Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського» Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Лабораторна робота № 4

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Вступ до паттернів проектування»

Тема проекту: «26. Download Manager»

Виконала:

студентка групи ІА-34

Мушта Анна

Перевірив:

Мягкий Михайло Юрійович

Дата здачі 13.10.2025

Захищено з балом

Київ 2025

Зміст

Тема: Download manager (iterator, command, observer, template method, composite, p2p) Інструмент для скачування файлів з інтернету по протоколах http або https з можливістю продовження завантаження в зупиненому місці, розподілу швидкостей активним завантаженням, ведення статистики завантажень, інтеграції в основні браузери (firefox, opera, internet explorer, chrome)

1.1. Теоретичні відомості

Шаблони проєктування (патерни) — це типові рішення часто повторюваних задач проєктування програмних систем. Вони забезпечують структурований підхід до створення архітектури, полегшують підтримку, масштабування та спілкування між розробниками. Використання шаблонів підвищує гнучкість і стійкість систем до змін вимог.

Singleton (Одинак)

Забезпечує існування лише одного екземпляра класу та надає глобальну точку доступу до нього. Використовується для спільних ресурсів (наприклад, конфігурацій, підключень).

Переваги: гарантія одного екземпляра, зручність доступу.

Недоліки: порушує принцип єдиної відповідальності, ускладнює тестування, вважається антишаблоном.

Iterator (Ітератор)

Дозволяє послідовно обходити елементи колекції без розкриття її внутрішньої структури.

Переваги: уніфікований обхід різних типів колекцій, гнучкість у зміні способу перебору.

Недоліки: недоцільний для простих структур, де достатньо звичайного циклу.

Proxy (Проксі)

Створює об’єкт-заступник, який контролює доступ до реального об’єкта або додає додаткову логіку (наприклад, кешування чи контроль запитів).

Переваги: можна впровадити проміжний рівень без зміни клієнтського коду, спрощує керування ресурсами.

Недоліки: може знижувати продуктивність і ускладнювати відлагодження.

State (Стан)

Дозволяє змінювати поведінку об’єкта залежно від його внутрішнього стану. Кожен стан реалізується окремим класом, що робить систему гнучкішою.

Переваги: спрощення коду контексту, легке додавання нових станів.

Недоліки: ускладнює структуру програми через велику кількість класів.

Strategy (Стратегія)

Дозволяє вибирати або змінювати алгоритм поведінки об’єкта під час виконання програми. Кожна стратегія оформлена окремим класом із єдиним інтерфейсом.

Переваги: можливість легко змінювати алгоритми, зменшення умовних операторів, покращення гнучкості коду.

Недоліки: ускладнення проєкту при невеликій кількості алгоритмів.

1.2. Хід роботи.

1. **Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.**

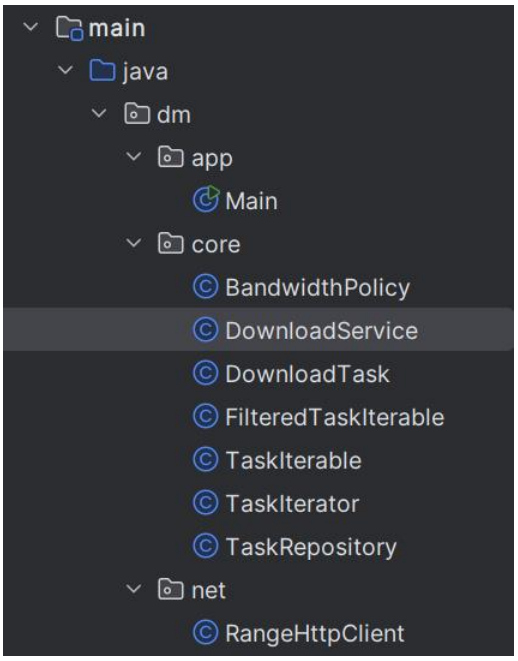

Рисунок 1 - Структура модулів і класів застосунку.

