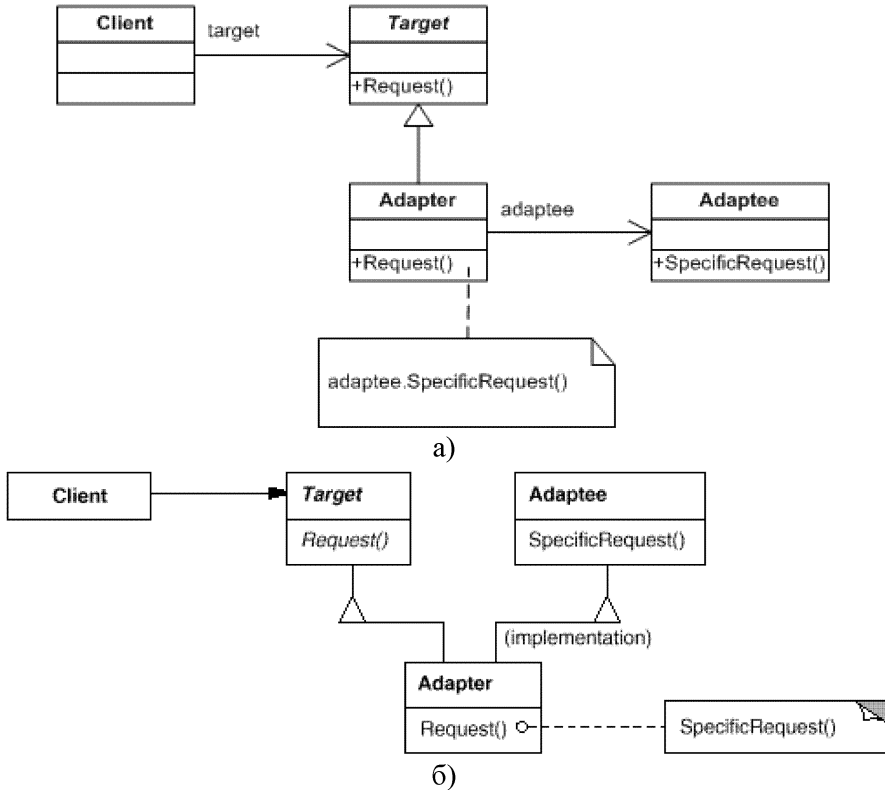
В залежності від схеми роботи з адаптованим об'єктом розрізняють два типи адаптерів:

- адаптер класу – використовує множинне наслідування для адаптації одного інтерфейсу до іншого;
- адаптер об'єкту – використовує композицію об'єктів.

Структура шаблону наведена на рис. 2.1.

Учасниками шаблону є:

- **Target** – цільовий:
 - визначає інтерфейс, що залежить від предметної області, і яким користується *Client*;
- **Client** – клієнт:
 - вступає у взаємини з об'єктами, що задовольняють інтерфейсу *Target*;
- **Adaptee** – адаптований:
 - визначає існуючий інтерфейс, який необхідно адаптувати;
- **Adapter** – адаптер:
 - адаптує інтерфейс *Adaptee* до інтерфейсу *Target*.



а) адаптер об'єкту, б) адаптер класу
Рисунок 2.1 – Структура шаблону Adapter

Клас *Adapter* приводить інтерфейс класу *Adaptee* у відповідність з інтерфейсом класу *Target*, що дозволяє об'єктові *Client* використовувати об'єкт *Adaptee* так, немов він є екземпляром класу *Target*.

Відносини. Клієнти визивають операції екземпляру адаптеру *Adapter*. В свою чергу адаптер визиває операції адаптованого об'єкту або класу *Adaptee*, який і виконує запит.

Результати. Шаблон Адаптер дозволяє включати вже існуючі об'єкти в нові об'єктні структури, незалежно від відмінностей в їхніх інтерфейсах.

2.2.2 Шаблон проектування Декоратор (Decorator)

Призначення. Покласти додаткові обов'язки (прозорі для клієнтів) на окремий об'єкт, а не на клас в цілому. Вирішення цієї задачі досягається шляхом динамічного додавання об'єкту нових обов'язків не вдаючись при цьому до породження підкласів (наслідування).

Також відомий під назвою обгортка (wrapper).

Застосування. Використовується коли:

- необхідно динамічно додати об'єктам додаткові обов'язки;
- необхідно реалізувати обов'язки, що можуть бути зняті з об'єкту;
- розширення шляхом породження підкласів з якихось причин незручно або неможливо.

Структура шаблону наведена на рис. 2.2.

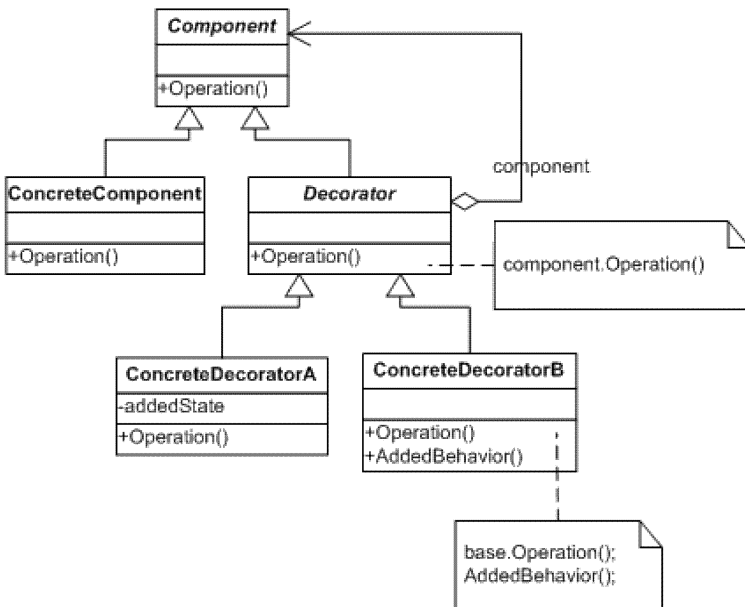


Рисунок 2.2 – Структура шаблону Decorator

Учасниками шаблону є:

- **Component** – компонент:
 - визначає інтерфейс для об'єктів, на які можуть бути

- динамічно покладені додаткові обов'язки;
- **ConcreteComponent** – конкретний компонент:
 - визначає об'єкт, на який покладаються додаткові обов'язки;
- **Decorator** – декоратор:
 - зберігає посилання на об'єкт *Component* та визначає інтерфейс, що відповідає інтерфейсу *Component*;
- **ConcreteDecorator** – конкретний декоратор:
 - покладає додаткові обов'язки на компонент.

Відносини. *Decorator* переадресує запити об'єкту *Component*.

Результати. Застосування декількох *Decorator* до одного *Component* дозволяє довільним чином поєднувати обов'язки, наприклад, одну властивість можна додати двічі.

Функціональність, що додається, реалізується в невеликих об'єктах. Перевага полягає в можливості динамічно додавати цю функціональність до або після основної функціональності об'єкта *ConcreteComponent*. Однак ланцюжок створюваних об'єктів завжди повинен закінчуватися об'єктом класу *ConcreteComponent*.

Дозволяє більшу гнучкість, ніж у статичного наслідування: можна додавати і видаляти обов'язки під час виконання програми в той час як при використанні наслідування треба було б створювати новий клас для кожного додаткового обов'язку. Даний шаблон дозволяє уникнути перевантажених методами класів на верхніх рівнях ієрархії – нові обов'язки можна додавати по мірі необхідності.

Decorator і його *Component* не є ідентичними, і, крім того, виходить, що система складається з великої кількості дрібних об'єктів, які схожі один на одного і розрізняються тільки способом взаємозв'язку, а не класом і не значеннями своїх внутрішніх змінних. Така система складна в вивченні та налагодженні.

2.2.3 Шаблон проектування Замісник (Proxy)

Призначення. Надати об'єкт, що контролює доступ до іншого об'єкту, перехоплюючи всі виклики (виконує функцію контейнеру). Тобто необхідно створити сурогат громіздкого об'єкту.

Також відомий під назвою сурогат (surrogate).

Застосування. Використовується коли необхідно управляти доступом до об'єкта для таких цілей:

- створювати громіздкі об'єкти "на вимогу" (Virtual Proxy);

- кешувати дані (SmartLink Proxy);
- динамічно обмежувати доступ (Security Proxy);
- забезпечувати доступ до об'єкта в іншому адресному просторі (Remote Proxy).

Структура шаблону наведена на рис. 2.3.

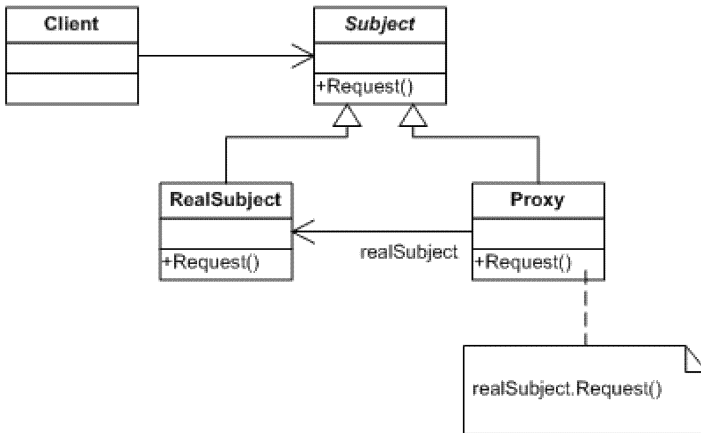


Рисунок 2.3 – Структура шаблону Proxy

Учасниками шаблону є:

- **Proxy** – замісник:
 - зберігає посилання, яке дозволяє заступникові звернутися до реального суб'єкту. Об'єкт класу *Proxy* може звертатися до об'єкту класу *Subject*, якщо інтерфейси класів *RealSubject* та *Subject* однакові;
 - надає інтерфейс, ідентичний інтерфейсу *Subject*, так що замісник завжди може бути підставлений замість реального суб'єкту;
 - контролює доступ до реального суб'єкту і може відповідати за його створення та видалення;
 - інші обов'язки залежать від виду замісника;
- **Subject** – суб'єкт:
 - визначає загальний для *RealSubject* і *Proxy* інтерфейс, так, що *Proxy* може бути використаний скрізь, де очікується *RealSubject*;
- **RealSubject** – реальний суб'єкт:

- визначає реальний об'єкт, що представляється замісником.

Відносини. При необхідності запити можуть бути переадресовані *Proxy* до *RealSubject*. Деталі залежать від виду замісника:

- віддалений замісник відповідає за кодування запиту та його аргументів та відправлення закодованого запита реальному суб'єкту в іншому адресному просторі;
- віртуальний замісник може кешувати додаткову інформацію про реальний суб'єкт, щоб відложити його створення;
- захищаючий замісник перевіряє, чи має об'єкт, що робить запит, необхідні права.

Зазначимо, що адаптер – забезпечує відмінний інтерфейс до об'єкту, проксі – той самий інтерфейс, а декоратор – розширений.

