Національний технічний університет України

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

Кафедра інформаційних систем та технологій

Лабораторна робота № 7

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 29.11.2025

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

Mediator (Посередник)

• Координує взаємодію між об’єктами через центральну сутність, що зменшує прямі залежності між ними.

• Переваги: зменшує зв’язність, полегшує керування комунікацією, спрощує тестування.

• Недоліки: центральний посередник може надмірно ускладнитися й перетворитися на «God Object».

Facade (Фасад)

• Надає спрощений інтерфейс для складної підсистеми, приховуючи її внутрішню реалізацію.

• Переваги: приховує деталі реалізації та забезпечує легкий доступ до підсистеми.

• Недоліки: може обмежувати можливість тонкого налаштування внутрішніх компонентів.

Bridge (Міст)

• Відокремлює абстракцію від її реалізації, дозволяючи змінювати їх незалежно одна від одної.

• Переваги: підвищує гнучкість, полегшує підтримку та запобігає надмірному розростанню кількості класів.

• Недоліки: додає складності за рахунок введення додаткових рівнів абстракції.

Template Method (Шаблонний метод)

• Формує загальну послідовність алгоритму в базовому класі, а конкретні кроки дозволяє реалізувати підкласам.

• Переваги: зменшує дублювання коду, дає змогу легко додавати нові варіанти алгоритму, гарантує визначений порядок виконання кроків.

• Недоліки: структура алгоритму є фіксованою, а підкласи залишаються прив’язаними до базової реалізації..

1.2. Хід роботи.

