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

Лабораторна робота №8

Технології розроблення програмного забезпечення

ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»

FTP-server

Виконав
студент групи ІА–22:
Білокур Євгеній
Перевірив:
Мягкий Михайло Юрійович

Київ 2024

Зміст

Тема: шаблони «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR».

Мета: ознайомитися з шаблонами проектування «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR» та набути практичних навичок їх застосування. Реалізувати частину функціоналу програми за допомогою одного з розглянутих шаблонів для досягнення конкретних функціональних можливостей та забезпечення ефективної взаємодії між класами.

Хід роботи

..22 FTP-server (state, builder, memento, template method, visitor, client-server)

FTP-сервер повинен вміти коректно обробляти і відправляти відповіді по протоколу FTP, з можливістю створення користувачів (з паролями) і доступних їм папок, розподілу прав за стандартною схемою (rwe), ведення статистики з'єднань, обмеження максимальної кількості підключень і максимальної швидкості поширення глобально і окремо для кожного облікового запису.

Короткі теоретичні відомості

Шаблони роботи з БД при розробці корпоративних додатків: Існує набір сучасних шаблонів, які використовуються для розробників широко-масштабних корпоративних додатків - з кількістю користувачів понад 100 і великою кількістю взаємопов'язаних програм.

Такі додатки вимагають окремих підходів до організації доступу до даних, оскільки кількість даних, що зберігаються, класів і клієнтів, що звертаються, зростає.

Шаблон «Active Record»:
Поєднує дані й логіку в одному об'єкті, що представляє рядок БД. Широко використовується через простоту, але складність класу зростає зі збільшенням запитів, тому логіку часто виносять окремо.

Шаблон «Table Data Gateway»:
Окремий клас для взаємодії з БД для кожного класу даних. Вся логіка запитів зосереджена в ньому, що підвищує гнучкість і тестованість. Часто абстрагується в базовий клас для зменшення дублювання.

Шаблон «Data Mapping»:
Перетворює об'єкти даних у формат, що приймається БД чи мережею.

Маппер пов'язує властивості об'єкта з колонками таблиці, вирішуючи невідповідності типів.

Шаблон «Composite»:

Дозволяє об’єднувати об’єкти в деревоподібну структуру для подання ієрархій типу «частина-ціле» і обробляти їх уніфіковано.

Структура шаблону «Composite»:

1. Компонент (Component)
    - Загальний інтерфейс для всіх об’єктів у структурі (наприклад, метод getPrice()).
    - Може мати стандартну реалізацію деяких методів.
2. Лист (Leaf)
    - Конкретний об’єкт без вкладених елементів (наприклад, Продукт).
    - Реалізує методи інтерфейсу (наприклад, повертає свою ціну).
3. Контейнер (Composite)
    - Об’єкт, що містить інші компоненти (наприклад, Коробка).
    - Зберігає колекцію дочірніх елементів і викликає їхні методи.
4. Клієнт (Client)
    - Працює з компонентами через загальний інтерфейс, не залежачи від їхньої конкретної реалізації.

Використання:

- Використовується, коли модель програми можна представити у вигляді дерева.
- Продукт і Коробка мають спільний інтерфейс із методом для отримання вартості.
- Коробка підсумовує ціни продуктів і вкладених коробок.

Переваги:

- Спрощує роботу з деревоподібними структурами.
- Полегшує додавання нових компонентів.

Недоліки:

- Занадто загальний дизайн класів.

Шаблон «Flyweight»

Призначення:

Шаблон використовується для оптимізації пам'яті в програмах з великою кількістю схожих об'єктів шляхом поділу загальних (внутрішніх) даних між цими об'єктами. Унікальні (зовнішні) дані зберігаються окремо в контексті.

Проблема:

Коли додаток створює багато схожих об'єктів, це може призвести до надмірного використання оперативної пам’яті. Наприклад, в грі тисячі частинок з однаковим кольором і спрайтом займають багато місця через дублювання цих даних.

Рішення:

Розділити об'єкти на:

1. **Внутрішні (загальні) дані:** спільні для всіх екземплярів (наприклад, спрайт, колір).
2. **Зовнішні (унікальні) дані:** специфічні для кожного екземпляра (наприклад, координати, швидкість).