2.2.4 Шаблон проектування Компонувальник (Composite)

Призначення. Об'єднує об'єкти в ієрархічну деревовидну структуру, і дозволяє уніфіковане звертання для кожного елемента дерева. Дозволяє користувачам будувати складні структури з простіших компонентів. Проектувальник може згрупувати дрібні компоненти для формування більших, які, в свою чергу, можуть стати основою для створення ще більших.

Застосування. Використовується коли:

- необхідно представити ієрархію об'єктів виду частина-ціле;
- необхідно щоб клієнти одноманітно трактували складені та індивідуальні об'єкти.

Структура шаблону наведена на рис. 2.4.

Учасниками шаблону є:

- **Component** – компонент:
 - оголошує інтерфейс для об'єктів, що компонуються;
 - надає відповідну реалізацію операцій за замовчуванням, загальну для всіх класів;
 - оголошує єдиний інтерфейс для доступу до нащадків та управління ними;
 - визначає інтерфейс для доступу до батька компонента в рекурсивній структурі і при необхідності реалізує його (можливість необов'язкова);
- **Leaf** – лист.

- об'єкт того ж типу що і *Composite*, але без реалізації контейнерних функцій;
- представляє листові вузли композиції і не має нащадків;
- визначає поведінку примітивних об'єктів в композиції;
- входить до складу контейнерних об'єктів;
- **Composite** – складений об'єкт.
 - визначає поведінку контейнерних об'єктів, у яких є нащадки;
 - зберігає ієрархію компонентів-нащадків;
 - реалізує пов'язані з управлінням нащадками (контейнерні) операції в інтерфейсі класу *Component*.

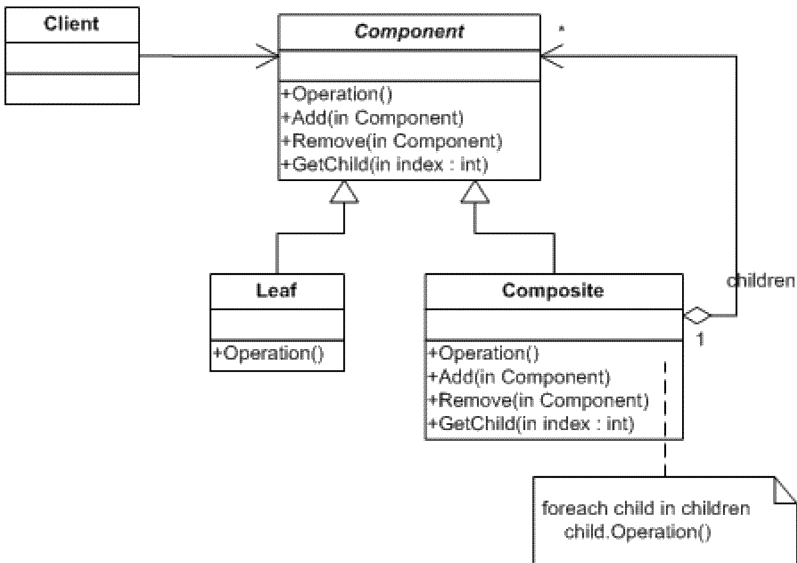


Рисунок 2.4 – Структура шаблону Composite

Відносини. Ключем до шаблону компонування є абстрактний клас, який є одночасно і примітивом, і контейнером (*Component*). У ньому оголошені методи, специфічні для кожного виду об'єкта (такі як *Operation*) і загальні для всіх складових об'єктів, наприклад операції для доступу і управління нащадками. Підкласи *Leaf* визначають примітивні об'єкти, які не є контейнерами. У них операція *Operation* реалізована відповідно до їх специфічних потреб. Оскільки у примітивних об'єктів немає нащадків, то жоден з цих підкласів не

реалізує операції, пов'язані з управлінням нащадками (*Add*, *Remove*, *GetChild*). Клас *Composite* складається з інших примітивніших об'єктів *Component*. Реалізована в ньому операція *Operation* викликає однойменну функцію відтворення для кожного нащадка, а операції для роботи з нащадками вже не порожні. Оскільки інтерфейс класу *Composite* відповідає інтерфейсу *Component*, то до складу об'єкта *Composite* можуть входити і інші такі ж об'єкти.

2.2.5 Шаблон проектування Міст (Bridge)

Призначення. Відокремити абстракцію від її реалізації таким чином, щоб перше та друге можна було змінювати незалежно одне від одного. Якщо для деякої абстракції можливо кілька реалізацій, зазвичай застосовують спадкування. Абстрактний клас визначає інтерфейс абстракції, а його конкретні підкласи по-різному реалізують його. Але такий підхід не завжди є достатньо гнучким. Спадкування жорстко прив'язує реалізацію до абстракції, що перешкоджає незалежній модифікації, розширенню та повторному використанню абстракції та її реалізації.

Застосування. Використовується коли:

- треба запобігти постійній прив'язці абстракції до реалізації (наприклад, коли реалізацію необхідно обрати під час виконання програми);
- як абстракції, так і реалізації повинні розширюватись новими підкласами (у цьому разі шаблон Міст дозволяє комбінувати різні абстракції та реалізації та змінювати їх незалежно одне від одного);
- зміни у реалізації не повинні впливати на клієнтів, тобто клієнтський код не повинен перекомпілюватись;
- треба повністю сховати від клієнтів реалізацію абстракції;
- треба розподілити одну реалізацію поміж кількох об'єктів (можливо застосовуючи підрахунок посилань), і при цьому приховати це від клієнту.

Структура шаблону наведена на рис. 2.5.

Учасниками шаблону є:

- **Abstraction** – абстракція:
 - визначає інтерфейс абстракції;
 - зберігає посилання на об'єкт типу *Implementor*;
- **RefinedAbstraction** – уточнена абстракція:
 - розширює інтерфейс, означений абстракцією *Abstraction*;

- **Implementor** – реалізатор:
 - визначає інтерфейс для класів реалізації;
- **ConcreteImplementor** – конкретний реалізатор:
 - містить конкретну реалізацію інтерфейсу класу *Implementor*.

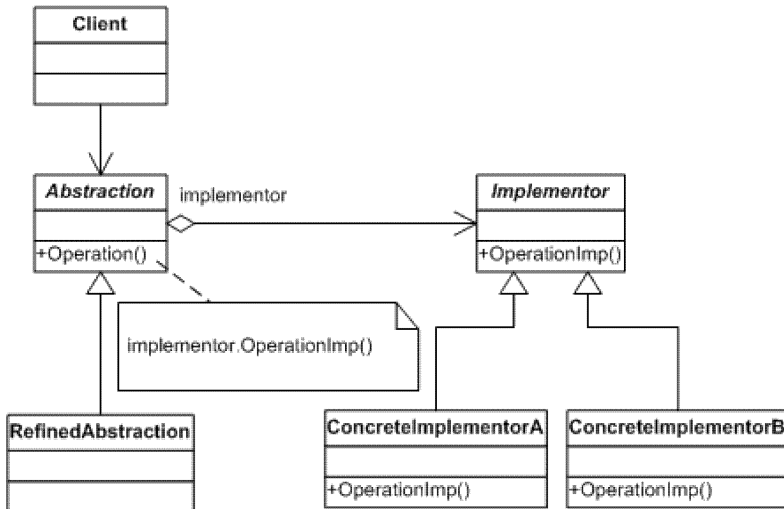


Рисунок 2.5 – Структура шаблону Bridge

Інтерфейс класів реалізації не зобов'язаний точно відповідати інтерфейсу класу *Abstraction*. Насправді обидва інтерфейси можуть бути зовсім різними. Зазвичай, інтерфейс класу *Implementor* надає тільки примітивні операції, а клас *Abstraction* визначає операції більш високого рівня, що базуються на цих примітивах

Відносини. Об'єкт *Abstraction* містить у собі *Implementor* і перенаправляє йому запити клієнта.

Результати. Відокремлення реалізації від інтерфейсу дає можливість конфігурації під час виконання *Implementor* та *Abstraction*. Крім того, слід зазначити, що розподіл класів *Abstraction* і *Implementor* усуває залежності від реалізації, що встановлюються на етапі компіляції: щоб змінити клас *Implementor* зовсім не обов'язково перекompілювати клас *Abstraction*.

2.2.6 Шаблон проектування Легковаговик (Flyweight)

Призначення. Використовується для ефективної підтримки (в першу чергу для зменшення затрат пам'яті) великої кількості дрібних об'єктів. Шаблон Легковаговик (або пристосуванець) використовує загальнодоступний легкий об'єкт (flyweight), який одночасно може використовуватися у великій кількості контекстів. Стан цього об'єкта поділяється на внутрішній, що містить інформацію, незалежну від контексту, і зовнішній, який залежить або змінюється разом з контекстом легковаговика. Об'єкти клієнтів відповідають за передачу зовнішнього стану легковаговика, коли йому це необхідно.

Застосування. Використовується коли:

- в програмі використовується велика кількість об'єктів;
- витрати на збереження високі через велику кількість об'єктів;
- більшість станів об'єктів можна зробити зовнішніми;
- велика кількість груп об'єктів може бути замінена відносно малою кількістю загальнодоступних об'єктів, однократно видаливши зовнішній стан;
- програма не залежить від ідентичності об'єктів.

Оскільки об'єкти-легковаговики можуть використовуватися колективно, то тести на ідентичність будуть повертати значення "істина" ("true") для концептуально різних об'єктів.

Структура шаблону наведена на рис. 2.6.

Учасниками шаблону є:

- **Flyweight** – пристосуванець:
 - оголошує інтерфейс, за допомогою якого пристосуванці можуть отримати зовнішній стан або якимось впливати на нього;
- **ConcreteFlyweight** – конкретний пристосуванець:
 - реалізує інтерфейс класу *Flyweight* і додає за необхідності внутрішній стан;
 - об'єкт класу *ConcreteFlyweight* розділяється. Будь-який стан, що зберігається у ньому, має бути внутрішнім, тобто незалежним від контексту;
- **UnsharedConcreteFlyweight** – нероздільний конкретний пристосуванець:
 - не всі підкласи *Flyweight* обов'язково розділяються;
- **FlyweightFactory** – фабрика пристосуванців:

- створює об'єкти – *Flyweight* (або надає примірник, що вже існує) та керує ними.
- забезпечує необхідне розділення пристосувань;
- **Client** – клієнт:
 - зберігає посилання на одного або кількох *Flyweight*;
 - обчислює або зберігає зовнішній стан *Flyweight*.

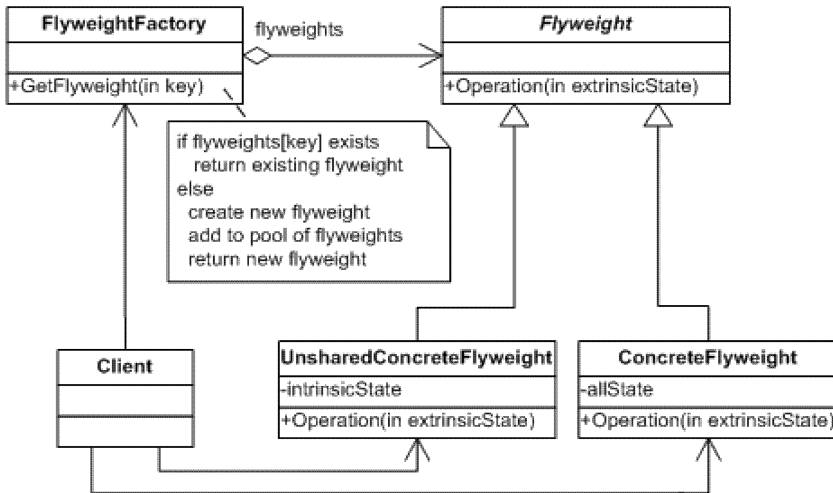


Рисунок 2.6 – Структура шаблону Flyweight

Відносини. Стан, необхідний пристосуванцю для нормальної роботи можна охарактеризувати як внутрішній або зовнішній.

