Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Технології розроблення програмного забезпечення

Лабораторна робота №6

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

Виконав:
студент групи ІА–22:
Клочков І. Ф.

Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: : Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator».

Мета: ознайомитися з шаблонами проектування «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator». Реалізувати частину функціоналу програми за допомогою одного з розглянутих шаблонів.

Хід роботи

..18 Shell (total commander) (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі - перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

1. Короткі теоретичні відомості

Принципи проектування SOLID

Принципи проектування SOLID були введені Робертом Мартіном в його статті "Об'єктноорієнтоване проектування" і відносяться до розробки класів систем. Ось вони:

Принцип єдиної відповідальності (single responsibility principle) свідчить про необхідність створення класів, що відповідають не більше ніж за одну річ. Проблема, яка виникає коли один клас містить безліч обов'язків, - раніше всього, погана читаність коду (як правило, такі класи розростаються до неймовірних розмірів і стає практично неможливою їх подальша підтримка), і складність зміни. Наприклад, якщо служба звітів тісно інтегрована із службою друку (у одному класі), то зміни в одній із служб напевно приведуть до поломки інших. Цей принцип також веде до зменшення зв'язаності компоненту (low coupling) і підвищення цілісності елементів системи (high cohesion).

Принцип відкритості/закритості проголошує необхідність додавання можливості розширення без зміни початкових кодів самого компонента. Таким чином існує можливість зміни поведінки компонента без зачіпання початкових кодів, що може знадобитися у разі тестування (заміна окремих компонент

заглушками) або додавання функціональності. Однією з реалізацій принципів відкритості/закритості є реалізація інтерфейсів або абстрактних класів для специфікації семантики необхідних дій (контракт послуг, що надаються), і надання можливостей до власної реалізації цих інтерфейсів (конкретна реалізація).

Принцип підстановки Барбари Лисков стверджує, що підкласи в ієрархії класів повинні зберігати функціональність, аналогічну батьківським класам. Подстановка підкласів повинна залишити програму функціонувати тим же чином (коректно). Проблема походить з наступних міркувань: припустимо, реалізується клас калькулятор; створюється дочірній клас "цифровий калькулятор", але в перевантаженні методу "сума" використовується множення. Синтаксично, таке використання дочірніх класів не несе помилок (компілюється і запускається). Семантично, результат роботи програми абсолютно поміняється при заміні батьківського класу підкласом (кожного разу при підсумовуванні ми будемо насправді множити). Принцип свідчить про необхідність зберігати семантику базового класу в уникнення безглуздих помилок в програмному забезпеченні.

Основною ідеєю **принципу розділення інтерфейсів** є розбиття загального "контракту" (чи набору послуг, що надаються) програми на безліч дрібних інтерфейсних модулів, які відповідають за одну конкретну роботу. Такий підхід нагадує принцип одного обов'язку, але відноситься до усієї системи в цілому, і спрощує читання, розуміння і модифікацію системи, оскільки користувачі інтерфейсу бачать лише той "розріз" системи, який їм в даний момент потрібний.

Принцип інверсії залежностей стверджує про необхідність усунення залежностей модулів верхнього рівня від модулів нижнього рівня, оскільки і ті і інші повинні посилатися на абстракції (інтерфейси). Абстракції, у свою чергу,

не повинні залежати від деталей, а навпаки. Найпростіше зрозуміти цей принцип на конкретному прикладі: нехай існує деяка служба, що виробляє конкретні розрахунки (назвемо її Служба 1); існує служба, яка має посилання на неї, і використовує для реалізації власної внутрішньої механіки (назвемо її служба 2). Реалізація цього принципу виглядатиме таким чином: у Служби 1 буде винесений загальний інтерфейс, і цей інтерфейс використовуватиметься в Службі 2 (підставлятися в конструктор, наприклад). Таким чином, служба 1 залежить від абстракції (загальний інтерфейс), і не впливає на роботу Служби 2 (модулі верхнього рівня не залежать від модулів нижнього рівня). Абстракція у свою чергу не залежить від деталей, оскільки є простою специфікацією можливих дій. Слід зазначити, що цей принцип добре поєднується з попередніми, і дозволяє складати чистіші застосування. Існує безліч реалізацій цього принципу - впровадження залежностей, система розширень, локатор служб.

Шаблон "Abstract Factory" (Абстрактна фабрика)

Призначення:

Шаблон "Abstract Factory" забезпечує створення сімейства пов'язаних або залежних об'єктів без вказівки їх конкретних класів. Це дозволяє клієнту працювати з об'єктами, не знаючи їх конкретної реалізації.