Об'єкти зберігають посилання на загальні дані, а унікальні — отримують із контексту.

Приклад:

1. У грі частинки мають спільні властивості (спрайт, колір) у класі ParticleType, тоді як унікальні дані (позиція, швидкість) залишаються в контексті.
2. У базі ветеринарної клініки для котів:
    - Загальні дані (порода, колір) зберігаються у класі CatBreed.
    - Унікальні дані (ім’я, вік, власник) — у класі Cat.

Структура:

1. **Flyweight (Легковаговик):**
   Оголошує інтерфейс для спільних даних.
2. **ConcreteFlyweight (Конкретний Легковаговик):**
   Реалізує внутрішній стан і забезпечує доступ до спільних даних.

3. **FlyweightFactory (Фабрика Легковаговиків):**
   Відповідає за створення і управління спільними об'єктами.
   Перевіряє, чи існує потрібний об'єкт, і створює його за необхідності.
4. **Client (Клієнт):**
   Використовує об'єкти Flyweight, додаючи до них зовнішні дані.

Переваги:

- Зменшує використання оперативної пам’яті завдяки спільним об'єктам.

Недоліки:

- Підвищує складність коду через введення додаткових класів.
- Витрачає більше процесорного часу на обробку зовнішніх даних.

Шаблон «Interpreter»

Призначення:

Шаблон використовується для подання граматики та інтерпретатора вибраної мови. Граматика описується термінальними та нетермінальними символами, кожен з яких інтерпретується в заданому контексті.
Клієнт передає сформовану пропозицію в термінах абстрактного синтаксичного дерева, де кожен вузол виразу інтерпретується рекурсивно.

Проблема:

Часто виникає потреба реалізувати задачу, яка потребує регулярного оновлення чи складного розбору текстів або виразів, наприклад, пошук рядків за шаблоном.

Рішення:

Створити інтерпретатор для заданої мови, який складається з:

- **Абстрактного виразу:** Базовий клас, що оголошує метод інтерпретації.
- **Термінальних виразів:** Описують кінцеві символи граматики.
- **Нетермінальних виразів:** Описують правила граматики, які можуть включати інші термінальні та нетермінальні вирази.
- **Контексту:** Зберігає глобальну інформацію для інтерпретації.

Клієнт будує абстрактне синтаксичне дерево, що складається з термінальних і нетермінальних вузлів. Дерево обчислює результат за допомогою рекурсивного виклику операції Розібрати().

Приклад:

Реалізація автоматичного розбору поставлених завдань із розбивкою на прості інструкції, зрозумілі для системи.

Структура:

1. **AbstractExpression (Абстрактний Вираз):**
   Оголошує метод Інтерпретувати(Context).
2. **TerminalExpression (Термінальний Вираз):**
   Реалізує метод для інтерпретації термінального символу граматики.
3. **NonterminalExpression (Нетермінальний Вираз):**
   Реалізує метод для інтерпретації нетермінального символу, що може включати інші вирази.
4. **Context (Контекст):**
   Зберігає дані для інтерпретації виразів (глобальна інформація).
5. **Client (Клієнт):**
   Створює синтаксичне дерево та ініціює інтерпретацію.

Переваги:

- Легко змінювати та розширювати граматику.
- Простота додавання нових способів обчислення виразів.

Недоліки:

- Ускладнення підтримки для граматики з великою кількістю правил.

Шаблон «Visitor»

Призначення:

Шаблон дозволяє визначати нові операції для класів ієрархії без зміни їх структури. Використовується для групування однотипних операцій, що застосовуються до різнотипних об'єктів, зберігаючи відкритість для розширення.

Переваги:

- Легке додавання нових операцій.
- Локалізація операцій у відвідувачах.

Недоліки:

- Складність додавання нових елементів у ієрархію (потрібно оновлювати всіх відвідувачів).

- Збільшення кількості методів у відвідувачах для підтримки нових елементів.

Проблема:

Необхідно реалізувати експорт даних графа з вузлами різного типу (міста, пам’ятки тощо) у формат XML.
Однак класи вузлів є стабільними і не можуть бути змінені. Це унеможливлює додавання в них нових методів для виконання операцій.