Внутрішній стан зберігається в об'єкті **ConcreteFlyweight**, в той час як зовнішній стан зберігається або обчислюється в **Client**. **Client** передає його **Flyweight** при виклику операцій.

Клієнти не повинні створювати екземпляри класу **ConcreteFlyweight** напряму, а можуть отримати їх тільки від об'єкту **FlyweightFactory**. Це дозволяє гарантувати коректне розділення.

Результати. Пристосуванці моделюють сутності, кількість яких занадто велика для подання об'єктами.

Крім того, внаслідок зменшення загального числа екземплярів і винесення стану економиться пам'ять.

Шаблон **Flyweight** доповнює шаблон **Factory** таким чином, що **Factory** при зверненні до неї клієнту для створення нового об'єкту шукає вже створений об'єкт з такими параметрами, що потребуються,

та повертає його клієнту. Якщо такого об'єкту немає, то фабрика створить новий.

2.2.7 Шаблон проектування Фасад (Facade)

Призначення. Об'єднання групи підсистем під один уніфікований інтерфейс, надаючи доступ до них через одну точку входу. Це дозволяє спростити роботу з підсистемами. Крім того цей шаблон дозволяє забезпечити уніфікований інтерфейс з набором розрізних реалізацій або інтерфейсів, наприклад, з підсистемою, якщо небажаним є високе зв'язування з цією підсистемою або реалізація підсистеми може змінитися.

Застосування. Використовується щоб:

- спростити доступ до складної системи;
- створити рівні доступу до системи;
- додати стійкість до змін підсистем;
- зменшити кількість сильних зв'язків між клієнтом та підсистемою, але залишити доступ до повної функціональності.

Структура шаблону наведена на рис. 2.7.

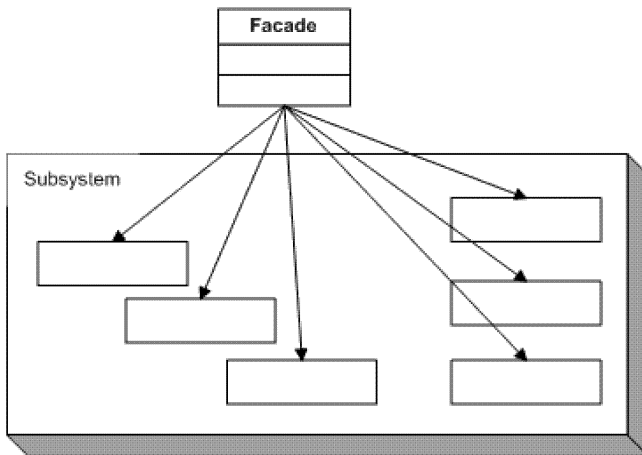


Рисунок 2.7 – Структура шаблону Facade

В цьому шаблоні визначається одна точка взаємодії з підсистемами – фасадний об'єкт.

Учасниками шаблону є:

- **Facade** – фасад;

- має відомості, яким класам підсистеми адресувати запит;
- делегує запити клієнтів відповідним об'єктам всередині підсистеми;
- **Класи підсистеми:**
 - реалізують функціональність підсистеми;
 - виконують роботу, що доручено об'єктом Facade;
 - нічого не знають про існування фасаду, тобто не зберігають посилань на нього.

Фасад:

- створює нові методи, котрі об'єднують виклики об'єктів системи і/або додають свою логіку;
- приховує підсистеми;
- зменшує кількість параметрів методів, шляхом попередньої підстановки визначених значень.

Реалізація інших компонентів підсистеми закрита і не доступна для зовнішніх компонентів. Фасадний об'єкт забезпечує реалізацію шаблону "Стійкий до змін" з точки зору захисту від змін у реалізації підсистеми.

Відносини. Клієнти спілкуються з підсистемою, надсилаючи запити фасаду. Він переадресує їх відповідним об'єктам всередині підсистеми. Хоч основну роботу виконуються об'єкти підсистеми, фасаду, можливо, доведеться перетворити свій інтерфейс на інтерфейси підсистеми.

Результати. Шаблон фасад ізолює клієнтів від компонентів підсистеми, зменшуючи тим самим число об'єктів, з якими клієнтам необхідно мати справу, та спрощує роботу з підсистемою.

Фасад також дозволяє послабити зв'язаність між підсистемою та її клієнтами. При цьому фасад не заважає додаткам напяму звертатися до класів підсистеми, якщо це необхідно.

2.3 Завдання на лабораторну роботу

2.3.1 Вивчити структурні шаблони проектування, їхню загальну характеристику та призначення. Для кожного з цих шаблонів:

- вивчити призначення шаблону, альтернативні назви, призначення, випадки коли його застосування є доцільним та результати такого застосування;
- знати особливості реалізації шаблону, споріднені шаблони,

відомі випадки його застосування в програмних додатках;

– вільно володіти структурою шаблону, призначенням його класів та відносинами між ними.

2.3.2 За допомогою джерел зі списку рекомендованих для виконання лабораторної роботи, виконати програмну реалізацію структурних шаблонів.

2.4 Зміст звіту

2.4.1 Тема та мета роботи.

2.4.2 Короткі теоретичні відомості.

2.4.3 Результати роботи.

2.4.4 Висновки, що відображують результати виконання роботи та їх критичний аналіз.

2.5 Контрольні запитання

2.5.1 Призначення структурних шаблонів проектування ПЗ.

2.5.2 Коротка характеристика кожного структурного шаблону.

2.5.3 Назви, призначення та мотивація шаблону Composite.

2.5.4 Структура шаблону Composite та його учасники.

2.5.5 Особливості реалізації шаблону Composite. Результат використання шаблону.

2.5.6 Назви, призначення та мотивація шаблону Decorator.

2.5.7 Структура шаблону Decorator та його учасники.

2.5.8 Особливості реалізації шаблону Decorator. Результат використання шаблону.

2.5.9 Назви, призначення та мотивація шаблону Proxy.

2.5.10 Структура шаблону Proxy та його учасники.

2.5.11 Особливості реалізації шаблону Proxy. Результат використання шаблону.

2.5.12 Види шаблону Proxy.

2.5.13 Шаблони, які використовуються сумісно з Composite, Decorator та Proxy.

2.5.14 Відмінність Decorator та Proxy в специфікації конструктора.

2.5.15 Назви, призначення та мотивація шаблону Flyweight.

2.5.16 Структура шаблону Flyweight та його учасники.

2.5.17 Особливості реалізації шаблону Flyweight. Результат

використання шаблону.

2.5.18 Назви, призначення та мотивація шаблону Adapter.

2.5.19 Структура шаблону Adapter та його учасники.

2.5.20 Особливості реалізації шаблону Adapter. Результат використання шаблону.

2.5.21 Назви, призначення та мотивація шаблону Bridge.

2.5.22 Структура шаблону Bridge та його учасники.

2.5.23 Особливості реалізації шаблону Bridge. Результат використання шаблону.

2.5.24 Назви, призначення та мотивація шаблону Facade.

2.5.25 Структура шаблону Facade та його учасники.

2.5.26 Особливості реалізації шаблону Facade. Результат використання шаблону.

2.5.27 Відмінність Adapter, Decorator та Proxu в специфікації конструктора.

2.5.28 Види адаптерів. Двосторонній та динамічний (pluggable) адаптери.

2.5.29 Шаблони, які використовуються сумісно з Flyweight, Adapter, Bridge, Facade.

ЛАБОРАТОРНА РОБОТА № 3 ШАБЛони ПОВЕДІНКИ. ЧАСТИНА 1

3.1 Мета роботи

Вивчити шаблони поведінки та отримати базові навички із застосування шаблонів Ітератор (Iterator), Посередник (Mediator), Спостерігач (Observer), Стратегія (Strategy), Ланцюг обов'язків (Chain of Responsibility), Відвідувач (Visitor).

3.2 Основні теоретичні відомості

3.2.1 Шаблон проектування Ітератор (Iterator)

Призначення. Надає спосіб послідовного доступу до всіх елементів складеного об'єкта, не розкриваючи його внутрішнього улаштування.

Складений об'єкт, скажімо список, дерево, хеш-таблиця або масив, повинен надавати спосіб доступу до своїх елементів, не розкриваючи їхню внутрішню структуру. Більш того, іноді треба обходити список по-різному, у залежності від задачі, що вирішується. При цьому немає ніякого бажання засмічувати інтерфейс класу Список усілякими операціями для усіх потрібних варіантів обходу, навіть якщо їх усі можна передбачити заздалегідь. Крім того, іноді треба, щоб в один момент часу існувало декілька активних операцій обходу списку.

Все це призводить до необхідності реалізації шаблону Ітератор. Його основна ідея у тому, щоб за доступ до елементів та обхід списку відповідав не сам список, а окремий об'єкт-ітератор.

У класі Ітератор означений інтерфейс для доступу до елементів списку. Об'єкт цього класу прослідковує поточний елемент, тобто він володіє інформацією, які з елементів вже відвідувались.

Також відомий під назвою курсор (Cursor).

Застосування. Використовується:

- для доступу до змісту агрегованих об'єктів не розкриваючи їхнє внутрішнє улаштування;
- для підтримки декількох активних обходів одного й того ж агрегованого об'єкта;
- для подання уніфікованого інтерфейсу з метою обходу різноманітних агрегованих структур (тобто для підтримки

поліморфної ітерації).

Структура шаблону наведена на рис. 3.1.

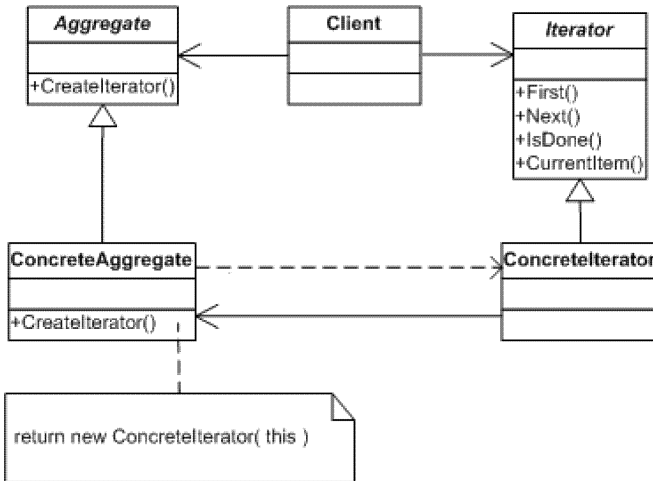


Рисунок 3.1 – Структура шаблону Iterator

Учасниками шаблону є:

- **Iterator** – ітератор:
 - визначає інтерфейс для доступу та обходу елементів
- **ConcreteIterator**:
 - реалізує інтерфейс класу **Iterator**;
 - слідкує за поточною позицією під час обходу агрегату;
- **Aggregate**:
 - визначає інтерфейс для створення об'єкта-ітератора;
- **ConcreteAggregate** :
 - реалізує інтерфейс створення ітератора та повертає екземпляр відповідного класу **ConcreteIterator**

Відносини. *ConcreteIterator* відслідковує поточний об'єкт у агрегаті та може вирахувати наступний.

