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 №**6**
з предмету «Технології розроблення програмного забезпечення»
на тему «**Шаблони “Abstract Factory”, “Factory Method”, “Memento”, “Observer”, “Decorator”**»

Виконав
студент групи ІА-23:
Воронюк Є. В.

Перевірив:
Мягкий М. Ю.

Київ 2024

Зміст

Мета: реалізація частини функціоналу робочої програми у вигляді класів та їхньої взаємодії із застосуванням одного із розглянутих шаблонів проектування.

Завдання 1: ознайомитися з короткими теоретичними відомостями.
Завдання 2: реалізувати частину функціонала робочої програми у вигляді класів і їх взаємодій для досягнення конкретних функціональних можливостей.
Завдання 3: застосування одного з даних шаблонів при реалізації програми.

Тема для виконання:

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:

Шаблон «Abstract Factory»

Шаблон "абстрактна фабрика" використовується для створення сімейств об'єктів без вказівки їх конкретних класів. Для цього виноситься загальний інтерфейс фабрики (AbstractFactory) і створюються його реалізації для різних сімейств продуктів. Хорошим прикладом використання абстрактної фабрики є ADO.NET: існує загальний клас DbProviderFactory, здатний створювати об'єкти типів DbConnection, DbDataReader, DbAdapter та ін.; існують реалізації цих фабрик і об'єктів - SqlProviderFactory, SqlConnection, SqlDataReader, SqlAdapter і так далі. Відповідно, якщо додатку необхідно працювати з різними базами даних (чи потрібна така можливість), то досить використати базові реалізації (Db.) і підставити відповідну фабрику у момент ініціалізації фабрики (Factory = new SqlProviderFactory()).

Цей шаблон передусім структурує знання про схожі об'єкти (що називаються сімействами, як класи для доступу до БД) і створює можливість взаємозаміни різних сімейств (робота з Oracle ведеться також, як і робота з SQL Server). Проте, при використанні такої схеми украй незручно розширювати фабрику - для додавання нового методу у фабрику необхідно додати його в усіх фабриках і створити відповідні класи, що створюються цим методом.

Переваги та недоліки:

+ Гарантує поєднання створюваних продуктів.
+ Звільняє клієнтський код від прив’язки до конкретних класів продукту.
+ Реалізує принцип відкритості/закритості.
- Вимагає наявності всіх типів продукту в кожній варіації.
- Ускладнює код програми внаслідок введення великої кількості додаткових класів.

Шаблон «Factory Method»

Шаблон "фабричний метод" визначає інтерфейс для створення об'єктів певного базового типу. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу. Тому шаблон "фабричний метод" носить ще назву "Віртуальний конструктор".

Розглянемо простий приклад. Нехай наше застосування працює з мережевими драйверами та використовує клас Packet для зберігання даних, що передаються в мережу. Залежно від використовуваного протоколу, існує два перевантаження - TcpPacket, UdpPacket. І відповідно два створюючі об'єкти (TcpCreator, UdpCreator) з фабричним методом (який створює відповідні реалізації).

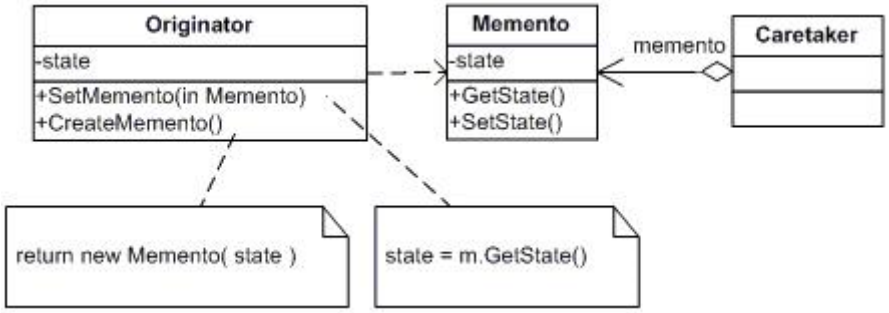
Проте базова функціональність (передача пакету, прийом пакету, заповнення пакету даними) нічим не відрізняється один від одного, відповідно поміщається у базовий клас PacketCreator. Таким чином поведінка системи залишається такою ж, проте з'являється можливість підстановки власних об'єктів в процес створення і роботи з пакетами.