Рішення:

Виділити нову поведінку в окремі класи-відвідувачі. Об’єкти вузлів передаються до методів відвідувача, який виконує необхідні дії залежно від типу вузла.

Структура:

1. **Visitor (Відвідувач):**
   Інтерфейс або базовий клас, що декларує методи для відвідування кожного типу елементів.
2. **ConcreteVisitor (Конкретний Відвідувач):**
   Реалізує методи для виконання операцій над конкретними типами елементів.
3. **Element (Елемент):**
   Інтерфейс або базовий клас, що визначає метод accept(Visitor).
4. **ConcreteElement (Конкретний Елемент):**
   Реалізує метод accept(), викликаючи відповідний метод відвідувача.
5. **ObjectStructure (Структура Об'єктів):**
   Контейнер для елементів, який дозволяє зручно перебирати їх та передавати відвідувачу.

Приклад з життя:

У компіляторі є об'єкти різних синтаксичних конструкцій (виклики методів, умовні вирази тощо). Для кожного з них реалізовано:

1. **Відвідувача перевірки типів.**
2. **Відвідувача генерації коду.**
   Нові етапи компіляції додаються шляхом створення додаткових відвідувачів.

Переваги:

- Легкість у додаванні нових операцій (наприклад, експорту в JSON).
- Зручність групування поведінки за різними відвідувачами.

Недоліки:

- Додавання нових типів вузлів вимагає змін у всіх існуючих відвідува

Реалізувати не менше 3-х класів відповідно до обраної теми

Структура класів

Структура проекту з реалізованими класами зображена на рисунку 1

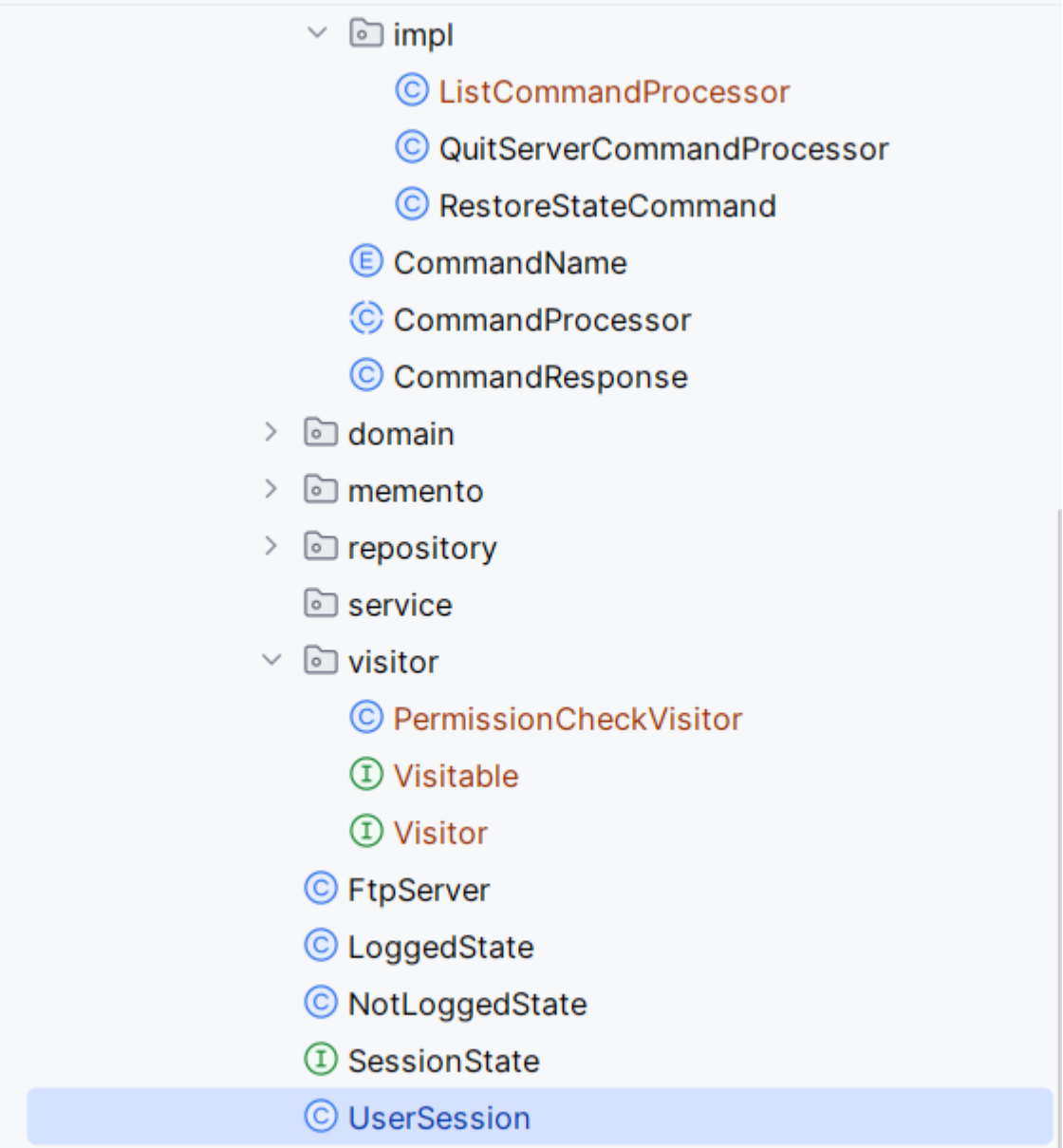

Рисунок 1 – Структура реалізованих класів

Опис класів

1. Visitor (Інтерфейс)

Цей інтерфейс визначає два методи:

- **visitFile(File file, User user)**: метод для відвідування файлів, в якому реалізується логіка перевірки доступу чи інших операцій, що залежать від файлів.
- **visitDirectory(Directory directory, User user)**: метод для відвідування каталогів, в якому реалізується логіка перевірки доступу чи інших операцій, що залежать від каталогів.

Цей інтерфейс реалізується в класах, які виконують операції над об'єктами File та Directory.

2. Visitable (Інтерфейс)

Цей інтерфейс представляє об'єкти, які можуть бути відвідані відвідувачем (Visitor). Клас, що реалізує цей інтерфейс, повинен забезпечувати реалізацію методу:

- **accept(Visitor visitor, User user)**: метод, який дозволяє передати об'єкт в певного відвідувача для виконання логіки (наприклад, перевірки прав доступу). Метод повертає результат виконаної операції для об'єкта (файлу або каталогу) для певного користувача.

3. PermissionCheckVisitor (Клас реалізує Visitor)

Цей клас реалізує інтерфейс Visitor і використовується для перевірки прав доступу до файлів та каталогів для конкретного користувача.

- **Конструктор**: приймає параметр access типу Permission.Access, який визначає тип доступу (наприклад, READ, WRITE, EXECUTE).
- **visitFile(File file, User user)**: перевіряє, чи має користувач відповідні права доступу до файлу. Для цього викликається метод hasAccess в об'єкті Permission, який визначає доступ для конкретного користувача на основі прав, власника та групи файлу.
- **visitDirectory(Directory directory, User user)**: подібно до методу для файлів, перевіряє доступ до каталогу за допомогою методу hasAccess класу Permission.

4. Permission (Клас для прав доступу)

Цей клас реалізує права доступу для файлів і каталогів. Він може містити:

- **Метод hasAccess**: перевіряє, чи має користувач доступ до певного файлу чи каталогу на основі його прав, власника та групи.

Реалізувати один з розглянутих шаблонів за обраною темою

Опис шаблону

Шаблон Visitor у контексті FTP-сервера

Шаблон **Visitor** дозволяє визначити окрему операцію для об'єктів, не змінюючи їх самих. Це досягається завдяки створенню окремого об'єкта-відвідувача, який може виконувати певні дії з об'єктами класів File та Directory (або іншими класами), не змінюючи їх внутрішню структуру. У контексті FTP-сервера цей шаблон використовуються для перевірки доступу до файлів і каталогів.