Результати. Шаблон підтримує різні способи перебору агрегату, одночасно можуть бути активні декілька переборів.

3.2.2 Шаблон проектування Посередник (Mediator)

Призначення. Визначає об'єкт, що інкапсулює спосіб взаємодії множини об'єктів. Посередник забезпечує слабку зв'язаність системи,

звільняючи об'єкти від необхідності явно посилатися один на одного, і дозволяючи тим самим незалежно змінювати взаємодії між ними.

Застосування. Використовується коли:

- існують об'єкти, зв'язки між котрими досить складні та чітко задані. Отримані при цьому залежності не структуровані та важкі для розуміння;
- не можна повторно використовувати об'єкт, оскільки він обмінюється інформацією з багатьма іншими об'єктами;
- поведінка, розподілена між кількома класами, повинна піддаватися налагодженню без створювання множини підкласів.

Структура шаблону наведена на рис. 3.2.

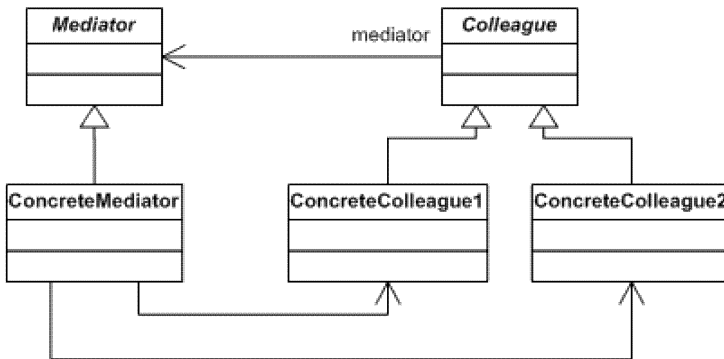


Рисунок 3.2 – Структура шаблону Mediator

Учасниками шаблону є:

- **Mediator** – посередник:
 - визначає інтерфейс для обміну інформацією з об'єктами Colleague;
- **ConcreteMediator** – конкретний посередник:
 - реалізує кооперативну поведінку, координуючи дії об'єктів Colleague;
 - володіє інформацією про колег, та підраховує їх;
- Класи **Colleague** – колеги:
 - кожному класу Colleague відомо про свій об'єкт Mediator;
 - усі колеги обмінюються інформацією виключно через посередника, інакше за його відсутності їм довелося б спілкуватися між собою напряму.

Відносини. Колеги посилають запити посередникові та

отримують запити від нього. *Посередник* реалізує кооперативну поведінку шляхом переадресації кожного запиту відповідному колезі (або декільком з них).

Результат. Шаблон знижує кількість класів, що породжуються, усуває зв'язаність між колегами, спрощує протоколи взаємодії об'єктів, абстрагує спосіб кооперації об'єктів та централізує управління.

3.2.3 Шаблон проектування Спостерігач (Observer)

Призначення. Визначає залежність типу “один до багатьох” між об'єктами таким чином, що при зміні стану одного об'єкта всі залежних від нього сповіщаються про цю подію.

Також відомий як “підлегли” (Dependents) або “видавець-передплатник” (Publisher-Subscriber).

Застосування. Використовується коли:

- існує, як мінімум, один об'єкт, що розсилає повідомлення і є не менше одного одержувача повідомлень, причому їхня кількість і склад можуть змінюватися під час роботи програми;
- коли при модифікації одного об'єкту необхідно змінити інші і не відомо, скільки таких об'єктів необхідно змінити;
- коли один об'єкт повинен сповіщати інших, не роблячи припущень про об'єкти, що сповіщаються, іншими словами потрібно, щоб об'єкти не були тісно зв'язані між собою.

Цей шаблон часто застосовують в ситуаціях, в яких відправника повідомлень не цікавить, що роблять одержувачі з наданою їм інформацією.

Структура шаблону наведена на рис. 3.3.

Учасниками шаблону є:

- **Subject** – суб'єкт:
 - має інформацію про своїх спостерігачів, за суб'єктом можуть «слідкувати» будь-яка кількість спостерігачів;
 - надає інтерфейс, що визначає методи для додавання, видалення та оповіщення спостерігачів;
- **Observer** – спостерігач:
 - визначає інтерфейс для оновлення об'єктів, які повинні бути сповіщені про зміну суб'єкту;
- **ConcreteSubject** – конкретний суб'єкт:
 - зберігає стан, що становить інтерес для конкретного

- спостерігача *ConcreteObserver*;
- надсилає інформацію своїм спостерігачам, коли відбувається зміна;
- **ConcreteObserver** – конкретний спостерігач:
 - зберігає посилання на об'єкт класу *ConcreteSubject*;
 - зберігає дані, які повинні бути узгоджені з даними суб'єкту;
 - реалізує інтерфейс оновлення, визначений в класі *Observer*, для підтримки узгодженості з суб'єктом.

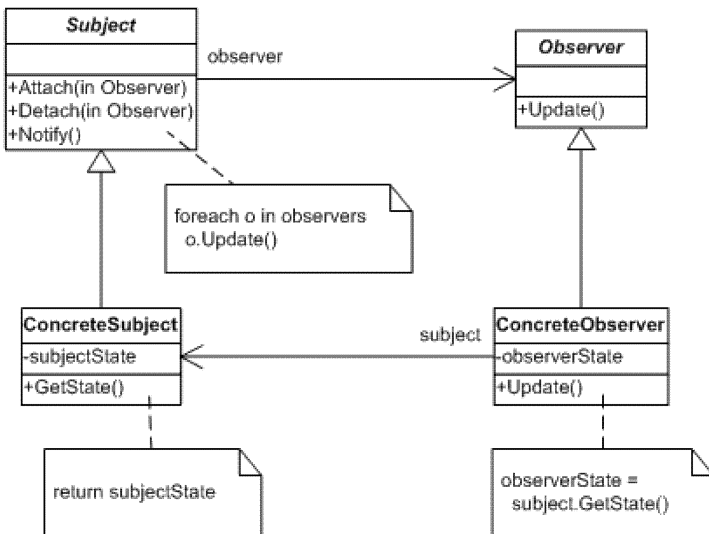


Рисунок 3.3 – Структура шаблону Observer

При зміні спостережуваного об'єкту, оповіщення спостерігачів може бути реалізоване за такими сценаріями:

- спостережуваний об'єкт надсилає, кожному із зареєстрованих спостерігачів, всю потенційно релевантну інформацію (примусове розповсюдження);
- спостережуваний об'єкт надсилає, кожному із зареєстрованих спостерігачів, лише повідомлення про те що інформація була змінена, а кожен із спостерігачів, за необхідності, самостійно здійснює запит необхідної інформації у спостережуваного об'єкта (розповсюдження за запитом).

Відносини. Об'єкт *ConcreteSubject* оповіщає своїх спостерігачів про будь-яку зміну, що могла би привести до неузгодженості станів спостерігача та суб'єкта.

Отримавши від конкретного суб'єкта сповіщення про зміну, об'єкт *ConcreteObserver* може запитати у суб'єкта додаткову інформацію, яку використає для того, щоб опинитися в стані, узгодженому зі станом суб'єкту.

Об'єкт *Observer*, що ініціює запит на зміну, відкладає своє оновлення до отримання сповіщення від суб'єкту. Операція *Notify* не завжди викликається суб'єктом. Її може викликати і спостерігач і сторонній об'єкт.

Результат. Шаблон дозволяє змінювати суб'єкти та спостерігачі незалежно один від одного. Суб'єкти дозволяється повторно використовувати без участі спостерігачів та навпаки. Це надає можливість додавати нових спостерігачів без модифікації суб'єкта чи інших спостерігачів.

3.2.4 Шаблон проектування Стратегія (Strategy)

Призначення. Суть шаблону полягає у тому, щоб створити декілька схем поведінки для одного об'єкту та винести в окремий клас.

Відомий ще під назвою політика (Policy).

Існують системи, поведінка яких визначається відповідно до певного роду алгоритмів. Всі вони подібні між собою: призначені для вирішення спільних задач, мають однаковий інтерфейс для користування, але відрізняються тільки "поведінкою", тобто реалізацією. Користувач, налаштувавши програму на потрібний алгоритм, отримує потрібний результат.

Застосування. Використовується коли:

- є багато споріднених класів, що відрізняються лише поведінкою (стратегія дозволяє зконфігурувати клас, задавши одну з можливих поведінок);
- необхідно мати декілька варіантів алгоритму (наприклад, можна визначити два варіанти алгоритму, один з яких потребує більше часу, а інший – більше пам'яті; стратегії дозволяється використовувати, коли варіанти алгоритмів реалізовані у вигляді ієрархій класів);
- в алгоритмі містяться складні специфічні дані, про які клієнт не повинний знати;

– в класі визначено багато поведінок, що представлені у вигляді розгалужених умовних операторів.

Структура шаблону наведена на рис. 3.4.

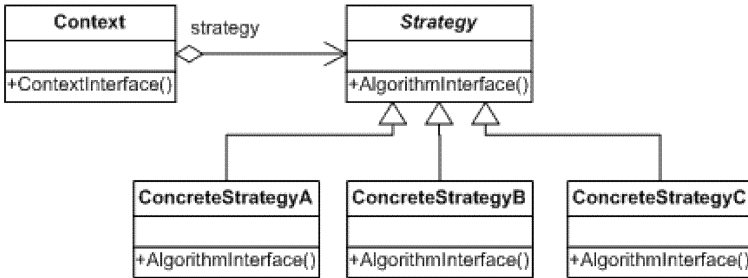


Рисунок 3.4 – Структура шаблону Strategy

Учасниками шаблону є:

- **Strategy** – стратегія:
 - визначає загальний інтерфейс для всіх алгоритмів, що підтримуються. Клас *Context* користується цим інтерфейсом для виклику конкретного алгоритму, визначеного в класі *ConcreteStrategy*;
- **ConcreteStrategy** – конкретна стратегія:
 - реалізує алгоритм, що використовує інтерфейс, визначений в класі *Strategy*;
- **Context** – контекст:
 - конфігурується об'єктом класу *ConcreteStrategy*;
 - зберігає посилання на об'єкт класу *Strategy*;
 - може визначати інтерфейс, що дозволяє об'єкту *Strategy* отримати доступ до даних контексту.

Створюється декілька класів *ConcreteStrategy*, кожен з яких містить один і той же поліморфний метод *AlgorithmInterface*. Об'єкт стратегії зв'язується з контекстним об'єктом (тим об'єктом, до якого застосовується алгоритм).

