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: «Патерни проектування.»

Тема проекту: «26. Download Manager»

Виконала:

студентка групи ІА-34

Мушта Анна

Перевірив:

Мягкий Михайло Юрійович

Дата здачі 22.11.2025

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

Шаблон **«Abstract Factory»** використовується для створення сімейств взаємопов'язаних об'єктів без вказівки їх конкретних класів. Він гарантує, що всі створені об'єкти (продукти) будуть узгоджуватися між собою (наприклад, усі елементи кімнати будуть одного стилю). Шаблон відокремлює логіку створення від використання. Його перевага — легкість додавання нових сімейств (нових стилів), але недолік — складність додавання нового типу продукту (нового елемента) до існуючих фабрик.

Шаблон **«Factory Method»** визначає інтерфейс для створення об'єкта, дозволяючи підкласам вирішувати, який саме клас інстанціювати. Це дозволяє створювати об'єкти не базового, а дочірнього типу. Також відомий як «Віртуальний конструктор». Він позбавляє клас-творець від прив'язки до конкретних класів продуктів і спрощує додавання нових продуктів, але може призвести до великих паралельних ієрархій класів.

Шаблон **«Memento»** використовується для збереження та відновлення внутрішнього стану об'єкта (Originator) без порушення його інкапсуляції. Об'єкт-знімок (Memento) зберігається об'єктом-опікуном (Caretaker). Шаблон спрощує структуру вихідного об'єкта, оскільки той не зберігає історію версій свого стану. Часто використовується для реалізації функціоналу undo/redo. Недолік — може вимагати багато пам’яті.

Шаблон **«Observer»** визначає залежність «один-до-багатьох», при якій зміна стану одного об'єкта (Subject) автоматично сповіщає всіх залежних об'єктів (Observers), які можуть змінити свій стан у відповідь. Реалізує принцип слабкого зв'язку (наприклад, підписка на YouTube-канал). Переваги: спостерігачів можна додавати та видаляти динамічно, а також підтримується паралельна обробка повідомлень. Недолік — послідовність розсилки повідомлень не гарантується.

Шаблон **«Decorator»** призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. Декоратор «обгортає» початковий об'єкт зі збереженням його функцій, додаючи додаткові дії. Це гнучкіший спосіб зміни поведінки, ніж просте спадкування.

1.2. Хід роботи.

