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 №5
«ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE».»

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

Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: Шаблони «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype».

Мета: ознайомитися з шаблонами проектування «Adapter», «Builder», «Command», «Chain of responsibility», «Prototype». 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

Анти-патерни (anti-patterns) — це погані рішення проблем, що часто використовуються в проектуванні. Вони є протилежністю «шаблонам проектування», які описують хороші практики. Анти-патерни в управлінні та розробці ПЗ включають:

Управління розробкою ПЗ:

- Дим і дзеркала: Демонстрація неповних функцій.
- Роздування ПЗ: Збільшення вимог до ресурсів у наступних версіях.
- Функції для галочки: Наявність непов'язаних функцій для реклами.

Розробка ПЗ:

- Інверсія абстракції: Приховування частини функціоналу.
- Невизначена точка зору: Моделі без чіткої специфікації.
- Великий клубок бруду: Система без чіткої структури.
- Бензинова фабрика: Необов'язкова складність дизайну.
- Затичка на введення даних: Недостатня обробка неправильного введення.

ООП:

- Базовий клас-утиліта: Спадкування замість делегування.
- Божественний об'єкт: Надмірна концентрація функцій в одному класі.
- Самотність: Надмірне використання патерну Singleton.

Анти-патерни в програмуванні:

- Непотрібна складність: Ускладнення рішень без необхідності.
- Дія на відстані: Несподівані взаємодії між різними частинами системи.
- Накопичити і запустити: Використання глобальних змінних для параметрів.
- Сліпа віра: Недостатня перевірка результатів або виправлень.
- Активне очікування: Використання ресурсів під час очікування події замість асинхронного підходу.
- Кешування помилки: Забуте скидання прапора після обробки помилки.
- Перевірка типу замість інтерфейсу: Перевірка типу замість використання інтерфейсу.
- Кодування шляхом виключення: Обробка випадків через винятки.
- Потік лави: Залишення поганого коду через високу вартість його видалення.
- Спагеті-код: Заплутаний код з некерованим виконанням.

Методологічні анти-патерни:

- Програмування методом копіювання-вставки: Копіювання коду замість створення спільних рішень.
- Дефакторінг: Заміна функціональності документацією.
- Золотий молоток: Впевненість у застосуванні одного рішення для всіх проблем.
- Передчасна оптимізація: Оптимізація без достатньої інформації.
- Винахід колеса: Створення нових рішень замість використання існуючих.
- Винахід квадратного колеса: Погане рішення, коли є хороше.
- Самознищення: Помилка, що призводить до фатальної поведінки програми.

Анти-патерни управління конфігурацією:

- DLL-пекло: Проблеми з версіями і доступністю DLL.
- Пекло залежностей: Проблеми з версіями залежних продуктів.

Організаційні анти-патерни:

- Аналітичний параліч: Надмірний аналіз без реалізації.
- Дійна корова: Неінвестування в розвиток продукту.
- Тривале старіння: Перенесення зусиль на портинг замість нових рішень.
- Скидування витрат: Перенесення витрат на інші відділи.
- Повзуче покращення: Додавання змін, що знижують загальну якість.
- Розробка комітетом: Розробка без централізованого керівництва.
- Ескалація зобов'язань: Продовження хибного рішення.
- Розповзання рамок: Втрата контролю над проектом.
- Замикання на продавці: Залежність від одного постачальника.

Шаблон "Adapter" (Адаптер)

Призначення:

Адаптер використовується для приведення інтерфейсу одного об’єкта до іншого, забезпечуючи сумісність між різними системами.

Проблема:

Існує потреба адаптувати різні формати даних (наприклад, XML до JSON) без зміни вихідного коду.

Рішення:

Створення адаптера, який «перекладає» інтерфейси одного об'єкта для іншого (наприклад, XML_To_JSON_Adapter).

Приклад:

Адаптер для зарядки ноутбука в різних країнах, що дозволяє підключити пристрій до різних типів розеток.

Переваги:

Приховує складність взаємодії різних інтерфейсів від користувача.

Недоліки:

Ускладнює код через додаткові класи.

Шаблон "Builder" (Будівельник)

Призначення:

Шаблон відділяє процес створення об'єкта від його представлення, що зручно для складних або багатоформатних об'єктів.

