Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №6

з дисципліни «Технології розроблення програмного забезпечення»

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

Варіант №15

Виконав:
студент групи ІА-23
Лядський Д.С.

Перевірив:
Мягкий М. Ю.

Київ 2024

Зміст

Тема

Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»

Мета

Дослідити та практично реалізувати шаблони проєктування: «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator». Ознайомитися з принципами їх використання, перевагами та недоліками в розробці програмного забезпечення. Навчитися застосовувати дані шаблони для розв'язання типових задач, пов'язаних зі створенням, збереженням стану, спостереженням за подіями та динамічним розширенням функціональності програм.

Короткі теоретичні відомості

Принципи проектування SOLID

Принципи проектування SOLID були введені Робертом Мартіном в його статті "Об'єктно-орієнтоване 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»

Призначення патерну: 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: существу-ет 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 под-ставить 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.

Проблема:

Уявіть, що ви пишете симулятор меблевого магазину. Ваш код містить:

1. Сімейство залежних продуктів. Скажімо, Крісло+Диван+Столик.

2. Кілька варіацій цього сімейства. Наприклад, продукти Крісло, Диван та Столик представлені в трьох різних стилях: Ар-деко,Вікторіанському і Модерн.

Вам потрібно створювати об’єкти продуктів у такий спосіб, щоб вони завжди пасували до інших продуктів того самого сімейства. Це дуже важливо, адже клієнти засмучуються, коли отримують меблі, що не можна поєднати між собою.

Рішення:

Для початку, патерн Абстрактна фабрика пропонує виділити загальні інтерфейси для окремих продуктів, що складають одне сімейство, і описати в них спільну для цих продуктів поведінку. Так, наприклад, усі варіації крісел отримають спільний інтерфейс Крісло, усі дивани реалізують ін терфейс Диван , тощо.

Всі варіації одного й того самого об’єкта мають жити в одній ієрархії класів. Конкретні фабрики відповідають певній варіації сімейства продуктів. Для кожної варіації сімейства продуктів ми повинні створити свою власну фабрику, реалізувавши абстрактний інтерфейс. Фабрики створюють продукти однієї варіації. Клієнтський код повинен працювати як із фабриками, так і з продуктами тільки через їхні загальні інтерфейси. Це дозволить подавати у ваші класи будь-які типи фабрик і виробляти будь-які типи продуктів, без необхідності вносити зміни в існуючий код.

Приклад з життя: Припустимо, ви вирішили повністю взяти під свій контроль ринок автомобілів. Як це зробити? Ви можете створити свою марку автомобіля, своє виробництво, провести масштабну рекламну компанію і т.д. Але, в цьому випадку вам доведеться битися з такими гігантами авторинку, як Toyota або Ford. Не факт, що з цієї боротьби ви вийдіть переможцем. Набагато кращим рішенням буде скупити заводи всіх цих компаній, продовжити випускати автомобілі під їх власними марками, а прибуток класти собі в кишеню.

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»

Призначення: 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 драйвер-мі і 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 тим же, проте з'являється можливість підстановки власних об'єктів в процес створення і роботи з пакетами.

Проблема: Уявіть, що ви створюєте програму керування вантажними перевезеннями. Спочатку ви плануєте перевезення товарів тільки вантажними автомобілями. Тому весь ваш код працює з об’єктами класу Вантажівка.

Згодом ваша програма стає настільки відомою, що морські перевізники шикуються в чергу і благають додати до програми підтримку морської логістики.

Додати новий клас не так просто, якщо весь код вже залежить від конкретних класів. Чудові новини, чи не так?! Але як щодо коду? Велика частина існуючого коду жорстко прив’язана до класів Вантажівок. Щоб додати до програми класи морських Суден, знадобиться перелопачувати весь код. Якщо ж ви вирішите додати до програми ще один вид транспорту, тоді всю цю роботу доведеться повторити.

У підсумку ви отримаєте жахливий код, переповнений умовними операторами, що виконують ту чи іншу дію в залежності від вибраного класу транспорту.

Рішення: Патерн Фабричний метод пропонує відмовитись від безпосереднього створення об’єктів за допомогою оператора new, замінивши його викликом особливого фабричного методу. Не лякайтеся, об’єкти все одно будуть створюватися за допомогою new, але робити це буде фабричний метод.

Щоб ця система запрацювала, всі об’єкти, що повертаються, повинні мати спільний інтерфейс. Підкласи зможуть виготовляти об’єкти різних класів, що відповідають одному і тому самому інтерфейсу. Наприклад, класи Вантажівка і Судно реалізують інтерфейс Транспорт з методом доставити. Кожен з цих класів реалізує метод по-своєму: вантажівки перевозять вантажі сушею, а судна — морем. Фабричний метод класу Дорожної Логістики поверне об’єкт вантажівку, а класу Морської Логістики — об’єкт-судно.

