МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І. І. МЕЧНИКОВА
ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ ТА ІНФОРМАЦІЙНИЇ ТЕХНОЛОГІЙ
КАФЕДРА МАТЕМАТИЧНОГО ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНИХ СИСТЕМ

Лабораторна робота №5

з дисципліни: «Інженерія програмного забезпечення»

за темою: «Патерн проектування Abstract Factory»

Виконав:
студент 3 курсу
Владислав Краковський

Перевірив:
викладач
Пенко В.Г.

Одеса 2025

Зміст

1 ЗАВДАННЯ РОБОТИ

Мета роботи: Ознайомитися з патерном проєктування “Абстрактна фабрика” (Abstract Factory), навчитися реалізовувати його на мові програмування C++, зрозуміти його призначення, переваги та недоліки, а також розібратися, де він застосовується на практиці.

Умова роботи:

Реалістичний приклад

Реалістичний приклад коду демонструє створення за допомогою різних фабрик кількох світів тварин для комп'ютерної гри. Хоча тварини, створювані фабриками Continent різні, взаємодія між тваринами залишається незмінною.

Рисунок 1.1 - Умова

Система контролю версій Git та рефакторинг

Варіанти самостійних завдань

У наведеному вище викладі кожного патерну наводяться структурний та реальний приклад. Реалізувати структурний приклад та зафіксувати його в системі контролю версій. Далі здійснити серію фіксацій, яка трансформує структурний приклад у реальний. Деякі з проміжних фіксацій повинні полягати у застосуванні того чи іншого прийому рефакторингу.

Рисунок 1.2 - Завдання з GitHub

2 ХІД РОБОТИ

Опис патерну Abstract Factory:

Abstract Factory (Абстрактна фабрика) — це породжуючий патерн проєктування, який надає інтерфейс для створення цілих сімейств взаємопов’язаних або взаємозалежних об’єктів без необхідності вказувати їх конкретні класи.

Він дозволяє клієнту отримувати різні продукти (елементи однієї групи), не залежачи від конкретної реалізації.

Як реалізується патерн Abstract Factory?

1. Створюється абстрактний інтерфейс фабрики, який оголошує методи створення різних типів продуктів (наприклад, CreateButton(), CreateCheckbox()).
2. Створюються конкретні фабрики, які реалізують цей інтерфейс і повертають об’єкти певних конкретних класів.
3. Для кожної групи продуктів створюються базові інтерфейси і конкретні реалізації.
4. Клієнт працює тільки з інтерфейсами фабрики та продуктів, не знаючи конкретних класів.

У C++ це реалізується через створення ієрархії фабрик (abstract + concrete) та ієрархії продуктів (abstract + concrete для кожної групи).

Для чого використовується патерн Abstract Factory?

- Коли потрібно створювати кілька пов’язаних об’єктів (сімейство продуктів), які мають працювати разом.
- Коли необхідно гарантувати сумісність продуктів у рамках однієї сім’ї.
- Для ізоляції коду від конкретних класів продуктів.

Типові приклади:

- Кросплатформені графічні інтерфейси (наприклад, різне оформлення для Windows, macOS, Linux).
- Системи плагінів.
- Створення взаємозалежних налаштувань у складних конфігураціях.

Переваги патерну Abstract Factory

- Забезпечує узгодженість продуктів — продукти з однієї фабрики гарантовано сумісні.
- Послаблює зв’язок між клієнтом і конкретними класами — клієнт залежить лише від абстракцій.
- Легко додавати нові сімейства продуктів — достатньо створити нову фабрику.

Недоліки патерну Abstract Factory

− Ускладнює структуру коду — збільшує кількість класів (фабрики, продукти, інтерфейси).
− Важко додавати нові типи продуктів — необхідно змінювати всі фабрики (але легко додавати сімейства).
− Може бути “зайвим” для простих задач, якщо немає групи взаємозалежних об’єктів.