Проблема:

Ускладнене створення об'єкта, наприклад, формування відповіді web-сервера з кількох частин (заголовки, статуси, вміст).

Рішення:

Кожен етап створення об'єкта абстрагується в окремий метод будівельника, що дозволяє контролювати процес побудови.

Приклад:

Будівництво будинку за етапами, де кожен етап виконується окремою командою будівельників.

Переваги:

Забезпечує гнучкість та незалежність від внутрішніх змін у процесі створення.

Недоліки:

Клієнт залежить від конкретних класів будівельників, що може обмежити можливості.

Шаблон "Command" (Команда)

Призначення:

Перетворює виклик методу в об'єкт-команду, що дозволяє гнучко керувати діями (додавати, скасовувати, комбінувати команди).

Проблема:

Як організувати обробку кліків у текстовому редакторі без дублювання коду для різних кнопок.

Рішення:

Створити окремий клас-команду для кожної дії, який буде викликати методи бізнес-логіки через об'єкт інтерфейсу.

Приклад:

Офіціант у ресторані приймає замовлення (команда) і передає кухарю для виконання (отримувач), без прямого контакту між клієнтом і кухарем.

Переваги:

- Знімає залежність між об’єктами, що викликають і виконують операції
- Дозволяє скасовувати, відкладати та комбінувати команди
- Підтримує принцип відкритості/закритості

Недоліки:

- Ускладнює код додатковими класами

Шаблон "Chain of Responsibility"

Призначення:

Ланцюг відповідальності дозволяє організувати обробку запиту, передаючи його послідовно через ланцюг об'єктів, поки не буде знайдено обробник, здатний виконати запит. Це зручно для побудови гнучкої системи обробників, коли важливо зменшити залежність клієнта від обробників і структурувати обробку.

Проблема:

Припустимо, потрібно реалізувати систему обробки онлайн-замовлень, де доступ мають тільки авторизовані користувачі, а адміністратори мають додаткові права. Ці перевірки слід виконувати послідовно. Однак при додаванні нових перевірок код стає перевантаженим умовними операторами, що ускладнює підтримку.

Рішення:

Створити для кожної перевірки окремий клас з методом, який виконує потрібні дії. Далі всі об'єкти-обробники об'єднуються в ланцюг, де кожен обробник має посилання на наступного. Запит передається першому обробнику ланцюга, який або самостійно обробляє його, або передає далі. Якщо обробник не може виконати дію, запит продовжує передаватися далі по ланцюгу, аж поки не знайдеться відповідний обробник або ланцюг закінчиться.

Приклад з життя:

У JavaScript події в браузерах можна обробляти на різних рівнях DOMдерева. Наприклад, для таблиці з рядками, кожен з яких має кнопку для видалення,

можна поставити обробник не на кожну кнопку, а на tbody, що містить усі рядки. Якщо подія виникає на кнопці, але вона не має обробника, подія "піднімається" по ланцюгу: від кнопки до рядка, потім до tbody, що дозволяє зручно обробляти події динамічно.

Переваги:

- Зменшує залежність між клієнтом і обробниками.
- Відповідає принципу єдиного обов’язку.
- Підтримує принцип відкритості/закритості.

Недоліки:

- Запит може залишитися без обробки, якщо жоден обробник його не обробить.

Шаблон "Prototype"

Призначення:

Шаблон "Prototype" використовується для створення об'єктів шляхом копіювання існуючого шаблонного об'єкта. Це дозволяє спрощувати процес створення об'єктів, коли їх структура заздалегідь відома, та уникати прямого створення нових екземплярів.

Проблема:

Якщо потрібно скопіювати об’єкт, звичайний спосіб — створити новий об'єкт і вручну копіювати його поля. Але деякі частини об’єкта можуть бути приватними, що ускладнює доступ до них і порушує інкапсуляцію.

Рішення:

Шаблон "Prototype" доручає самим об’єктам реалізовувати метод clone(), який повертає їх копію. Це дозволяє створювати нові об'єкти без прив'язки до їхніх конкретних класів, залишаючи логіку копіювання всередині класу.

