Міністерство освіти і науки України Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №6

Технології розробки програмного забезпечення

«Патерни проєктування»

«Веб-браузер»

Виконав:
студент групи ІА-33
Мартинюк Ю.Р.

Перевірив:
Мягкий Михайло Юрійович

Київ 2025

Тема: Патерни проєктування

Мета: Вивчити структуру шаблонів “Abstract Factory”, “Factory Method”, “Memento”, “Observer”, “Decorator” та навчитися застосовувати їх в реалізації програмної системи.

Зміст

Завдання

• Ознайомитись з короткими теоретичними відомостями.

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

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

Теоретичні відомості

Abstract Factory

Патерн Abstract Factory забезпечує створення сімейств пов’язаних об’єктів без вказування їхніх конкретних класів. Він визначає інтерфейс для створення різних типів продуктів, дозволяючи змінювати цілі набори об’єктів, не змінюючи

код клієнта. Зазвичай використовується тоді, коли система повинна бути незалежною від способу створення та представлення об’єктів, а також коли потрібно гарантувати узгодженість між продуктами, що належать до однієї родини.

Factory Method

Factory Method визначає інтерфейс для створення об’єктів, але дозволяє підкласам вирішувати, який саме клас створювати. Таким чином, він делегує створення об’єктів підкласам, уникаючи прямої залежності від конкретних реалізацій. Цей патерн полегшує розширення системи новими типами об’єктів без зміни вже існуючого коду, що підвищує гнучкість і розширюваність програми.

Memento

Патерн Memento використовується для збереження та відновлення попереднього стану об’єкта без порушення інкапсуляції. Він зберігає знімок внутрішнього стану об’єкта у спеціальному об’єкті Memento, який потім може бути використаний для відновлення цього стану. Це часто застосовується у програмах із функціональністю “скасування” (Undo) або відновлення попередніх версій даних.

Observer

Observer визначає залежність “один-до-багатьох” між об’єктами, при якій зміна стану одного об’єкта автоматично повідомляє всі залежні об’єкти. Об’єкт-спостерігач (Observer) підписується на події суб’єкта (Subject) і отримує сповіщення про зміни. Цей патерн часто використовується для реалізації систем оповіщення, реактивних інтерфейсів і подій у GUI.

Decorator

Патерн Decorator дозволяє динамічно додавати нову поведінку або функціональність об’єктам без зміни їхнього коду. Він обгортає об’єкт у

спеціальний клас-декоратор, який реалізує той самий інтерфейс і делегує виклики до оригінального об’єкта, додаючи нову логіку. Це забезпечує гнучку альтернативу наслідуванню при розширенні функціональності.

Тема проєкту

6. Web-browser (proxy, chain of responsibility, factory method, template method, visitor, p2p) Веб-браузер повинен мати можливість зробити наступне: мати адресний рядок для введення адреси сайту, переміщатися і відображати структуру html документа, переглядати підключений javascript та css файли, перегляд всіх підключених ресурсів (зображень), коректна обробка відповідей з сервера (коди відповідей HTTP) – переходи при перенаправленнях, відображення сторінок 404 і 502/503.

