Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

ПРАКТИЧНА РОБОТА № 4

З дисципліни «Інженерія програмного забезпечення»

на тему «ШАБЛОНИ ПОВЕДІНКИ. ITERATOR, MEDIATOR, OBSERVER, STRATEGY, CHAIN OF RESPONSIBILITY.»

Виконав:
студент групи ІО-42
Куліков М. М.
Залікова: 4214

Перевірив:
Ст. викладач кафедри ОТ
Васильєва М. Д.

Київ – 2025

Практична робота №4

Тема: «Шаблони поведінки. Iterator, Mediator, Observer, Strategy, Chain of responsibility.».

Мета: Ознайомлення з видами шаблонів проектування ПЗ. Вивчення шаблонів поведінки. Отримання базових навичок з застосування шаблонів Iterator, Mediator, Observer, Chain of Responsibility.

Виконання роботи:

Завдання 1

1. Повторити шаблони поведінки для проектування ПЗ. Знати загальну характеристику шаблонів поведінки та призначення кожного з них.
2. Детально вивчити шаблони поведінки для проектування ПЗ – Iterator, Mediator, Observer, Strategy, Chain of Responsibility. Для кожного з них:
    - вивчити Шаблон, його призначення, мотивацію, випадки коли його застосування є доцільним та результати такого застосування;
    - знати особливості реалізації Шаблону, споріднені шаблони, відомі випадки його застосування в програмних додатках;
    - вільно володіти структурою Шаблону, призначенням його класів та відносинами між ними;
    - вміти розпізнавати Шаблон в UML діаграмі класів та будувати сирцеві коди Java-класів, що реалізують шаблон.
3. В підготованому проєкті створити програмний пакет work4. В пакеті розробити інтерфейси і класи, що реалізують завдання 1 та 2 (згідно варіанту) з застосуванням одного чи декількох шаблонів (п.2). В класах, що розробляються, повністю реалізувати методи, пов'язані з функціюванням Шаблону. Методи, що реалізують бізнес-логіку, закрити заглушками з виводом на консоль інформації про викликаний метод та його аргументи.
4. За допомогою автоматизованих засобів виконати повне документування розроблених класів (також методів і полів), при цьому документація має в достатній мірі висвітлювати роль певного класу в загальній структурі Шаблону та особливості конкретної

реалізації.

Визначимо варіант для індивідуальних завдань: 4214%13 = 2, 4214%7 = 0.

Варіант 2 до завдання 1:

Необхідно розробити класи, які інкапсулюють лінійний список символьних рядків і забезпечують можливість різних способів обходу елементів, не розкриваючи користувачу внутрішню структуру списку.

Потрібно реалізувати:

- Звичайний послідовний обхід усіх рядків списку у порядку їх зберігання.
- Обхід із додатковою фільтрацією, тобто можливість проходити лише ті елементи, які відповідають певним умовам (наприклад, рядки певної довжини, рядки, що починаються з певної літери, містять певний символ тощо).
- Вимоги:
  - Користувач не повинен знати, як саме реалізовано список – доступ до елементів здійснюється лише через спеціальні об’єкти для обходу.
  - Повинна бути можливість легко змінювати або додавати нові критерії фільтрації.
  - У методі main продемонструвати:
    - створення списку рядків;
    - звичайний послідовний обхід усіх елементів;
    - обхід із фільтрацією за певною умовою (наприклад, лише рядки довжиною більше 3 або ті, що починаються з літери "A").

Код до завдання 1:

Вміст файлу main.java:

```
import work4.Iterator.*;
/**
 * Головний клас для демонстрації роботи власної реалізації патерну Ітератор.
 * Цей клас показує, як створити список рядків (`StringList`), обійти його
 * за допомогою стандартного ітератора, а також як використовувати відфільтровані
 * ітератори з різними умовами, що задаються за допомогою лямбда-виразів.
 *
 * @author Broniev
 * @version 1.0
 */
public class Main {
    public static void main(String[] args) {
        String divider = ">----------------------------------------";

        StringList list = new StringList();
```

```
        list.add("Hello");
        list.add("I");
        list.add("added some text");
        list.add("right here");
        list.add("$1");
        list.add("$A");

        System.out.println("Standart iterated list: ");
        Iterator it = list.createIterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
        System.out.println(divider);

        System.out.println("Filtered iterated list (elements with more then 3 characters): ");
        Iterator filtered1 = list.createFilteredIterator(s -> s.length() > 3);
        while (filtered1.hasNext()) {
            System.out.println(filtered1.next());
        }
        System.out.println(divider);

        System.out.println("Filtered iterated list (elements with letter a inside (include uppercase)): ");
        Iterator filtered2 = list.createFilteredIterator(s -> s.toLowerCase().contains("a"));
        while (filtered2.hasNext()) {
            System.out.println(filtered2.next());
        }
    }
}
```

Вміст файлу Iterator.java:

```
package work4.Iterator;

/**
 * Інтерфейс Ітератора (Iterator).
 * Визначає стандартні методи для обходу елементів колекції,
 * не розкриваючи її внутрішню структуру.
 */
public interface Iterator {
    /**
     * Перевіряє, чи є ще елементи для обходу.
     * @return {@code true}, якщо ітерація має наступний елемент.
     */
    boolean hasNext();
    /**
     * Повертає наступний елемент в ітерації.
     * @return Наступний елемент.
     */
    String next();
}
```

Вміст файлу IterableCollection.java:

```
package work4.Iterator;
/**
 * Інтерфейс Агрегата (Aggregate) або Колекції.
 * Змушує класи, що його реалізують, надавати методи ("фабрики")
 * для створення ітераторів.
 */
```

```
public interface IterableCollection {
    /**
     * Створює стандартний ітератор для обходу всіх елементів.
     * @return Новий екземпляр ітератора.
     */
    Iterator createIterator();

    /**
     * Створює ітератор, що обходить лише елементи, які відповідають умові.
     * @param condition Умова для фільтрації.
     * @return Новий екземпляр відфільтрованого ітератора.
     */
    Iterator createFilteredIterator(StringCondition condition);
}
```

Вміст файлу StringCondition.java:

```
package work4.Iterator;
/**
 * Функціональний інтерфейс, що виступає як "умова" або "фільтр" для рядків.
 * Використовується для передачі логіки фільтрації у {@link FilteredStringIterator}.
 */
public interface StringCondition {
    /**
     * Перевіряє, чи задовольняє рядок умову.
     * @param s Рядок для перевірки.
     * @return {@code true}, якщо умова виконана, інакше {@code false}.
     */
    boolean condition(String s);
}
```

Вміст файлу StringIterator.java:

```
package work4.Iterator;
import java.util.List;

/**
 * Клас Конкретного Ітератора (Concrete Iterator).
 * Реалізує базову логіку послідовного обходу списку рядків.
 */
public class StringIterator implements Iterator {
    protected List<String> list;
    protected int index;

    /**
     * Конструктор, що приймає колекцію для обходу.
     * @param list Колекція, яку потрібно ітерувати.
     */
    StringIterator(List<String> list) {
        this.list = list;
        this.index = 0;
    }

    /**
     * Перевіряє, чи є ще елементи для обходу.
     * @return {@code true}, якщо ітерація має наступний елемент.
     */
    @Override
    public boolean hasNext() {
```

```
        return index < list.size();
    }

    /**
     * Повертає наступний елемент в ітерації.
     * @return Наступний елемент.
     */
    @Override
    public String next() {
        return list.get(index++);
    }
}
```