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

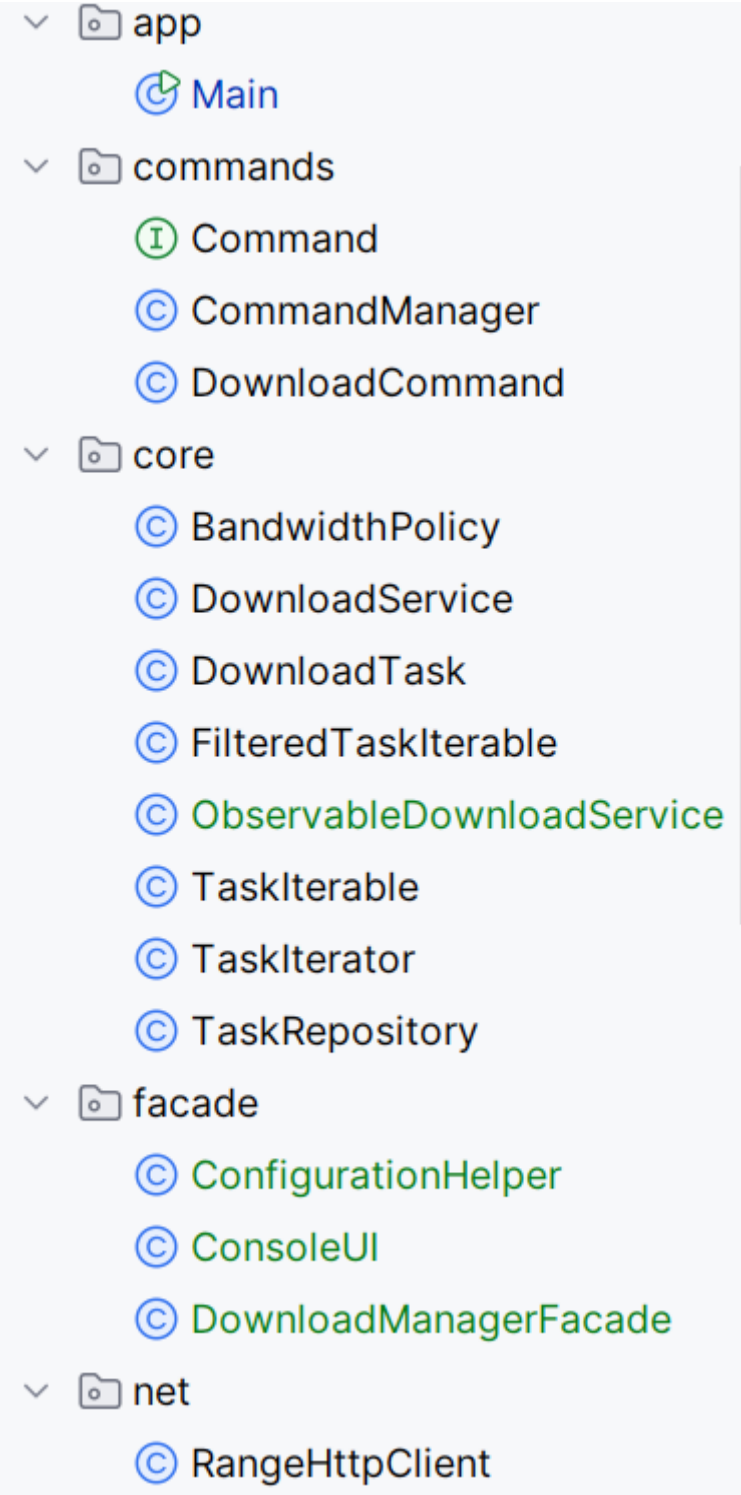


Рисунок 1 - Структура модулів і класів застосунку.

Проєкт розбитий на пакети **dm.app**, **dm.core**, **dm.net**, **dm.commands**, **dm.observers** та **dm.facade**.

У **dm.core** реалізовано шаблон **Iterator** для ефективного 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.
- **ObservableDownloadService** - розширює DownloadService, додаючи підтримку Observer Pattern для сповіщення про події завантажень.
- **BandwidthPolicy** - ліміт швидкості.
- **RangeHttpClient** - HTTP клієнт із підтримкою Range/відновлення.

У **dm.facade** реалізовано шаблон **Facade** для спрощення взаємодії зі складною підсистемою download-manager, об'єднуючи множину компонентів у простий уніфікований інтерфейс.

- **DownloadManagerFacade** - головний клас фасаду. У конструкторі він ініціалізує всі підсистеми (ObservableDownloadService,

CommandManager), створює та підписує observer'ів (ConsoleObserver, StatisticsObserver). Надає клієнту спрощений API для керування завантаженнями, доступу до функціоналу undo/redo та статистики.

- **ConsoleUI** - допоміжний клас для форматованого виведення інформації у консоль. Надає статичні методи для показу привітального повідомлення (showWelcome()), форматованого списку завантажень (showTaskList()) та повідомлень про успіх/помилки (showSuccess(), showError()). Централізує логіку представлення.
- **ConfigurationHelper** - допоміжний клас для централізації налаштувань системи (шлях до бази даних, розмір batch для ітератора, ліміт швидкості завантаження). Використовує Fluent Interface для зручної конфігурації.

У **dm.commands** реалізовано шаблон **Command** для інкапсуляції операцій:

- **Command** - інтерфейс команди з методами execute(), undo(), getDescription().
- **DownloadCommand** - конкретна команда для операцій pause/resume з підтримкою скасування.
- **CommandManager** - invoker, що управляє історією команд та забезпечує undo/redo функціонал.

У **dm.observers** реалізовано шаблон **Observer** для моніторингу подій завантажень:

- **DownloadObserver** - інтерфейс спостерігача з методами для обробки подій (onDownloadStarted, onProgressUpdate, onDownloadCompleted, onDownloadPaused, onDownloadError).
- **ConsoleObserver** - конкретний Observer, що виводить інформацію про події в консоль у форматованому вигляді.
- **StatisticsObserver** - конкретний Observer, що збирає статистику завантажень (кількість запусків, завершень, пауз, помилок, загальний обсяг даних).

У **dm.app.Main** - консольний інтерфейс, який:

- Використовує TaskIterable/FilteredTaskIterable (Iterator Pattern) у команді list для перегляду задач.
- Використовує CommandManager (Command Pattern) для виконання операцій pause/resume з підтримкою undo/redo.
- Підписує ConsoleObserver та StatisticsObserver на події ObservableDownloadService (Observer Pattern).

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 або PAUSED). Клієнтський код у Main працює з цією колекцією через звичний цикл for-each, не знаючи про внутрішню структуру сховища та спосіб вибірки.

Шаблон **Command** забезпечує гнучке управління операціями завантаження, дозволяючи скасовувати та повторювати дії, вести історію команд та відокремлювати ініціатора дії від її виконавця.

Шаблон **Observer** забезпечує слабкий зв'язок між бізнес-логікою завантажень та способами відображення інформації. Завдяки цьому можна легко додавати нові способи реагування на події (логування у файл, відправка повідомлень тощо) без зміни основного коду DownloadService.