Як працює шаблон Visitor в FTP-сервері:

1. **Визначення об'єктів, які приймають відвідувачів (Visitable)**
    - Класи **File** і **Directory** реалізують інтерфейс Visitable, що дозволяє відвідувати їх іншими об'єктами, такими як перевірка прав доступу.
    - Кожен з цих класів має метод accept, який приймає відвідувача та передає йому себе для виконання відповідної операції.
2. **Операції для відвідувача (Visitor)**
    - Клас **PermissionCheckVisitor** реалізує інтерфейс Visitor і визначає операції для перевірки прав доступу користувача до файлів та каталогів. Для цього він використовує методи класів **File** і **Directory** для перевірки прав доступу користувача.
    - Кожен конкретний відвідувач (наприклад, для перевірки прав доступу) реалізує методи **visitFile** та **visitDirectory**, що виконують перевірку доступу до відповідного ресурсу (файл чи каталог).
3. **Поглиблення доступу**
    - Замість того, щоб кожен об'єкт **File** або **Directory** мав свою логіку для перевірки доступу, це робить окремий відвідувач, який перевіряє права на доступ до ресурсів централізовано, забезпечуючи гнучкість.
    - Відвідувач може додавати нові види операцій без необхідності змінювати класи **File** чи **Directory**, що полегшує розширення системи.

Компоненти реалізації шаблону Visitor:

1. **Visitable (Інтерфейс)**
    - Клас **File** та клас **Directory** реалізують цей інтерфейс. Метод **accept** дозволяє передавати об'єкт до відвідувача для виконання операції над ним.
    - Це дає змогу зовнішньому об'єкту (відвідувачу) виконувати операції над ресурсами, не змінюючи їх внутрішню структуру.
2. **Visitor (Інтерфейс)**
    - Визначає методи **visitFile** та **visitDirectory**, які виконують специфічні дії для файлів і каталогів.
    - У випадку FTP-сервера це буде перевірка прав доступу користувача до файлів та каталогів.
3. **PermissionCheckVisitor (Конкретний відвідувач)**
    - Реалізує логіку перевірки доступу до файлів і каталогів.
    - Використовує клас **Permission** для визначення доступу для конкретного користувача, перевіряючи доступ на основі прав користувача, власника та групи файлу чи каталогу.
4. **Permission (Клас для прав доступу)**
    - Визначає, чи має користувач доступ до ресурсу (файлу чи каталогу). Для цього клас **Permission** використовує метод **hasAccess**, який перевіряє права доступу за умовами прав користувача, власника та групи.

Переваги використання шаблону Visitor:

- **Розширюваність**: Завдяки шаблону **Visitor** додавання нових операцій не вимагає змін в класах **File** та **Directory**. Це знижує складність підтримки та розширення коду.
- **Чіткий поділ відповідальності**: Клас **File** і клас **Directory** відповідають лише за свою структуру, тоді як операції (перевірка доступу) реалізуються окремо через відвідувачів.
- **Гнучкість**: Якщо потрібно додати нову операцію, наприклад, змінити логіку перевірки доступу, це робиться через реалізацію нового відвідувача без зміни існуючих класів.

- **Зменшення дублювання**: Шаблон дозволяє уникнути дублювання коду, оскільки перевірка доступу виконується в єдиному місці (у відвідувачі), а не в кожному об'єкті (файл чи каталог).

Проблеми, які вирішує шаблон Visitor:

- Без шаблону **Visitor** кожен клас **File** і **Directory** мав би свою реалізацію для перевірки доступу, що призводить до дублювання коду та ускладнює підтримку. Шаблон **Visitor** вирішує цю проблему, централізуючи логіку перевірки доступу в окремому класі відвідувачі.
- Це також спрощує додавання нових операцій, не вносячи змін у самі об'єкти.

Таким чином, шаблон **Visitor** забезпечує гнучкість, розширюваність і зручність при роботі з файлами та каталогами на FTP-сервері, дозволяючи виконувати операції (наприклад, перевірку доступу) без зміни самих об'єктів файлів і каталогів.