Відносини. Класи *Strategy* та *Context* взаємодіють для реалізації обраного алгоритму. Контекст може передати до стратегії всі необхідні алгоритму дані під час його виклику. Замість цього контекст може дозволити звертатися до своїх операцій в необхідні моменти, передавши посилання на самого себе операціям класу *Strategy*.

Контекст переадресує запити своїх клієнтів до об'єкту-стратегії.

Зазвичай клієнт створює об'єкт *ConcreteStrategy* та передає його контексту, після чого клієнт «спілкується» виключно з контекстом. Часто в розпорядженні клієнта знаходиться декілька класів *ConcreteStrategy*, які він може обирати.

І Стратегія, і Декоратор можуть використовуватися для зміни поведінки конкретних класів. Переваги Стратегії полягають в тому, що інтерфейс кастомізації не співпадає з публічним інтерфейсом та може бути більш зручним, а недолік в тому, що необхідно з самого початку проектувати клас з можливістю реєстрації стратегій.

3.2.5 Шаблон проектування Ланцюг обов'язків (Chain of Responsibility)

Призначений для організації в системі рівнів відповідальності, коли запит повинен бути оброблений декількома об'єктами.

Застосування. Використовується коли:

- є більш ніж один об'єкт, здатний обробити запит, при цьому реальний обробник заздалегідь не відомий і повинен бути знайдений автоматично;
- необхідно відправити запит одному з декількох об'єктів, не вказуючи явно одержувача;
- набір об'єктів, що можуть обробити запит повинен задаватися динамічно під час виконання програми.

Структура шаблону наведена на рис. 3.5.

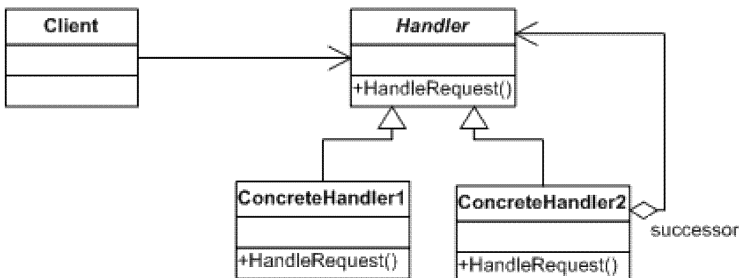


Рисунок 3.5 – Структура шаблону Chain of Responsibility

Учасниками шаблону є:

- **Handler** – обробник:
 - визначає інтерфейс для обробки запитів;
 - (необов'язково) реалізує зв'язок з приймачем;

- **ConcreteHandler** – конкретний обробник:
 - обробляє запит, за який відповідає;
 - має доступ до свого приймача;
 - якщо *ConcreteHandler* здатен обробити запит, то так і робить, якщо ні, то відправляє його своєму приймачу;
- **Client** – клієнт:
 - відправляє запит деякому об'єкту *ConcreteHandler* в ціпочці.

Відносини. Коли клієнт ініціює запит, він просувається по ціпочці доти, поки деякий об'єкт *ConcreteHandler* не візьме на себе відповідальність за його обробку.

Результат. Шаблон послаблює зв'язаність (об'єкт не зобов'язаний "знати" про існування всіх інших об'єктів та про те, хто саме опрацює його запит) та дозволяє підвищити гнучкість при розподілі обов'язків між об'єктами. Але немає гарантій, що запит буде оброблений, оскільки він не має явного одержувача.

3.2.6 Шаблон проектування Відвідувач (Visitor)

Призначення. дозволяє відділити певний алгоритм від елементів, на яких алгоритм має бути виконаний, таким чином можливо легко додати або ж змінити алгоритм без змін щодо елементів системи. Практичним результатом є можливість додавання нових операцій в існуючі структури об'єкта без зміни цих структур.

Відвідувач дозволяє додавати нові віртуальні функції в родинні класи без зміни самих класів, натомість, один відвідувач створює клас, який реалізує всі відповідні спеціалізації віртуальної функції. Відвідувач бере приклад посилення в якості вхідних даних і реалізується шляхом подвійної диспетчеризації.

Застосування. Використовується коли:

- в структурі наявні об'єкти декількох класів з різними інтерфейсами і необхідно виконувати над ними операції, що залежать від конкретних класів;
- над об'єктами, що входять в состав структури, необхідно виконувати різноманітні, не зв'язані між собою операції та не бажано «ускладнювати» класи цими операціями;
- класи, що устанавлюють структуру об'єктів, змінюються рідко, але часто додаються нові операції над структурою.

Структура шаблону наведена на рис. 3.6.

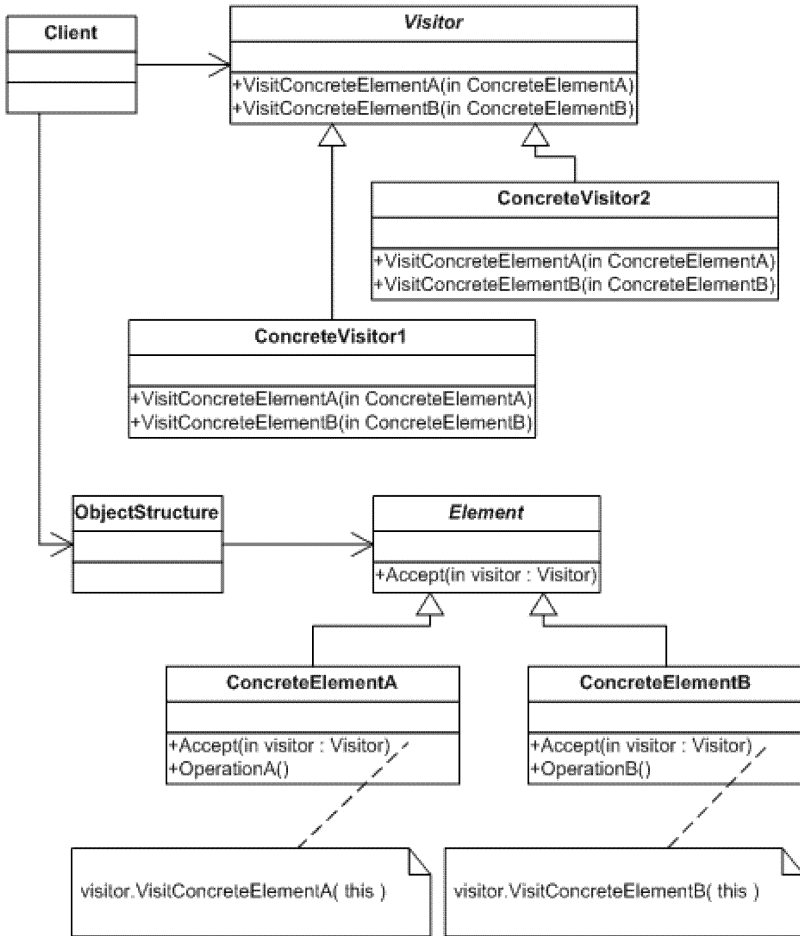


Рисунок 3.6 – Структура шаблону Visitor

Учасниками шаблону є:

– **Visitor** – відвідувач:

- оголошує операцію *Visit* для кожного класу *ConcreteElement* в структурі об'єктів. Ім'я та сигнатура цієї операції ідентифікують клас, який посилає відвідувачу запит *Visit*. Це дозволяє відвідувачу визначити елемент якого конкретного класу він відвідує. Маючи цю інформацію, відвідувач може звертатися к елементу

- безпосередньо через його інтерфейс;
- **ConcreteVisitor**– конкретний відвідувач:
 - реалізує всі операції, оголошені в класі *Visitor*. Кожна операція реалізує фрагмент алгоритму, визначеного для класу відповідного об'єкту в структурі. Клас *ConcreteVisitor* надає контекст для цього алгоритму та зберігає його локальний стан. Часто в цьому стані акумулюються результати, отримані в процесі обходу структури;
 - **Element** – елемент:
 - визначає операцію *Accept*, яка приймає відвідувача в якості аргументу
 - **ConcreteElement**– конкретний елемент:
 - реалізує операцію *Accept*;
 - **ObjectStructure** – структура об'єктів:
 - може перелічувати свої елементи;
 - може надавати відвідувачу високорівневий інтерфейс для відвідування власних елементів;
 - може бути як складеним об'єктом (компонувальник), так і колекцією, наприклад, списком або множиною.

Відносини. Клієнт, який використовує даний шаблон, повинен створити об'єкт класу *ConcreteVisitor*, а потім відвідати кожен елемент структури. *Visitor* оголошує операцію *VisitConcreteElement* для кожного класу *ConcreteElement* (ім'я та сигнатура даної операції ідентифікують клас, елемент якого відвідує *Visitor* – тобто, відвідувач може звертатися до елементу безпосередньо). *ConcreteVisitor* реалізує всі операції, оголошені в класі *Visitor*.

Кожна операція реалізує фрагмент алгоритму, визначеного для класу відповідного об'єкта в структурі. Клас *ConcreteVisitor* надає контекст для цього алгоритму і зберігає його локальний стан. *Element* визначає операцію *Accept*, яка приймає *Visitor* як аргумент, *ConcreteElement* реалізує операцію *Accept*, яка приймає *Visitor* як аргумент. *ObjectStructure* може перерахувати свої аргументи і надати відвідувачеві високорівневий інтерфейс для відвідування своїх елементів.

Логічним є використання цього шаблону, якщо в структурі присутні об'єкти багатьох класів з різними інтерфейсами, і необхідно виконати над ними операції, що залежать від конкретних класів, або

якщо класи, що визначають структуру об'єктів змінюються рідко, але нові операції над цією структурою додаються часто.

Результати. Шаблон спрощує додавання нових операцій, об'єднує споріднені операції в класі *Visitor*. При цьому ускладнюється додавання нових класів *ConcreteElement*, оскільки потрібно оголошення нової абстрактної операції в класі *Visitor*.

Сукупна структура об'єктів елементу може визначатися за допомогою шаблону Компонувальник (Composite). Для обходу може використовуватися Ітератор (Iterator).

Шаблон Відвідувач демонструє класичний прийом відновлення інформації про втрачені типи, не вдаючись до понижуючого приведення типів (динамічне приведення).

3.3 Завдання на лабораторну роботу

3.3.1 Вивчити шаблони поведінки, їхню загальну характеристику та призначення. Для кожного з цих шаблонів:

- вивчити призначення шаблону, альтернативні назви, призначення, випадки коли його застосування є доцільним та результати такого застосування;
- знати особливості реалізації шаблону, споріднені шаблони, відомі випадки його застосування в програмних додатках;
- вільно володіти структурою шаблону, призначенням його класів та відносинами між ними.

3.3.2 За допомогою джерел зі списку рекомендованих для виконання лабораторної роботи, виконати програмну реалізацію шаблонів поведінки.

3.4 Зміст звіту

3.4.1 Тема та мета роботи.

3.4.2 Короткі теоретичні відомості.

3.4.3 Результати роботи.

3.4.4 Висновки, що відображують результати виконання роботи та їх критичний аналіз.

3.5 Контрольні запитання

3.5.1 Призначення шаблонів поведінки для проектування ПЗ.

3.5.2 Коротка характеристика кожного розглянутого шаблону

поведінки.