Шаблон **Facade** застосовано для надання уніфікованого та спрощеного інтерфейсу до складної підсистеми управління завантаженнями, яка включає Command Manager, Observable Download Service та Observer'и. Клас DownloadManagerFacade спрощує ініціалізацію, керування командами та взаємодію з підсистемою, приховуючи від клієнтського коду (Main) її внутрішню складність, залежності та логіку налаштування (наприклад, підписку Observer'ів).

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

Я обрала Facade Pattern для спрощення взаємодії зі складною системою download-manager, тому що він ідеально підходить для об'єднання множини підсистем у простий уніфікований інтерфейс. Нижче детальніше поясню, чому саме цей патерн є оптимальним вибором.

1. Складна система з багатьма компонентами

У моєму download-manager на момент реалізації Facade вже були впроваджені три патерни проєктування: Iterator для обходу завдань, Command для керування операціями з підтримкою undo/redo, та Observer для моніторингу подій завантажень. Це призвело до того, що клієнтський код (Main.java) містив велику кількість ініціалізаційної логіки: створення ObservableDownloadService, CommandManager, підписку observer'ів (ConsoleObserver, StatisticsObserver), налаштування конфігурації.

Facade Pattern дозволяє приховати всю цю складність за простим інтерфейсом, де одним викликом конструктора ініціалізується вся система.

2. Централізація конфігурації та UI

Facade Pattern також дозволив створити допоміжні класи для централізації налаштувань (ConfigurationHelper) та форматованого виведення (ConsoleUI). Це відокремлює логіку конфігурації та представлення від бізнес-логіки, роблячи систему більш модульною.

Тож перейдемо до реалізації

1. DownloadManagerFacade.java - головний фасад системи

```java
package dm.facade;

import dm.commands.*;
import dm.core.*;
import dm.observers.*;

import java.nio.file.Path;

public class DownloadManagerFacade {  2 usages  new *

    private final ObservableDownloadService service;  10 usages
    private final CommandManager commandManager;  8 usages
    private final ConsoleObserver consoleObserver;  2 usages
    private final StatisticsObserver statsObserver;  4 usages

    public DownloadManagerFacade(String dbPath) throws Exception {  1 usage  new *
        this.service = new ObservableDownloadService(Path.of(dbPath));
        this.commandManager = new CommandManager();

        this.consoleObserver = new ConsoleObserver();
        this.statsObserver = new StatisticsObserver();
```

```
    service.attach(consoleObserver);
    service.attach(statsObserver);
}
public int addDownload(String url, String targetFile) throws Exception {  new *
    return service.add(url, Path.of(targetFile));
}


public void pauseDownload(int taskId) throws Exception {  new *
    Command pauseCmd = new DownloadCommand(service, taskId, DownloadCommand.Action.PAUSE);
    commandManager.execute(pauseCmd);
}

public void resumeDownload(int taskId) throws Exception {  new *
    Command resumeCmd = new DownloadCommand(service, taskId, DownloadCommand.Action.RESUME);
    commandManager.execute(resumeCmd);
}

public void undoLastOperation() throws Exception {  new *
    commandManager.undo();
}

public void redoLastOperation() throws Exception {  new *
    commandManager.redo();
}

public void showCommandHistory() {  new *
    commandManager.showHistory();
}

public void showStatistics() {  new *
    statsObserver.printStatistics();
}

public Iterable<DownloadTask> getAllTasks() throws Exception {  new *
    return new TaskIterable(service.getRepository(),  batchSize: 50);
}

public Iterable<DownloadTask> getTasksByStatus(DownloadTask.Status status) throws Exception {
    TaskIterable base = new TaskIterable(service.getRepository(),  batchSize: 50);
    return new FilteredTaskIterable(base,  DownloadTask t -> t.status == status);
}

public void setSpeedLimit(long bytesPerSecond) {  new *
    service.setLimit(bytesPerSecond);
}

    public void shutdown() throws Exception {  new *
        statsObserver.printStatistics();
        service.close();
    }

    public boolean canUndo() {  new *
        return commandManager.canUndo();
    }

    public boolean canRedo() {  new *
        return commandManager.canRedo();
    }
}
```

Рисунок 2 - Клас DownloadManagerFacade - головний фасад, що об'єднує всі підсистеми

DownloadManagerFacade є центральним компонентом патерну. У конструкторі він ініціалізує ObservableDownloadService, CommandManager, створює та підписує observer'ів (ConsoleObserver та StatisticsObserver). Далі він надає простий API для роботи з системою: addDownload() для додавання завантажень, pauseDownload() та resumeDownload() для керування ними, undoLastOperation() та redoLastOperation() для скасування/повторення операцій, showStatistics() для перегляду статистики, getAllTasks() та getTasksByStatus() для отримання списків завантажень.