Проєкт розбитий на пакети dm.app, dm.core, dm.net. У dm.core реалізовано шаблон Iterator для обходу задач завантажень.

- DownloadTask - сутність задачі (id, url, target, status, прогрес).
- TaskRepository - доступ до SQLite; метод listRange(offset, limit) віддає записи батчами. Aggregate (джерело даних для ітератора).
- TaskIterable - колекція, що вміє створювати ітератор (iterator()); зберігає repo і розмір батча. Iterable.
- TaskIterator - послідовно підвантажує задачі через listRange, методи hasNext()/next(). Iterator.
- FilteredTaskIterable - обгортка над TaskIterable (наприклад, лише COMPLETED або PAUSED). Варіант ітератора з фільтрацією.
- DownloadService - керує паузою/відновленням, лімітом швидкості та надає доступ до TaskRepository.

- BandwidthPolicy — ліміт швидкості.
- RangeHttpClient — HTTP клієнт із підтримкою Range/відновлення.
- dm.app.Main — консольний інтерфейс; команда list використовує TaskIterable/FilteredTaskIterable для демонстрації шаблону.

Також шаблон Iterator застосовано для ефективного перегляду великої кількості задач з БД без завантаження всіх записів у пам’ять: TaskIterable/TaskIterator підвантажують їх батчами через TaskRepository.listRange(), а FilteredTaskIterable дозволяє додавати фільтри (наприклад, лише COMPLETED). Клієнтський код у Main працює з цією колекцією через звичний цикл for-each, не знаючи про внутрішню структуру сховища та спосіб вибірки.

2. Реалізувати шаблон iterator з не менш ніж 3-ма класами

Шаблон Iterator використано для безпечного та ефективного перегляду великої кількості задач завантаження, що зберігаються у SQLite. Рішення не завантажує всю таблицю в пам’ять одразу: TaskIterable/TaskIterator підтягують дані порційно (батчами) через метод репозиторію TaskRepository.listRange(...). Додатково запроваджено FilteredTaskIterable, який дозволяє застосовувати предикат (наприклад, показати лише COMPLETED чи PAUSED) без зміни клієнтського коду та без дублювання логіки вибірки. Клієнтський код у Main споживає колекцію через звичний цикл for-each, не знаючи про внутрішню структуру сховища та конкретний спосіб отримання записів.

```java
package dm.core;

public class TaskIterable implements Iterable<DownloadTask> {
    private final TaskRepository repo;  2 usages
    private final int batchSize;  2 usages

    public TaskIterable(TaskRepository repo, int batchSize) {
        this.repo = repo;
        this.batchSize = Math.max(1, batchSize);
    }

    @Override
    public java.util.Iterator<DownloadTask> iterator() {
        return new TaskIterator(repo, batchSize);
    }
}
```

Рисунок 2 - TaskIterable. Колекція, що створює ітератор (Iterable).

TaskIterable виконує роль Aggregate/Iterable: інкапсулює посилання на TaskRepository та обраний розмір батча. Об’єкт цього класу створює конкретний ітератор через метод iterator(), повертаючи інстанс TaskIterator. Такий поділ відповідальності дає змогу легко змінювати політику завантаження (наприклад, інший розмір батча) без втручання в клієнтський код. Крім того, клас є єдиною точкою входу для побудови різних варіацій ітераторів (звичайний чи фільтрований).

```
public class TaskIterator implements Iterator<DownloadTask> {  1 usage
    private final TaskRepository repo;  2 usages
    private final int batchSize;  2 usages
    private int cursor = 0;  2 usages
    private final List<DownloadTask> buffer = new ArrayList<>();  4 usages
    private boolean noMoreData = false;  3 usages

    public TaskIterator(TaskRepository repo, int batchSize) {  1 usage
        this.repo = repo;
        this.batchSize = batchSize;
    }

    @Override
    public boolean hasNext() {
        if (!buffer.isEmpty()) return true;
        if (noMoreData) return false;
        loadBatch();
        return !buffer.isEmpty();
    }

    @Override
    public DownloadTask next() {
        if (!hasNext()) throw new java.util.NoSuchElementException();
        return buffer.remove( index: 0);
    }

    private void loadBatch() {  1 usage
        try {
            List<DownloadTask> chunk = repo.listRange(cursor, batchSize);
            if (chunk.isEmpty()) {
                noMoreData = true;
                return;
            }
            buffer.addAll(chunk);
            cursor += chunk.size();
        } catch (SQLException e) {
            noMoreData = true;
            throw new RuntimeException("DB error while iterating tasks", e);
        }
    }
}
```

