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 №5

Технології розроблення програмного забезпечення

ШАБЛОНИ «ADAPTER», «BUILDER», «COMMAND», «CHAIN OF RESPONSIBILITY», «PROTOTYPE»

FTP-server

Виконав

студент групи ІА–22:

Білокур Євгеній

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» 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.

1. Короткі теоритичні 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 DOM-дерева. 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.

Реалізувати не менше 3-х класів відповідно до обраної теми

Структура класів

Структура проекту з реалізованими класами зображена на рисунку 1

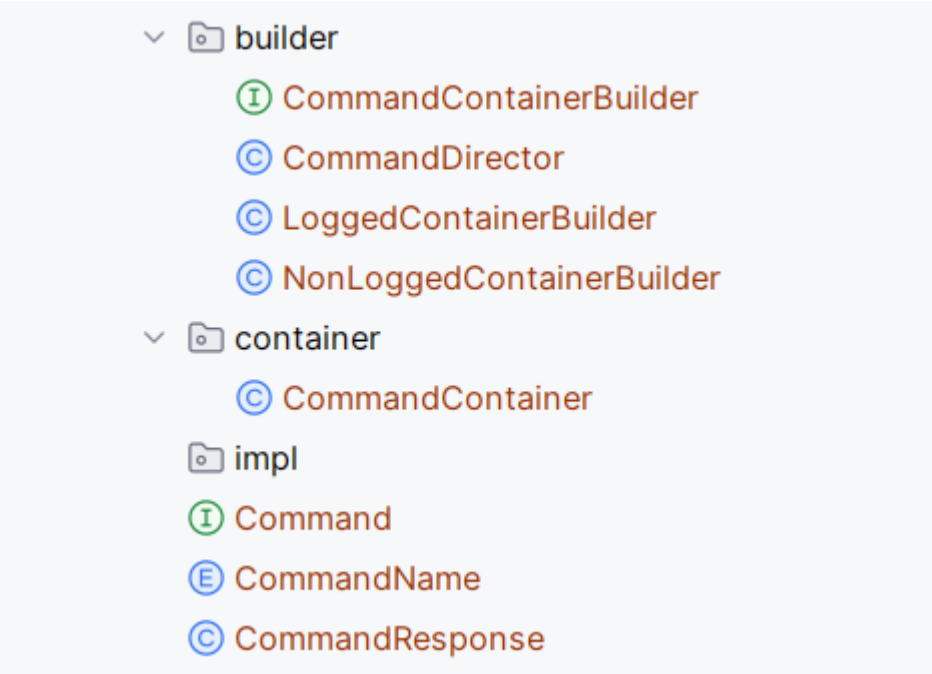

Рисунок 1 – Структура реалізованих класів

Опис класів

1. **CommandContainerBuilder** (інтерфейс) — визначає контракт для побудови об’єктів типу CommandContainer. Цей інтерфейс містить метод build(), який реалізується в класах-нащадках для створення екземпляра CommandContainer.
2. **LoggedContainerBuilder** і **NonLoggedContainerBuilder** (класи) — конкретні реалізації інтерфейсу CommandContainerBuilder. Вони використовуються для побудови різних конфігурацій CommandContainer. Наприклад, LoggedContainerBuilder може додавати команди, доступні лише для авторизованих користувачів, тоді як NonLoggedContainerBuilder — для неавторизованих.
3. **CommandContainer** — клас, який виступає як "контейнер команд" і є кінцевим продуктом, що створюється за допомогою CommandContainerBuilder. У цьому класі також є вкладений клас Builder, який є реалізацією шаблону Builder. Вкладений Builder містить методи addCommand і setUnknownCommand, які додають команди до контейнера, а також метод build, що повертає екземпляр CommandContainer.
4. **CommandDirector** — клас, який відповідає за управління процесом побудови об'єкта CommandContainer. Він приймає на вхід об'єкт CommandContainerBuilder і використовує його для створення контейнера команд.

