Національний технічний університет України

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

Кафедра інформаційних систем та технологій

Лабораторна робота № 9

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування.»

Тема проекту: «26. Download Manager»

Виконала:

студентка групи ІА-34

Мушта Анна

Перевірив:

Мягкий Михайло Юрійович

Дата здачі 13.12.2025

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

1. Клієнт-серверна архітектура
   Цей тип архітектури складається з двох основних компонентів: клієнтів, які взаємодіють з користувачем, та серверів, що обробляють і зберігають дані.

- Тонкий клієнт: більшість обчислень і операцій виконуються на сервері. Прикладом є веб-застосунки.
- Товстий клієнт: більшість логіки обробляється на стороні клієнта. Приклади: десктопні програми або мобільні додатки (наприклад, Viber, Outlook, GTA).
- SPA (Single Page Application): це проміжний варіант, де частина логіки обробляється на клієнтській стороні, а з сервером відбувається взаємодія через Web API.

Структура:
Клієнт → Проміжна частина (middleware) → Сервер

- Клієнтська частина: забезпечує інтерфейс і взаємодію з користувачем, а також підключення до сервера.
- Серверна частина: містить бізнес-логіку та обробку даних.

2. **Peer-to-Peer (P2P) архітектура**
   Призначення: У такій архітектурі кожен вузол одночасно є і клієнтом, і сервером.
   Основні принципи:

- Децентралізація: немає єдиного центрального сервера.
- Рівноправність вузлів: кожен вузол може як отримувати, так і надавати ресурси.

- Розподіл ресурсів: ресурси, такі як обчислювальна потужність, файли та пам'ять, розподіляються між вузлами.

Приклади: BitTorrent, Skype, Zoom, блокчейн.
Проблеми: безпека, синхронізація, пошук даних.

3. **Сервіс-орієнтована архітектура (SOA)**
   Це підхід, що базується на використанні розподілених сервісів, які мають чітко визначені інтерфейси для взаємодії.
   Взаємодія між сервісами здійснюється через протоколи, такі як HTTP, SOAP або REST.

- Кожен сервіс виконує певну бізнес-функцію (наприклад, перевірка складу товарів).
- Сервіси взаємодіють через обмін повідомленнями, не маючи спільної бази даних.
- Для пошуку сервісів використовуються реєстри або каталоги.

Зазвичай SOA використовується разом з Enterprise Service Bus (ESB) — шиною для обміну даними. SOA також є основою для мікросервісної архітектури.

4. **Мікросервісна архітектура**
   У мікросервісній архітектурі додаток складається з маленьких незалежних сервісів, кожен з яких виконує конкретну функцію.

- Кожен мікросервіс має власний процес і базу даних.
- Кожен мікросервіс має чітко визначену відповідальність.
- Мікросервіси можна розгортати незалежно один від одного.

Протоколи взаємодії: HTTP/HTTPS, WebSockets, AMQP.
Переваги: висока масштабованість, можливість простого оновлення та автономність кожного сервісу.

1.2. Хід роботи.

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

1. Шаблон "Composite"

Призначення: шаблон використовується для об'єднання об'єктів у дерево, щоб керувати ними як єдиним цілим.

Класи:

- DownloadTask: абстрактний клас, що реалізує загальну логіку для завантаження файлів. Він визначає шаблонний метод для запуску завантаження і викликає абстрактні методи для конкретних реалізацій.

```
abstract class DownloadTask {  14 usages  3 inheritors
    final String id = UUID.randomUUID().toString();  4 usages
    protected volatile boolean cancelled = false;  2 usages
    protected final List<ProgressObserver> observers = new CopyOnWriteArrayList<>();
    protected final AtomicLong downloaded = new AtomicLong( initialValue: 0);  6 usages
    protected volatile long totalSize = -1;  11 usages
    protected final String sourceUrl;  1 usage
    protected final String targetFile;  4 usages

    DownloadTask(String sourceUrl, String targetFile) {  3 usages
        this.sourceUrl = sourceUrl;
        this.targetFile = targetFile;
    }

    public String getId() { return id; }  5 usages
    public void addObserver(ProgressObserver o){ observers.add(o); }  1 usage
    public void removeObserver(ProgressObserver o){ observers.remove(o); }  no usages

    public final void runTask() {  2 usages
        try {
            preCheck();
            openResources();
            download();
            finalizeDownload();
            notifyAllObservers( finished: true);
        } catch (CancelledException e) {
            notifyAllObservers( finished: false);
            System.out.println("Download cancelled: " + id);
        } catch (Exception e) {
            notifyAllObservers( finished: false);
            System.err.println("Error in download " + id + ": " + e.getMessage());
            e.printStackTrace();
        }
    }
    protected abstract void preCheck() throws Exception;  1 usage  3 implementations
    protected abstract void openResources() throws Exception;  1 usage  3 implementations
    protected abstract void download() throws Exception;  1 usage  3 implementations
    protected abstract void finalizeDownload() throws Exception;  1 usage  3 implementations
```

```
    protected void notifyAllObservers(boolean finished) {  6 usages
        for (ProgressObserver o : observers) {
            o.onProgress(id, downloaded.get(), totalSize, finished);
        }
    }

    public void cancel() { cancelled = true; }  1 usage

    protected void checkCancelled() throws CancelledException {  3 usages
        if (cancelled) throw new CancelledException();
    }
}
```