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

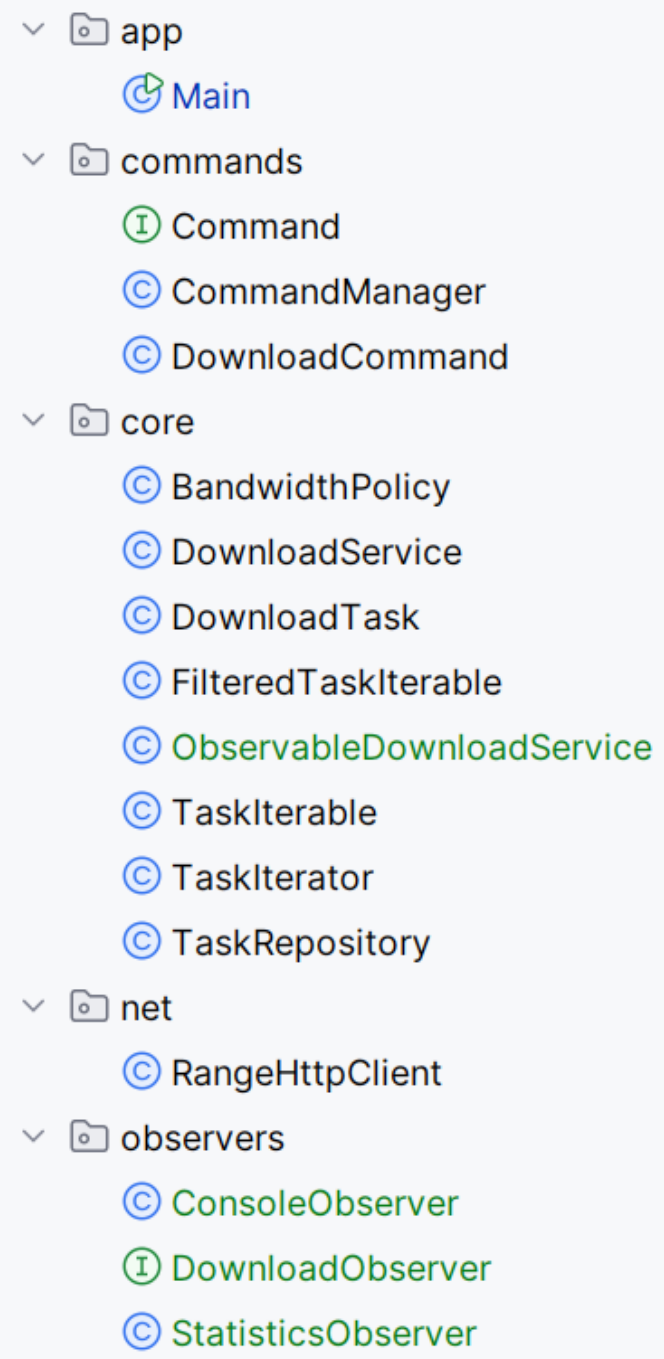


Рисунок 1 - Структура модулів і класів застосунку.

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

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.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.

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

Я обрала Observer Pattern для розширення функціоналу download-manager, тому що він ідеально доповнює вже реалізовану архітектуру та відповідає потребам системи моніторингу завантажень.

1. Моніторинг стану завантажень у реальному часі

У моєму download-manager важливо відстежувати поточний стан кожного завантаження: коли воно розпочалося, який прогрес виконання, чи

виникли помилки. Observer Pattern дозволяє автоматично сповіщати різні частини системи про зміни стану без створення жорсткої залежності між компонентами.

Завдяки цьому я можу легко додавати нові способи реагування на події (наприклад, логування у файл, відправка повідомлень, оновлення GUI) без зміни основної бізнес-логіки завантажень.

2. Множинні підписники на одні події

Однією з сильних сторін Observer Pattern є можливість мати декілька незалежних спостерігачів, які одночасно отримують сповіщення про одну й ту саму подію. У моєму випадку це означає, що одна подія "завантаження розпочато" може одночасно:

- Виводитись у консоль для користувача
- Записуватись у статистику для аналізу
- Логуватись у файл (при необхідності)

Кожен Observer обробляє подію по-своєму, але всі вони отримують інформацію одночасно та незалежно один від одного.

3. Збір статистики та аналітики

Observer Pattern ідеально підходить для збору статистики використання системи. StatisticsObserver може непомітно для користувача підраховувати кількість запущених, завершених, призупинених завантажень та загальний обсяг завантажених даних.

Це корисно для аналізу роботи системи, виявлення проблемних місць та надання користувачу детальної інформації про використання download-manager.

Тож перейдемо до реалізації

1. DownloadObserver.java - інтерфейс спостерігача з методами для різних подій

```java
public interface DownloadObserver {  11 usages  2 implementations  new *

    void onDownloadStarted(int taskId, String url);  1 usage  2 implementations  new *

    void onProgressUpdate(int taskId, long downloaded, long total);  1 usage  2 imp

    void onDownloadCompleted(int taskId);  1 usage  2 implementations  new *

    void onDownloadPaused(int taskId);  1 usage  2 implementations  new *

    void onDownloadError(int taskId, String error);  1 usage  2 implementations  new *
}
```

Рисунок 2 – Інтерфейс DownloadObserver з методами для сповіщення про події завантаження

Інтерфейс визначає п'ять методів для різних типів подій: початок завантаження, оновлення прогресу, завершення, пауза та помилка. Це дозволяє спостерігачам реагувати на всі важливі зміни стану.

2. **ConsoleObserver.java -** конкретний Observer для виведення подій у консоль

```java
    @Override  1 usage  new *
    public void onDownloadStarted(int taskId, String url) {
        System.out.println("[STARTED] Task #" + taskId + ": " + url);
    }

    @Override  1 usage  new *
    public void onProgressUpdate(int taskId, long downloaded, long total) {
        if (total > 0) {
            double percent = (downloaded * 100.0) / total;
            System.out.printf("[PROGRESS] Task #%d: %.1f%% (%d/%d bytes)%n",
                    taskId, percent, downloaded, total);
        } else {
            System.out.printf("[PROGRESS] Task #%d: %d bytes downloaded%n",
                    taskId, downloaded);
        }
    }

    @Override  1 usage  new *
    public void onDownloadCompleted(int taskId) {
        System.out.println("[COMPLETED] Task #" + taskId + " finished successfully!");
    }

@Override  1 usage  new *
public void onDownloadPaused(int taskId) {
    System.out.println("[PAUSED] Task #" + taskId);
}

@Override  1 usage  new *
public void onDownloadError(int taskId, String error) {
    System.err.println("[ERROR] Task #" + taskId + ": " + error);
}
}
```

Рисунок 3 – Клас ConsoleObserver, що виводить інформацію про події в консоль

ConsoleObserver реалізує інтерфейс DownloadObserver і відповідає за виведення інформації користувачу. При оновленні прогресу він розраховує відсоток завершення та форматує вивід у зручному для читання вигляді.

