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 №4

з дисципліни "Технології розроблення програмного забезпечення"

Виконав

студент групи ІА–33:

Китченко Д.В.

Перевірив:

Мягкий М.Ю.

Київ 2025

Зміст

Вступ

Метою даної лабораторної роботи є вивчення та практичне застосування патернів проектування у розробці програмного забезпечення, зокрема реалізація поведінкового патерну Iterator. Патерни проектування є перевіреними рішеннями типових проблем, які виникають під час розробки програмного забезпечення. Використання патернів дозволяє створювати гнучкі, масштабовані та легко підтримувані системи. Завданням роботи є реалізація патерну Iterator для системи управління архівами. Iterator дозволяє послідовно обходити елементи колекції без розкриття її внутрішньої структури. У рамках роботи було розроблено:

- інтерфейс ArchiveIterator;
- два конкретні ітератори: DepthFirstIterator та BreadthFirstIterator;
- інтеграція патерну в існуючу систему управління архівами.

Теоретичні відомості

Поняття шаблону проєктування

Будь-який патерн проєктування, використовуваний при розробці інформаційних систем, являє собою формалізований опис, який часто зустрічається в завданнях проєктування, вдале рішення даної задачі, а також рекомендації по застосуванню цього рішення в різних ситуаціях. Правильно сформульований патерн проєктування дозволяє, відшукавши одного разу вдале рішення, користуватися ним знову і знову.

Застосування патернів проєктування підвищує стійкість системи до зміни вимог та спрощує неминуче подальше доопрацювання системи. Сукупність патернів проєктування, по суті, являє собою єдиний словник проєктування, який, будучи уніфікованим засобом, незамінний для спілкування розробників один одним.

Шаблон "Ітератор"(Iterator)

Призначення: "Iterator"(Ітератор) являє собою шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх

механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків: колекція відповідає за зберігання даних, ітератор – за прохід по колекції.

Проблема: Колекції можуть мати складну внутрішню структуру (дерева, графи). Клієнтський код повинен мати можливість послідовно обходити елементи колекції, не заглиблюючись в деталі її реалізації. Крім того, може виникнути потреба в різних алгоритмах обходу (наприклад, в глибину або в ширину), і додавання цих алгоритмів безпосередньо в клас колекції "розмиває" її основну відповідальність – зберігання даних.

Рішення: Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий клас. Об'єкт-ітератор відстежує стан обходу, поточну позицію і кількість елементів, що залишилися. Це дозволяє створювати різні класи ітераторів для однієї колекції, не змінюючи її вихідний код.

