Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота №6

Технології розроблення програмного забезпечення

« Патерни проектування.»

«System activity monitor»

Виконав
студент групи ІА–32:
Феклістов Д.С.

Київ 2025

Зміст

1. Варіант та Мета роботи

Варіант №10: VCS all-in-one (Універсальний клієнт систем контролю версій: **Git, SVN, Mercurial**).

Мета роботи: Ознайомитися, вивчити структуру та реалізувати **породжувальний патерн Фабричний Метод (Factory Method)**. Застосувати його для створення об'єктів різних клієнтів систем контролю версій (GitClient, SvnClient, MercurialClient), щоб забезпечити розширюваність системи.

2. Теоретичні відомості

Патерн Фабричний Метод (Factory Method)

Призначення: Фабричний Метод — це породжувальний патерн, який надає інтерфейс для створення об'єктів, але дозволяє підкласам (конкретним фабрикам) визначати, **який саме клас інстанціювати**. Це перекладає відповідальність за створення об'єкта з класу-клієнта (який використовує продукт) на його підкласи-творці.

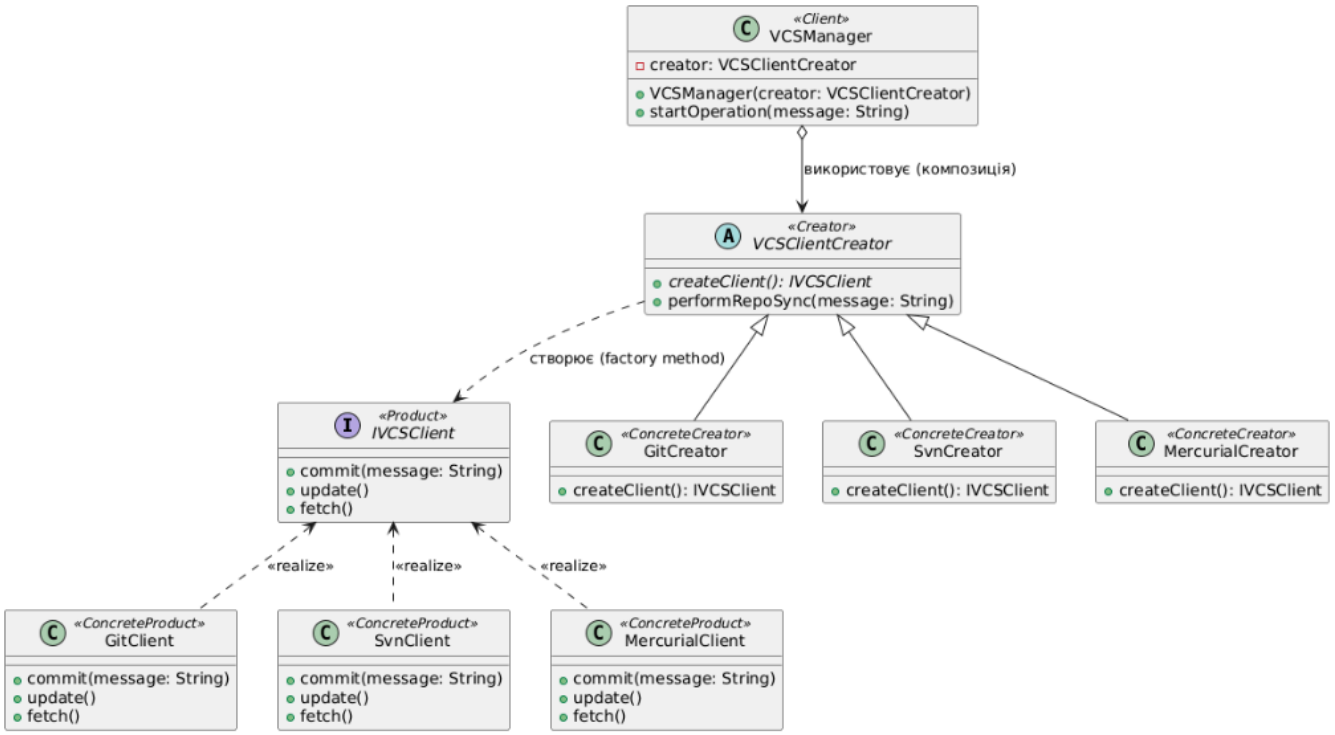
Перевага: Реалізує принцип **відкритості/закритості (OCP)**: код клієнта (наш VCSManager) стає незалежним від конкретних класів продуктів, що дозволяє легко додавати нові типи VCS-клієнтів без зміни основного менеджера.