3.5.3 Назви, призначення та мотивація шаблону Iterator.

3.5.4 Структура шаблону Iterator та його учасники.

3.5.5 Особливості реалізації шаблону Iterator. Результат використання шаблону.

3.5.6 Назви, призначення та мотивація шаблону Mediator.

3.5.7 Структура шаблону Mediator та його учасники.

3.5.8 Особливості реалізації шаблону Mediator. Результат використання шаблону.

3.5.9 Назви, призначення та мотивація шаблону Observer.

3.5.10 Структура шаблону Observer та його учасники.

3.5.11 Особливості реалізації шаблону Observer. Результат використання шаблону.

3.5.12 Шаблони, які використовуються сумісно з Iterator, Mediator та Observer.

3.5.13 Назви, призначення та мотивація шаблону Strategy.

3.5.14 Структура шаблону Strategy та його учасники.

3.5.15 Особливості реалізації шаблону Strategy. Результат використання шаблону.

3.5.16 Назви, призначення та мотивація шаблону Chain of Responsibility.

3.5.17 Структура шаблону Chain of Responsibility та його учасники.

3.5.18 Особливості реалізації шаблону Chain of Responsibility. Результат використання шаблону.

3.5.19 Назви, призначення та мотивація шаблону Visitor.

3.5.20 Структура шаблону Visitor та його учасники.

3.5.21 Особливості реалізації шаблону Visitor. Результат використання шаблону.

3.5.22 Шаблони, які використовуються сумісно з Strategy, Chain of Responsibility та Visitor.

ЛАБОРАТОРНА РОБОТА № 4 ШАБЛони ПОВЕДІНКИ. ЧАСТИНА 2

4.1 Мета роботи

Вивчити шаблони поведінки та отримати базові навички із застосування шаблонів Знімок (Memento), Стан (State), Команда (Command), Інтерпретатор (Interpreter), Шаблонний метод (Template method).

4.2 Основні теоретичні відомості

4.2.1 Шаблон проектування Знімок (Memento)

Призначення. Не порушуючи інкапсуляції, фіксує та виносить за межі об'єкта його внутрішній стан так, щоб пізніше можна було відновити з нього об'єкт.

Також відомий під назвою лексема (Token).

Застосування. Використовується коли:

- необхідно зберегти миттєвий знімок стану об'єкта (або його частини), щоб згодом його можна було відтворити у цьому стані;
- безпосереднє вилучення цього стану розкриває деталі реалізації та порушує інкапсуляцію об'єкта.

Структура шаблону наведена на рис. 4.1.

Учасниками шаблону є:

- **Memento** – контекст (зберігач):
 - зберігає внутрішній стан об'єкта *Originator*. Обсяг інформації, що зберігається, може бути різним та визначається потребами хазяїна;
 - забороняє доступ усім іншим об'єктам окрім хазяїна;
- **Originator** – хазяїн:
 - створює знімок, що утримує поточний внутрішній стан;
 - використовує знімок для відтворення внутрішнього стану;
- **CareTaker** – опікун (посильний):
 - відповідає за зберігання знімку;
 - не проводить жодних операцій над знімком та не має уяви про його внутрішній зміст.

По суті знімок має два інтерфейси. Опікун *Caretaker* користується лише вузьким інтерфейсом знімку – він може лише передавати знімок іншим об'єктам. Напроти, хазяїн користується

широким інтерфейсом, котрий забезпечує доступ до всіх даних, необхідних для відтворення об'єкта (чи його частини) у попередньому стані. Ідеальний варіант – коли тільки хазяїну, що створив знімок, відкритий доступ до внутрішнього стану знімку

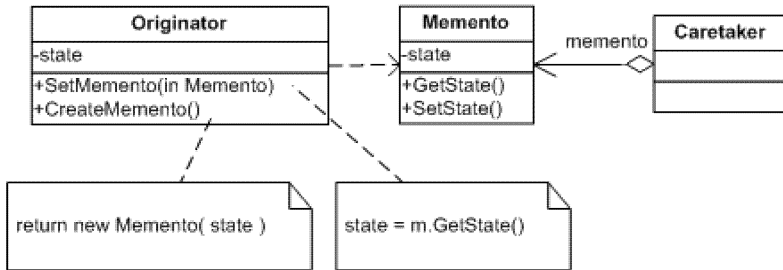


Рисунок 4.1 – Структура шаблону Memento

Відносини. Опікун запитує знімок у хазяїна, деякий час тримає його у себе, опісля повертає хазяїну. Іноді цього не відбувається, бо хазяїн не має необхідності відтворювати свій попередній стан;

Знімки пасивні. Тільки хазяїн, що створив знімок, має доступ до інформації про стан.

Результати. При використанні шаблону не розкривається інформація, яка доступна тільки *Originator*, спрощується структура *Originator*. Але з використанням *Memento* можуть бути пов'язані значні витрати, якщо *Originator* повинен копіювати великий обсяг інформації, або якщо копіювання повинно проводитися часто.

4.2.2 Шаблон проектування Стан (State)

Призначення. Дозволяє об'єктові варіювати свою поведінку у залежності від внутрішнього стану. Ззовні здається, що змінився клас об'єкта.

Застосування. Використовується коли:

- поведінка об'єкта залежить від його стану та повинно змінюватись під час виконання програми;
- у коді операцій зустрічаються умовні оператори, що складаються з багатьох частин, у котрих вибір гілки залежить від стану. Зазвичай у такому разі стан представлено константами, що перелічуються. Часто одна й та ж структура умовного оператора повторюється у декількох операціях. Шаблон Стан пропонує

помістити кожну гілку у окремий клас. Це дозволить трактувати стан об'єкта як самостійний об'єкт, котрий можна змінитися незалежно від інших.

Структура шаблону наведена на рис. 4.2.

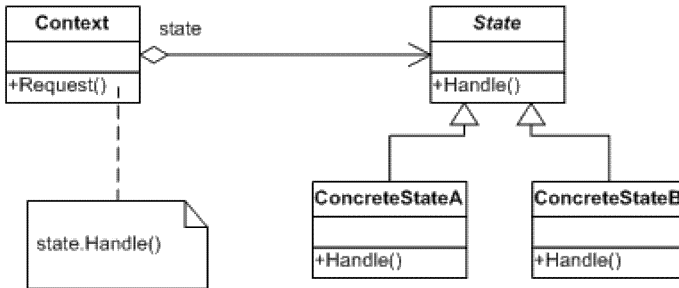


Рисунок 4.2 – Структура шаблону State

Учасниками шаблону є:

- **Context** – контекст:
 - визначає інтерфейс, що є корисним для клієнтів;
 - зберігає екземпляр підкласу *ConcreteState*, котрим визначається поточний стан;
- **State** – стан:
 - визначає інтерфейс для інкапсуляції поведінки, асоційованої з конкретним станом контексту *Context*;
- Підкласи **ConcreteState** – конкретні стани:
 - кожний підклас реалізує поведінку, асоційовану з деяким станом контексту *Context*.

Відносини. Клас *Context* делегує залежні від стану запити до поточного об'єкта *ConcreteState*.

Контекст може передати себе у якості аргументу об'єкта *State*, котрий буде обробляти запит. Це надає можливість об'єкта-стану при необхідності отримати доступ до контексту.

Context – це головний інтерфейс для клієнтів. Клієнти можуть конфігурувати контекст об'єктами стану *State*. Один раз сконфігурувавши контекст, клієнти вже не повинні напямучу зв'язуватися з об'єктами стану.

Або *Context*, або підкласи *ConcreteState* можуть вирішити, за яких умов та у якій послідовності відбувається зміна станів.

Результати. Шаблон локалізує залежну від стану поведінку і

ділить її на частини, що відповідають станам, переходи між станами стають явними.

4.2.3 Шаблон проектування Команда (Command)

Призначення. Інкапсулює запит у формі об'єкта, дозволяючи тим самим задавати параметри клієнтів для обробки відповідних запитів, ставити запити у чергу або протоколювати їх, а також підтримувати скасовування операцій.

Також відомий під назвою дія (Action), транзакція (Transaction).

Застосування. Використовується коли:

- треба параметризувати об'єкти дією. У процедурній мові таку параметризацію можна виразити за допомогою функції оберненого виклику (Callback), тобто такою функцією, котра реєструється для того, щоб бути викликаного пізніше. Команди є об'єктно-орієнтованою альтернативою функціям оберненого виклику;
- визначати, ставити у чергу та виконувати запити у різний час. Строк життя об'єкта Команди не обов'язково залежить від строку життя початкового запиту. Якщо отримувач вдається реалізувати таким чином, щоб він не залежав від адресного простору, то об'єкт-команду можна передати іншому процесу, котрий займеться його виконанням;
- потрібна підтримка скасовування операцій. Операція Execute об'єкта Команда може зберегти стан, що необхідний для скасування дій, виконаних Командою. У цьому разі у інтерфейсі класу Command повинна бути додаткова операція Unexecute, котра скасовує дії, виконанні попереднім викликом операції Execute. Виконані команди зберігаються у списку історії. Для реалізації довільної кількості рівнів скасування та повтору команд треба обходити цей список відповідно в оберненому та прямому напрямках, викликаючи під час відвідування кожного елементу операцію Unexecute або Execute;
- підтримати протоколювання змін, щоб їх можна було виконати повторно після аварійної зупинки системи. Доповнивши інтерфейс класу Command операціями зберігання та завантаження, можна вести протокол змін у внутрішній пам'яті. Для поновлення після збою треба буде завантажити збереженні команди з диску та повторно виконати їх за допомогою операції Execute;
- треба структурувати систему на основі високорівневих операцій, що побудовані з примітивних. Така структура є типовою для

інформаційних систем, що підтримують транзакції. Транзакція інкапсулює множину змін даних. Шаблон Команда дозволяє моделювати транзакції. В усіх команд є спільний інтерфейс, що надає можливість працювати однаково з будь-якими транзакціями. За допомогою цього шаблону можна легко додавати у систему нові види транзакцій.

Структура шаблону наведена на рис. 4.3.

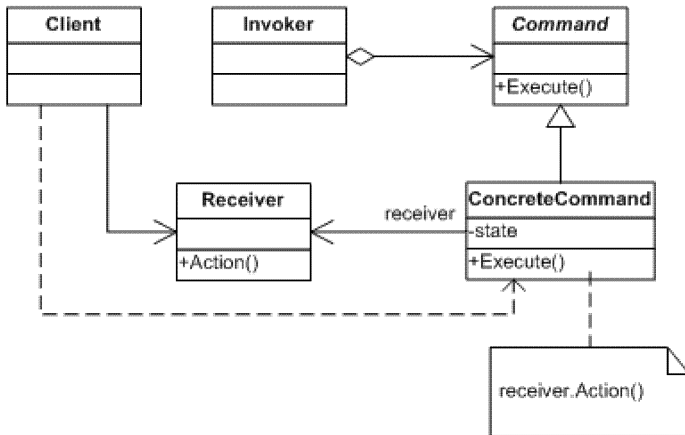


Рисунок 4.3– Структура шаблону Command

Учасниками шаблону є:

- **Command** – команда:
 - оголошує інтерфейс для виконання операції;
- **ConcreteCommand** – конкретна команда:
 - визначає зв'язок між об'єктом-отримувачем *Receiver* та дією;
 - реалізує операцію *Execute* шляхом виклику відповідних операцій об'єкта *Receiver*;
- **Client** – клієнт:
 - створює об'єкт класу *ConcreteCommand* та встановлює його отримувача;
- **Invoker** – виклиач:
 - звертається до команди щоб та виконала запит;
- **Receiver** – отримувач:
 - має у своєму розпорядженні уся інформація про способи

виконання операцій, необхідних для задоволення запиту.