Проблема:

Необхідно створити різні типи об'єктів, які належать до певної категорії (наприклад, для різних операційних систем або стилів інтерфейсів), але потрібно уникнути прямого створення конкретних класів, щоб забезпечити гнучкість і розширюваність.

Рішення:

Створення абстрактної фабрики, яка визначає інтерфейси для створення об'єктів, але дозволяє конкретним фабрикам створювати свої власні реалізації. Клієнт працює з абстрактними фабриками, що дозволяє йому створювати об'єкти певної сім'ї без знання їх конкретної реалізації.

Приклад:

У програмуванні для різних операційних систем можна створити окремі фабрики для Windows і Linux, кожна з яких буде створювати інтерфейси і компоненти, специфічні для цієї системи.

Переваги:

- Покращує розширюваність системи.
- Знижує залежності між конкретними класами.
- Дозволяє легко додавати нові сімейства об'єктів без зміни існуючого коду.

Недоліки:

- Може збільшити складність коду через додаткові класи та інтерфейси.
- Може привести до надмірної абстракції, яка ускладнює розуміння коду.

Шаблон "Factory Method" (Фабричний метод)

Призначення:

Шаблон "Factory Method" дозволяє створювати об'єкти без вказівки їх конкретного класу. Цей шаблон дозволяє підкласи змінювати тип створюваних об'єктів.

Проблема:

Необхідно створити об'єкти, але точний тип об'єкта не може бути визначений до моменту виконання програми, і створення різних об'єктів має бути централізовано контрольованим.

Рішення:

Створення абстрактного класу з фабричним методом, який делегує створення конкретного об'єкта підкласам. Кожен підклас реалізує фабричний метод, що відповідає за створення конкретних об'єктів.

Приклад:

У бібліотеці для обробки зображень можна створити фабричний метод для створення різних типів фільтрів (наприклад, фільтри для чорно-білих або

кольорових зображень), який буде визначати тип фільтра залежно від контексту.

Переваги:

- Спрощує код клієнта, оскільки він не потребує знань про точні типи об'єктів.
- Легко додаються нові типи об'єктів, не змінюючи існуючий код.

Недоліки:

- Може призвести до створення великої кількості підкласів.
- Вимагає підтримки додаткової абстракції, що може ускладнити код.

Шаблон "Memento" (Мементо)

Призначення:

Шаблон "Memento" дозволяє зберігати та відновлювати стан об'єкта без порушення інкапсуляції. Це корисно для реалізації функцій "відміни" в програмах.

Проблема:

Необхідно зберігати стан об'єкта для подальшого відновлення без прямого доступу до його внутрішніх даних.

Рішення:

Мементо дозволяє зберігати знімок внутрішнього стану об'єкта в окремому об'єкті (мементо) і пізніше відновлювати цей стан, не порушуючи принцип інкапсуляції.

Приклад:

У текстовому редакторі можна зберігати історію змін і використовувати мементо для реалізації функції "відміна", щоб повернутися до попереднього стану документа.

Переваги:

- Забезпечує безпечне збереження і відновлення стану об'єкта.
- Покращує гнучкість у реалізації функцій, таких як "відміна" та "повтор".

Недоліки:

- Зберігання стану може займати багато пам'яті, якщо об'єкти мають великий стан.
- Ускладнює код через необхідність створення мементо для кожного об'єкта.

Шаблон "Observer" (Спостерігач)

Призначення:

Шаблон "Observer" дозволяє створювати залежність один до багатьох, де зміни в одному об'єкті автоматично сповіщають усіх його спостерігачів.

Проблема:

Необхідно оновлювати кілька об'єктів одночасно, коли змінюється стан одного з них, без безпосереднього зв'язку між ними.

Рішення:

Створення спостерігача, який підписується на зміни в іншому об'єкті. Коли об'єкт змінюється, спостерігачі отримують повідомлення про ці зміни.

Приклад:

В інтерфейсі користувача може бути використаний шаблон "Observer", щоб оновлювати різні елементи інтерфейсу, коли змінюється дані, наприклад, оновлення балансу користувача в реальному часі.

Переваги:

- Спрощує управління залежностями між об'єктами.
- Покращує масштабованість, оскільки нові спостерігачі можуть бути додані без зміни коду спостережуваного об'єкта.

Недоліки:

- Може призвести до надмірної кількості оновлень, якщо спостерігачів багато.
- Ускладнює тестування через велику кількість взаємодій.

Шаблон "Decorator" (Декоратор)

Призначення:

Шаблон "Decorator" дозволяє динамічно додавати нову поведінку об'єкту, не змінюючи його структуру. Це дає можливість додавати нові можливості об'єкта в процесі виконання.