Учасники патерну у проєкті:

Учасник	Клас у проєкті	Роль
Product	IVCSClient	Інтерфейс для всіх клієнтів VCS.
Concrete Product	GitClient, SvnClient, MercurialClient	Конкретні реалізації команд commit/update/fetch.
Creator	VCSClientCreator	Абстрактний клас, що оголошує метод createClient().
Concrete Creator	GitCreator, SvnCreator, MercurialCreator	Класи, що перевизначають createClient(), повертаючи конкретний клієнт.

3. Діаграма Класів (UML)

Діаграма демонструє, як **VCSClientCreator** оголошує фабричний метод createClient(), а його підкласи (GitCreator, SvnCreator, MercurialCreator) відповідають за створення конкретних клієнтів (GitClient, SvnClient, MercurialClient), які реалізують інтерфейс IVCSClient. Клієнт (VCSManager) взаємодіє тільки з абстракціями.



4. Хід роботи

1. **Створення контракту (Product):** Визначено інтерфейс IVCSClient з уніфікованими операціями (commit, update, fetch).
2. **Реалізація продуктів (Concrete Products):** Створено GitClient, SvnClient та MercurialClient для імплементації цих операцій із специфічними виводами.
3. **Створення абстрактної фабрики (Creator):** Створено абстрактний клас VCSClientCreator, що містить **абстрактний фабричний метод createClient()**. Додано загальний метод performRepoSync() для виконання послідовності команд.
4. **Реалізація конкретних фабрик (Concrete Creators):** Створено GitCreator, SvnCreator та MercurialCreator, кожен з яких реалізує createClient() для повернення відповідного VCS-клієнта.
5. **Тестування (Client):** Клас VCSManager використовує об'єкт абстрактного VCSClientCreator, що дозволяє змінювати тип VCS простою зміною об'єкта Creator у точці входу (Main.java).

Результат виконання програми

Наведений нижче консольний вивід підтверджує, що для кожного об'єкта VCSManager створюється коректний тип клієнта:

```
=== VCS All-in-One Application ===

[Manager]: Отримано запит на синхронізацію репозиторію.
-> [Factory]: Створення клієнта Git.

--- Запуск операції ---
[Git]: Executing 'git pull' (Update).
[Git]: Staging changes & committing: "Added feature X."
[Git]: Executing 'git fetch' (Fetch metadata).
--- Операція завершена. ---

==================================

[Manager]: Отримано запит на синхронізацію репозиторію.
-> [Factory]: Створення клієнта SVN.

--- Запуск операції ---
[SVN]: Executing 'svn update' (Update).
[SVN]: Committing changes to repository trunk. Log: "Fixed bug in module Y."
[SVN]: Fetch command is not applicable. Data fetched during update.
--- Операція завершена. ---

==================================
```

```
[Manager]: Отримано запит на синхронізацію репозиторію.
-> [Factory]: Створення клієнта Mercurial.

--- Запуск операції ---
[Hg]: Executing 'hg pull -u' (Update & Merge).
[Hg]: Recording changes with 'hg commit -m': "Experimental change for speed optimization."
[Hg]: Executing 'hg pull' (Fetch/Pull changes).
--- Операція завершена. ---
```

5. Код Програми (Java)

Наведено ключові файли, які демонструють реалізацію патерну.

vcs/clients/IVCSClient.java (Product)

```
package vcs.clients;


public interface IVCSClient {
    void commit(String message);
    void update();
    void fetch();
}
```

vcs/clients/GitClient.java (Concrete Product)

```
package vcs.clients;


public class GitClient implements IVCSClient {
    @Override
    public void commit(String message) {
        System.out.println("[Git]: Staging changes & committing: \"" +
message + "\"");
    }


    @Override
    public void update() {
        System.out.println("[Git]: Executing 'git pull' (Update).");
    }


    @Override
    public void fetch() {
        System.out.println("[Git]: Executing 'git fetch' (Fetch
metadata).");
    }
}
```