Патерн: Factory Method

Посилання на Github: https://github.com/masonabor/web-browser-lab

Хід роботи

Діаграма класів

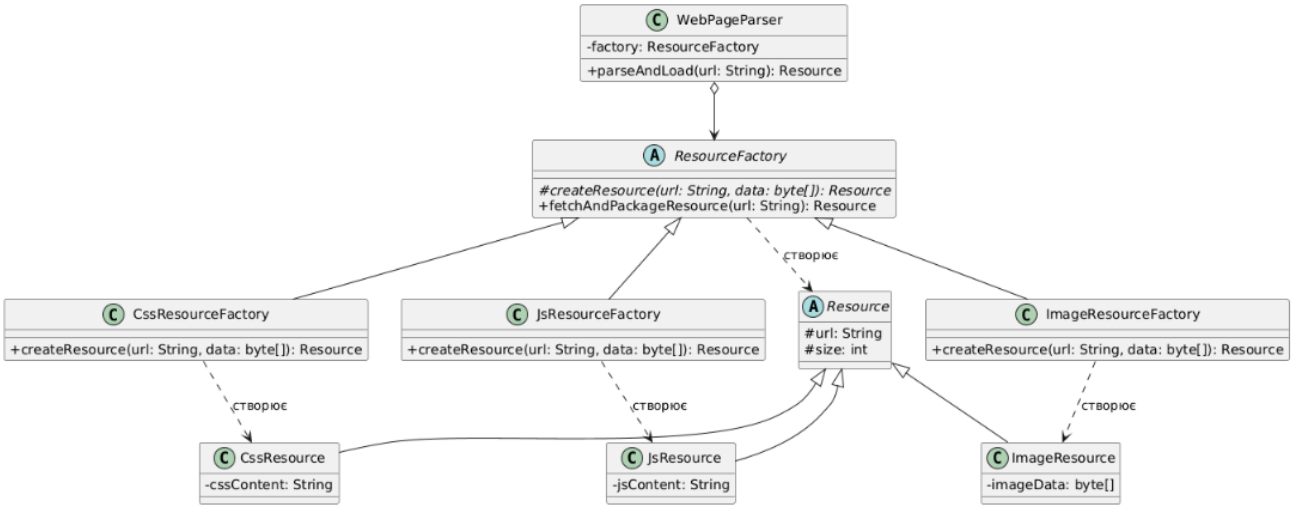


Рис.1 – Діаграма класів для ResourceFactory (патерн Factory Method)

Опис діаграми класів

Ця діаграма класів ілюструє, як процес створення об'єктів-ресурсів (наприклад, CSS, JS, Image) інкапсульовано за допомогою патерну "Фабричний Метод" (Factory Method).

Замість того, щоб один клас WebPageParser містив складну логіку (великий switch або if-else блок) для вирішення, який тип ресурсу створити (new CssResource(), new JsResource() тощо), ця відповідальність делегується ієрархії спеціалізованих класів-"фабрик".

Учасники патерну "Фабричний Метод"

На діаграмі чітко видно всіх ключових учасників патерну:

1. Клієнт (Client)

- Клас: WebPageParser
- Роль: Це ініціатор запиту на створення. Він знає, що йому потрібен Resource, але не знає, як створити його конкретний підтип.
- Взаємодія: WebPageParser має посилання на абстрактну ResourceFactory. Він викликає загальний метод fetchAndPackageResource(), щоб отримати готовий об'єкт Resource. (Примітка: в реалізації WebPageParser сам вирішує, яку *конкретну* фабрику створити, залежно від типу URL).

2. Абстрактний Продукт (Abstract Product)

- Клас: Resource
- Роль: Це загальний абстрактний клас для всіх об'єктів (продуктів), які можуть бути створені. WebPageParser працює з об'єктами саме цього типу, не знаючи про їхні конкретні реалізації.

- Ключова деталь: Визначає загальні поля, такі як url та size.

3. Конкретні Продукти (Concrete Products)
   - Класи: CssResource, JsResource, ImageResource
   - Роль: Це конкретні реалізації Resource. Кожен клас додає власні специфічні дані (наприклад, cssContent, jsContent, imageData). Саме ці об'єкти є кінцевою метою створення.
4. Абстрактний Творець (Abstract Creator)
   - Клас: ResourceFactory
   - Роль: Це абстрактний клас "фабрики". Він надає загальний інтерфейс для створення продуктів.
   - Ключова деталь: Він містить абстрактний фабричний метод createResource(). Він також може містити загальну логіку (як fetchAndPackageResource()), яка є спільною для всіх творців, але делегує сам крок створення об'єкта своїм підкласам.
5. Конкретні Творці (Concrete Creators)
   - Класи: CssResourceFactory, JsResourceFactory, ImageResourceFactory
   - Роль: Це реальні "робочі" фабрики. Кожен із них успадковує ResourceFactory та надає конкретну реалізацію фабричного методу createResource().
   - Взаємодія:
     - CssResourceFactory реалізує createResource() для повернення нового об'єкта CssResource.
     - JsResourceFactory реалізує createResource() для повернення нового об'єкта JsResource.
     - ImageResourceFactory реалізує createResource() для повернення нового об'єкта ImageResource.

Як працює потік (The Flow)

1. WebPageParser аналізує HTML і знаходить URL ресурсу, наприклад, "style.css".
2. На основі розширення файлу .css, він приймає рішення і створює екземпляр CssResourceFactory.
3. WebPageParser викликає загальний метод factory.fetchAndPackageResource("style.css") (де factory - це CssResourceFactory).
4. Метод fetchAndPackageResource() (всередині ResourceFactory) виконує загальну логіку (наприклад, завантажує дані).
5. Потім він викликає *абстрактний* метод createResource().

