Міністерство освіти і науки України
Хмельницький національний університет
Кафедра комп’ютерної інженерії та інформаційних систем

ЛАБОРАТОРНА РОБОТА № 5-6

з дисципліни: Об’єктно-орієнтовані технології програмування

Виконав студент: Роюк Р.В.
Група: КІ2м-24-2
Перевірив: Лисенко С.М.

2025

Завдання: написати програмне забезпечення, що описує застосування патерну «Ітератор» та патерну «Інтерпретатор».

Отже, маємо програму, яка реалізує патерни "Ітератор" та "Інтерпретатор" для логістичної компанії, та задамо мінімальний сценарій роботи: у нас є список вантажів, який ми будемо обходити через ітератор; користувач зможе написати команду доставки у вигляді тексту, а проограма зможе це інтерпретувати.

```
#include <iostream>
#include <vector>
#include <string>
#include <memory>
#include <sstream>

//
// ===== Патерн Ітератор =====
//

// Клас вантажу
class Cargo {
public:
    std::string name;
    Cargo(const std::string& name) : name(name) {}
};

// Колекція вантажів
class CargoCollection {
private:
    std::vector<Cargo> cargos;
public:
    void addCargo(const Cargo& cargo) {
        cargos.push_back(cargo);
    }

    std::vector<Cargo>::iterator begin() { return cargos.begin(); }
    std::vector<Cargo>::iterator end() { return cargos.end(); }
};

//
// ===== Патерн Інтерпретатор =====
//

// Абстрактний клас виразу
class Expression {
public:
    virtual std::string interpret() const = 0;
    virtual ~Expression() = default;
};
```

```cpp
// Конкретні вирази
class CargoExpression : public Expression {
    std::string cargoName;
public:
    explicit CargoExpression(const std::string& name) : cargoName(name) {}
    std::string interpret() const override {
        return "Вантаж: " + cargoName;
    }
};

class TransportExpression : public Expression {
    std::string method;
public:
    explicit TransportExpression(const std::string& method) : method(method) {}
    std::string interpret() const override {
        return "Метод доставки: " + method;
    }
};

class DeliveryExpression : public Expression {
    std::shared_ptr<Expression> cargo;
    std::shared_ptr<Expression> transport;
public:
    DeliveryExpression(std::shared_ptr<Expression> cargo,
std::shared_ptr<Expression> transport)
        : cargo(std::move(cargo)), transport(std::move(transport)) {}

    std::string interpret() const override {
        return cargo->interpret() + " | " + transport->interpret();
    }
};

// Інтерпретатор команд
std::shared_ptr<Expression> parseCommand(const std::string& input) {
    std::stringstream ss(input);
    std::string command, item, by, method;

    ss >> command >> item >> by >> method;

    if (command == "DELIVER" && by == "BY") {
        auto cargoExpr = std::make_shared<CargoExpression>(item);
        auto transportExpr = std::make_shared<TransportExpression>(method);
        return std::make_shared<DeliveryExpression>(cargoExpr, transportExpr);
    }

    return nullptr;
}

//
// ===== Демонстрація =====
```

```
//

int main() {
    std::cout << "=== Iterator Pattern ===" << std::endl;
    CargoCollection collection;
    collection.addCargo(Cargo("Телевізор"));
    collection.addCargo(Cargo("Стілець"));
    collection.addCargo(Cargo("Холодильник"));

    for (auto it = collection.begin(); it != collection.end(); ++it) {
        std::cout << "Обробка вантажу: " << it->name << std::endl;
    }

    std::cout << "\n=== Interpreter Pattern ===" << std::endl;
    std::string command = "DELIVER Ліки BY Літак";
    auto expression = parseCommand(command);

    if (expression) {
        std::cout << "Команда: " << expression->interpret() << std::endl;
    } else {
        std::cout << "Невідома команда." << std::endl;
    }

    return 0;
}
```

Патерн **«Ітератор»** дозволяє послідовно обходити елементи складного об'єкта (наприклад, список вантажів чи маршрутів), не розкриваючи його внутрішню реалізацію. У коді це реалізовано як:

- CargoCollection– вміщує список вантажів;
- begin()/end() – повертають стандартні ітератори STL для обходу елементів;
- for-цикл – використовує ітератор для поелементного обходу;

```
•   for (auto it = collection.begin(); it != collection.end(); ++it) {
•       std::cout << "Обробка вантажу: " << it->name << std::endl;
•   }
•
```

Цей цикл по черзі обробляє кожен об'єкт типу Cargo, використовуючи стандартні ітератори STL, які надає вектор std::vector<Cargo>.

Перевагами патерну «Ітератор» є:

- Можливість обходу структури без зміни коду;
- Приховує внутрішню реалізацію контейнера;
- Можливість реалізації власних складних ітераторів, якщо потрібно (наприклад, обхід у зворотному порядку);

Патерн **«Інтерпретатор»** визначає мову, її граматику, і забезпечує механізм для інтерпретації виразів цієї мови. Застосовується для обробки простих формальних мов або команд (наприклад: "DELIVER apple BY TRUCK").

У коді це реалізовано як:

- Expression – абстрактний інтерфейс з методом interpret();
- CargoExpression – інтерпретує частину виразу, яка стосується вантажу;
- TransportExpression – інтерпретує транспортний метод;
- DeliveryExpression – об’єднує обидва вирази в одне повідомлення;
- parseCommand – парсить текстову команду та створює дерево виразів;

Команда у вигляді тексту:

```
•  "DELIVER Ліки BY Літак"
•
```

буде мати такий вигляд:

```
"Вантаж: Ліки | Метод доставки: Літак"
```

У коді це працює як:

```
auto expression = parseCommand("DELIVER Ліки BY Літак");
std::cout << expression->interpret();
```

parseCommand() робить наступне:

- парсить команду по частинах;

- створює об'єкти:

  - CargoExpression("Ліки")
  - TransportExpression("Літак")

- об'єднує їх у DeliveryExpression, яка викликає interpret() для обох.

Перевагами патерну «Інтерпретатор» є:

- Дає змогу створити власну мову команд (DSL);
- Зручно для автоматизації рутинних процесів (наприклад: "DELIVER Хліб BY Дрон");
- Можна легко додавати нові правила (наприклад, час доставки, напрямок тощо).

Висновок: У ході виконання лабораторної роботи було реалізовано два шаблони проєктування — ітератор та інтерпретатор — у контексті логістичної системи. Патерн «Ітератор» забезпечив зручний і універсальний спосіб доступу до елементів колекції вантажів, що дало змогу організувати обхід без прив’язки до конкретної структури зберігання. Це сприяє підвищенню гнучкості та підтримуваності коду. Патерн «Інтерпретатор» дозволив реалізувати базову мову команд доставки, яку система може розпізнавати та виконувати. Такий підхід може бути основою для розробки власної командної мови або автоматизованої обробки запитів у майбутніх версіях системи.