Всередині кожного методу Facade приховує складність створення команд та взаємодії з підсистемами. Наприклад, метод pauseDownload() створює об'єкт DownloadCommand з потрібними параметрами та передає його CommandManager для виконання.

2. ConsoleUI.java - helper для форматованого виведення

```
package dm.facade;

import dm.core.DownloadTask;

public class ConsoleUI {  13 usages  new *

    private static final String SEPARATOR = "=".repeat( count: 50);  6 usages

    public static void showWelcome() {  2 usages  new *
        System.out.println(SEPARATOR);
        System.out.println("  Download Manager - Simplified Facade Interface");
        System.out.println(SEPARATOR);
        System.out.println("Available commands:");
        System.out.println("  add <url> <file>  - Add new download");
        System.out.println("  pause <id>        - Pause download");
        System.out.println("  resume <id>       - Resume download");
        System.out.println("  list [status]     - List downloads");
        System.out.println("  stats             - Show statistics");
        System.out.println("  undo              - Undo last operation");
        System.out.println("  redo              - Redo operation");
        System.out.println("  history           - Show command history");
        System.out.println("  help              - Show this help");
        System.out.println("  exit              - Exit program");
        System.out.println(SEPARATOR);
    }
```

```
    public static void showTaskList(Iterable<DownloadTask> tasks, String filter) {  1 usage  new *
        System.out.println("\n" + SEPARATOR);
        System.out.println("  Downloads" + (filter != null ? " (" + filter + ")" : ""));
        System.out.println(SEPARATOR);

        boolean hasAny = false;
        for (DownloadTask t : tasks) {
            hasAny = true;
            String progress = formatProgress(t.lastByte, t.totalBytes);
            System.out.printf("#%-3d [%-9s] %s%n", t.id, t.status, t.url);
            System.out.printf("     Progress: %s -> %s%n", progress, t.target);
        }

        if (!hasAny) {
            System.out.println("  No downloads found.");
        }
        System.out.println(SEPARATOR + "\n");
    }

    private static String formatProgress(long downloaded, long total) {  1 usage  new *
        if (total > 0) {
            double percent = (downloaded * 100.0) / total;
            return String.format("%.1f%% (%s / %s)",
                    percent, formatBytes(downloaded), formatBytes(total));
        } else {
            return String.format("%s / unknown", formatBytes(downloaded));
        }
    }

    private static String formatBytes(long bytes) {  3 usages  new *
        if (bytes < 1024) return bytes + " B";
        if (bytes < 1024 * 1024) return String.format("%.1f KB", bytes / 1024.0);
        if (bytes < 1024 * 1024 * 1024) return String.format("%.1f MB", bytes / (1024.0 * 1024));
        return String.format("%.1f GB", bytes / (1024.0 * 1024 * 1024));
    }

    public static void showSuccess(String message) {  1 usage  new *
        System.out.println("[OK] " + message);
    }

    public static void showError(String message) {  6 usages  new *
        System.err.println("[ERROR] " + message);
    }

    public static void showInfo(String message) {  3 usages  new *
        System.out.println("[INFO] " + message);
    }
}
```

Рисунок 3 - Клас ConsoleUI для форматованого виведення інформації в консоль

ConsoleUI є допоміжним класом, що спрощує роботу з виведенням інформації. Він надає статичні методи для показу привітального повідомлення (showWelcome()), форматованого списку завантажень (showTaskList()), повідомлень про успіх (showSuccess()), помилки (showError()) та інформаційних повідомлень (showInfo()).

Клас також містить приватні методи для форматування: formatProgress() розраховує відсоток виконання та форматує прогрес, formatBytes() конвертує байти в KB/MB/GB для зручного читання. Це відокремлює логіку представлення від бізнес-логіки та робить код більш чистим.