6. Оскільки реальним об'єктом є CssResourceFactory, саме його реалізація createResource() виконується, створюючи та повертаючи об'єкт CssResource.
7. WebPageParser отримує готовий CssResource, але продовжує працювати з ним через абстрактний тип Resource.

Частина коду програми з використанням патерну Chain of responsibility

Resource.java

```
package com.edu.web.restservicewebbrowser.domain.resource;

import com.edu.web.restservicewebbrowser.visitor.resource.IResourceVisitor;

public abstract class Resource {
   protected String url;

   public abstract void visit(IResourceVisitor visitor) throws Exception;

   public Resource(String url) {
      this.url = url;
   }

   public String getUrl() {
      return url;
   }

   @Override
   public String toString() {
      return this.getClass().getSimpleName() + " [url=" + url + "]";
   }
}
```

CssResource.java

```
package com.edu.web.restservicewebbrowser.factory.resource;

import com.edu.web.restservicewebbrowser.domain.resource.CssResource;
import com.edu.web.restservicewebbrowser.domain.resource.Resource;
```

```
public class CssResourceFactory extends ResourceFactory {

    @Override
    protected Resource createResource(String url, byte[] rawData) {
        String cssContent = new String(rawData);

        return new CssResource(url, cssContent);
    }
}
```

ImageResource.java

```
package com.edu.web.restservicewebbrowser.domain.resource;

import com.edu.web.restservicewebbrowser.visitor.resource.IResourceVisitor;

public class ImageResource extends Resource {
    private final byte[] imageData;

    public ImageResource(String url, byte[] imageData) {
        super(url);
        this.imageData = imageData;
    }

    public byte[] getImageData() {
        return imageData;
    }

    @Override
    public void visit(IResourceVisitor visitor) throws Exception {
        visitor.visit(this);
    }
}
```

JsResource.java

```
package com.edu.web.restservicewebbrowser.domain.resource;

import com.edu.web.restservicewebbrowser.visitor.resource.IResourceVisitor;

public class JsResource extends Resource {
    private final String jsContent;
```

```
    public JsResource(String url, String jsContent) {
        super(url);
        this.jsContent = jsContent;
    }

    public String getJsContent() {
        return jsContent;
    }

    @Override
    public void visit(IResourceVisitor visitor) throws Exception {
        visitor.visit(this);
    }
}
```

ResourceFactory.java

```
package com.edu.web.restservicewebbrowser.factory.resource;


import com.edu.web.restservicewebbrowser.domain.resource.Resource;


public abstract class ResourceFactory {

    protected abstract Resource createResource(String url, byte[] rawData);

    public Resource fetchAndPackageResource(String url) {
        byte[] rawData = downloadFromUrl(url);

        Resource resource = createResource(url, rawData);

        System.out.println("Фабрика " + this.getClass().getSimpleName() + " створила:
" + resource.toString());
        return resource;
    }

    private byte[] downloadFromUrl(String url) {
        System.out.println("Завантаження даних з " + url + "...");
        // додати реальний виклик http
```

```
        return ("(сирі байти для " + url + ")").getBytes();
    }
}
```

JsResource.java

```
package com.edu.web.restservicewebbrowser.factory.resource;

import com.edu.web.restservicewebbrowser.domain.resource.JsResource;
import com.edu.web.restservicewebbrowser.domain.resource.Resource;

public class JsResourceFactory extends ResourceFactory {

    @Override
    protected Resource createResource(String url, byte[] rawData) {
        String jsContent = new String(rawData);

        return new JsResource(url, jsContent);
    }
}
```

ImageResource.java

```
package com.edu.web.restservicewebbrowser.factory.resource;

import com.edu.web.restservicewebbrowser.domain.resource.ImageResource;
import com.edu.web.restservicewebbrowser.domain.resource.Resource;

public class ImageResourceFactory extends ResourceFactory {

    @Override
    protected Resource createResource(String url, byte[] rawData) {
        return new ImageResource(url, rawData);
    }
}
```

CssResource.java

```
package com.edu.web.restservicewebbrowser.factory.resource;

import com.edu.web.restservicewebbrowser.domain.resource.CssResource;
import com.edu.web.restservicewebbrowser.domain.resource.Resource;
```

```
public class CssResourceFactory extends ResourceFactory {

    @Override
    protected Resource createResource(String url, byte[] rawData) {
        String cssContent = new String(rawData);

        return new CssResource(url, cssContent);
    }
}
```