Рисунок 3 – TaskIterator. Власне ітератор, що підвантажує батчами (Iterator).

TaskIterator реалізує інтерфейс Iterator<DownloadTask> і відповідає за послідовний доступ до елементів. Ключова логіка зосереджена у зв’язці методів hasNext()/next() та приватної операції підвантаження loadBatch(): якщо локальний буфер спорожнів, ітератор звертається до БД через listRange(...) і наповнює буфер наступною порцією записів. Такий підхід дозволяє працювати зі списками будь-якого розміру, зберігаючи стабільну продуктивність і низьке споживання пам’яті. У випадку помилки БД ітератор коректно зупиняється й повідомляє про збій через виняток.

Рисунок 4 - FilteredTaskIterable. Фільтрований обхід (декоратор над Iterable).

FilteredTaskIterable реалізує патерн «декоратора» поверх TaskIterable, додаючи фільтрацію на льоту за довільним предикатом (Predicate<DownloadTask>). Клієнтський код, як і раніше, ітерується звичайним for-each, а відбір елементів виконується всередині спеціалізованого ітератора, що загортає базовий. Це мінімізує дублювання коду та дозволяє легко вводити нові режими перегляду (наприклад, лише

RUNNING або лише ERROR) без зміни базового ітератора чи репозиторію. Підхід зберігає інваріанти колекції та не порушує інкапсуляцію.

```
public List<DownloadTask> listRange(int offset, int limit) throws SQLException {
    List<DownloadTask> out = new ArrayList<>();
    try (PreparedStatement ps = con.prepareStatement(
            sql: "SELECT * FROM tasks ORDER BY id LIMIT ? OFFSET ?")) {
        ps.setInt( parameterIndex: 1, Math.max(0, limit));
        ps.setInt( parameterIndex: 2, Math.max(0, offset));
        try (ResultSet rs = ps.executeQuery()) {
            while (rs.next()) out.add(map(rs));
        }
    }
    return out;
}
```

Рисунок 5 - TaskRepository.listRange(...). Батч-вибірка з БД (підтримка Iterator).

Метод listRange(int offset, int limit) є джерелом даних для ітератора: він повертає рівно ту кількість записів, що потрібна на поточному кроці обходу. Завдяки цьому ітератор може масштабовано обробляти великі таблиці, не витрачаючи надлишкові ресурси на попереднє завантаження. Також метод зберігає стабільний порядок (наприклад, за id), що гарантує детермінований обхід незалежно від розміру набору.

5. Реалізація системи

Додавання кількох задач для ітерації.

На етапі перевірки користувач додає декілька завантажень командою add <url> <file>, створюючи потрібний обсяг даних у БД. Це одразу відображається у внутрішній таблиці tasks, а статуси задач оновлюються в процесі роботи. Наявність різних станів (наприклад, COMPLETED та PAUSED) дозволяє перевірити як базовий обхід, так і роботу фільтрації.

```
<=Task created: #5% EXECUTING [7m 39s]                <==========----> 75% EXECUTING [7m 40s]
> :app:run
<=Task created: #6% EXECUTING [8m 7s]                 <==========----> 75% EXECUTING [8m 12s]
> :app:run
<=Task created: #7% EXECUTING [8m 27s]:\Temp\10MB_2.bin  <==========----> 75% EXECUTING [8m 28s]
> :app:run
<==========----> 75% EXECUTING [9m 9s]
```

Імітація стану “paused” для демонстрації фільтра.