3. ConfigurationHelper.java - helper для централізації налаштувань

```java
package dm.facade;

public class ConfigurationHelper {  9 usages  new *

    private static final String DEFAULT_DB_PATH = "download.db";  1 usage
    private static final int DEFAULT_BATCH_SIZE = 50;  1 usage
    private static final long DEFAULT_SPEED_LIMIT = 0; // unlimited  1 usage

    private String dbPath;  4 usages
    private int batchSize;  4 usages
    private long speedLimit;  5 usages

    public ConfigurationHelper() {  3 usages  new *
        this.dbPath = DEFAULT_DB_PATH;
        this.batchSize = DEFAULT_BATCH_SIZE;
        this.speedLimit = DEFAULT_SPEED_LIMIT;
    }

    public ConfigurationHelper withDatabase(String dbPath) {  2 usages  new *
        this.dbPath = dbPath;
        return this;
    }
```

```java
public ConfigurationHelper withBatchSize(int batchSize) {  2 usages  new *
    this.batchSize = Math.max(1, batchSize);
    return this;
}

public ConfigurationHelper withSpeedLimit(long bytesPerSecond) {  2 usages  new *
    this.speedLimit = Math.max(0, bytesPerSecond);
    return this;
}

public String getDbPath() {  1 usage  new *
    return dbPath;
}

public int getBatchSize() {  no usages  new *
    return batchSize;
}

public long getSpeedLimit() {  2 usages  new *
    return speedLimit;
}
```


```java
public ConfigurationHelper withBatchSize(int batchSize) {  2 usages  new *
    this.batchSize = Math.max(1, batchSize);
    return this;
}

public ConfigurationHelper withSpeedLimit(long bytesPerSecond) {  2 usages  new *
    this.speedLimit = Math.max(0, bytesPerSecond);
    return this;
}

public String getDbPath() {  1 usage  new *
    return dbPath;
}

public int getBatchSize() {  no usages  new *
    return batchSize;
}

public long getSpeedLimit() {  2 usages  new *
    return speedLimit;
}
```


Рисунок 4 - Клас ConfigurationHelper для управління конфігурацією системи

ConfigurationHelper централізує всі налаштування системи: шлях до бази даних, розмір batch для Iterator Pattern, ліміт швидкості завантаження. Він використовує Fluent Interface (builder-подібний підхід) через методи withDatabase(), withBatchSize(), withSpeedLimit(), які повертають this для зручного ланцюжка викликів.

Клас надає готові конфігурації через статичні методи forTesting() (для тестового середовища з меншими параметрами) та forProduction() (для production з оптимальними налаштуваннями). Це дозволяє легко перемикатися між різними конфігураціями залежно від потреб.

4. Main.java - спрощений клієнтський код з використанням Facade

```
package dm.app;

import dm.core.DownloadTask;
import dm.facade.*;

import java.util.Locale;
import java.util.Scanner;

public class Main {  Anchoys19 *
    public static void main(String[] args) throws Exception {  Anchoys19 *
        ConfigurationHelper config = new ConfigurationHelper();

        DownloadManagerFacade manager = new DownloadManagerFacade(config.getDbPath());

        if (config.getSpeedLimit() > 0) {
            manager.setSpeedLimit(config.getSpeedLimit());
        }

        ConsoleUI.showWelcome();

        try (Scanner sc = new Scanner(System.in)) {
            while (true) {
                System.out.print("> ");
                if (!sc.hasNextLine()) break;
                String line = sc.nextLine().trim();
                if (line.isEmpty()) continue;

                String[] parts = line.split( regex: "\\s+");
                String cmd = parts[0].toLowerCase(Locale.ROOT);

                try {
                    switch (cmd) {
                        case "add" -> {
                            if (parts.length < 3) {
                                ConsoleUI.showError("Usage: add <url> <file>");
                                break;
                            }
                            int id = manager.addDownload(parts[1], parts[2]);
                            ConsoleUI.showSuccess("Task created: #" + id);
                        }
```

```
                case "pause" -> {
                    if (parts.length < 2) {
                        ConsoleUI.showError("Usage: pause <id>");
                        break;
                    }
                    int id = Integer.parseInt(parts[1]);
                    manager.pauseDownload(id);
                }

                case "resume" -> {
                    if (parts.length < 2) {
                        ConsoleUI.showError("Usage: resume <id>");
                        break;
                    }
                    int id = Integer.parseInt(parts[1]);
                    manager.resumeDownload(id);
                }
```

```
case "list" -> {
    String filter = (parts.length >= 2) ? parts[1].toLowerCase(Locale.ROOT) : "all";

    Iterable<DownloadTask> tasks;
    switch (filter) {
        case "completed" -> tasks = manager.getTasksByStatus(DownloadTask.Status.COMPLETED);
        case "paused"    -> tasks = manager.getTasksByStatus(DownloadTask.Status.PAUSED);
        case "running"   -> tasks = manager.getTasksByStatus(DownloadTask.Status.RUNNING);
        case "error"     -> tasks = manager.getTasksByStatus(DownloadTask.Status.ERROR);
        default          -> tasks = manager.getAllTasks();
    }

    ConsoleUI.showTaskList(tasks, filter.equals("all") ? null : filter);
}

case "stats" -> {
    manager.showStatistics();
}
```

```
case "undo" -> {
    if (manager.canUndo()) {
        manager.undoLastOperation();
    } else {
        ConsoleUI.showInfo("Nothing to undo");
    }
}

case "redo" -> {
    if (manager.canRedo()) {
        manager.redoLastOperation();
    } else {
        ConsoleUI.showInfo("Nothing to redo");
    }
}

case "history" -> {
    manager.showCommandHistory();
}
```

```
            case "help" -> {
                ConsoleUI.showWelcome();
            }

            case "exit" -> {
                ConsoleUI.showInfo("Shutting down...");
                manager.shutdown();
                return;
            }

            default -> ConsoleUI.showError("Unknown command. Type 'help' for available commands.");
        }
    catch (NumberFormatException e) {
        ConsoleUI.showError("Invalid number format");
    catch (Exception e) {
        ConsoleUI.showError(e.getMessage());
```