- DownloadGroup: реалізує DownloadTask і дозволяє додавати кілька завдань у групу для одночасного керування.

```
class DownloadGroup extends DownloadTask {  no usages
    private final List<DownloadTask> children = new ArrayList<>();  2 usages
    DownloadGroup(String name) { super( sourceUrl: "group://" + name, name); }
    public void add(DownloadTask t) { children.add(t); }
    @Override protected void preCheck() throws Exception {}  1 usage
    @Override protected void openResources() throws Exception {}  1 usage
    @Override protected void download() throws Exception {  1 usage
        for (DownloadTask t : children) {
            checkCancelled();
            t.runTask();
        }
    }
    @Override protected void finalizeDownload() throws Exception {}  1 usage
}
```

Основна логіка:

- Клієнт може додавати завдання у групу через DownloadGroup і запускати їх через метод runTask.
- Для кожного елементу групи виконується однакова логіка завантаження через методи шаблону.

2. Шаблон "Flyweight"

Призначення: зменшення кількості об'єктів шляхом поділу спільних даних між ними.

Класи:

- Peer: представляє вузол у P2P-мережі, який містить ресурси для обміну.
- P2PManager: управляє списком пірів та забезпечує пошук файлів серед них.

```
class P2PManager {  4 usages
    private final List<Peer> peers = new ArrayList<>();  3 usages
    P2PManager() {  1 usage
        // default seed folder "seeds" in current dir with one peer (local)
        Path seeds = Paths.get( first: "seeds");
        try { Files.createDirectories(seeds); } catch (IOException ignored) {}
        peers.add(new Peer( id: "local", seeds));
    }
    void addPeer(Peer p) { peers.add(p); }  no usages
    Optional<Path> request(String resourceName) {  2 usages
        for (Peer p : peers) {
            Optional<Path> found = p.find(resourceName);
            if (found.isPresent()) return found;
        }
        return Optional.empty();
    }
}
```

Основна логіка:

- Всі пірі мають спільний набір файлів для завантаження, що дозволяє зменшити кількість створюваних об'єктів і заощадити пам'ять.
- Кожен пір може бути знайдений за ім'ям файлу та передати його клієнту для завантаження.

3. Шаблон "Iterator"

Призначення: надає спосіб доступу до елементів колекції без її розкриття.

Клас:

- SegmentIterator: ітератор, що дозволяє по черзі обробляти сегменти файлів під час завантаження.
- SingleSegmentIterator: конкретна реалізація ітератора для обробки одного сегмента.

```
class Segment {  4 usages
    final long start;  1 usage
    final long end;  1 usage
    Segment(long s, long e) { start = s; end = e; }  1 usage
}

interface SegmentIterator extends Iterator<Segment> {}  1 usage  1 implementation

class SingleSegmentIterator implements SegmentIterator {  no usages
    private boolean used = false;  2 usages
    private final Segment seg;  2 usages
    SingleSegmentIterator(long total) {  no usages

        seg = new Segment( s: 0, Math.max(0, total - 1));
    }
    @Override public boolean hasNext() { return !used; }
    @Override public Segment next() { used = true; return seg; }
}
```

4. Шаблон "Command"

Призначення: для реалізації операцій як об'єктів, що можна передавати, скасовувати або повторювати.