Переваги та недоліки:

+ Позбавляє клас від прив’язки до конкретних класів продуктів.
+ Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
+ Спрощує додавання нових продуктів до програми.
- Може призвести до створення великих паралельних ієрархій класів.

Шаблон «Memento»

Структура:



Шаблон використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції. Об'єкт "мементо" служить виключно для збереження змін над початковим об'єктом (Originator). Лише початковий об'єкт має можливість зберігати і отримувати стан об'єкту "мементо" для власних цілей, цей об'єкт є "порожнім" для кого-небудь ще. Об'єкт Caretaker використовується для передачі і зберігання мементо об'єктів в системі. Таким чином вдається досягти наступних цілей:

- Зберігання стану повністю відділяється від початкових об'єктів, що полегшує їх реалізацію;
- Передача об'єктів мементо лягає на плечі Caretaker об'єктів, що дозволяє гнучкіше управляти станами об'єктів і спростити дизайн класів початкових об'єктів;
- Збереження і відновлення стану реалізовані у вигляді двох простих методів і є закритими для кого-небудь ще окрім початкових об'єктів, таким чином не порушуючи інкапсуляцію.

Шаблон «мементо» дуже зручно використати разом з шаблоном «команда» для реалізації «скасовних» дій – дані про дію зберігаються в мементо, а команда має можливість вважати і відновити початкове положення відповідних об’єктів.

Проблема:

Припустімо, ви пишете програму текстового редактора. Крім звичайного редагування, ваш редактор дозволяє змінювати форматування тексту, вставляти малюнки та інше.

В певний момент ви вирішили надати можливість скасовувати усі ці дії. Для цього вам потрібно зберігати поточний стан редактора перед тим, як виконати будь-яку дію. Якщо користувач вирішить скасувати свою дію, ви візьмете копію стану з історії та відновите попередній стан редактора.

Перед виконанням команди ви можете зберегти копію стану редактора, щоб потім мати можливість скасувати операцію.

Щоб зробити копію стану об’єкта, достатньо скопіювати значення полів. Таким чином, якщо ви зробили клас редактора достатньо відкритим, то будь-який інший клас зможе зазирнути всередину, щоб скопіювати його стан.

Здавалося б, які проблеми? Тепер будь-яка операція зможе зробити резервну копію редактора перед виконанням своєї дії. Але такий наївний підхід забезпечить вам безліч

проблем у майбутньому. Адже, якщо ви вирішите провести рефакторинг — прибрати або додати кілька полів до класу редактора — доведеться змінювати код усіх класів, які могли копіювати стан редактора.

Рішення:

Усі проблеми, описані вище, виникають через порушення інкапсуляції, коли одні об’єкти намагаються зробити роботу за інших, проникаючи до їхньої приватної зони, щоб зібрати необхідні для операції дані.

Патерн Знімок доручає створення копії стану об’єкта самому об’єкту, який цим станом володіє. Замість того, щоб робити знімок «ззовні», наш редактор сам зробить копію своїх полів, адже йому доступні всі поля, навіть приватні.

Патерн пропонує тримати копію стану в спеціальному об’єкті-знімку з обмеженим інтерфейсом, що дозволяє, наприклад, дізнатися дату виготовлення або назву знімка. Проте, знімок повинен бути відкритим для свого творця і дозволяти прочитати та відновити його внутрішній стан.

Знімок повністю відкритий для творця, але лише частково відкритий для опікунів.