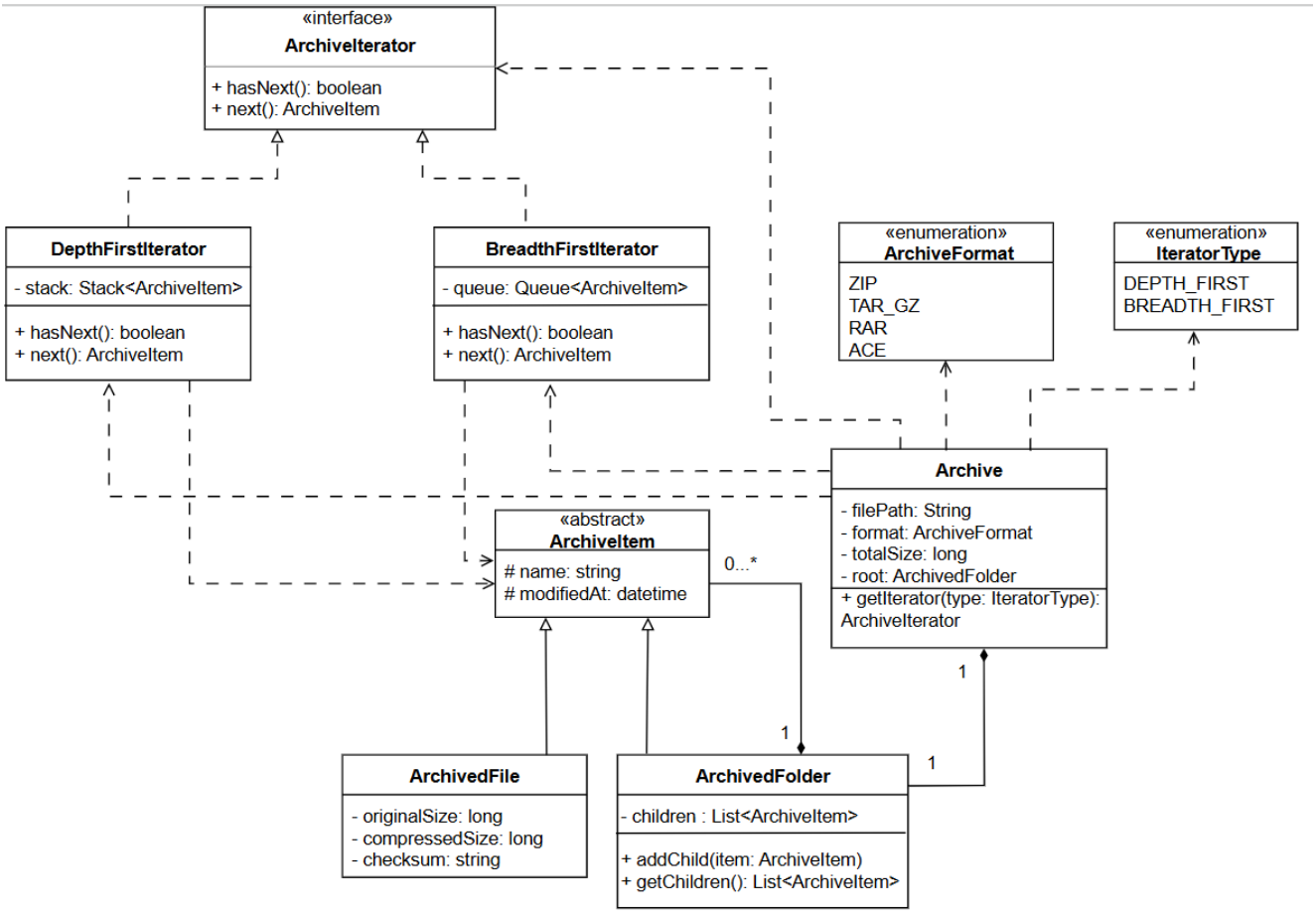
Хід роботи

В програмній системі "Архіватор" клас Archive представляє собою контейнер, що містить ієрархічну структуру файлів та папок (ArchiveItem, ArchivedFolder). Виникла задача надати клієнтському коду (класу SimpleMainWindow) простий та уніфікований спосіб обходу всіх елементів архіву, незалежно від рівня їх вкладеності.

Пряма реалізація рекурсивного обходу в клієнтському коді (SimpleMainWindow) призвела б до змішування обов'язків (логіка відображення та логіка обходу даних) і зробила б систему негнучкою до зміни алгоритму обходу.

Для вирішення цієї проблеми було обрано патерн "Ітератор", який ідеально підходить для відокремлення алгоритму обходу від структури даних.

На основі обраного патерну було спроектовано наступну структуру класів, яка була інтегрована в існуючу архітектуру системи.



Діаграма демонструє:

- інтерфейс ArchiveIterator з методами hasNext() та next();
- два конкретні ітератори: DepthFirstIterator (використовує Stack) та BreadthFirstIterator (використовує Queue);
- клас Archive, який створює ітератори;
- зв'язки між класами: реалізація інтерфейсу (пунктирна лінія з трикутником) та залежність (пунктирна лінія зі стрілкою).

Відповідно до завдання було реалізовано 3 нових типи: один інтерфейс та два класи, що його реалізують.