Приклад з життя:

У виробництві перед масовим випуском виготовляються прототипи, що дозволяють протестувати виріб. Ці прототипи служать шаблонами і не беруть участі в подальшому виробництві.

Переваги:

- Дозволяє клонувати об’єкти без прив'язки до конкретного класу.
- Зменшує дублювання коду ініціалізації.
- Прискорює процес створення об'єктів.
- Є альтернативою підкласам при створенні складних об’єктів.

Недоліки:

- Складність клонування об'єктів, що містять посилання на інші об'єкти.

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

2.1 Структура класів

Структура проекту з реалізованими класами зображена на рисунку 1.

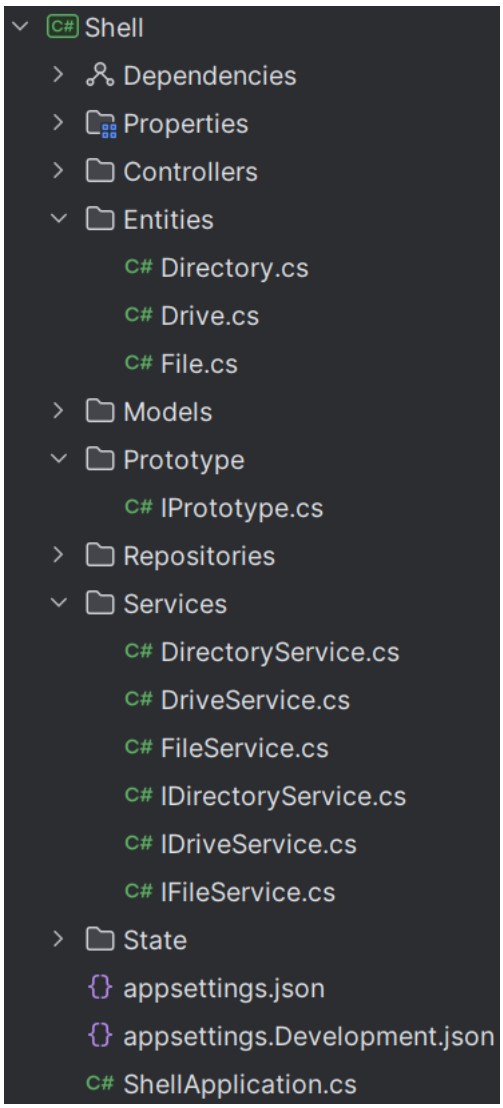

Рисунок 1 – структура проекту

2.2 Опис класів

IPrototype: Це інтерфейс, який визначає спільний контракт для різних прототипів (наприклад File, Directory). Він має метод Clone, який реалізують класи прототипів і можуть створювати власні копії незалежно від їх реалізацій. Кожен прототип імплементує метод клонування під себе.

File: Цей клас представляє файл, реалізує інтерфейс прототипу IPrototype та є конкретною реалізацію прототипу. Він дозволяє отримати власну копію за допомогою методу Clone.

Directory: Цей клас представляє папку, реалізує інтерфейс прототипу IPrototype та є конкретною реалізацію прототипу. Він дозволяє отримати власну копію за допомогою методу Clone.

FileService: Цей клас є сервісом для управління файлами. Він виконує основні операції над файлами, включаючи: створення, отримання, копіювання, оновлення, видалення. Також це клієнт прототипу, який може запросити створення копії файлу.

DirectoryService: Цей клас є сервісом для управління папками. Він виконує основні операції над папками, включаючи: створення, отримання, копіювання, оновлення, видалення. Також це клієнт прототипу, який може запросити створення копії папки.

3. Реалізація шаблону за обраною темою

3.1 Опис шаблону

Для реалізації функціональності файлової оболонки я використав шаблон проектування Прототип (Prototype). Цей патерн дозволив створювати нові об'єкти шляхом копіювання вже існуючих. Це досягається через загальний інтерфейс, який визначає метод для створення копії об'єкта. Таким чином, прототип забезпечує альтернативний спосіб створення об'єктів, минаючи традиційне використання конструктора.

Шаблон особливо корисний, коли створення об'єктів є складним або ресурсоємним процесом, і клонування існуючого об'єкта є простішим та ефективнішим.