Класи:

- StartCommand: виконує завдання на сервері за допомогою потоку.
- CancelCommand: скасовує поточне завантаження.

```
class StartCommand implements Command {  1 usage
    private final DownloadTask task;  2 usages
    private final ExecutorService exec;  2 usages
    StartCommand(DownloadTask task, ExecutorService exec) {  1 usage
        this.task = task; this.exec = exec; }
    @Override public void execute() { exec.submit(task::runTask); }  1 usage
}

class CancelCommand implements Command {  no usages
    private final DownloadTask task;  2 usages
    CancelCommand(DownloadTask task) { this.task = task; }  no usages
    @Override public void execute() { task.cancel(); }  1 usage
}
```

5. Шаблон "Template Method"

Призначення: надає загальний шаблон для виконання завдань, дозволяючи конкретним підкласам реалізовувати специфічну поведінку.

Клас:

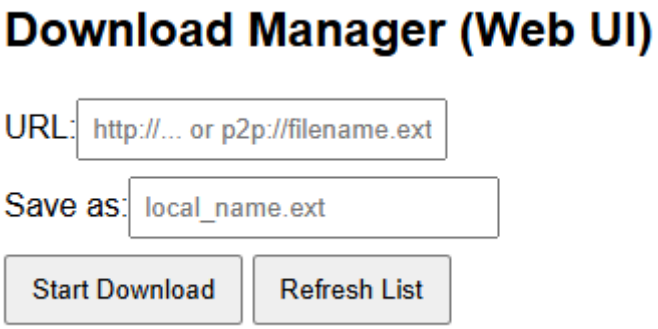
- DownloadTask: визначає шаблон для всіх завдань завантаження, що включає методи для перевірки, відкриття ресурсів, завантаження та фіналізації.

```
abstract class DownloadTask {  14 usages  3 inheritors
    final String id = UUID.randomUUID().toString();  4 usages
    protected volatile boolean cancelled = false;  2 usages
    protected final List<ProgressObserver> observers = new CopyOnWriteArrayList<>();
    protected final AtomicLong downloaded = new AtomicLong( initialValue: 0);  6 usages
    protected volatile long totalSize = -1;  11 usages
    protected final String sourceUrl;  1 usage
    protected final String targetFile;  4 usages

    DownloadTask(String sourceUrl, String targetFile) {  3 usages
        this.sourceUrl = sourceUrl;
        this.targetFile = targetFile;
    }

    public String getId() { return id; }  5 usages
    public void addObserver(ProgressObserver o){ observers.add(o); }  1 usage
    public void removeObserver(ProgressObserver o){ observers.remove(o); }  no usages
```

Результати



Маємо простий інтерфейс веб-застосунку для завантаження файлів. Користувач може вказати URL або шлях до файлу в системі P2P, а також зазначити ім'я файлу, під яким він хоче зберегти завантажений ресурс. Кнопки "Start Download" і "Refresh List" дають змогу почати завантаження або оновити список завдань, що вже обробляються.

Це зображення демонструє веб-застосунок після початку завантаження. Після натискання на кнопку "Start Download", додаток відображає статус завантаження для кожного завдання. У нашому прикладі завантажується файл з Вікіпедії за допомогою вказаного URL. Кожне завдання має унікальний ID і відображається прогрес завантаження у вигляді бару.

1.3. Висновки.

У ході виконання лабораторної роботи №9, присвяченої вивченню та застосуванню шаблонів проектування, я реалізував функціональність однофайлового менеджера завантажень, що використовує шаблони «Composite», «Flyweight», «Template Method», «Command» та «Observer» для досягнення різних функціональних можливостей. Кожен з цих шаблонів дозволяє вирішити окремі задачі, пов’язані з обробкою даних, керуванням завантаженнями та взаємодією компонентів системи.

1. Шаблон «Composite»:

Цей шаблон був використаний для групування завдань завантаження. У рамках програми був реалізований клас DownloadGroup, який дозволяє об’єднувати кілька завдань завантаження у єдину групу. Клас DownloadGroup реалізує шаблон «Composite» і дозволяє додавати завдання до групи, а також одночасно запускати всі завдання в групі. Це значно спрощує управління кількома завданнями, дозволяючи працювати з ними як з єдиним цілим, не розрізняючи окремі елементи та групи. Задачі групи виконуються по черзі, при цьому кожне завдання може реалізовувати свою специфічну поведінку, але управління ними здійснюється централізовано через групу.