Діаграма класів

Діаграма класів, які реалізовують паттерн Visitor зображена на рисунку 2

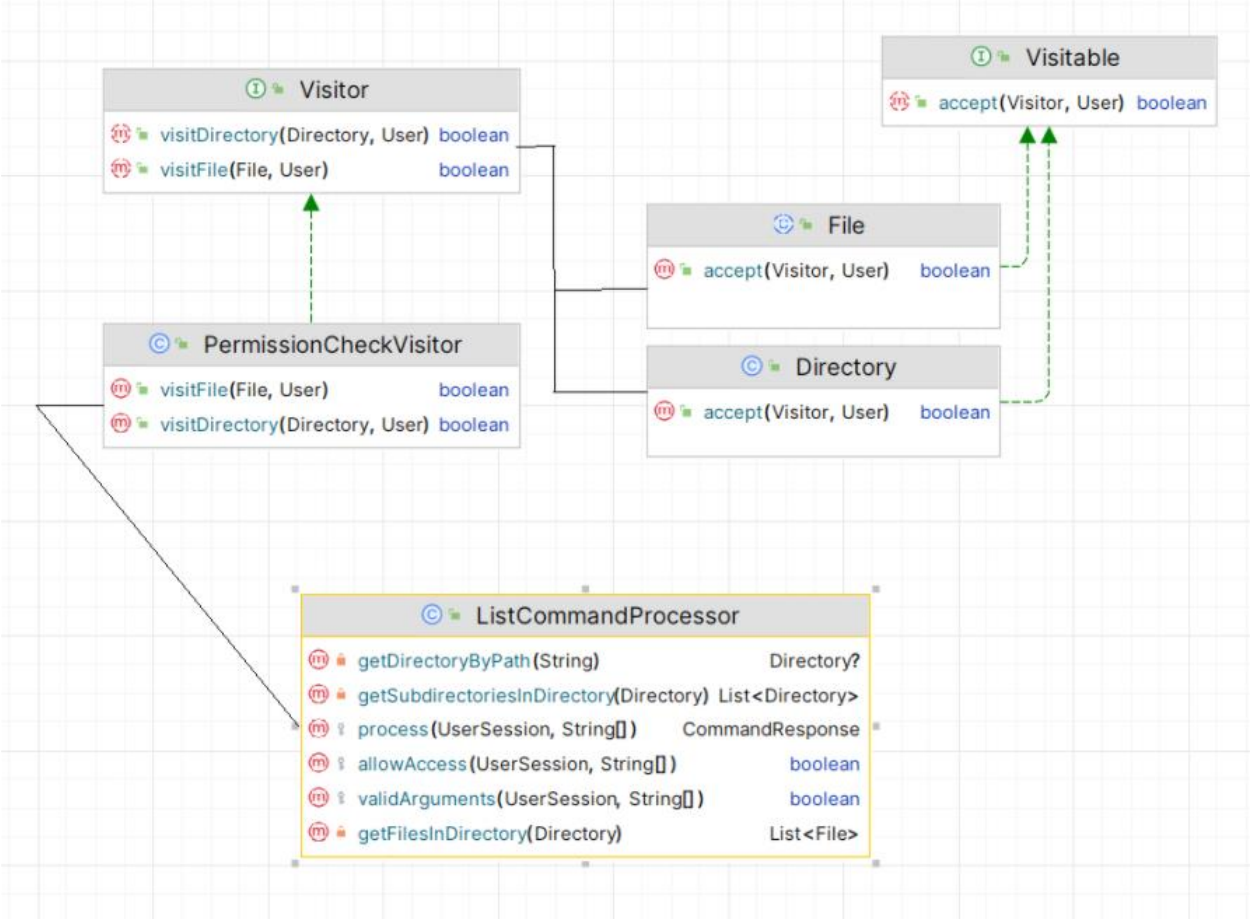

Рисунок 2 – реалізація паттерну Visitor

Висновки та код

Код можна знайти за посиланням - https://github.com/B1lok/trpz

Висновок: У ході виконання лабораторної роботи було реалізовано шаблон проектування **Visitor**. Це дозволило централізувати виконання операцій над об'єктами (такими як файли та каталоги), виділивши логіку перевірки доступу в окремому класі відвідувачі. Завдяки цьому, кожен об'єкт не потребує власної реалізації для обробки операцій, що зменшує дублювання коду та спрощує його підтримку.

Реалізований підхід продемонстрував переваги гнучкості, масштабованості та легкості розширення системи. Додавання нових операцій, таких як перевірка інших прав доступу або нових типів операцій, можна здійснити без змін у класах об'єктів (файлів та каталогів), що значно підвищує читаємість і зручність підтримки коду.