3. **StatisticsObserver.java -** конкретний Observer для збору статистики

```
public class StatisticsObserver implements DownloadObserver {  2 usages  new *

    private int totalStarted = 0;  2 usages
    private int totalCompleted = 0;  3 usages
    private int totalPaused = 0;  2 usages
    private int totalErrors = 0;  3 usages
    private final Map<Integer, Long> downloadedBytes = new HashMap<>();  2 usages

    @Override  1 usage  new *
    public void onDownloadStarted(int taskId, String url) {
        totalStarted++;
    }

    @Override  1 usage  new *
    public void onProgressUpdate(int taskId, long downloaded, long total) {
        downloadedBytes.put(taskId, downloaded);
    }

    @Override  1 usage  new *
    public void onDownloadCompleted(int taskId) {
        totalCompleted++;
    }
    @Override  1 usage  new *
    public void onDownloadPaused(int taskId) {
        totalPaused++;
    }

    @Override  1 usage  new *
    public void onDownloadError(int taskId, String error) {
        totalErrors++;
    }

    /**
     * Показати зібрану статистику
     */
    public void printStatistics() {  2 usages  new *
        System.out.println("\n=== Download Statistics ===");
        System.out.println("Total started: " + totalStarted);
        System.out.println("Total completed: " + totalCompleted);
        System.out.println("Total paused: " + totalPaused);
        System.out.println("Total errors: " + totalErrors);
```

```
        long totalBytes = downloadedBytes.values().stream() Stream<Long>
                .mapToLong(Long::longValue) LongStream
                .sum();
        System.out.println("Total downloaded: " + totalBytes + " bytes");
        System.out.println("============================\n");
    }

    public int getTotalCompleted() {  no usages  new *
        return totalCompleted;
    }

    public int getTotalErrors() {  no usages  new *
        return totalErrors;
    }
}
```

Рисунок 4 - Клас StatisticsObserver, що збирає статистику завантажень

StatisticsObserver непомітно для користувача відстежує всі події та накопичує статистику: кількість запущених, завершених, призупинених завантажень та помилок. Також він підраховує загальний обсяг завантажених даних через Map, де ключ - це ID завдання, а значення - кількість завантажених байтів.