Проблема:

Необхідно розширювати функціональність об'єкта, але без створення великої кількості підкласів і порушення принципу відкритості/закритості.

Рішення:

Створення декоратора, який обгортає існуючий об'єкт і додає нову поведінку. Декоратори можуть бути складені, що дозволяє динамічно комбінувати різні додаткові функціональності.

Приклад:

У графічному редакторі можна додавати нові фільтри або ефекти до зображень, використовуючи декоратори для розширення базових можливостей без зміни існуючого коду.

Переваги:

- Дозволяє гнучко додавати нову функціональність.
- Спрощує модифікацію об'єктів без змін у їхній базовій реалізації.

Недоліки:

- Може призвести до надмірної кількості декораторів, що ускладнює розуміння системи.
- Вимагає додаткових витрат на виконання через обгортання об'єктів.

2. Реалізація не менше 3-х класів відповідно до обраної теми.

2.1 Структура класів

Структура проекту з реалізованими класами зображена на рисунку 1.

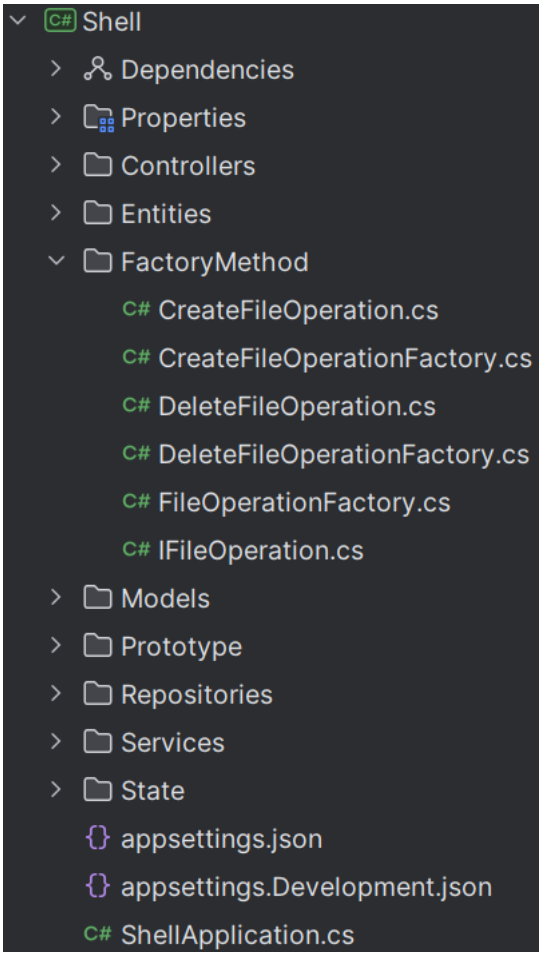

Рисунок 1 – структура проекту

2.2 Опис класів

IFileOperation: Це інтерфейс, який визначає спільний контракт для різних файлових операцій (наприклад CreateFileOperation, DeleteFileOperation). Він має загальний метод Execute, який реалізують класи операцій.

CreateFileOperation: Цей клас представляє операцію створення файлу, реалізує інтерфейс IFileOperation та є конкретною реалізацію операції. Він відповідає за логіку створення файлу.

DeleteFileOperation: Цей клас представляє операцію видалення файлу, реалізує інтерфейс IFileOperation та є конкретною реалізацію операції. Він відповідає за логіку видалення файлу.

FileOperationFactory: Це інтерфейс, який визначає спільний контракт для різних фабричних методів (наприклад CreateFileOperationFactory, DeleteFileOperationFactory). Він має загальний метод CreateOperation, який реалізують класи фабричних методів.

CreateFileOperationFactory: Цей клас представляє фабричний метод для створення операції створення файлу, реалізує інтерфейс FileOperationFactory та є конкретною реалізацію фабрики. Він відповідає за логіку отримання конкретної операції.

DeleteFileOperationFactory: Цей клас представляє фабричний метод для створення операції видалення файлу, реалізує інтерфейс FileOperationFactory та є конкретною реалізацію фабрики. Він відповідає за логіку отримання конкретної операції.