Інтерфейс ArchiveIterator

Створено загальний інтерфейс ArchiveIterator, який визначає єдиний контракт для всіх ітераторів. Він вимагає реалізації двох методів: hasNext() для перевірки наявності наступного елемента та next() для його отримання.

```
package iterator;

import model.ArchiveItem;

public interface ArchiveIterator {
    boolean hasNext();
    ArchiveItem next();
}
```

Метод hasNext() перевіряє, чи є наступний елемент для обходу.

Повертає true, якщо елементи ще залишились, та false інакше.

Метод next() повертає наступний елемент архіву типу ArchiveItem.

ArchiveItem є абстрактним класом, від якого успадковуються ArchivedFile та ArchivedFolder.

Клас DepthFirstIterator (Обхід в глибину)

Перша конкретна реалізація ітератора, що використовує структуру даних Stack (стек) для забезпечення обходу дерева файлів у глибину (Depth-First Search).

```
package iterator;

import model.ArchiveItem;
import model.ArchivedFolder;

import java.util.List;
```

```
import java.util.Stack;

public class DepthFirstIterator implements ArchiveIterator {
    private Stack<ArchiveItem> stack;

    public DepthFirstIterator(ArchiveItem root) {
        this.stack = new Stack<>();
        if( root !=null) {
            this.stack.push(root);
        }
    }
    @Override
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    @Override
    public ArchiveItem next() {
        ArchiveItem current = stack.pop();
        if (current instanceof ArchivedFolder) {
            ArchivedFolder folder = (ArchivedFolder) current;
            List<ArchiveItem> children = folder.getChildren();
            for(int i = children.size()-1; i>=0; i--){
                stack.push(children.get(i));
            }
        }
        return current;
    }
}
```

Принцип роботи:

1. У конструкторі кореневий елемент поміщається в стек.
2. Метод next() витягує елемент зі стеку (pop).
3. Якщо елемент – папка, всі її діти додаються в стек у зворотному порядку, щоб зберегти правильний порядок обходу.
4. Стек забезпечує обхід LIFO (Last In, First Out), що дає обхід в глибину.

Клас BreadthFirstIterator (Обхід в ширину)

Друга, альтернативна реалізація, що використовує Queue (чергу) для обходу в ширину (Breadth-First Search). Наявність двох реалізацій демонструє гнучкість патерну.

```
package iterator;
```

```
import model.ArchiveItem;
import model.ArchivedFolder;
import java.util.LinkedList;
import java.util.Queue;

public class BreadthFirstIterator implements ArchiveIterator {
    private Queue<ArchiveItem> queue;

    public BreadthFirstIterator(ArchivedFolder root) {
        this.queue = new LinkedList<>();
        if(root != null) this.queue.add(root);
    }
    @Override
    public boolean hasNext() {
        return !queue.isEmpty();
    }

    @Override
    public ArchiveItem next() {
        ArchiveItem current = queue.poll();
        if(current instanceof ArchivedFolder){
            ArchivedFolder folder = (ArchivedFolder) current;
            for(ArchiveItem child : folder.getChildren()){
                queue.add(child);
            }
        }
        return current;
    }
}
```

Відмінності від DepthFirstIterator:

1. Використовується Queue замість Stack.
2. Елементи додаються в кінець черги (add), а витягуються з початку (poll).
3. Queue забезпечує FIFO (First In, First Out), що дає обхід по рівнях дерева.
4. Не потрібен зворотний цикл – елементи додаються у природному порядку.

Інтеграція патерну в існуючу систему

Для інтеграції патерну було модифіковано клас-агрегат Archive, додавши в нього фабричні методи для створення ітераторів. Також було оновлено клієнтський клас SimpleMainWindow для використання нового механізму.