Загалом, **CommandContainerBuilder**, **LoggedContainerBuilder**, **NonLoggedContainerBuilder**, **CommandContainer.Builder**, і **CommandDirector** разом реалізують шаблон Builder для побудови різних конфігурацій контейнера команд CommandContainer

Реалізувати один з розглянутих шаблонів за обраною темою

Опис шаблону

У нашій реалізації шаблону Builder є такі основні компоненти:

1. **Інтерфейс CommandContainerBuilder** — визначає методи, необхідні для побудови об’єкта CommandContainer. Він визначає загальні кроки побудови для різних конфігурацій об'єкта CommandContainer.
2. **Класи LoggedContainerBuilder і NonLoggedContainerBuilder** — конкретні реалізації інтерфейсу CommandContainerBuilder. Ці класи використовуються для побудови контейнера команд для авторизованих (LoggedContainerBuilder) і неавторизованих (NonLoggedContainerBuilder) користувачів. У кожному класі визначено конкретні дії для додавання потрібних команд в CommandContainer залежно від типу користувача.
3. **Клас CommandContainer** — кінцевий продукт, який будується за допомогою CommandContainerBuilder. Він є контейнером для команд FTP-сервера, і його можна налаштовувати за допомогою Builder. CommandContainer має внутрішній клас Builder, який надає зручний інтерфейс для побудови об'єкта CommandContainer.
4. **Клас CommandDirector** — координує процес побудови CommandContainer, приймаючи на вхід об’єкт CommandContainerBuilder. CommandDirector визначає послідовність викликів, які необхідно зробити для побудови повного контейнера команд. Цей клас дозволяє легко змінювати побудову об'єкта без змін у класах, які його будують.

Проблема, яку допоміг вирішити шаблон Builder:

До використання цього паттерну створення об'єктів CommandContainer із різними конфігураціями могло бути складним і вимагало б великої кількості конструкторів або довгих умовних операторів. За допомогою Builder стало можливим створення конфігурованих об'єктів CommandContainer через простий та зрозумілий інтерфейс.

Переваги застосування паттерну Builder:

1. **Гнучкість**: Можна створювати об'єкти з різними конфігураціями, не засмічуючи код великим числом конструкторів.

2. **Чистий код**: Розподілення коду побудови на окремі класи (LoggedContainerBuilder, NonLoggedContainerBuilder) забезпечує зрозумілість і підтримуваність.
3. **Розширюваність**: Легко додати новий тип контейнера команд, реалізувавши новий Builder-клас без змін у вже існуючому коді.

Діаграма класів

Діаграма класів, які реалізовують паттерн Builder зображена на рисунку 2

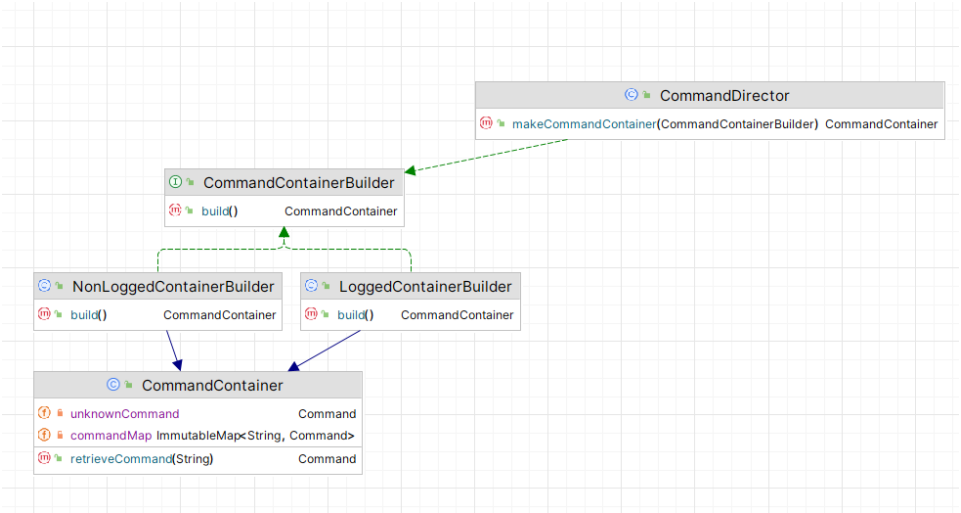

Рисунок 2 – реалізація паттерну Builder

Висновки та код

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

Висновок: У ході виконання лабораторної роботи було розглянуто і реалізовано шаблон проектування Builder для покращення архітектури FTP-сервера. Шаблон Builder спростив створення складних об'єктів із різними конфігураціями, у цьому випадку — контейнерів команд для різних станів користувача. Завдяки Builder'у було зменшено кількість конструкцій if-else і поліпшено структуру коду, що дозволяє легко додавати нові типи контейнерів без значних змін у коді системи.