ParsingServiceImpl.java

```
package com.edu.web.restservicewebbrowser.service.impl;

import com.edu.web.restservicewebbrowser.domain.resource.Resource;
import com.edu.web.restservicewebbrowser.factory.resource.CssResourceFactory;
import com.edu.web.restservicewebbrowser.factory.resource.ImageResourceFactory;
import com.edu.web.restservicewebbrowser.factory.resource.JsResourceFactory;
import com.edu.web.restservicewebbrowser.factory.resource.ResourceFactory;
import com.edu.web.restservicewebbrowser.repository.WebPageRepository;
import com.edu.web.restservicewebbrowser.service.ParsingService;
import com.edu.web.restservicewebbrowser.service.ResourceService;
import jakarta.ejb.Asynchronous;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;


@ApplicationScoped
public class ParsingServiceImpl implements ParsingService {

    @Inject
    private WebPageRepository webPageRepository;

    @Inject
    private ResourceService resourceService;
```

```
    @Override
    @Asynchronous
    public void parseAndSaveAll(String pageUrl) {
        System.out.println("Розпочато асинхронний парсинг для: " + pageUrl);
        try {
            Document doc = Jsoup.connect(pageUrl).get();

            int webPageId = webPageRepository.saveAndGetId(pageUrl, doc.html());

            Elements cssLinks = doc.select("link[rel=stylesheet]");
            for (Element link : cssLinks) {
                processResource(link.absUrl("href"), webPageId);
            }

            Elements jsLinks = doc.select("script[src]");
            for (Element link : jsLinks) {
                processResource(link.absUrl("src"), webPageId);
            }

            Elements imgLinks = doc.select("img[src]");
            for (Element link : imgLinks) {
                processResource(link.absUrl("src"), webPageId);
            }

            System.out.println("Парсинг та збереження для " + pageUrl + " (ID=" +
webPageId + ") завершено.");

        } catch (Exception e) {
            System.err.println("Помилка парсингу: " + e.getMessage());
        }
    }

    private void processResource(String url, int webPageId) {
        try {
            ResourceFactory factory;

            if (url.endsWith(".css")) {
                factory = new CssResourceFactory();
            } else if (url.endsWith(".js")) {
                factory = new JsResourceFactory();
            } else if (url.endsWith(".png") || url.endsWith(".jpg") || url.endsWith(".svg")) {
```

```
            factory = new ImageResourceFactory();
        } else {
            return;
        }

        Resource resource = factory.fetchAndPackageResource(url);

        resourceService.saveResource(resource, webPageId);

    } catch (Exception e) {
        System.err.println("Не вдалося обробити ресурс: " + url + " | " +
e.getMessage());
    }
  }
}
```

ResourceServiceImpl.java

```
package com.edu.web.restservicewebbrowser.service.impl;

import com.edu.web.restservicewebbrowser.domain.resource.Resource;
import com.edu.web.restservicewebbrowser.repository.ResourceRepository;
import com.edu.web.restservicewebbrowser.service.ResourceService;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

@ApplicationScoped
public class ResourceServiceImpl implements ResourceService {

  @Inject
  private ResourceRepository resourceRepository;

  public void saveResource(Resource resource, int webPageId) {
    resourceRepository.save(resource, webPageId);
  }
}
```

Опис коду з використання патерну Chain of responsibility

Цей код демонструє, як асинхронний сервіс парсингу (ParsingServiceImpl) використовує набір патернів для ефективної обробки та збереження веб-ресурсів.

Замість того, щоб один гігантський клас ParsingServiceImpl знав, *як* створювати CssResource і *як* зберігати JsResource, ця логіка розумно розділена: патерн **Factory Method** використовується для *створення* об'єктів-ресурсів, а патерн **Visitor** (для якого підготовлено код) використовується для їх *обробки* (наприклад, для збереження).

Ролі учасників патерну "Factory Method"