Така схема дозволяє творцям робити знімки та віддавати їх на зберігання іншим об’єктам, що називаються опікунами. Опікунам буде доступний тільки обмежений інтерфейс знімка, тому вони ніяк не зможуть вплинути на «нутрощі» самого знімку. У потрібний момент опікун може попросити творця відновити свій стан, передавши йому відповідний знімок.

У нашому прикладі з редактором опікуном можна зробити окремий клас, який зберігатиме список виконаних операцій. Обмежений інтерфейс знімків дозволить демонструвати користувачеві гарний список з назвами й датами виконаних операцій. Коли ж користувач вирішить скасувати операцію, клас історії візьме останній знімок зі стека та надішле його об’єкту редактора для відновлення.

Переваги та недоліки:

+ Не порушує інкапсуляцію вихідного об’єкта.

+ Спрощує структуру вихідного об’єкта. Не потрібно зберігати історію версій свого стану.

- Вимагає багато пам’яті, якщо клієнти дуже часто створюють знімки.

- Може спричинити додаткові витрати пам’яті, якщо об’єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.

Шаблон «Observer»

Шаблон визначає залежність "один-до-багатьох" таким чином, що коли один об'єкт змінює власний стан, усі інші об'єкти отримують про це сповіщення і мають можливість змінити власний стан також.

Розглянемо цей шаблон на прикладі. Припустимо, є деяка банківська система і декілька користувачів переглядають баланс на рахунку пана И. У цей момент пан И. кладе на свій рахунок деяку суму, яка міняє загальний баланс. Кожен з користувачів, що переглядали баланс, отримує про це звістку (для користувачів ця звістка може бути прозорою - просто зміна цифр, або попередження про те, що баланс змінився). Раніше неможливі дії для користувачів (переведення до іншої категорії клієнтів і тому подібне) стають доступними.

Цей шаблон дуже широко поширений в шаблоні MVVM і механізмі "прив'язок" (bindings) в WPF і частково в WinForms. Інша назва шаблону - підписка/розсилка. Кожен з оглядачів власноручно підписується на зміни конкретного об'єкту, а об'єкти зобов'язані сповіщати своїх передплатників про усі свої зміни (на даний момент конкретних механізмів автоматичного сповіщення про зміну стану в .NET мовах не існує).

Переваги та недоліки:

+ Ви можете підписувати і відписувати одержувачів «на льоту».
+ Видавці не залежать від конкретних класів підписників і навпаки.
+ Реалізує принцип відкритості/закритості.
- Підписники сповіщуються у випадковій послідовності.

Шаблон «Decorator»

Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. Декоратор деяким чином "обертає" (агрегація) початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому.

Простим прикладом є накладення смуги прокрутки до усіх візуальних елементів. Кожен об'єкт, який може прокручуватися, обертається в прокручуваному елементі, і при необхідності з'являється смуга прокрутки. Початкові функції елементу (наприклад, рядки статусу) залишаються незмінними.

Переваги та недоліки:

+ Дозволяє мати кілька дрібних об’єктів, замість одного об’єкта «на всі випадки життя».
+ Дозволяє додавати обов’язки «на льоту».
+ Більша гнучкість, ніж у спадкування.
- Велика кількість крихітних класів.
- Важко конфігурувати об’єкти, які загорнуто в декілька обгорток одночасно.

Хід роботи::

Реалізація патерну:

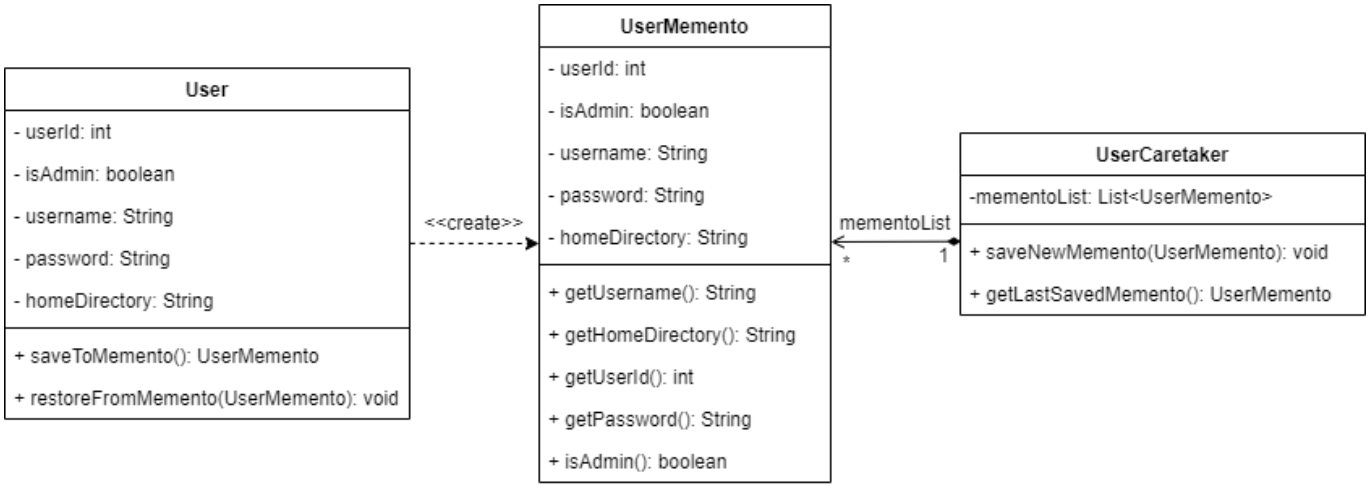


Рисунок 1 – реалізація патерну Memento

Опис реалізації патерну:

Патерн Memento використовується для забезпечення можливості зміни паролю чи імені користувача та відкатування до попереднього стану (стану до змін). Зберігання стану об'єкта та відновлення його попереднього стану дає можливість користувачам безпечно змінювати свої облікові дані – юзернейм та пароль, та скасовувати небажані зміни. Зміна облікових даних на сервері відбувається за допомогою команди «ALTER», а повернення попередніх даних – за допомогою команди «RESTORE».

Опис класів на діаграмі:

User (Originator): клас, стан якого необхідно зберігати та відновлювати. Містить поля, які описують поточний стан користувача – userId, isAdmin, username, password, homeDirectory.

Методи: saveToMemento():UserMemento — створює та повертає об'єкт UserMemento, який містить копію поточного стану користувача. restoreFromMemento (UserMemento): void — відновлює стан користувача на основі переданого мементо (UserMemento).

UserMemento: клас, що інкапсулює стан об’єкта User на момент його збереження. Містить тільки ті поля від юзера, які необхідно зберегти для відновлення стану. Забезпечує іммутабельність, тому що дозволяє лише отримувати значення – гетери, а не замінювати їх.

Методи: гетери для доступу до полів – getUserId(), getUsername(), getPassword(), getHomeDirectory(), isAdmin().

UserCaretaker: клас керує збереженням і доступом до об’єктів UserMemento. Має поле mementoList – список збережених станів, та методи: saveNewMemento (UserMemento): void – додає новий стан користувача до списку; getLastSave Memento(): UserMemento – повертає останній збережений стан користувача.

Логіка роботи:

Збереження стану – об'єкт User створює UserMemento через метод saveToMemento(). UserCaretaker зберігає цей об'єкт у своєму списку за допомогою saveNewMemento().

Відновлення стану - UserCaretaker повертає останній збережений стан через getLastSavedMemento(). Об'єкт User використовує цей об'єкт для відновлення свого стану через метод restoreFromMemento().

Вихідний код системи:

Вихідний код системи розміщено на GitHub: https://github.com/dobrogo-dnia/SDT-labs

Висновок: під час виконання даної лабораторної роботи було реалізовано частину функціоналу робочої програми у вигляді класів та їхньої взаємодії із застосуванням одного із розглянутих шаблонів проектування – а саме шаблону Memento. Існуючий код було оновлено для використання відповідного патерну.