Приклад з життя: Розглянемо приклад з менеджером по найму. Неможливо одній людині провести співбесіди з усіма кандидатами на всі вакансії. Залежно від вакансії він повинен розподілити етапи співбесіди між різними людьми. Простими словами: Менеджер надає спосіб делегування логіки створення екземпляра дочірнім класами.

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»

Призначення: 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 ментo 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 "ментo" дуже зручно використати разом з шаблоном "команда" для реалізації "скасовних" дій - дані про дію зберігаються в ментo, 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. 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.

Приклад з життя: Коли вам потрібно зберігати миттєві знімки стану об’єкта (або його частини) для того, щоб об’єкт можна було відновити в тому самому стані. Патерн Знімок дозволяє створювати будь-яку кількість знімків об’єкта і зберігати їх незалежно від об’єкта, з якого роблять знімок. Знімки часто використовують не тільки для реалізації операції скасування, але й для транзакцій, коли стан об’єкта потрібно «відкотити», якщо операція не була вдалою.

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» С

Шаблон визначає залежність "один-ко-многим" 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).

Проблема: Уявіть, що ви маєте два об’єкти: Покупець і Магазин. До магазину мають ось-ось завезти новий товар, який цікавить покупця. Покупець може щодня ходити до магазину, щоб перевіряти наявність товару. Але через це він буде дратуватися, даремно витрачаючи свій дорогоцінний час. З іншого боку, магазин може розсилати спам кожному своєму покупцеві. Багатьох покупців це засмутить, оскільки товар специфічний і потрібний не всім.

Виходить конфлікт: або покупець гає час на періодичні перевірки, або магазин розтрачує ресурси на непотрібні сповіщення.

Рішення: Давайте називати Видавцями ті об’єкти, які містять важливий або цікавий для інших стан. Решту об’єктів, які хотіли б відстежувати зміни цього стану, назвемо Підписниками. Патерн Спостерігач пропонує зберігати всередині об’єкта видавця список посилань на об’єкти підписників. Причому видавець не повинен вести список підписки самостійно. Він повинен надати методи, за допомогою яких підписники могли б додавати або прибирати себе зі списку. Тепер найцікавіше. Коли у видавця відбуватиметься важлива подія, він буде проходитися за списком передплатників та сповіщувати їх про подію, викликаючи певний метод об’єктів-передплатників. Побачивши, як добре все працює, ви можете виділити загальний інтерфейс і для всіх видавців, який буде складатися з методів підписки та відписки. Після цього підписники зможуть працювати з різними типами видавців, і отримувати від них сповіщення через єдиний метод.

Приклад з життя: Після того, як ви оформили підписку на журнал, вам більше не потрібно їздити до супермаркета та дізнаватись, чи вже вийшов черговий номер. Натомість видавництво надсилатиме нові номери поштою прямо до вас додому, відразу після їхнього виходу. Видавництво веде список підписників і знає, кому який журнал слати. Ви можете в будь-який момент відмовитися від підписки, й журнал перестане до вас надходити.

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»

Призначення: 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.

Проблема: Ви працюєте над бібліотекою сповіщень, яку можна підключати до різноманітних програм, щоб отримувати сповіщення про важливі події. Основою бібліотеки є клас Notifier з методом send, який приймає на вхід рядок повідомлення і надсилає його всім адміністраторам електронною поштою. Стороння програма повинна створити й налаштувати цей об’єкт, вказавши, кому надсилати сповіщення, та використовувати його щоразу, коли щось відбувається. В якийсь момент стало зрозуміло, що користувачам не вистачає одних тільки emailсповіщень. Деякі з них хотіли б отримувати сповіщення про критичні проблеми через SMS. Інші хотіли б отримувати їх у вигляді Facebook-повідомлень. Корпоративні користувачі хотіли би бачити повідомлення у Slack. Спершу ви додали кожен з типів сповіщень до програми, успадкувавши їх від базового класу Notifier. Тепер користувачі могли вибрати один з типів сповіщень, який і використовувався надалі. Але потім хтось резонно запитав, чому не можна увімкнути кілька типів сповіщень одночасно? Адже, якщо у вашому будинку раптом почалася пожежа, ви б хотіли отримати сповіщення по всіх каналах, чи не так? Ви зробили спробу реалізувати всі

можливі комбінації підкласів сповіщень, але після того, як додали перший десяток класів, стало зрозуміло, що такий підхід неймовірно роздуває код програми. Рішення: Одним зі способів, що дозволяє обійти ці проблеми, є заміна спадкування агрегацією або композицією . Це той випадок, коли один об’єкт утримує інший і делегує йому роботу, замість того, щоб самому успадкувати його поведінку. Саме на цьому принципі побудовано патерн Декоратор. Декоратор має альтернативну назву — обгортка. Вона більш вдало описує суть патерна: ви розміщуєте цільовий об’єкт у іншому об’єкті-обгортці, який запускає базову поведінку об’єкта, а потім додає до результату щось своє.