- **"Клієнт" (Client)**
  - **Клас:** ParsingServiceImpl
  - **Роль:** Виступає в ролі "Клієнта" для Фабричного Методу. Його відповідальність — проаналізувати HTML (за допомогою Jsoup), визначити, який тип ресурсу потрібен (на основі розширення файлу), і **створити відповідну фабрику** (CssResourceFactory, JsResourceFactory тощо).
  - **Взаємодія:** Клієнт викликає загальний метод factory.fetchAndPackageResource(url), отримуючи назад абстрактний Resource, не знаючи деталей його створення.
- **"Абстрактний Творець" (Creator)**
  - **Клас:** ResourceFactory
  - **Роль:** Це абстрактний клас, який визначає "контракт" для всіх фабрик. Він містить **фабричний метод** createResource(), який змушує підкласи реалізувати логіку створення. Він також надає спільну логіку (fetchAndPackageResource), яка використовує цей фабричний метод.
- **"Конкретні Творці" (Concrete Creators)**
  - **Класи:** CssResourceFactory, JsResourceFactory, ImageResourceFactory
  - **Роль:** Це реальні "робочі" фабрики. Кожна реалізує метод createResource() і повертає **конкретний продукт**: CssResource, JsResource або ImageResource.
- **"Абстрактний Продукт" (Product)**
  - **Клас:** Resource
  - **Роль:** Це загальний абстрактний клас для всіх ресурсів, які можна створити.
- **"Конкретні Продукти" (Products)**
  - **Класи:** CssResource, JsResource, ImageResource
  - **Роль:** Це кінцеві об'єкти, які створюються фабриками та містять специфічні дані (cssContent, jsContent тощо).

Ролі учасників патерну "Visitor" (Підготовка)

Код також демонструє підготовку до використання патерну **Visitor** для обробки

створених ресурсів.

- **"Елемент" (Element)**
  - **Клас:** Resource (та його нащадки JsResource, ImageResource тощо)
  - **Роль:** Це об'єкти, які можна "відвідати" (тобто обробити). Абстрактний Resource оголошує метод visit(IResourceVisitor).
  - **Взаємодія:** Кожен конкретний клас (JsResource, ImageResource) реалізує цей метод, викликаючи visitor.visit(this). Це ключовий елемент патерну ("подвійна диспетчеризація"), який дозволяє візитору викликати правильний метод для правильного типу, уникаючи перевірок instanceof.
- **"Відвідувач" (Visitor)**
  - **Інтерфейс:** IResourceVisitor (згадується в Resource.java)
  - **Роль:** (Хоча реалізація IResourceVisitor не показана, його роль мається на увазі). Це буде інтерфейс для класів, які містять логіку, що має бути застосована до ресурсів. Наприклад, ResourceServiceImpl буде використовувати реалізацію ResourceSaveVisitor для збереження кожного типу ресурсу в свою окрему таблицю в БД.

Переваги підходу

Ця архітектура є чудовим прикладом застосування патернів. ParsingServiceImpl (клієнт) повністю відокремлений від логіки *створення* ресурсів (за це відповідає **Factory Method**). Крім того, ResourceServiceImpl (який отримує ці ресурси) буде відокремлений від логіки *обробки* кожного типу (за це відповідатиме **Visitor**). Якщо завтра знадобиться додати FontResource, достатньо буде додати FontResource.java, FontResourceFactory.java та оновити IResourceVisitor, не чіпаючи існуючу логіку парсингу чи збереження.