Рисунок 5 - Спрощений Main.java з використанням DownloadManagerFacade

Оновлений Main.java демонструє переваги Facade Pattern. Замість багатьох рядків ініціалізації (створення сервісу, менеджера команд, observer'ів, їх підписки) тепер достатньо двох рядків: створення ConfigurationHelper та ініціалізація DownloadManagerFacade.

Всі операції виконуються через простий API фасаду: manager.addDownload(), manager.pauseDownload(), manager.resumeDownload() тощо. Також використовується ConsoleUI для форматованого виведення, що робить інтерфейс більш зрозумілим та естетичним.

3. Продемонструвати результати

```
> Task :app:dm.app.Main.main()
================================================
  Download Manager - Simplified Facade Interface
================================================
Available commands:
  add <url> <file>  - Add new download
  pause <id>        - Pause download
  resume <id>       - Resume download
  list [status]     - List downloads
  stats             - Show statistics
  undo              - Undo last operation
  redo              - Redo operation
  history           - Show command history
  help              - Show this help
  exit              - Exit program
================================================
```

Рисунок 6 - Запуск програми з форматованим привітальним повідомленням

При запуску програми через Facade Pattern користувач бачить форматоване привітальне повідомлення з чітким списком доступних команд. Це створює ConsoleUI.showWelcome(), який є частиною Facade підсистеми.

```
add http://example.com/file1.zip test1.zip
[STARTED] Task #7: http://example.com/file1.zip
[OK] Task created: #7
```

Рисунок 7 - Створення завантаження через спрощений API Facade

Команда add демонструє спрощений API - замість створення Command об'єкта вручну, користувач просто викликає manager.addDownload(). Система автоматично створює завдання, запускає його, а Observer'и сповіщають про події ([STARTED], [ERROR]). Повідомлення "[OK] Task created: #6" виводиться через ConsoleUI.showSuccess().

```
> stats

=== Download Statistics ===
Total started: 2
Total completed: 0
Total paused: 0
Total errors: 2
Total downloaded: 0 bytes
============================
```

Рисунок 8 - Форматований список завантажень через ConsoleUI

Команда list показує всі завантаження у форматованому вигляді. ConsoleUI.showTaskList() створює таблицю з рамками, відображає статус кожного завантаження та прогрес у зрозумілому форматі (замість "(0/-1)" тепер "0 B / unknown"). Кожне завдання займає два рядки для кращої читабельності.

```
help
==================================================
  Download Manager - Simplified Facade Interface
==================================================
Available commands:
  add <url> <file>  - Add new download
  pause <id>        - Pause download
  resume <id>       - Resume download
  list [status]     - List downloads
  stats             - Show statistics
  undo              - Undo last operation
  redo              - Redo operation
  history           - Show command history
  help              - Show this help
  exit              - Exit program
==================================================
```

Рисунок 9 - Команда help показує довідку через Facade

Нова команда help (доступна завдяки Facade) дозволяє користувачу в будь-який момент переглянути список доступних команд. Це покращує user experience та робить систему більш зручною для використання.

```
> pause
> [ERROR] Usage: pause <id>
```