Приклад з життя: Будь-який одяг — це аналог Декоратора. Застосовуючи Декоратор, ви не змінюєте початковий клас і не створюєте дочірніх класів. Так само з одягом: вдягаючи светра, ви не перестаєте бути собою, але отримуєте нову властивість — захист від холоду. Ви можете піти далі й одягти зверху ще один декоратор — плащ, щоб захиститися від дощу.

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

Завдання

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціонала робочої програми у вигляді класів і їх взаємодій для досягнення конкретних функціональних можливостей.

3. Застосування одного з даних шаблонів при реалізації програми.

Обрана тема

15 E-mail клієнт (singleton, builder, decorator, template method, interpreter, SOA)

Поштовий клієнт повинен нагадувати функціонал поштових програм Mozilla Thunderbird, The Bat і т.д. Він повинен сприймати і коректно обробляти pop3/smtp/imap протоколи, мати функції автонастройки основних поштових провайдерів для України (gmail, ukr.net, i.ua), розділяти повідомлення на папки/категорії/важливість, зберігати чернетки незавершених повідомлень, прикріплювати і обробляти прикріплені файли.

Хід роботи

Реалізація патерну Decorator:

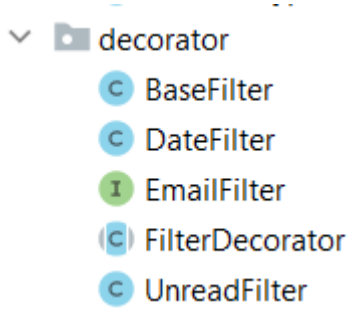

Рисунок 1. Структура класів при реалізації патерну Decorator

```
5   import java.util.List;
6
    9 usages   4 implementations
7   public interface EmailFilter {
        4 implementations
8       List<EmailMessage> filter(List<EmailMessage> emails);
9   }
10
```

Рисунок 2. Інтерфейс EmailFilter

Цей інтерфейс містить метод filter, який приймає список електронних листів і повертає відфільтрований список.

```
import java.util.List;

2 usages  2 inheritors
abstract class FilterDecorator implements EmailFilter {
    protected EmailFilter filter;

    2 usages
    public FilterDecorator(EmailFilter filter) { this.filter = filter; }

    2 overrides
    @Override
    public List<EmailMessage> filter(List<EmailMessage> emails) { return filter.filter(emails); }
}
```

Рисунок 3. Абстрактний клас FilterDecorator

Цей клас містить посилання на інший об'єкт EmailFilter і делегує виконання методу filter.

```
2 usages
public class BaseFilter implements EmailFilter{
    @Override
    public List<EmailMessage> filter(List<EmailMessage> emails) {
        return emails;
    }
}
```

Рисунок 4. Клас BaseFilter

```java
import java.time.LocalDate;
import java.util.List;
import java.util.stream.Collectors;

2 usages
public class DateFilter extends FilterDecorator {
    2 usages
    private LocalDate startDate;
    2 usages
    private LocalDate endDate;

    1 usage
    public DateFilter(EmailFilter filter, LocalDate startDate, LocalDate endDate) {
        super(filter);
        this.startDate = startDate;
        this.endDate = endDate;
    }

    @Override
    public List<EmailMessage> filter(List<EmailMessage> emails) {
        return super.filter(emails).stream()
                .filter(email -> email.getSentDate().isAfter(startDate) && email.getSentDate().isBefore(endDate))
                .collect(Collectors.toList());
    }
}
```

Рисунок 5. Клас DateFilter

DateFilter додає додаткову фільтрацію до результату базового фільтра, враховуючи діапазон дат.

```java
import java.util.List;
import java.util.stream.Collectors;

2 usages
public class UnreadFilter extends FilterDecorator {
    1 usage
    public UnreadFilter(EmailFilter filter) { super(filter); }

    @Override
    public List<EmailMessage> filter(List<EmailMessage> emails) {
        return emails.stream()
                .filter(email -> email.getEmailStatus() != null && email.getEmailStatus().equals(EmailStatus.UNREAD))
                .collect(Collectors.toList());
    }
}
```

Рисунок 6. Клас UnreadFilter

Як це працює:

- Базовий фільтр (BaseFilter) спочатку повертає всі електронні листи.
- Фільтри-декоратори додають логіку:
  - UnreadFilter відбирає лише непрочитані листи.

  - DateFilter обмежує результат до заданого діапазону дат.
- Комбіновані фільтри дозволяють накладати кілька умов послідовно.

Цей підхід робить код гнучким, дозволяючи легко додавати нові фільтри без зміни існуючих класів.

Висновок

У ході виконання лабораторної роботи були досліджені шаблони проєктування «Abstract Factory», «Factory Method», «Memento», «Observer» та «Decorator».