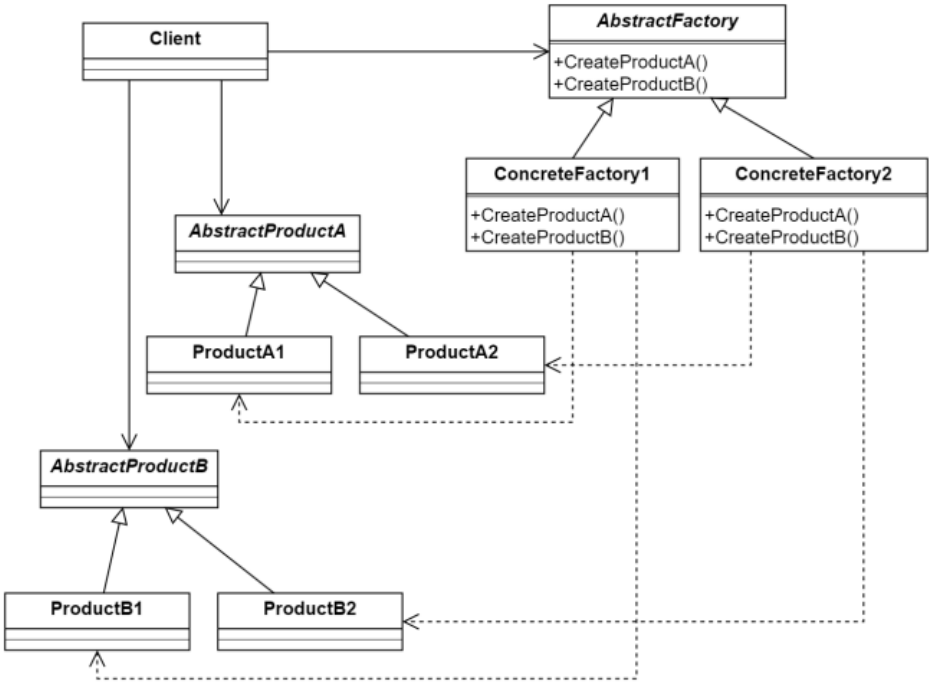
Контрольні запитання

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика» має на меті надати єдиний інтерфейс для створення сімейств взаємопов'язаних або залежних об'єктів, не вказуючи їхні конкретні класи. Це патерн, який допомагає створювати групи об'єктів, що мають бути сумісними між собою. Наприклад, уявіть, що ви створюєте інтерфейс користувача, який може мати два стилі: "Темний" та "Світлий". Вам потрібна родина об'єктів: Button, Checkbox, Window. Абстрактна фабрика GUIFactory матиме методи createButton(), createCheckbox(). А конкретні реалізації DarkFactory та LightFactory створюватимуть DarkButton і

DarkCheckbox або LightButton і LightCheckbox відповідно, гарантуючи, що всі елементи інтерфейсу будуть з одного стилю.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

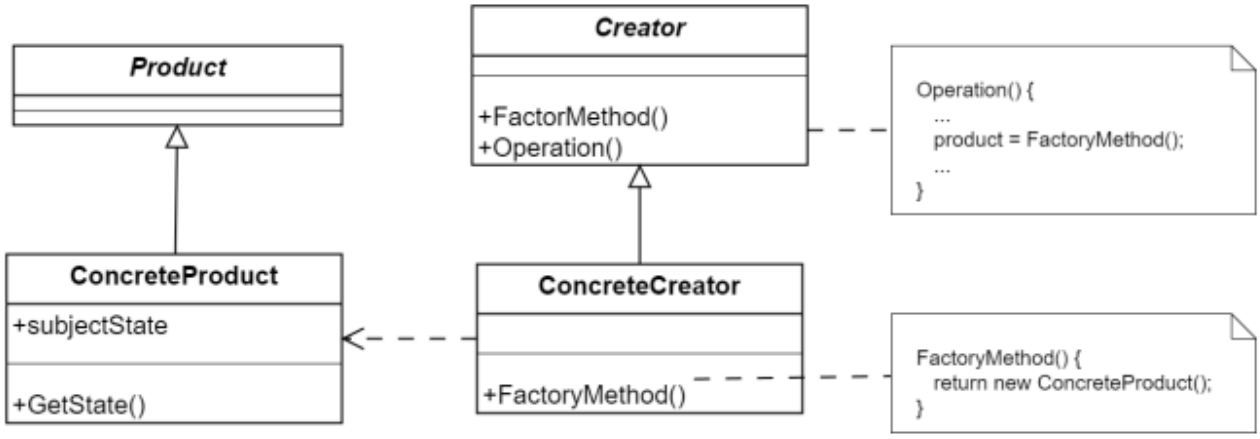
До шаблону «Абстрактна фабрика» входять класи Клієнта (Client), Абстрактної Фабрики (AbstractFactory), Конкретних Фабрик (ConcreteFactory) та ієрархії Абстрактних і Конкретних Продуктів. Взаємодія відбувається так: Клієнт вирішує, яка "тема" або "сімейство" йому потрібні, і створює екземпляр відповідної Конкретної Фабрики (наприклад, new DarkFactory()). Потім Клієнт працює з цією фабрикою виключно через інтерфейс Абстрактної Фабрики. Він просить фабрику створити продукти, наприклад, factory.createButton(). Клієнт не знає (і не повинен знати), що він отримав саме DarkButton, він просто працює з ним через загальний інтерфейс Button. Це гарантує, що всі отримані продукти сумісні.

4. Яке призначення шаблону «Фабричний метод»?