У ролі отримувача може виступати будь-який клас.

Відносини. Клієнт створює об'єкт *ConcreteCommand* та встановлює для нього отримувача. Викликач *Invoker* зберігає об'єкт *ConcreteCommand*. Викликач надсилає запит, викликаючи операцію команди *Execute*. Якщо підтримується скасування виконаних дій, то *ConcreteCommand* перед викликом *Execute* зберігає інформацію про стан, достатню для виконання скасування. Об'єкт *ConcreteCommand* викликає операції отримувача для виконання запиту.

Результати. Шаблон *Command* розриває зв'язок між об'єктом, який ініціює операції, і об'єктом, що має інформацію про те, як її виконати. Створюється об'єкт *Command*, який можна розширювати і маніпулювати ним як об'єктом. Із простих команд можна збирати складені.

4.2.4 Шаблон проектування Інтерпретатор (Interpreter)

Призначення. Для заданої мови визначає представлення її граматики, а також інтерпретатор речень цієї мови. У разі, якщо якась задача виникає досить часто, є сенс подати її конкретні проявлення у вигляді речень простою мовою. Потім можна буде створити інтерпретатор, котрий вирішує задачу, аналізуючи речення цієї мови.

Наприклад, пошук рядків за зразком – досить розповсюджена задача. Регулярні вирази – це стандартна мова для завдання зразків пошуку.

Застосування. Використовується коли є мова для інтерпретації, речення котрої можна подати у вигляді абстрактних синтаксичних дерев. Найкраще шаблон працює коли:

- граматика проста. Для складних граматик ієрархія класів стає занадто громіздкою та некерованою. У таких випадках краще застосовувати генератори синтаксичних аналізаторів, оскільки вони можуть інтерпретувати вирази, не будуючи абстрактних синтаксичних дерев, що заощаджує пам'ять, а можливо і час;

- ефективність не є головним критерієм. Найефективніші інтерпретатори зазвичай не працюють безпосередньо із деревами, а спочатку транслюють їх в іншу форму. Так, регулярний вираз часто перетворюють на скінченний автомат. Але навіть у цьому разі сам транслятор можна реалізувати за допомогою шаблону інтерпретатор.

Структура шаблону наведена на рис. 4.4.

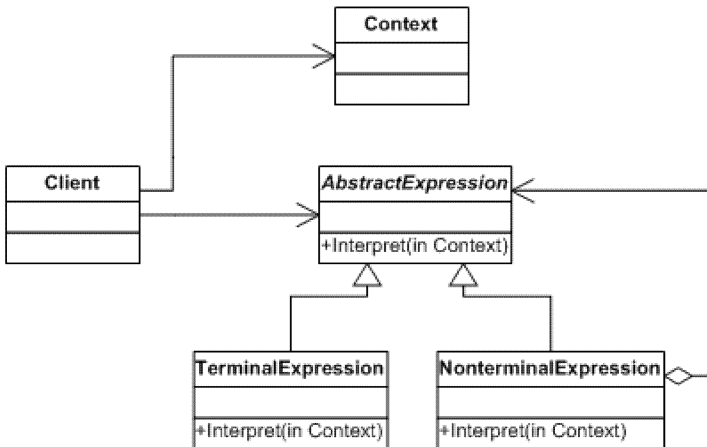


Рисунок 4.4 – Структура шаблону Interpreter

Учасниками шаблону є:

- **AbstractExpression** – абстрактний вираз:
 - оголошує абстрактну операцію *Interpret*, загальну для усіх вузлів у абстрактному синтаксичному дереві;
- **TerminalExpression** – термінальний вираз:
 - реалізує операцію *Interpret* для термінальних символів граматики;
 - необхідний окремий екземпляр для кожного термінального символу у реченні;
- **NonterminalExpression** – нетермінальний вираз:
 - по одному такому класу потребується для кожного граматичного правила;
 - зберігає змінні екземпляру типу *AbstractExpression* для кожного символу;
 - реалізує операцію *Interpret* для нетермінальних символів граматики. Ця операція рекурсивно викликає себе для змінних, що зберігають символи;
- **Context** – контекст:
 - містить інформацію, глобальну по відношенню до інтерпретатору;
- **Client** – клієнт:

- буде (або отримує у готовому вигляді) абстрактне синтаксичне дерево, що репрезентує окреме речення мовою з даною граматикою. Дерево складено з екземплярів класів *NonterminalExpression* та *TerminalExpression*;
- викликає операцію *Interpret*.

Відносини. Клієнт буде (або отримує у готовому вигляді) речення у вигляді абстрактного синтаксичного дерева, у вузлах якого знаходяться об'єкти класів *NonterminalExpression* та *TerminalExpression*. Далі клієнт ініціалізує контекст та викликає операцію *Interpret*.

У кожному вузлі виду *NonterminalExpression* через операції *Interpret* визначається операція *Interpret* для кожного підвиразу. Для класу *TerminalExpression* операція *Interpret* визначає базу рекурсії.

Операції *Interpret* у кожному вузлі використовують контекст для зберігання та доступу до стану інтерпретатору.

Результати. Завдяки шаблону граматику стає легко розширювати і змінювати, реалізації класів, що описують вузли абстрактного синтаксичного дерева схожі (легко кодуються). Можна легко змінювати спосіб обчислення виразів. Але важко супроводжувати граматику з великим числом правил.

4.2.5 Шаблон проектування Шаблонний метод (Template method)

Призначення. Визначає основу алгоритму та дозволяє підкласам перевизначити деякі кроки алгоритму, не змінюючи структуру в цілому.

Застосування. Слід використовувати коли:

- треба одноразово використати інваріантні частини алгоритму, залишаючи реалізацію поведінки, що змінюється, на розсуд підкласів;
- треба відокремити та локалізувати в одному класі поведінку, що є загальною для усіх підкласів, щоб запобігти дублюванню коду. Це хороший приклад техніки “винесення за лапки з метою узагальнення”, що описана в роботі Вільяма Опдайка та Ральфа Джонсона. Спочатку ідентифікуються відмінності в існуючому кодї, а потім вони виносяться у окремі операції. У кінцевому підсумку

відмінні фрагменти коду замінюються шаблонним методом, з котрого викликаються нові операції;

– для управління розширеннями підкласів. Можна визначити шаблонний метод таким чином, що він буде викликати операції-зачіпки у означених точках, дозволивши тим самим розширення тільки у цих точках.

Структура шаблону наведена на рис. 4.5.

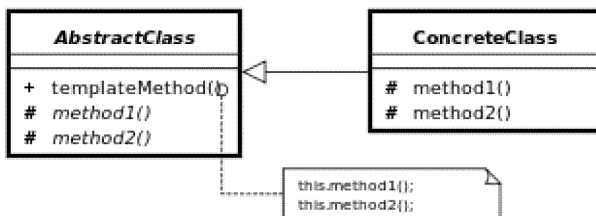


Рисунок 4.5 – Структура шаблону Template method

Учасниками шаблону є:

- **AbstractClass** – абстрактний клас:
 - визначає абстрактні примітивні операції, що замінюються у конкретних підкласах для реалізації кроків алгоритму;
 - реалізує шаблонний метод, що визначає скелет алгоритму. Шаблонний метод викликає примітивні операції, а також операції, означенні у класі AbstractClass, чи в інших об'єктах;
- **ConcreteClass** – конкретний клас:
 - реалізує примітивні операції, що виконують кроки алгоритму у спосіб, котрий залежить від підкласу.

Відносини. ConcreteClass припускає, що інваріантні кроки алгоритму будуть виконані у AbstractClass.

4.3 Завдання на лабораторну роботу

4.3.1 Вивчити шаблони поведінки, їхню загальну характеристику та призначення. Для кожного з цих шаблонів:

– вивчити призначення шаблону, альтернативні назви, призначення, випадки коли його застосування є доцільним та результати такого застосування;

- знати особливості реалізації шаблону, споріднені шаблони, відомі випадки його застосування в програмних додатках;

- вільно володіти структурою шаблону, призначенням його класів та відносинами між ними.

4.3.2 За допомогою джерел зі списку рекомендованих для виконання лабораторної роботи, виконати програмну реалізацію шаблонів поведінки.

4.4 Зміст звіту

4.4.1 Тема та мета роботи.

4.4.2 Короткі теоретичні відомості.

4.4.3 Результати роботи.

4.4.4 Висновки, що відображують результати виконання роботи та їх критичний аналіз.

4.5 Контрольні запитання

4.5.1 Призначення шаблонів поведінки для проектування ПЗ.

4.5.2 Коротка характеристика кожного розглянутого шаблону поведінки.

4.5.3 Назви, призначення та мотивація шаблону Memento.

4.5.4 Структура шаблону Memento та його учасники.

4.5.5 Особливості реалізації шаблону Memento. Результат використання шаблону.

4.5.6 Назви, призначення та мотивація шаблону State.

4.5.7 Структура шаблону State та його учасники.

4.5.8 Особливості реалізації шаблону State. Результат використання шаблону.

4.5.9 Назви, призначення та мотивація шаблону Command.

4.5.10 Структура шаблону Command та його учасники.

4.5.11 Особливості реалізації шаблону Command. Результат використання шаблону.

4.5.12 Назви, призначення та мотивація шаблону Interpreter.

4.5.13 Структура шаблону Interpreter та його учасники.

4.5.14 Особливості реалізації шаблону Interpreter. Результат використання шаблону.

4.5.15 Шаблони, які використовуються сумісно з Memento, State, Command та Interpreter.

ЛАБОРАТОРНА РОБОТА № 5

ПРАКТИЧНЕ ВИКОРИСТАННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ

5.1 Мета роботи

Отримати практичні навички використання шаблонів проектування при розробці програмного продукту.

5.2 Завдання на лабораторну роботу

Використовуючи знання, отримані при виконанні лабораторних робіт 1-4, вирішити прикладні задачі.

5.2.1 Задача 1: Модель «Годинник на ратуші»

У місті працюють «Ремісники», що використовують для організації часу у своїй роботі «Годинник» на міській ратуші.

«Пекар» о 4 ранку розпалює піч, о 5 замішує тісто, о 6 ставить тісто випікатись, о 7 витягає випічку з печі, о 8 відкриває пекарню для продажу випічки, о 16 зачиняє пекарню. «Чоботяр» о 10 відкриває майстерню, об 11 випиває гальбу пива, о 15 починає співати пісню, о 18 зачиняє майстерню і йде до шинку. «Шинкар» о 16 починає варити пиво, о 17 відчиняє шинок, о 23 зачиняє шинок.