Archive.java

```
package model;
import iterator.ArchiveIterator;
import iterator.BreadthFirstIterator;
import iterator.DepthFirstIterator;
import iterator.IteratorType;
import java.time.LocalDateTime;

public class Archive {
    private String filePath;
    private ArchiveFormat format;
    private long totalSize;
    private ArchivedFolder rootFolder;

    public Archive(String filePath, ArchiveFormat format) {
        this.filePath = filePath;
        this.format = format;
        this.rootFolder = new ArchivedFolder("/", LocalDateTime.now());
    }
    public ArchiveIterator getIterator(IteratorType type){
```

```
        switch (type){
            case DEPTH_FIRST:
                return new DepthFirstIterator(rootFolder);
            case BREADTH_FIRST:
                return new BreadthFirstIterator(rootFolder);
            default:
                return new DepthFirstIterator(rootFolder);
        }
    }
    public ArchivedFolder getRootFolder() {
        return rootFolder;
    }
}
```

Метод getIterator() приймає параметр типу IteratorType і повертає відповідний екземпляр ітератора. Це дозволяє клієнтському коду не знати про конкретні класи ітераторів та працювати тільки через інтерфейс ArchiveIterator.

Використання в клієнтському коді (SimpleMainWindow)

Фрагмент коду з класу SimpleMainWindow.java

```
  private void openArchive() {
      int selectedRow = table.getSelectedRow();
      if (selectedRow >= 0) {
          String path = (String) tableModel.getValueAt(selectedRow, 1);
          try {
              Archive archive = facade.open(path);
              logArea.append("Відкрито архів: " + archive.getFilePath() + "\n");

              StringBuilder content = new StringBuilder("Вміст архіву:\n");
              DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm");
// Клієнтський код отримує ітератор і працює з ним через абстрактний інтерфейс
```

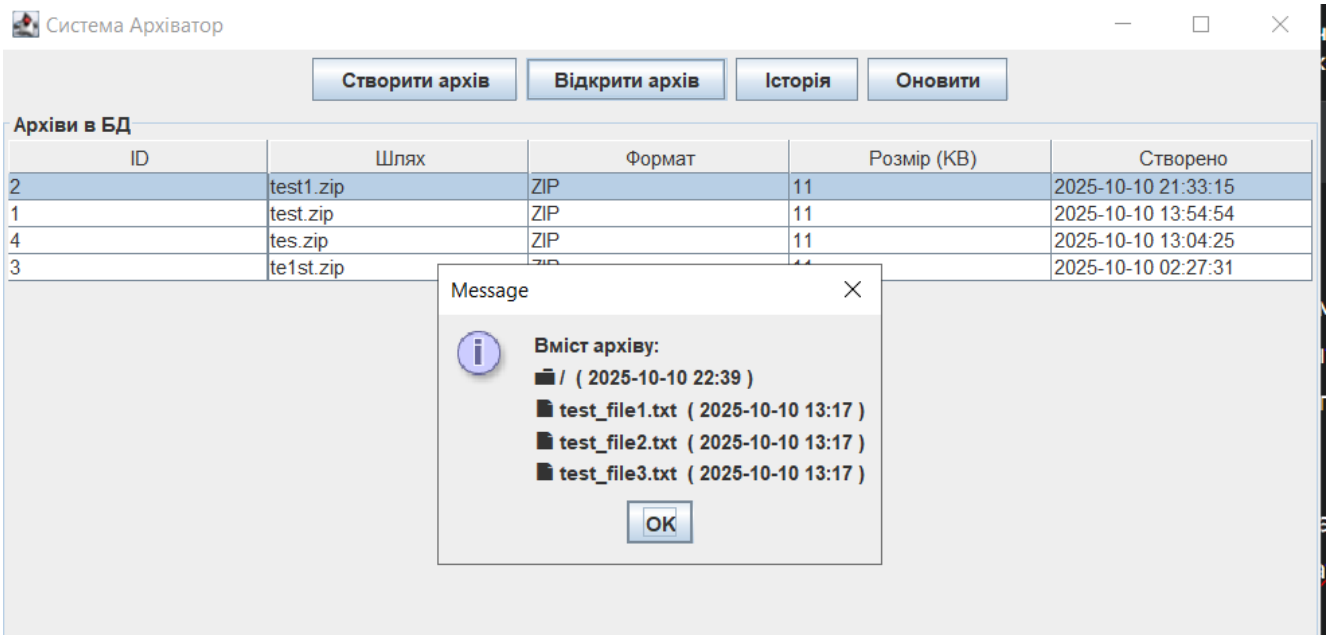
```
            ArchiveIterator iterator = archive.getIterator(IteratorType.BREADTH_FIRST);
            while (iterator.hasNext()){
                ArchiveItem item = iterator.next();
                String type = (item instanceof ArchivedFile) ? "\uD83D\uDCC4" :
"\uD83D\uDCC1";
                String timeString = item.getModificationDate().format(formatter);

                content.append(type).append(" ").append(item.getName()).append("  ( ")
                        .append(timeString)
                        .append(" )\n");;
            }
            JOptionPane.showMessageDialog(null, content.toString());
          } catch (Exception e) {
            logArea.append("Помилка відкриття: " + e.getMessage() + "\n");
          }
        } else {
          JOptionPane.showMessageDialog(null, "Виберіть архів для відкриття");
        }
    }
```