Призначення шаблону «Фабричний метод» (Factory Method) — це визначити інтерфейс для створення об'єкта, але дозволити підкласам вирішувати, екземпляр якого саме класу створювати. Цей патерн делегує логіку інстанціювання (створення) об'єкта підкласам. Основна ідея — замінити прямий виклик конструктора (new Product()) на виклик спеціального "фабричного методу" (createProduct()). Це дозволяє головному класу працювати з абстрактним

"Продуктом", не знаючи, яку з його багатьох варіацій він насправді створив.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

До шаблону «Фабричний метод» входять класи Творця (Creator), Конкретних Творців (ConcreteCreators), Продукту (Product) та Конкретних Продуктів (ConcreteProducts). Взаємодія така: Клієнтський код зазвичай працює з Конкретним Творцем (наприклад, new CssResourceFactory()). Творець, окрім фабричного методу, може мати й іншу корисну логіку (наприклад, ваш fetchAndPackageResource()). Коли ця логіка доходить до моменту, де потрібно створити об'єкт, вона викликає свій фабричний метод createResource(). Оскільки цей метод реалізований у Конкретному Творці (CssResourceFactory), він створює та повертає відповідний Конкретний Продукт (CssResource). Головний клас Творця отримує цей продукт і продовжує з ним працювати через загальний інтерфейс Resource.

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

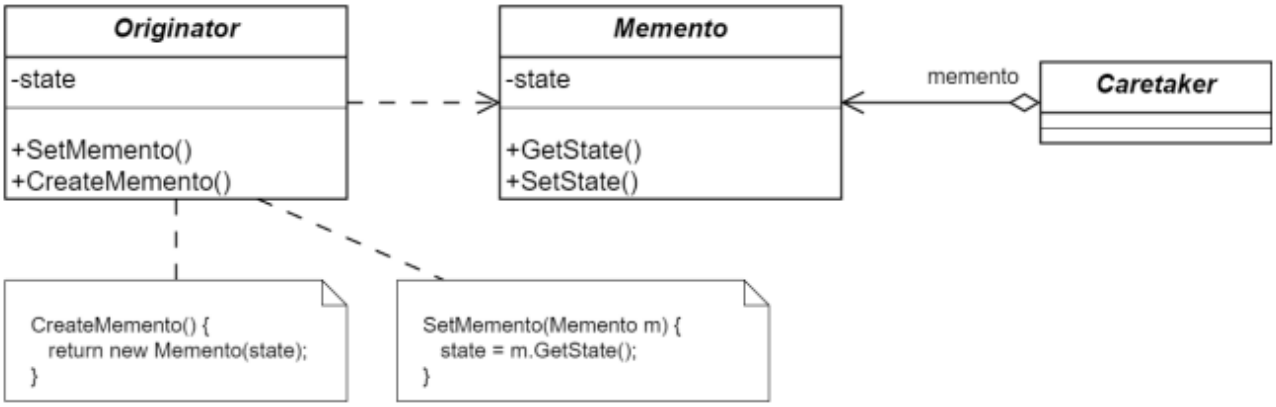
Відмінність між шаблонами «Абстрактна фабрика» та «Фабричний метод» є ключовою. Фабричний метод — це один метод, який делегує створення одного об'єкта підкласам (зазвичай через успадкування). Він вирішує проблему створення одного об'єкта. Абстрактна фабрика — це об'єкт, який має багато методів для створення групи або сімейства взаємопов'язаних об'єктів (наприклад, createButton() та createCheckbox() в одній фабриці). Вона вирішує проблему створення сумісних наборів об'єктів. Часто Абстрактна фабрика всередині себе може використовувати Фабричні методи для реалізації своїх методів.

8. Яке призначення шаблону «Знімок»?

Призначення шаблону «Знімок» (Memento) — це надати можливість зберегти та

згодом відновити внутрішній стан об'єкта, не порушуючи при цьому його інкапсуляцію. Тобто, ми хочемо отримати "фотографію" стану об'єкта, не даючи нікому "залізти" в його приватні поля. Це класичний патерн для реалізації функції "Undo" (Скасувати).

9. Нарисуйте структуру шаблону «Знімок».