2. Шаблон «Flyweight»:

Шаблон «Flyweight» використаний для оптимізації пам’яті при роботі з великою кількістю однакових об'єктів у P2P мережі. В рамках цієї роботи було реалізовано клас P2PManager та Peer, де кожен пір має доступ до загальних ресурсів, що зберігаються в локальній папці seeds/. Всі пірі використовують спільний пул об'єктів для обміну файлами, що дозволяє зменшити кількість дубльованих об’єктів і зберегти пам’ять. Патерн дозволяє економити ресурси, коли багато об'єктів поділяють однакові дані (наприклад, дані файлів).

3. Шаблон «Template Method»:

Шаблон «Template Method» використано для визначення загального шаблону виконання завдання завантаження. Абстрактний клас DownloadTask визначає основні етапи завантаження (перевірка, відкриття ресурсів, завантаження та завершення), а конкретні класи для HTTP та P2P завантажень реалізують специфічну логіку для кожного з етапів. Це дозволяє стандартизувати процес завантаження для різних типів завдань (HTTP, P2P), зберігаючи при цьому можливість гнучкої реалізації кожного етапу.

4. Шаблон «Command»:

Шаблон «Command» використано для реалізації операцій скасування і відновлення завдань завантаження. Клас StartCommand відповідає за ініціалізацію завдання на виконання, а клас CancelCommand дозволяє скасувати виконання завдання. Завдяки використанню цього шаблону, кожна операція (старт, скасування, пауза, відновлення) інкапсулюється в окремий об'єкт команди, що дозволяє реалізувати такі функціональності, як скасування та повторне виконання операцій. Така структура дає можливість легко додавати нові операції без зміни основної логіки програми.

5. Шаблон «Observer»:

Шаблон «Observer» був використаний для реалізації спостереження за прогресом завантаження. Клас ProgressObserver дозволяє спостерігати за завантаженням файлів та отримувати оновлення щодо стану завантаження. При цьому інтерфейс спостерігача дозволяє додавати і видаляти спостерігачів, що робить систему гнучкою і розширюваною. Цей шаблон

використовується для відслідковування прогресу кожного завдання та надання інформації користувачеві про стан завантаження.

6. Реалізація веб-сервера:

Один з важливих аспектів цієї роботи полягає в реалізації простого веб-сервера за допомогою класу HttpServer з пакету com.sun.net.httpserver. Це дозволяє користувачеві взаємодіяти з менеджером завантажень через веб-інтерфейс, запускати нові завдання, перевіряти їхній статус і переглядати список активних завантажень.

7. Робота з P2P:

Імітація P2P-завантаження через локальну папку seeds/ була здійснена за допомогою класів P2PManager та Peer. Взаємодія між клієнтами в P2P-мережі здійснюється через механізм пошуку файлів у мережі. Це дозволяє забезпечити спільний доступ до файлів між кількома користувачами без необхідності центрального сервера.

8. Висновки:

Реалізація цієї лабораторної роботи дозволила застосувати на практиці кілька шаблонів проектування, що забезпечили модульність і гнучкість програми. Використання шаблонів «Composite», «Flyweight», «Template Method», «Command» і «Observer» дозволило реалізувати ефективну систему для завантаження файлів з підтримкою кількох архітектурних рішень, таких як клієнт-сервер і P2P. Веб-інтерфейс дозволяє зручно керувати завантаженнями, а реалізація шаблонів дозволяє легко розширювати функціональність програми у майбутньому.

Програма продемонструвала вдалу інтеграцію різних архітектурних патернів і ефективне використання можливостей мови Java для розробки ефективних розподілених систем..

1.4 Контрольні запитання

1. Клієнт-серверна архітектура
   Клієнт-серверна архітектура - це модель побудови програмних систем, де функціонал розподіляється між двома основними компонентами: сервером і клієнтом.

- Сервер: Цей компонент відповідає за зберігання, обробку та надання даних.
- Клієнт: Це програма, через яку користувач взаємодіє з системою, відправляючи запити до сервера для отримання даних або виконання операцій.
  Архітектура може бути двошаровою (клієнт і сервер) або тришаровою (клієнт, проміжне середовище, сервер).
  Переваги такої моделі: централізоване управління, легкість в оновленні і можливість масштабування.

2. **Сервіс-орієнтована архітектура (SOA)**
   SOA є стилем побудови систем, де основними елементами є сервіси - незалежні програмні компоненти, що надають певні бізнес-функції. Кожен сервіс має чітко визначений інтерфейс і взаємодіє з іншими сервісами через стандартизовані протоколи (наприклад, HTTP, SOAP, REST).
   SOA дозволяє досягти масштабованості, гнучкості та повторного використання компонентів, забезпечуючи слабке зв’язування між частинами системи.