Рисунок 10 - Обробка помилок через ConsoleUI

При неправильному введенні команди (без параметрів) ConsoleUI.showError() виводить зрозуміле повідомлення "[ERROR] Usage: pause <id>". Це демонструє, що Facade не тільки спрощує API, але й покращує обробку помилок.

```
> pause 1
[PAUSED] Task #1
Paused task #1
> undo
Undone: PAUSE task #1
```

Рисунок 11 - Функціонал undo працює через Facade

Команда undo демонструє, що Facade не руйнує попередню функціональність - Command Pattern все ще працює. DownloadManagerFacade просто делегує виклик до CommandManager.undo(), приховуючи складність взаємодії.

```
> stats

=== Download Statistics ===
Total started: 2
Total completed: 0
Total paused: 0
Total errors: 2
Total downloaded: 0 bytes
============================
```

Рисунок 12 - Статистика через Facade

Команда stats викликає manager.showStatistics(), який внутрішньо звертається до StatisticsObserver.printStatistics(). Це показує, як Facade об'єднує всі підсистеми - Observer Pattern збирає дані, а Facade надає простий спосіб їх переглянути.

1.3. Висновки.

Facade Pattern не замінює попередні патерни (Iterator, Command, Observer), а доповнює їх, надаючи простий уніфікований інтерфейс. Всі підсистеми продовжують працювати як раніше:

- Iterator Pattern використовується для ефективного перегляду завдань (getAllTasks(), getTasksByStatus())
- Command Pattern забезпечує undo/redo функціонал (undoLastOperation(), redoLastOperation())
- Observer Pattern моніторить події завантажень (ConsoleObserver, StatisticsObserver)

Facade просто приховує складність їх взаємодії та ініціалізації, надаючи клієнту простий API. Це демонструє ключову перевагу патерну - зменшення складності системи для кінцевого користувача без втрати функціональності.

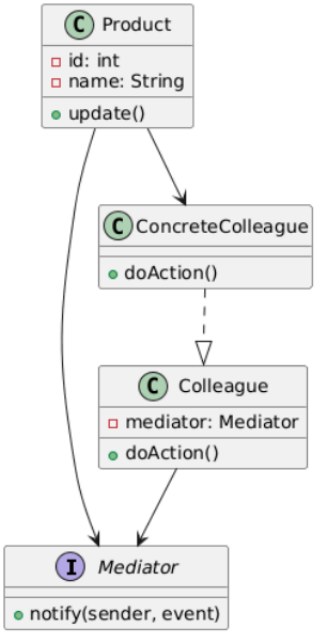
Додатково, створення helper-класів (ConsoleUI, ConfigurationHelper) як частини Facade підсистеми демонструє, що Facade може включати не тільки основний клас-обгортку, але й допоміжні компоненти для покращення зручності використання системи.

1.4 Контрольні запитання

1. Яке призначення шаблону «Посередник»?

Організовує взаємодію між об’єктами через центральний елемент, що дає змогу зменшити прямі зв’язки між ними.

2. Нарисуйте структуру шаблону «Посередник».



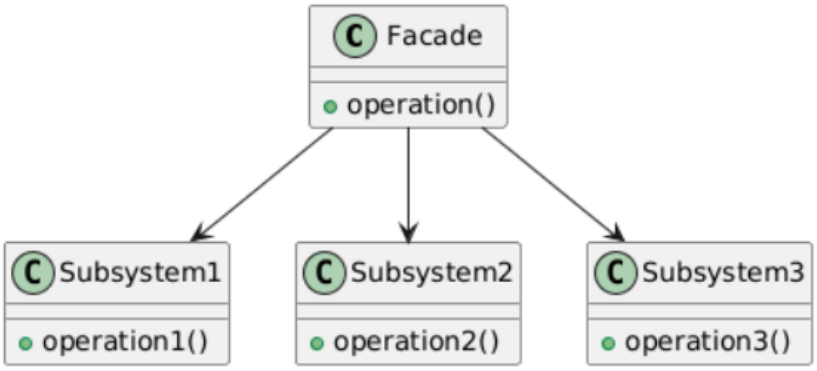
3. Які класи входять в шаблон «Посередник», та яка між ними взаємодія?

• **Mediator (інтерфейс)** - задає методи, через які відбувається взаємодія.
• **ConcreteColleague** - конкретна реалізація учасника взаємодії.
• **Colleague** - об’єкти, що обмінюються даними через медіатора.
Принцип взаємодії: колеги не мають інформації один про одного, вони знають лише медіатора; саме він керує їхньою комунікацією.