4. **ObservableDownloadService.java** - Subject, що керує спостерігачами

```
public class ObservableDownloadService extends DownloadService {  3 usages  new *

    private final List<DownloadObserver> observers = new ArrayList<>();  8 usages
    private final TaskRepository repo;  6 usages
    private final RangeHttpClient http = new RangeHttpClient();  1 usage
    private final ExecutorService pool = Executors.newFixedThreadPool( nThreads: 3);  1 usage
    private final Map<Integer, RangeHttpClient.InterruptFlag> flags = new ConcurrentHashMap<>();
    private final BandwidthPolicy policy = new BandwidthPolicy();  1 usage

    public ObservableDownloadService(Path sqliteDb) throws Exception {  1 usage  new *
        super(sqliteDb);
        this.repo = super.getRepository();
    }

    public void attach(DownloadObserver observer) {  2 usages  new *
        if (!observers.contains(observer)) {
            observers.add(observer);
        }
    }

    public void detach(DownloadObserver observer) {  no usages  new *
        observers.remove(observer);
    }
```

```
private void notifyStarted(int taskId, String url) {  1 usage  new *
    for (DownloadObserver observer : observers) {
        observer.onDownloadStarted(taskId, url);
    }
}

private void notifyProgress(int taskId, long downloaded, long total) {
    for (DownloadObserver observer : observers) {
        observer.onProgressUpdate(taskId, downloaded, total);
    }
}

private void notifyCompleted(int taskId) {  1 usage  new *
    for (DownloadObserver observer : observers) {
        observer.onDownloadCompleted(taskId);
    }
}
 private void notifyPaused(int taskId) {  1 usage  new *
     for (DownloadObserver observer : observers) {
         observer.onDownloadPaused(taskId);
     }
 }

 private void notifyError(int taskId, String error) {  1 usage  ne
     for (DownloadObserver observer : observers) {
         observer.onDownloadError(taskId, error);
     }
 }
 public void resume(int id) throws Exception {
     DownloadTask t = repo.findById(id);
     if (t == null) throw new IllegalArgumentException("No such task: " + id);

     RangeHttpClient.InterruptFlag flag = new RangeHttpClient.InterruptFlag();
     flags.put(id, flag);

     repo.updateStatus(id, DownloadTask.Status.RUNNING, t.lastByte);
     notifyStarted(id, t.url.toString());

     pool.submit(() -> {
```

```
            try {
                LongSupplier lim = policy::getLimit;
                RangeHttpClient.Result r = http.download(
                        t.url, t.target, t.lastByte,
                        ( long bytes,  long total) -> {
                            try {
                                repo.updateProgress(id, bytes, total);
                                notifyProgress(id, bytes, total);
                            }
                            catch (SQLException e) { /* log */ }
                        },
                        flag, lim);

                repo.updateStatus(id, DownloadTask.Status.COMPLETED,
                        r.contentLength > 0 ? r.contentLength : t.lastByte);
                notifyCompleted(id);
            } catch (Exception e) {
                try {
                    repo.updateStatus(id, DownloadTask.Status.ERROR, t.lastByte);
                    notifyError(id, e.getMessage());
                }
                catch (SQLException ignored) {}
            }
        });
    }

    @Override  3 usages  new *
    public void pause(int id) throws Exception {
        super.pause(id);
        notifyPaused(id);
    }
}
```

Рисунок 5 - Клас ObservableDownloadService, що розширює DownloadService з підтримкою Observer Pattern

ObservableDownloadService розширює базовий DownloadService, додаючи функціонал управління спостерігачами. Він зберігає список підписаних Observer'ів та сповіщає їх про всі події через приватні методи notify. При кожній зміні стану завантаження (старт, прогрес, завершення, пауза, помилка) викликається відповідний notify-метод, який проходить по всім спостерігачам та викликає їхні методи обробки подій.

3. Продемонструвати результати

```
> add http://example.com/file2.zip test2.zip
[STARTED] Task #4: http://example.com/file2.zip
```

Рисунок 5 - Створення завантажень з автоматичним сповіщенням ConsoleObserver

При додаванні нових завантажень ConsoleObserver автоматично виводить повідомлення [STARTED] з номером завдання та URL. Це демонструє, що спостерігач отримав подію про початок завантаження.

```
> pause 3
[PAUSED] Task #3
Paused task #3
> pause 4
[PAUSED] Task #4
Paused task #4
```

Рисунок 6 - Призупинення завантаження з виведенням події [PAUSED]

При виконанні команди pause ConsoleObserver виводить повідомлення про призупинення завдання. Це показує, що Observer Pattern коректно працює разом з Command Pattern - команда виконується, а спостерігачі отримують сповіщення.

```
> list paused
#3 [PAUSED] http://example.com/file1.zip (0/-1) -> test1.zip
#4 [PAUSED] http://example.com/file2.zip (0/-1) -> test2.zip
```

Рисунок 7 - Перегляд списку завантажень та їхніх статусів

Команда list показує поточний стан всіх завантажень. Статуси PAUSED, ERROR, RUNNING змінюються завдяки сповіщенням через Observer Pattern.

```
stats

=== Download Statistics ===
Total started: 6
Total completed: 0
Total paused: 2
Total errors: 6
Total downloaded: 0 bytes
============================
```

Рисунок 8 - Статистика, зібрана StatisticsObserver

Команда stats демонструє роботу StatisticsObserver. Він незалежно від ConsoleObserver відстежував всі події та зібрав статистику: кількість запущених завантажень (Total started), завершених (Total completed), призупинених (Total paused), помилок (Total errors) та загальний обсяг завантажених даних.

```
> undo
[PAUSED] Task #3
Undo: Paused task #3
Undone: RESUME task #3
> redo
[STARTED] Task #3: http://example.com/file1.zip
Resumed task #3
Redone: RESUME task #3
```