3. Реалізація шаблону за обраною темою

3.1 Опис шаблону

Для реалізації функціональності файлової оболонки я використав шаблон проектування Фабричний метод(Factory method). Цей патерн дозволив легко створювати, отримувати та використовувати операції над файлами. Це досягається через загальний інтерфейс операцій, який визначає метод для виконання операції, та загальний інтерфейс фабричного методу, який визначає метод для створення та отримання операції, та їх конкретних реалізацій. Таким чином, фабричний метод забезпечує альтернативний, гнучкий та швидкий спосіб створення об'єктів, де клієнт, який використовує ці об’єкти, не залежить від конкретних класів-продуктів, а працює лише через абстракцію. Цей шаблон особливо корисний, коли потрібно взаємодіяти з великими сімействами об’єктів та ізолювати клієнтський код від конкретних класів.

Опис реалізації:

Основні компоненти шаблону:

- IFileOperation – загальний інтерфейс операцій, який визначає метод запуску.
- CreateFileOperation – конкретна операція, яка відповідає за створення файлів.

- DeleteFileOperation – конкретна операція, яка відповідає за видалення файлів.
- FileOperationFactory – загальний інтерфейс, який визначає фабричний метод створення операцій.
- CreateFileOperationFactory – конкретний фабричний метод, який відповідає за створення операції створення файлу.
- DeleteFileOperationFactory – конкретний фабричний метод, який відповідає за створення операції видалення файлу.

Проблема, яку допоміг вирішити шаблон Factory method:
До використання цього патерну, код створення об’єктів залежав би від клієнтського коду. Це б призвело до поганої підтримуваності, складності розширення і неможливості повторного використання.

Використовуючи шаблон Factory method, вдалося:

- Зробити створення об’єктів дуже простим.
- Забезпечити незалежність клієнтського коду від конкретних класів.
- Забезпечити принцип відкритості/закритості.
- Забезпечити високу гнучкість, підтримку та розширюваність системи.

Переваги застосування шаблону Factory method:

- Позбавляє клас від прив’язки до конкретних класів продуктів.
- Виділяє код виробництва в одне місце, спрощуючи підтримку коду.
- Спрощує додавання нових продуктів до програми.
- Полегшує тестування програми.

3.2 Діаграма класів

Діаграма класів, які реалізують шаблон Фабричний метод(Factory method) зображена на рисунку 2.

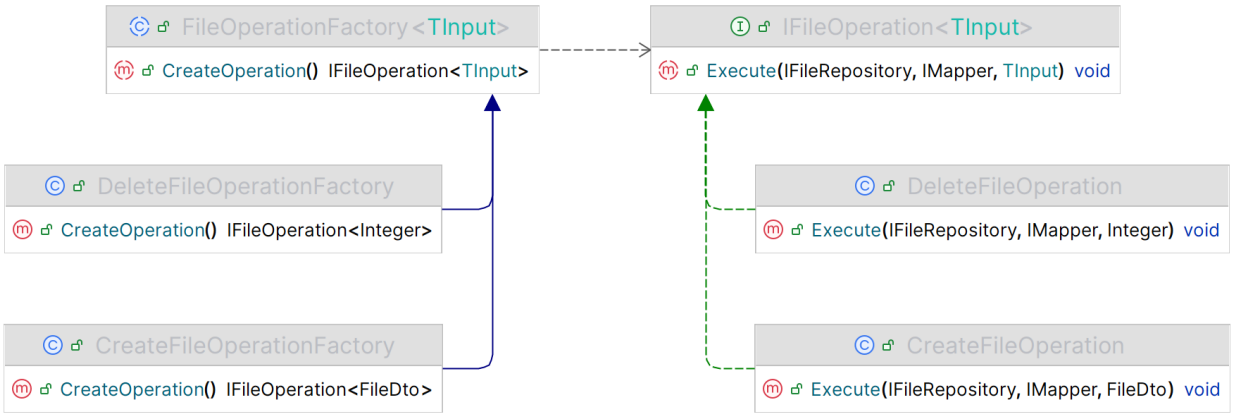

Рисунок 2 – реалізація шаблону фабричний метод(Factory method)

Посилання на код: https://github.com/ivanchikk/trpz

Висновок: виконавши дану лабораторну роботу, я використав шаблон проектування Фабричний метод(Factory method) для файлової оболонки, за допомогою якого було впроваджено ефективне створення нових команд та їх виконання. Були реалізовані інтерфейс та класи операцій IFileOperation, CreateFileOperation, DeleteFileOperation та інтерфейс та класи фабричних методів FileOperationFactory, CreateFileOperationFactory, DeleteFileOperationFactory. Завдяки цьому шаблону вдалося позбавитися прив’язки до конкретних класів продуктів, спростити процес додавання та створення нових операцій, та полегшити процес тестування системи. Це дозволило зробити файлову оболонку більш швидкою, гнучкою і легкою у підтримці та розширенні.