Клієнтський код не знає про Stack або Queue всередині ітераторів. Він працює тільки з інтерфейсом ArchiveIterator, що демонструє принцип інкапсуляції.

Результати роботи

Після реалізації та інтеграції патерну "Ітератор" функціонал програми було перевірено. При виборі архіву в головному вікні та натисканні кнопки "Відкрити архів" клієнтський код (SimpleMainWindow) успішно отримує ітератор від об'єкта Archive та використовує його для обходу всіх елементів. На наведеному нижче скріншоті показано результат роботи методу openArchive. Вікно повідомлення JOptionPane відображає повний список файлів та папок, отриманих за допомогою ітератора, разом з їхніми датами модифікації. Це демонструє, що патерн працює коректно, і клієнтський код отримує доступ до всієї ієрархії архіву.



Питання до лабораторної роботи

1. Що таке шаблон проєктування?

Це формалізований, перевірений часом опис вдалого рішення типової проблеми, що часто зустрічається при проєктуванні програмних систем.

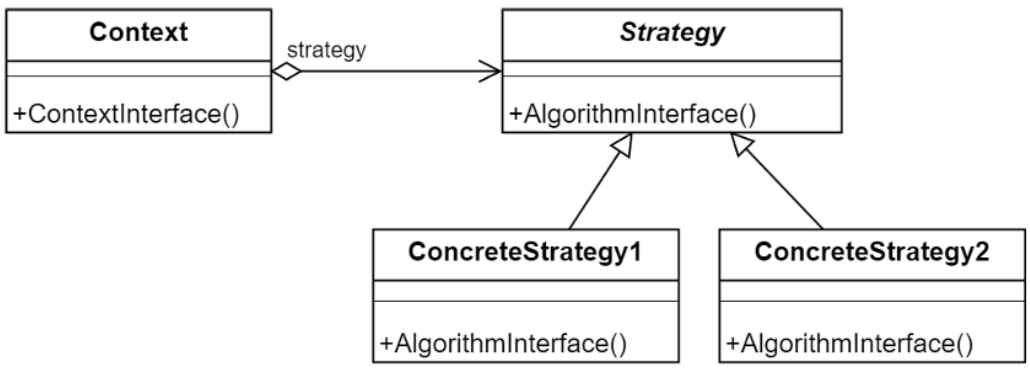
2. Навіщо використовувати шаблони проєктування?

Вони надають спільний словник для розробників, підвищують стійкість системи до змін, спрощують інтеграцію та дозволяють повторно використовувати перевірені архітектурні рішення.

3. Яке призначення шаблону «Стратегія»?

Дозволяє визначати сімейство алгоритмів, інкапсулювати кожен з них і робити їх взаємозамінними. Це дає змогу змінювати алгоритм незалежно від клієнтського коду, що його використовує.