vcs/creators/VCSClientCreator.java (Creator)

```java
package vcs.creators;

import vcs.clients.IVCSClient;

public abstract class VCSClientCreator {

    // Фабричний метод, який має бути реалізований підкласами
    public abstract IVCSClient createClient();

    // Загальна бізнес-логіка, яка використовує створюваний продукт
    public void performRepoSync(String message) {
        // ВИКЛИК ФАБРИЧНОГО МЕТОДУ: створення продукту делегується підкласу
        IVCSClient client = createClient();

        System.out.println("\n--- Запуск операції ---");

        client.update();
        client.commit(message);
        client.fetch();

        System.out.println("--- Операція завершена. ---");
    }
}
```

vcs/creators/GitCreator.java (Concrete Creator)

```java
package vcs.creators;

import vcs.clients.IVCSClient;
import vcs.clients.GitClient;

public class GitCreator extends VCSClientCreator {
    @Override
    public IVCSClient createClient() {
        System.out.println("-> [Factory]: Створення клієнта Git.");
        return new GitClient();
    }
}
```

vcs/manager/VCSManager.java (Client)

```
package vcs.manager;


import vcs.creators.VCSClientCreator;


public class VCSManager {


   private final VCSClientCreator creator; // Залежність від абстрактного
творця


   public VCSManager(VCSClientCreator creator) {
       this.creator = creator;
   }


   // Клієнт використовує загальний метод, не знаючи про конкретний тип
VCS
   public void startOperation(String commitMessage) {
       System.out.println("\n[Manager]: Отримано запит на синхронізацію
репозиторію.");
       creator.performRepoSync(commitMessage);
   }
}
```

6. Висновок

В ході виконання лабораторної роботи №6 було успішно реалізовано **породжувальний патерн Фабричний Метод (Factory Method)**. Архітектура універсального клієнта VCS (**VCS all-in-one**) була модифікована таким чином, що:

1. **Створення** об'єктів VCS-клієнтів було винесено в окрему ієрархію класів-творців (Creator).
2. Клас-клієнт (VCSManager) став повністю **незалежним** від конкретних типів VCS (GitClient, SvnClient тощо), працюючи лише з абстракціями (IVCSClient та VCSClientCreator).
3. Це забезпечило відповідність **Принципу відкритості/закритості (OCP)**, дозволяючи легко додавати підтримку нових систем контролю версій (наприклад, Perforce або TFS) шляхом створення нового Concrete Creator без модифікації основного коду менеджера.

7. Відповіді на питання

1. Яке призначення шаблону «Фабричний Метод»?
   Надає інтерфейс для створення об'єктів у базовому класі, але дозволяє підкласам змінювати клас створюваного об'єкта, інкапсулюючи логіку інстанціювання.
2. Яка різниця між Фабричним Методом та Абстрактною Фабрикою?
   Фабричний Метод (Factory Method) використовує спадкування для створення одного продукту. Абстрактна Фабрика (Abstract Factory) використовує композицію для створення сімейства пов'язаних продуктів.
3. Яке призначення шаблону «Адаптер» (Adapter)?
   Дозволяє об'єктам з несумісними інтерфейсами працювати разом, створюючи посередника, який перетворює один інтерфейс на очікуваний іншим класом.
4. Яке призначення шаблону «Будівельник» (Builder)?
   Відокремлює складний процес конструювання об'єкта від його представлення, дозволяючи створювати об'єкти покроково та отримувати різні конфігурації з одного й того ж коду конструювання.
5. Яке призначення шаблону «Одинак» (Singleton)?
   Гарантує, що для класу існує лише один екземпляр, і надає глобальну точку доступу до нього.
6. Яке призначення шаблону «Прототип» (Prototype)?
   Створення нових об'єктів шляхом клонування (копіювання) існуючого об'єкта, що ефективніше за створення "з нуля" для складних об'єктів.
7. Яке призначення шаблону «Проксі» (Proxy)?
   Надати об'єкт-замісник (посередника) для іншого об'єкта, щоб контролювати доступ до нього або додавати допоміжну логіку (кешування, логування).
8. Яке призначення шаблону «Фасад» (Facade)?
   Надати єдиний спрощений інтерфейс до складної, заплутаної підсистеми класів.
9. Яке призначення шаблону «Декоратор» (Decorator)?
   Динамічно додавати нові обов'язки об'єкту, обгортаючи його в класи-декоратори, що є гнучкою альтернативою спадкуванню.
10. Яке призначення шаблону «Міст» (Bridge)?
    Відокремити абстракцію від її реалізації, щоб вони могли змінюватися незалежно одна від одної.