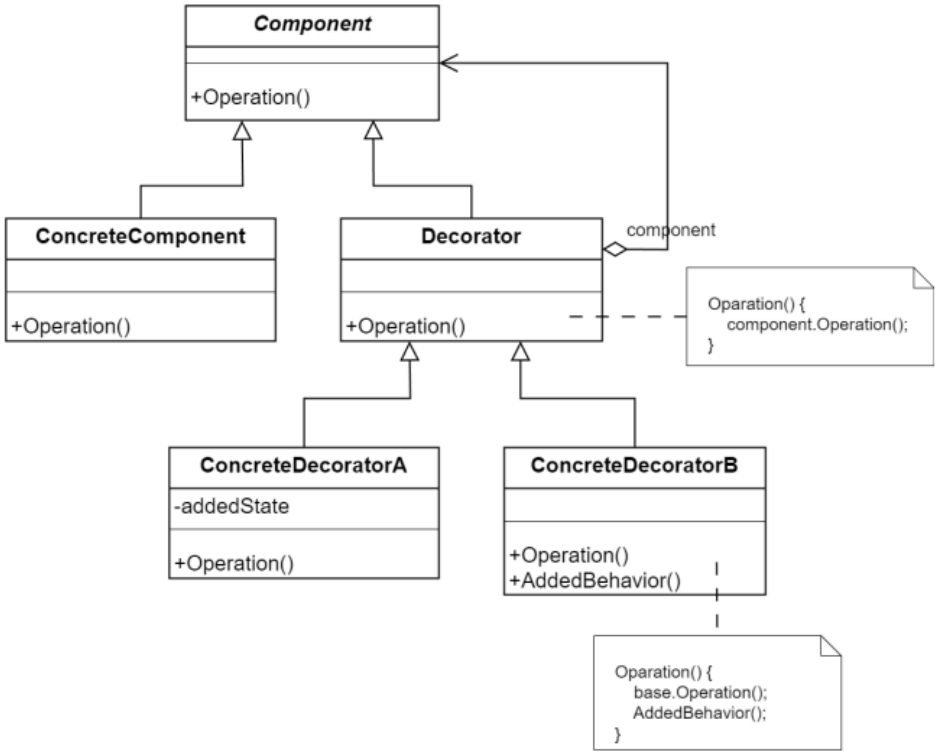
10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

До шаблону «Знімок» входять класи Творець (Originator), Знімок (Memento) та Опікун (Caretaker). Взаємодія відбувається так: Опікун (наприклад, HistoryManager) просить Творця (наш Editor) зберегти стан. Творець створює новий об'єкт Знімок і копіює туди свій поточний стан (наприклад, текст, позицію курсора). Творець повертає цей Знімок Опікуну. Опікун кладе цей Знімок у свій список, не заглядаючи всередину. Коли користувач натискає "Скасувати", Опікун дістає останній Знімок зі списку і повертає його Творцю через спеціальний метод restore(memento). Творець бере Знімок, витягує з нього дані (оскільки він має до них доступ) і відновлює свій стан.

11. Яке призначення шаблону «Декоратор»?

Призначення шаблону «Декоратор» (Decorator) — це надати можливість динамічно додавати нові обов'язки або поведінку об'єкту, не змінюючи його вихідний код. Це гнучка альтернатива успадкуванню. Замість того, щоб створювати підкласи EncryptedFileStream, CompressedFileStream, ви берете базовий FileStream і "загортаєте" його в об'єкт-декоратор EncryptionDecorator, а потім результат загортаєте в CompressionDecorator.

12. Нарисуйте структуру шаблону «Декоратор».



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

До шаблону «Декоратор» входять Інтерфейс Компонента, Конкретний Компонент (базовий об'єкт), Абстрактний Декоратор та Конкретні Декоратори. Взаємодія відбувається по ланцюжку. Клієнт бере Конкретний Компонент (наприклад, FileStream) і загортає його в Конкретний Декоратор (наприклад, new EncryptionDecorator(fileStream)). Потім він може загорнути цей декоратор в інший (new CompressionDecorator(encryptedStream)). Коли Клієнт викликає метод (наприклад, write()) у зовнішнього декоратора (CompressionDecorator), той спочатку виконує свою роботу (стиснення), а потім викликає метод write() у внутрішнього об'єкта, який він загортає (EncryptionDecorator), і так далі, аж до базового FileStream.

14. Які є обмеження використання шаблону «декоратор»?

Існують певні обмеження використання шаблону «Декоратор». По-перше, він може призвести до створення великої кількості дрібних об'єктів та класів, що може "засмітити" проєкт. По-друге, процес створення об'єкта може стати досить складним — іноді важко розібратися, в якому порядку потрібно "загортати" декоратори, і клієнтський код, що відповідає за цю збірку, може стати громіздким. Також, оскільки декоратори просто "прокидають" виклик, може бути складно ідентифікувати конкретний загорнутий об'єкт, що іноді створює проблеми при порівнянні об'єктів (equals()).

Висновки

Під час виконання даної лабораторної роботи я ознайомився та вивчив такі патерни програмування: abstact factory, factory method, memento, observer, decorator. На власному застосунку я реалізував патерн Factory Method для створення фабрик, які створюють окремі ресурси, визначені в цій фабриці.