Рисунок 9 - Інтеграція Command Pattern з Observer Pattern через undo/redo

Демонстрація сумісності Observer Pattern з раніше реалізованим Command Pattern. При виконанні undo для скасування паузи ConsoleObserver виводить Undo: Resumed task, а при redo - повідомлення про повторне призупинення. Історія команд показує всі виконані операції.

```
exit

=== Download Statistics ===
Total started: 7
Total completed: 0
Total paused: 3
Total errors: 7
Total downloaded: 0 bytes
==========================
```

Рисунок 10 - Автоматичне виведення фінальної статистики при виході

При завершенні роботи програми (команда exit) автоматично виводиться фінальна статистика, зібрана StatisticsObserver. Це демонструє, що спостерігач працював протягом всієї сесії та зберігав дані про всі події.

1.3. Висновки.

Під час тестування спостерігалися помилки HTTP 404 та SSL сертифікатів, оскільки використовувалися тестові URL-адреси. Проте це не впливає на демонстрацію Observer Pattern, оскільки основна мета - показати, що:

1. При кожній зміні стану завантаження (start, pause, error, complete) автоматично сповіщаються всі підписані спостерігачі
2. ConsoleObserver коректно виводить події в консоль у читабельному форматі
3. StatisticsObserver коректно збирає статистику операцій незалежно від ConsoleObserver
4. Спостерігачі працюють незалежно один від одного та не знають про існування інших Observer'ів
5. ObservableDownloadService (Subject) не залежить від конкретних реалізацій спостерігачів

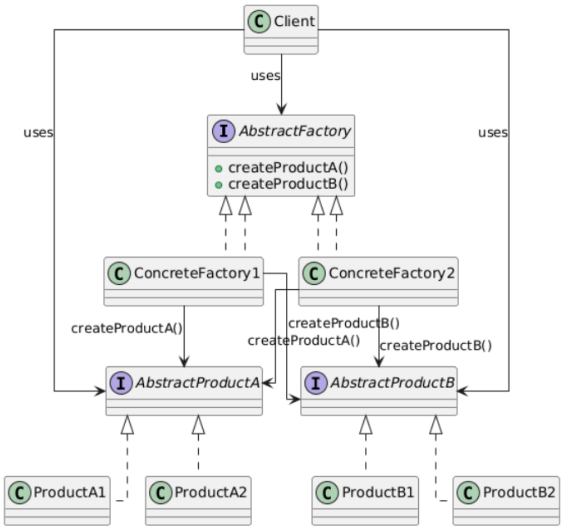
Це підтверджує правильність реалізації патерну Observer та його інтеграцію з існуючою архітектурою download-manager.

1.4 Контрольні запитання

1. Яке призначення шаблону «Абстрактна фабрика»?

Дозволяє створювати сімейства взаємопов’язаних об’єктів без вказівки їх конкретних класів. Забезпечує узгодженість об’єктів одного стилю та відокремлює процес їх створення від використання.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



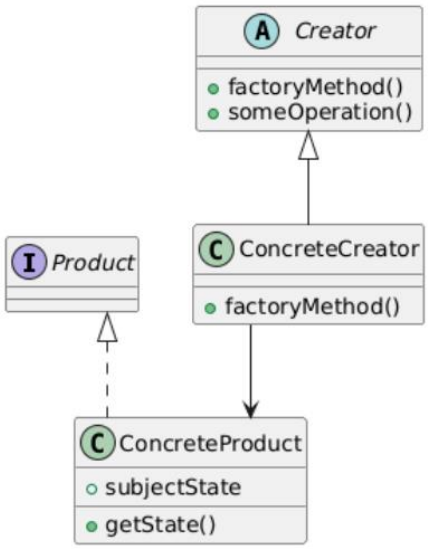
3. Які класи входять в шаблон «Абстрактна фабрика», та яка між ними взаємодія?

- AbstractFactory — інтерфейс для створення продуктів.
- ConcreteFactory — реалізація конкретної фабрики, створює конкретні
- продукти.
- AbstractProduct — інтерфейс продукту.
- ConcreteProduct — реалізація конкретного продукту.
  Взаємодія: клієнт використовує фабрику через AbstractFactory для створення продуктів одного стилю.

4. Яке призначення шаблону «Фабричний метод»?

Визначає інтерфейс для створення об’єктів певного базового типу і дозволяє підставляти підкласи без зміни коду клієнта.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

- Creator — базовий клас, що визначає фабричний метод.
- ConcreteCreator — реалізація фабричного методу, створює
- конкретний продукт.
- Product — базовий інтерфейс продукту.
- ConcreteProduct — конкретна реалізація продукту.
  Взаємодія: клієнт використовує Creator для створення об’єкта через фабричний метод, не знаючи конкретного підкласу.