4. Нарисуйте структуру шаблону «Стратегія».



5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

Strategy (Інтерфейс): Оголошує загальний інтерфейс для всіх алгоритмів.

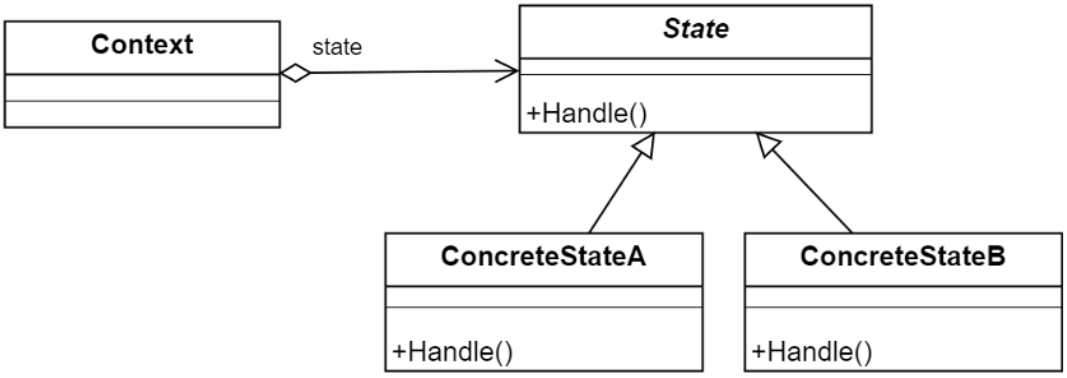
ConcreteStrategy (Класи): Реалізують конкретні алгоритми.

Context (Клас): Містить посилання на об'єкт-стратегію і взаємодіє з ним через загальний інтерфейс Strategy.

6. Яке призначення шаблону «Стан»?

Дозволяє об'єкту змінювати свою поведінку при зміні його внутрішнього стану. Зовні це виглядає так, ніби об'єкт змінив свій клас.

7. Нарисуйте структуру шаблону «Стан».



8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

State (Інтерфейс): Оголошує інтерфейс для поведінки, пов'язаної зі станом.

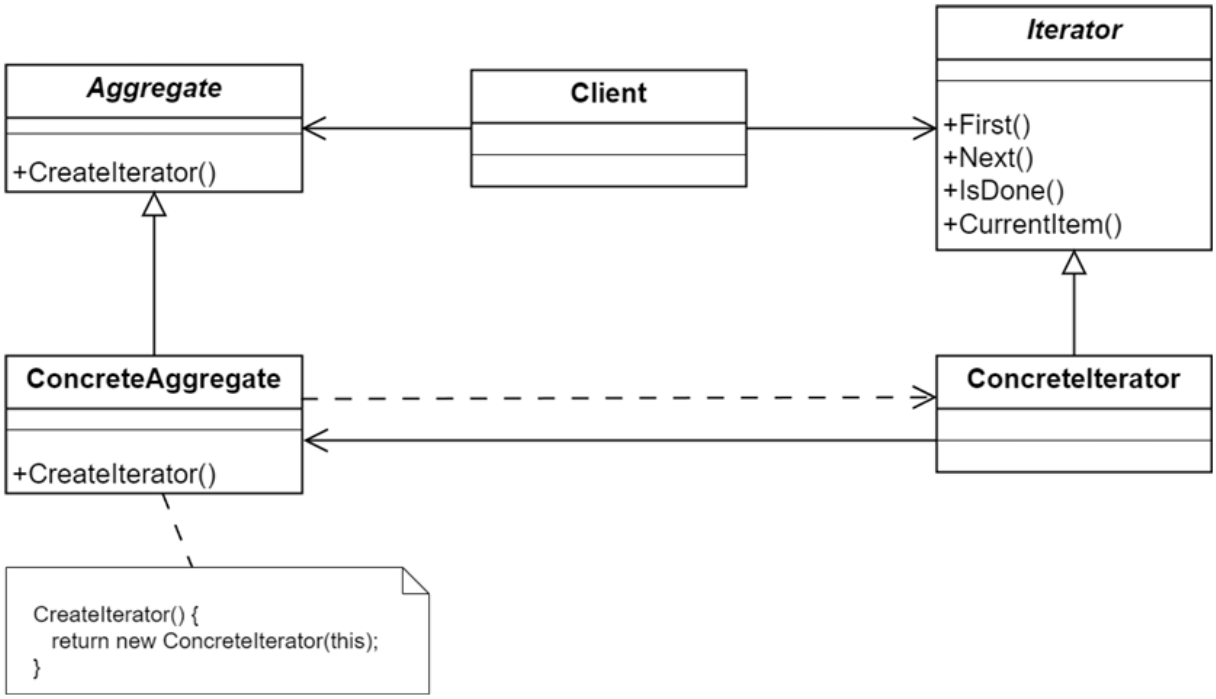
ConcreteState (Класи): Реалізують поведінку для конкретних станів.