4. Яке призначення шаблону «Фасад»?

Забезпечує єдиний спрощений інтерфейс для складної підсистеми, приховуючи внутрішню логіку та деталі її роботи.

5. Нарисуйте структуру шаблону «Фасад».



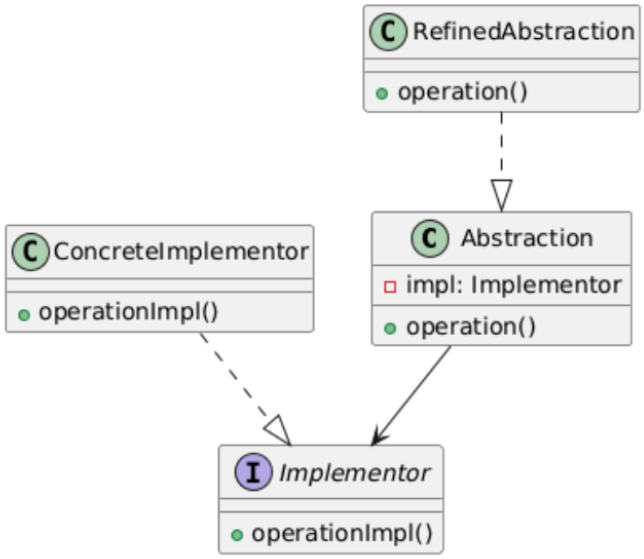
6. Які класи входять в шаблон «Фасад», та яка між ними взаємодія?

• **Facade** - забезпечує клієнтам спрощений інтерфейс для роботи з системою.
• **Subsystem** - внутрішні компоненти, з яких складається підсистема.
Взаємодія: клієнт працює виключно через фасад, а той уже координує роботу підсистем.

7. Яке призначення шаблону «Міст»?

Відокремлює абстракцію від її реалізації, дозволяючи змінювати їх незалежно одна від одної.

8. Нарисуйте структуру шаблону «Міст».



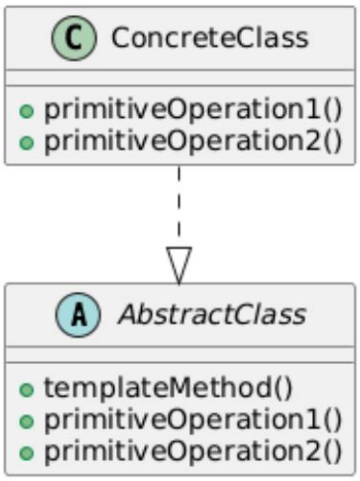
9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?

• **Abstraction** - базовий абстрактний інтерфейс.
• **RefinedAbstraction** - конкретизована версія цієї абстракції.
• **Implementor** - інтерфейс, який описує спосіб реалізації.
• **ConcreteImplementor** - конкретне втілення реалізації.
Взаємодія: абстракція передає виконання реалізації через інтерфейс Implementor, при цьому обидві ієрархії залишаються незалежними одна від одної.

10.Яке призначення шаблону «Шаблоний метод»?

Задає основну послідовність дій у базовому класі, а окремі кроки алгоритму делегує підкласам для специфічної реалізації.

11.Нарисуйте структуру шаблону «Шаблоний метод».



12.Які класи входять в шаблон «Шаблоний метод», та яка між ними взаємодія?

• **AbstractClass** - містить Template Method і оголошує абстрактні методи, які мають реалізувати підкласи.
• **ConcreteClass** - надає конкретну реалізацію окремих етапів алгоритму.
Взаємодія: Template Method викликає абстрактні методи, реалізовані в підкласах; підкласи змінюють лише певні кроки алгоритму, не змінюючи його структури.

13.Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?

Template Method формує загальну структуру алгоритму та передає виконання окремих його частин підкласам.
Factory Method визначає спосіб створення об’єктів: базовий клас задає інтерфейс, а підкласи вирішують, який саме об’єкт створити.

Основна різниця: Template Method регулює послідовність виконання операцій, тоді як Factory Method відповідає за процес створення об’єктів.

14.Яку функціональність додає шаблон «Міст»?

• Дозволяє змінювати абстракції та їхні реалізації окремо одна від одної.
• Полегшує масштабування системи без необхідності створювати велику кількість підкласів.
• Забезпечує гнучкість під час додавання нових типів абстракцій або нових реалізацій.