Усі «ремісники» прислухаються до того, яка година, і відповідно до цього вирішують, що їм слід робити. «Годинник» дозволяє чути усім, хто його слухає, котра година, тобто виступає спостережуваним суб'єктом, а ремісники – спостерігачами

Використати шаблон проектування «Одинак» (Singleton) для опису класу «Годинник», шаблон проектування «Спостерігач» (Observer) для опису взаємодії нащадків класу «Ремісник» (observer) з класом «Годинник» (observable) та шаблон проектування «Ітератор» (Iterator) для організації послідовного звертання до списку спостерігачів.

5.2.2 Задача 2: Модель «Святий Миколай»

Єдиним джерелом подарунків чемним дітям та покарання неслухняним є «Святий Миколай», який приймає у дітей замовлення на подарунки та дарує їм їх, або карає за погані вчинки. У цьому «Святому Миколаю» допомагає «Фабрика добрих подарунків» та «Фабрика поганих подарунків», кожна з яких створює один їстівний

та один неїстівний подарунок (методи абстрактної фабрики та абстрактні продукти відповідно).

«Фабрика добрих подарунків» створює солодощі та іграшки, а «Фабрика поганих подарунків» створює гіркі пілюлі та різки.

Для того, щоб отримати подарунок, діти надсилають повідомлення у вигляді двох чисел: кількості хороших і поганих вчинків, які вони скоїли за останній рік.

«Миколай» збирає усі повідомлення від дітей та вирішує, надіслати дитині добрий чи поганий подарунок в залежності від того, яких вчинків більше (добрих чи поганих) – призначає кожній дитині у списку відповідну конкретну фабрику.

У день Святого Миколая повинні спрацювати призначені їм фабрики, а «Миколай» надсилає дітям створені для них подарунки.

Використати шаблон проектування «Одинак» (Singleton) для опису класу «Святий Миколай», шаблон проектування «Абстрактна фабрика» (Abstract Factory) для опису створення подарунків (abstract/concrete product) на фабриках (abstract/concrete factory) та шаблон проектування «Ітератор» (Iterator) для організації послідовного звертання до списку дітей та призначених їм фабрик.

5.3 Зміст звіту

5.3.1 Тема та мета роботи.

5.3.2 Результати роботи.

5.3.3 Висновки, що містять відповіді на контрольні запитання, а також відображують результати виконання роботи та їх критичний аналіз.

5.4 Контрольні запитання

5.4.1 Які шаблони проектування було використано при розробці програми.

5.4.2 Обґрунтуйте доречність використання цих шаблонів при вирішенні поставленої задачі.

ЛИТЕРАТУРА

1. Фаулер М. UML. Основы, 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2004. – 192 с.
2. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2001. – 368 с.
3. Зандстра Мэтт. PHP: объекты, шаблоны и методики программирования / Мэтт Зандстра. – 2-е изд.. – М.: Вильямс, 2009. – 480 с. – <http://www.williamspublishing.com/PDF/978-5-8459-1586-3/part.pdf>
4. Джошуа Кериевски. Рефакторинг с использованием шаблонов. – М.: Вильямс, 2006. – 400 с.
5. Крэг Ларман. Применение UML и шаблонов проектирования. Второе издание. – М.: Вильямс, 2004. – 624 с.
6. Марк Гранд Шаблоны проектирования в JAVA. Каталог популярных шаблонов проектирования, проиллюстрированных при помощи UML = Patterns in Java, Volume 1. A Catalog of Reusable Design Patterns Illustrated with UML. – М.: «Новое знание», 2004. – 560 с.
7. <http://codelab.ru/c/patterns/>

Додаток А

Коротка довідка з мови UML

UML (англ. Unified Modeling Language) – уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування. Є невід'ємною частиною уніфікованого процесу розробки програмного забезпечення. UML призначена для візуалізації, проектування, моделювання та документування при розробці програмного забезпечення. UML не є мовою програмування, але існують засоби кодогенерації на основі UML.

Діаграма класів – статичне представлення структури моделі. Подає статичні (декларативні) елементи, такі як: класи, типи даних, їх зміст та відношення. Діаграма класів, також, може містити позначення для пакетів та може містити позначення для вкладених пакетів. Також, діаграма класів може містити позначення деяких елементів поведінки, однак, їх динаміку розкрито в діаграмах інших типів.

Клас позначається прямокутником з трьох частин: верхня містить ім'я класу, середня список атрибутів класу, нижня - список операцій класу. Абстрактні класи та абстрактні операції відрізняються нахильним написом їхніх імен.

Додатково елементи списків можуть позначатись типами та областю видимості.

Квантор видимості може приймати одне з трьох можливих значень:

- "+" означає область видимості типу загальнодоступний;
- "#" означає область видимості типу захищений (protected);
- "-" означає область видимості типу закритий (private).

Для операції за необхідності може бути представлений псевдокод її реалізації в прямокутнику із загнутим кутком, що зв'язаний з ім'ям операції пунктирною лінією.

Відношення між класами позначається різними видами ліній і стрілок. Існують наступні види відношень.

Асоціація показує, що об'єкти однієї сутності (класу) пов'язані з об'єктами іншої сутності. Позначається суцільною лінією з деякими спеціальними символами, що характеризують властивості асоціації. В якості таких додаткових спеціальних символів можуть використовуватися ім'я асоціації, що показує природу відношення, при цьому в імені вказується напрям читання зв'язку за допомогою

трикутного маркеру, а також ролі, які грає кожен з класів в цьому відношенні, і кратність, тобто кількість об'єктів зв'язаних відношенням.

Асоціація може бути спрямованою - односторонньою (позначається відкритою стрілкою, що вказує на напрям асоціації) та неспрямованою. В односторонній асоціації обидва класи входять у відношення, але тільки один з них знає про існування цього відношення. Спрямована асоціація використовується коли один клас спілкується з іншим за допомогою інстанціювання.

У відношенні між двома класами сама асоціація також може мати властивості і тому також може бути представлена у вигляді класу.

В асоціації можуть брати участь більше двох класів, такі асоціації позначаються лініями, один кінець яких йде до класового блоку, а інший до загального ромбу.

Агрегація – різновид асоціації, при відношенні між цілим і його частинами. Як тип асоціації, агрегація може бути іменованою. Агрегація не може включати відразу декілька класів. Агрегація зустрічається, коли один клас є колекцією або контейнером інших. Час існування класів-частин не залежить від часу існування класу-цілого. Якщо контейнер буде знищений, то його вміст – ні. Графічно агрегація позначається порожнім ромбом на блоці класу-цілого і лінією, що йде від цього ромба до класу-частини.

Композиція – більш суворий варіант агрегації. Під час композиції має місце жорстка залежність часу існування класу-цілого та класів-часток. Якщо контейнер буде знищений, то весь його вміст буде також знищено. Графічно позначається як і агрегація, але з зафарбованим ромбом.

Узагальнення (Наслідкування) – показує, що один з двох зв'язаних класів (підтип) є більш конкретною формою іншого (супертипу), який називається узагальненням першого. На практиці це означає що будь-який екземпляр підтипу є також примірником супертипу. Графічно генералізація представляється суцільною лінією з закритою стрілкою (порожнім трикутником) у супертипа. Генералізація також відома як наслідування або зв'язок типу "is a".

Реалізація – відношення між двома елементами моделі, в якому один елемент (клієнт) реалізує поведінку, задану іншим (постачальником, інтерфейсом). Графічно реалізація представляється

також як і генералізація, але з пунктирною лінією.

Залежність – це відношення використання, при якому зміна в специфікації одного тягне за собою зміну іншого, причому протилежне не обов'язково. Графічно позначається пунктирною стрілкою, що йде від залежного елемента до того, від якого він залежить.

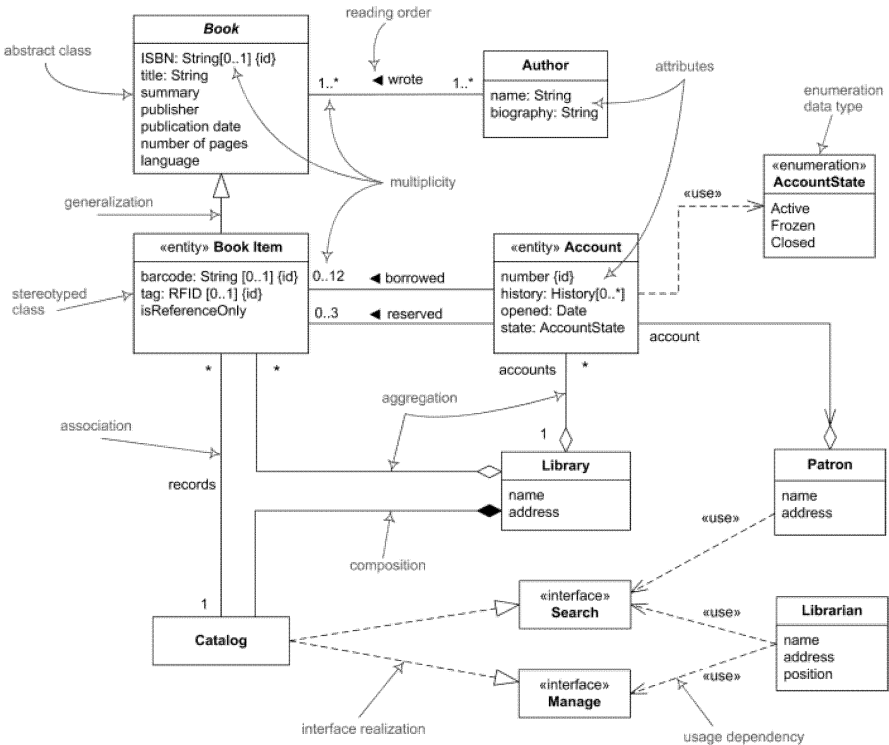


Рисунок А.1 – Приклад діаграми класів

КОНТРОЛЬНА РОБОТА ДЛЯ ЗАОЧНИКІВ

1. Ознайомитися з шаблонами проектування, їхніми загальними характеристиками, призначенням та прикладами використання.

2. З лабораторних робіт обрати по два шаблони проектування кожного типу. Тобто обрати 2 твірних шаблони проектування (ЛР1), два структурні шаблони проектування (ЛР2), два шаблони поведінки (ЛР3, ЛР4).

3. Для обраних шаблонів:

- вивчити призначення шаблону, альтернативні назви, призначення, випадки коли його застосування є доцільним та результати такого застосування;

- знати особливості реалізації шаблону, споріднені шаблони, відомі випадки його застосування в програмних додатках;

- вільно володіти структурою шаблону, призначенням його класів та відносинами між ними;

- за допомогою джерел зі списку рекомендованих виконати програмну реалізацію шаблону (використати будь-яку відому об'єктно-орієнтовану мову програмування).

4. В звіті навести короткі теоретичні відомості щодо шаблону та текст програмної реалізації шаблону.

5. При захисті контрольної роботи знати:

- що таке шаблони проектування та їхнє призначення;

- які типи шаблонів проектування існують;

- для обраних шаблонів знати призначення шаблону, альтернативні назви, випадки коли його застосування є доцільним та результати такого застосування;

- вміти пояснити особливості програмної реалізації шаблону, вільно володіти структурою шаблону, призначенням його класів та відносинами між ними.