Context (Клас): Зберігає посилання на поточний стан і делегує йому виконання роботи. Може змінювати свій стан.

9. Яке призначення шаблону «Ітератор»?

Надає уніфікований спосіб послідовного доступу до елементів колекції (агрегату), не розкриваючи її внутрішньої структури.

10. Нарисуйте структуру шаблону «Ітератор».



11. Які класи входять в шаблон «Ітератор», та яка між ними взаємодія?

Iterator (Інтерфейс): Визначає методи для обходу (hasNext, next).

ConcreteIterator (Клас): Реалізує алгоритм обходу і відстежує поточну позицію.

Aggregate (Інтерфейс): Визначає метод для створення ітератора.

ConcreteAggregate (Клас): Реалізує метод створення ітератора, повертаючи екземпляр ConcreteIterator.

12. В чому полягає ідея шаблона «Одинак»?

Гарантувати, що клас матиме лише один екземпляр (об'єкт), і надати глобальну точку доступу до цього екземпляра. Клас сам контролює створення екземпляра: приватний конструктор забороняє створення нових об'єктів ззовні, а статичний метод getInstance() повертає той самий єдиний екземпляр при кожному виклику.

13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

Singleton вважають анти-шаблоном через наступні проблеми:

1. Порушує Single Responsibility Principle – клас відповідає і за свою логіку, і за контроль кількості екземплярів.
2. Ускладнює тестування – складно підмінити Singleton mock-об'єктом для unit-тестів, неможливо створити окремі екземпляри для паралельних тестів.