Опис реалізації:

Основні компоненти шаблону:

- IProtoype – загальний інтерфейс, який визначає метод клонування Clone.
- File – конкретний прототип, який відповідає за копіювання файлів.
- Directory – конкретний прототип, який відповідає за копіювання папок.
- FileService – клієнт, який використовує прототип файлів для своїх операцій.
- DirectoryService – клієнт, який використовує прототип папок для своїх операцій.

Проблема, яку допоміг вирішити шаблон Prototype:

До використання цього патерну, код для створення копії об’єктів міг би бути складним та переплутаним з багатьма рядками переназначення полів або використання великих та складних конструкторів. Це б призвело до поганої підтримуваності, складності розширення та читабельності.

Використовуючи шаблон Prototype, вдалося:

- Зменшити витрати на створення об'єктів, завдяки можливості клонувати вже існуючі.
- Підвищити продуктивність системи, оминаючи багаторазові виклики конструкторів.
- Спростити створення складних структур, завдяки реалізації глибокого клонування.
- Зменшити кількість коду, використовуючи механізм клонування себе.

Переваги застосування шаблону Prototype:

- Клонування об’єктів без прив’язки до їх конкретних класів.
- Менша кількість повторювань коду ініціалізації об’єктів.
- Прискорює створення об’єктів.
- Альтернатива створенню підкласів під час конструювання складних об’єктів.

3.2 Діаграма класів

Діаграма класів, які реалізують шаблон Прототип(Prototype) зображена на рисунку 2.

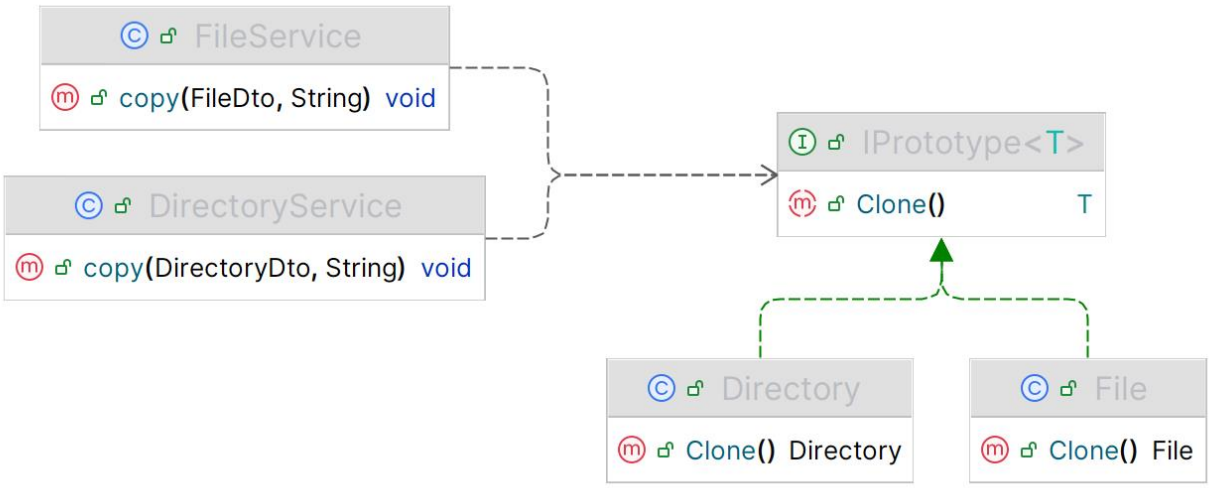

Рисунок 2 – реалізація шаблону Прототип(Prototype)

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

Висновок: виконавши дану лабораторну роботу, я використав шаблон проектування Прототип(Prototype) для файлової оболонки, за допомогою якого було впроваджено ефективне копіювання файлів та папок. Були реалізовані інтерфейс та класи прототипів IPrototype, File, Directory. Завдяки цьому шаблону вдалося зробити файлову оболонку більш швидкою, більш гнучкою і легкою у підтримці та розширенні, бо кожен прототип містить свою логіку створення власної копії. Це дозволило уникнути використання довгих, складних та ресурсоємних способів створення файлів та папок, що призвело б до заплутаного коду та сповільненню системи.