3. **Принципи SOA**
   Основні принципи SOA:

- Слабке зв'язування: сервіси повинні бути мінімально залежними один від одного.
- Повторне використання: кожен сервіс можна використовувати в різних додатках.
- Інтероперабельність: взаємодія сервіса через відкриті стандарти (HTTP, XML, JSON, SOAP).
- Самодостатність: сервіс містить всю необхідну логіку для виконання своєї функції.
- Виявлення: сервіси можна знайти через спеціальні реєстри, що надають опис доступних функцій та їх інтерфейсів.

4. **Взаємодія сервісів у SOA**
   Сервіси взаємодіють через обмін повідомленнями (через HTTP-запити або SOAP-повідомлення). Кожен сервіс має власний

інтерфейс (WSDL, API-документація), що дає змогу іншим сервісам або клієнтам звертатися до нього. Взаємодія може бути як синхронною (запит-відповідь), так і асинхронною (через черги повідомлень). Важливо, що сервіси не обов'язково мають спільну базу даних або прямий зв'язок між собою, вони обмінюються лише даними у стандартизованій формі (наприклад, у форматах JSON або XML).

5. **Пошук сервісів і виконання запитів у SOA**
   У SOA сервіси реєструються в каталозі (реєстрі сервісів), де зберігається інформація про доступні функції, протоколи, адреси та формати повідомлень. Розробники можуть знайти потрібний сервіс у цьому каталозі та отримати його опис. Запити до сервісів виконуються за допомогою HTTP-звернень або через API (SOAP/REST), де кожен запит має чітко визначену структуру, а відповіді приходять у форматах JSON або XML.

6. **Переваги та недоліки клієнт-серверної архітектури**
   Переваги:

- Централізоване управління та зберігання даних.
- Легке оновлення - змінюється лише серверна частина.
- Можливість підключення великої кількості клієнтів.
- Вищий рівень безпеки, оскільки дані зберігаються на сервері.
  Недоліки:
- Залежність від працездатності сервера - якщо сервер недоступний, система не працює.
- Можливе перевантаження сервера при великій кількості запитів.
- Потрібне постійне мережеве з'єднання.

7. **Переваги та недоліки однорангової (P2P) архітектури**
   Переваги:

- Відсутність центрального сервера, що забезпечує відсутність єдиної точки відмови.
- Рівноправність вузлів - кожен вузол може бути як клієнтом, так і сервером, що дозволяє рівномірно розподіляти навантаження.

- Легкість масштабування - можна додавати нові вузли для розширення мережі.
  Недоліки:
- Проблеми з безпекою і автентифікацією.
- Складність пошуку ресурсів у великій мережі.
- Труднощі в синхронізації даних між вузлами та узгодженні їх стану.

8. **Мікросервісна архітектура**
   Мікросервісна архітектура передбачає побудову системи із малих, незалежних сервісів, кожен з яких виконує певну бізнес-функцію. Кожен мікросервіс має власну базу даних і може бути розгорнутий та оновлений окремо від інших. Це забезпечує високу гнучкість, масштабованість і автономність команд розробників.
   Мікросервіси зазвичай використовують REST API або повідомлення через брокери для взаємодії.
9. **Протоколи взаємодії в мікросервісній архітектурі**
   Основні протоколи взаємодії між мікросервісами:
- HTTP/HTTPS: стандартні веб-запити, найчастіше використовуються для REST API.
- WebSockets: для двостороннього зв'язку в реальному часі (чати, повідомлення).
- AMQP (Advanced Message Queuing Protocol): для обміну повідомленнями через черги (наприклад, RabbitMQ).
  Також використовуються інші протоколи, такі як gRPC, MQTT, GraphQL, залежно від вимог системи.

10. **Сервіс-орієнтована архітектура та бізнес-логіка**
    Підхід, коли між шаром веб-контролерів і шаром доступу до даних розробляється шар бізнес-логіки у вигляді сервісів, не є повноцінною SOA. У цьому випадку сервіси не є незалежними компонентами, оскільки вони тільки розділяють логіку всередині одного застосунку (монолітна архітектура). В реальній SOA сервіси повинні бути розподіленими, автономними, і їх можна масштабувати й оновлювати окремо.