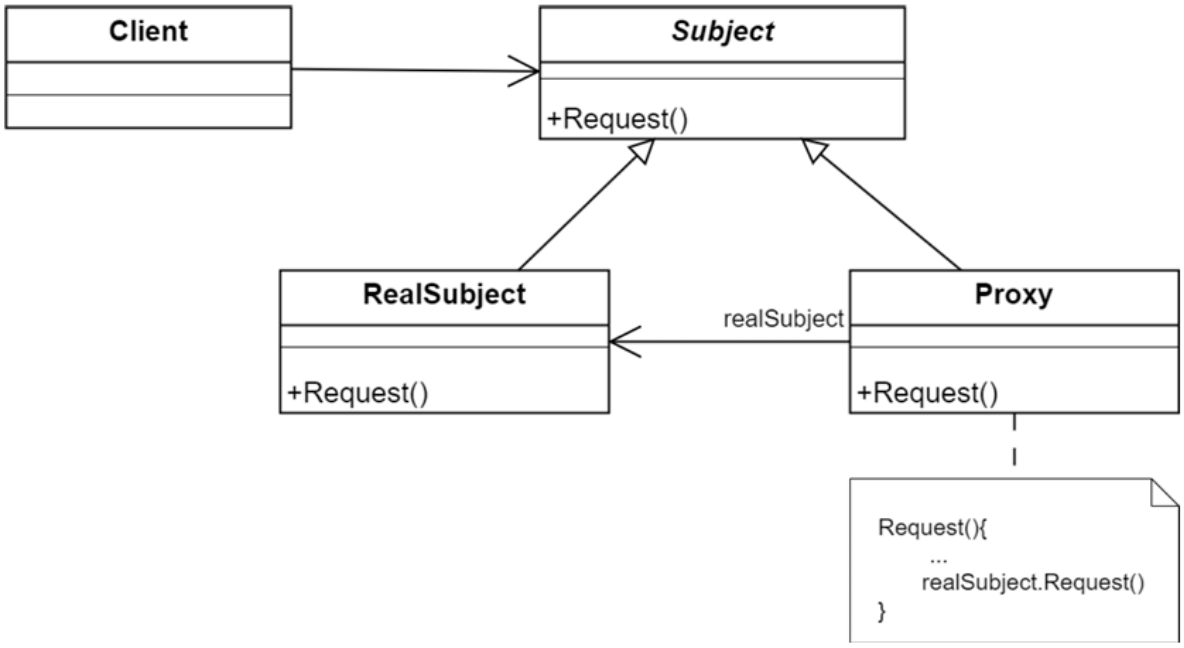
3. Прихований глобальний стан – створює неявні залежності між класами, що знижує модульність коду.
4. Проблеми з багатопоточністю – потребує додаткової синхронізації для коректної роботи в багатопоточному середовищі.
5. Порушує Dependency Injection – класи безпосередньо звертаються до Singleton, замість отримувати залежності ззовні.

Сучасні підходи рекомендують використовувати Dependency Injection контейнери замість Singleton.

14. Яке призначення шаблону «Проксі»?

Надає об'єкт-замінник (сурогат), який контролює доступ до іншого об'єкта. Проксі має той самий інтерфейс, що й реальний об'єкт, тому клієнт може працювати з ним прозорим чином. Проксі може виконувати додаткову логіку до або після делегування виклику реальному об'єкту.

15. Нарисуйте структуру шаблону «Проксі».



16. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

Subject (Інтерфейс) – визначає загальний інтерфейс для RealSubject та Proxy. Завдяки йому Proxy може бути використаний замість RealSubject.

RealSubject (Клас) – реальний об'єкт, який виконує основну бізнес-логіку. Містить ресурсомісткі або захищені операції.

Proxy (Клас) – містить посилання на об'єкт RealSubject і реалізує той самий інтерфейс Subject. Перехоплює виклики клієнта, може виконувати додаткову логіку (перевірка прав, логування, кешування), а потім делегує виклик реальному об'єкту.

Взаємодія:

1. Клієнт звертається до Proxy через інтерфейс Subject.
2. Proxy виконує додаткову логіку (наприклад, перевірка прав доступу).
3. Proxy делегує виклик об'єкту RealSubject.
4. RealSubject виконує реальну роботу і повертає результат.
5. Proxy може обробити результат (кешувати, логувати) і повернути його клієнту.

Клієнт не знає, працює він з Proxy чи з RealSubject – обидва мають однаковий інтерфейс.

Висновки

В ході виконання даної лабораторної роботи було досягнуто поставлену мету: вивчено теоретичні основи патернів проектування та отримано практичний досвід застосування шаблону "Ітератор". Було успішно реалізовано та інтегровано патерн в програмну систему "Архіватор", що дозволило вирішити задачу обходу ієрархічної структури даних. Реалізація патерну дозволила відокремити логіку обходу від структури даних та від клієнтського коду, що зробило архітектуру системи більш гнучкою, чистою та розширюваною.

Створення двох різних реалізацій ітератора (DepthFirstIterator та BreadthFirstIterator) наочно продемонструвало головну перевагу патерну – можливість змінювати алгоритми, не зачіпаючи клієнтський код.