Щоб продемонструвати відбір за статусом, одна із задач свідомо

переводиться у PAUSED командою pause <id>. Так ми створюємо різнорідний набір для подальшого обходу: частина записів готова, частина в паузі. Це наочно показує, що фільтрований ітерабель повертає лише ті елементи, які задовольняють предикат.

```
<=Paused #6======----> 75% EXECUTING [10m 53s]
> :app:run
<<<<==========----> 75% EXECUTING [11m 3s]
```

Перевірка команди list через ітератор.
Команда list виконує обхід колекції через TaskIterable/TaskIterator, підвантажуючи записи батчами (типове значення — 50). Клієнт бачить рівний стрімінговий вивід елементів, навіть якщо задач багато, — без помітних затримок чи піків споживання пам'яті. Завдяки прозорій буферизації ітератора розмір даних не впливає на стабільність інтерфейсу.

```
#5 [COMPLETED] http://speed.hetzner.de/10MB.bin (1059/1059) -> C:\Temp\10MB_1.bin
#6 [PAUSED] http://speed.hetzner.de/10MB.bin (1059/1059) -> C:\Temp\10MB_2.bin
#7 [COMPLETED] http://speed.hetzner.de/100MB.bin (1059/1059) -> C:\Temp\100MB_1.bin
```

Перевірка фільтрації: list completed.
У цьому режимі використовується FilteredTaskIterable з предикатом t -> t.status == COMPLETED. Виводяться тільки завершені задачі, причому вся логіка відбору захована всередині ітерабельного декоратора. Клієнтський код не змінюється: зовнішній інтерфейс залишається ідентичним звичайному обходу.

```
#5 [COMPLETED] http://speed.hetzner.de/10MB.bin (1059/1059) -> C:\Temp\10MB_1.bin
#7 [COMPLETED] http://speed.hetzner.de/100MB.bin (1059/1059) -> C:\Temp\100MB_1.bin
```

Перевірка фільтрації: list paused (за потреби з власним batchSize).
Аналогічно, предикат t -> t.status == PAUSED повертає лише задачі в паузі, що дозволяє візуально підтвердити коректність фільтрації. Додатковий параметр batchSize демонструє контроль над розміром порцій: це зручно, якщо потрібно зменшити кількість запитів або, навпаки, знизити миттєве навантаження на пам'ять. Результати узгоджуються зі станами в БД і відтворюють очікувану поведінку ітератора.

```
#6 [PAUSED] http://speed.hetzner.de/10MB.bin (1059/1059) -> C:\Temp\10MB_2.bin
> st paused
<==========----> 75% EXECUTING [13m 16s]
```

1.3. Висновки.

У роботі реалізовано поведінковий шаблон Iterator для масштабованого перегляду задач завантаження з локальної БД SQLite. Створені класи TaskIterable, TaskIterator та FilteredTaskIterable забезпечують послідовний обхід записів порціями (батчами) і фільтрацію за станом без зміни клієнтського коду. Це знімає залежність інтерфейсу від внутрішньої структури сховища, зменшує споживання пам’яті та підвищує стабільність при великій кількості записів.

Інтеграція з наявними компонентами (TaskRepository.listRange(...), DownloadService, CLI Main) довела практичну придатність рішення: команди list, list completed, list paused коректно відображають дані й демонструють прозоре використання ітератора. Під час тестування підтверджено роботу з різними статусами задач, відновленням завантажень і можливістю керування продуктивністю через розмір батча. Надалі шаблон легко розширити (наприклад, зворотний обхід, пагінація з курсорами, комбіновані фільтри), не змінюючи клієнтський код — лише додаючи нові варіанти ітерабельних обгорток.