7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

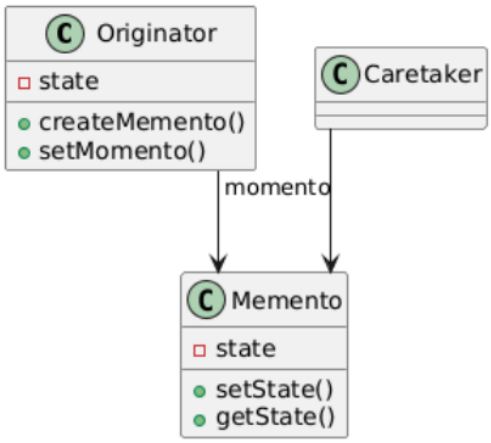
Abstract Factory створює сімейства взаємопов’язаних об’єктів (кілька типів продуктів), узгоджених між собою.

Factory Method створює один тип об’єкта і дозволяє підставляти підкласи без зміни коду клієнта.

8. Яке призначення шаблону «Знімок»?

Дозволяє зберігати та відновлювати стан об’єкта без порушення інкапсуляції. Використовується для реалізації undo/redo.

9. Нарисуйте структуру шаблону «Знімок».



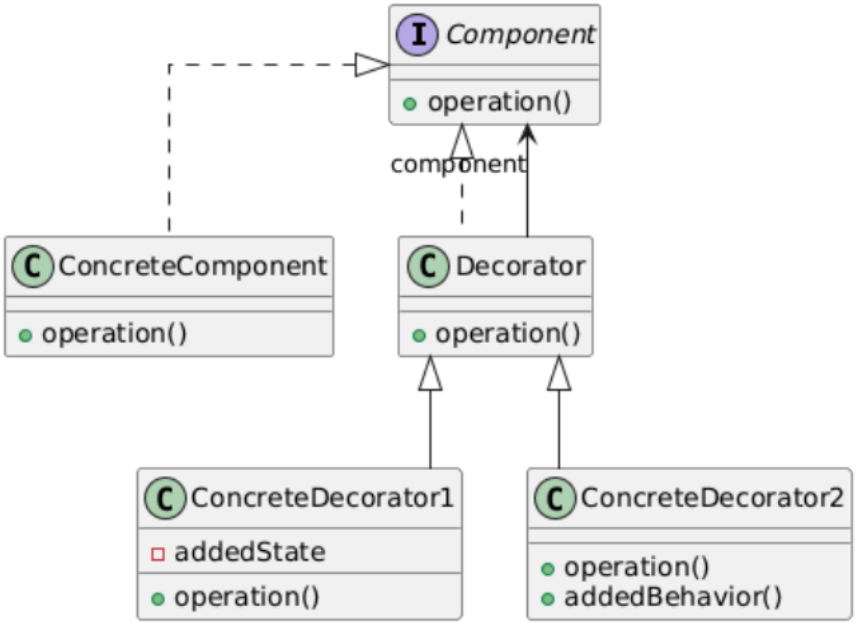
10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

- Originator — об’єкт, стан якого потрібно зберегти.
- Memento — об’єкт для зберігання стану.
- Caretaker — управляє збереженими станами.
  Взаємодія: Originator створює Memento, Caretaker зберігає його, а пізніше передає назад для відновлення стану

11. Яке призначення шаблону «Декоратор»?

Дозволяє динамічно додавати функціональні можливості об’єкту без зміни його коду, обертаючи його в новий клас.

12. Нарисуйте структуру шаблону «Декоратор».



13. Які класи входять в шаблон «Декоратор», та яка між ними взаємодія?

- Component — базовий інтерфейс або клас об’єкта.
- ConcreteComponent — конкретний об’єкт, який можна декорувати.
- Decorator — базовий клас для декораторів, містить посилання на
- Component.
- ConcreteDecorator — додає нові обов’язки до об’єкта.
  Взаємодія: Decorator обертає Component і виконує додаткові дії під час виклику operation().

14. Які є обмеження використання шаблону «декоратор»?

Велика кількість дрібних класів ускладнює систему. Складно конфігурувати об’єкти, які обгорнуті в декілька обгорток одночасно.