Запишемо розв’язок поставленої задачі з використання патерну Abstract Factory:

```
#include <iostream>
#include <memory>
using namespace std;

// === Абстрактные продукты ===
class Herbivore {
public:
    virtual void EatGrass() const = 0;
    virtual ~Herbivore() = default;
};

class Carnivore {
public:
    virtual void Eat(const Herbivore& herbivore) const = 0;
    virtual ~Carnivore() = default;
};

// === Конкретные травоядные ===
class Wildebeest : public Herbivore {
public:
    void EatGrass() const override {
        cout << "Wildebeest grazes.\n";
    }
};

class Bison : public Herbivore {
public:
    void EatGrass() const override {
        cout << "Bison grazes.\n";
    }
```

```
};

// === Конкретные хищники ===
class Lion : public Carnivore {
public:
    void Eat(const Herbivore& herbivore) const override {
        cout << "Lion eats Wildebeest.\n";
    }
};

class Wolf : public Carnivore {
public:
    void Eat(const Herbivore& herbivore) const override {
        cout << "Wolf eats Bison.\n";
    }
};

// === Абстрактная фабрика континента ===
class ContinentFactory {
public:
    virtual shared_ptr<Herbivore> CreateHerbivore() const = 0;
    virtual shared_ptr<Carnivore> CreateCarnivore() const = 0;
    virtual ~ContinentFactory() = default;
};

// === Конкретные фабрики континентов ===
class AfricaFactory : public ContinentFactory {
public:
    shared_ptr<Herbivore> CreateHerbivore() const override {
        return make_shared<Wildebeest>();
    }
```

```
    shared_ptr<Carnivore> CreateCarnivore() const override {
        return make_shared<Lion>();
    }
};

class AmericaFactory : public ContinentFactory {
public:
    shared_ptr<Herbivore> CreateHerbivore() const override {
        return make_shared<Bison>();
    }

    shared_ptr<Carnivore> CreateCarnivore() const override {
        return make_shared<Wolf>();
    }
};

// === Мир животных ===
class AnimalWorld {
private:
    shared_ptr<Herbivore> herbivore;
    shared_ptr<Carnivore> carnivore;

public:
    AnimalWorld(shared_ptr<ContinentFactory> factory) {
        herbivore = factory->CreateHerbivore();
        carnivore = factory->CreateCarnivore();
    }

    void RunFoodChain() const {
        herbivore->EatGrass();
        carnivore->Eat(*herbivore);
    }
```

```
};

// === main() ===
int main() {
    shared_ptr<ContinentFactory> africa =
make_shared<AfricaFactory>();
    AnimalWorld world1(africa);
    world1.RunFoodChain();

    shared_ptr<ContinentFactory> america =
make_shared<AmericaFactory>();
    AnimalWorld world2(america);
    world2.RunFoodChain();

    return 0;
}
```

Подивимось скріншот з результатом на ілюстрації 2.1:

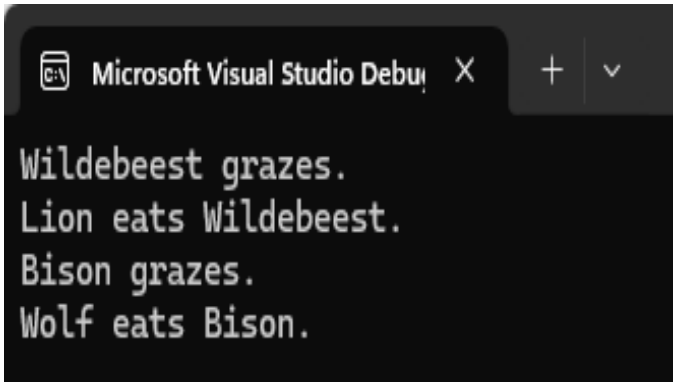

Рисунок 2.1 — Результат роботи програми

Програма була завантажена на GitHub згідно з завданням, ілюстрація 2.2:

Рисунок 2.2 - GitHub

Висновок

Патерн “Abstract Factory” — це потужний інструмент для створення груп пов’язаних об’єктів із чіткою структурою залежностей. Його застосування забезпечує масштабованість, гнучкість і узгодженість у розробці складних систем. Разом з тим, використання Abstract Factory призводить до збільшення кількості класів і підвищує складність архітектури, тому його доцільно застосовувати лише у випадках, коли потрібне створення цілих сімейств сумісних продуктів.