Вміст файлу FilteredStringIterator.java:

```
package work4.Iterator;
import java.util.List;

/**
 * Спеціалізований Конкретний Ітератор, який додає логіку фільтрації.
 * Пропускає елементи, що не відповідають заданій умові {@link StringCondition}.
 */
public class FilteredStringIterator extends StringIterator{
    private final StringCondition condition;

    /**
     * Конструктор, що приймає колекцію та умову фільтрації.
     * @param list Колекція для обходу.
     * @param condition Умова для перевірки елементів.
     */
    FilteredStringIterator(List<String> list, StringCondition condition) {
        super(list);
        this.condition = condition;
        moveToNextValid();
    }

    /**
     * Внутрішній метод, що пересуває індекс до наступного елемента,
     * який відповідає умові фільтра.
     */
    private void moveToNextValid() {
        while (index < list.size() && !condition.condition(list.get(index))) {
            index++;
        }
    }

    /**
     * Перевіряє, чи є ще елементи для обходу.
     * @return {@code true}, якщо ітерація має наступний елемент.
     */
    @Override
    public boolean hasNext() {
        return index < list.size();
    }

    /**
     * Повертає наступний елемент в ітерації.
     * @return Наступний елемент.
     */
```

```
    @Override
    public String next() {
        String value = list.get(index++);
        moveToNextValid();
        return value;
    }
}
```

Вміст файлу StringList.java:

```
package work4.Iterator;
import java.util.ArrayList;
import java.util.List;

/**
 * Клас Конкретного Агрегата (Concrete Aggregate).
 * Це колекція, що зберігає рядки і реалізує фабричні методи
 * для створення відповідних ітераторів.
 */
public class StringList implements IterableCollection {
    private List<String> data = new ArrayList<>();

    /**
     * Додає новий рядок до колекції.
     * @param s Рядок для додавання.
     */
    public void add(String s){
        data.add(s);
    }

    /**
     * Створює стандартний ітератор для обходу всіх елементів.
     * @return Новий екземпляр ітератора.
     */
    @Override
    public Iterator createIterator(){
        return new StringIterator(data);
    }

    /**
     * Створює ітератор, що обходить лише елементи, які відповідають умові.
     * @param condition Умова для фільтрації.
     * @return Новий екземпляр відфільтрованого ітератора.
     */
    @Override
    public Iterator createFilteredIterator(StringCondition condition) {
        return new FilteredStringIterator(data, condition);
    }
}
```

Результат виконання програми:

```
Standart iterated list:
Hello
I
added some text
right here
$1
$A
>--------------------------------------
Filtered iterated list (elements with more then 3 characters):
Hello
added some text
right here
>--------------------------------------
Filtered iterated list (elements with letter a inside (include uppercase)):
added some text
$A
```

Рисунок 1 – результат виконання програми згідно першого завдання

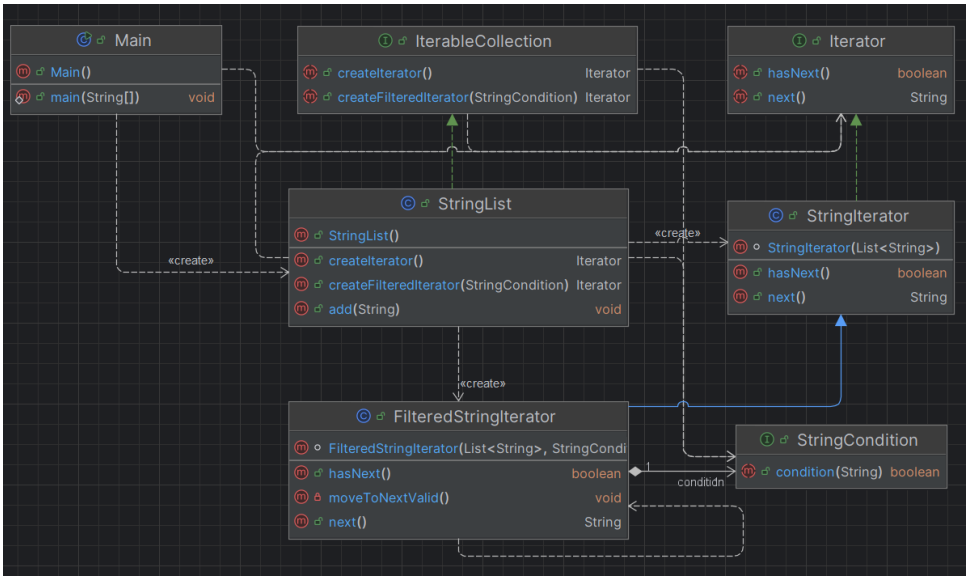

Рисунок 2 – Побудована в IntelliJ IDEA UML діаграма для першого завдання

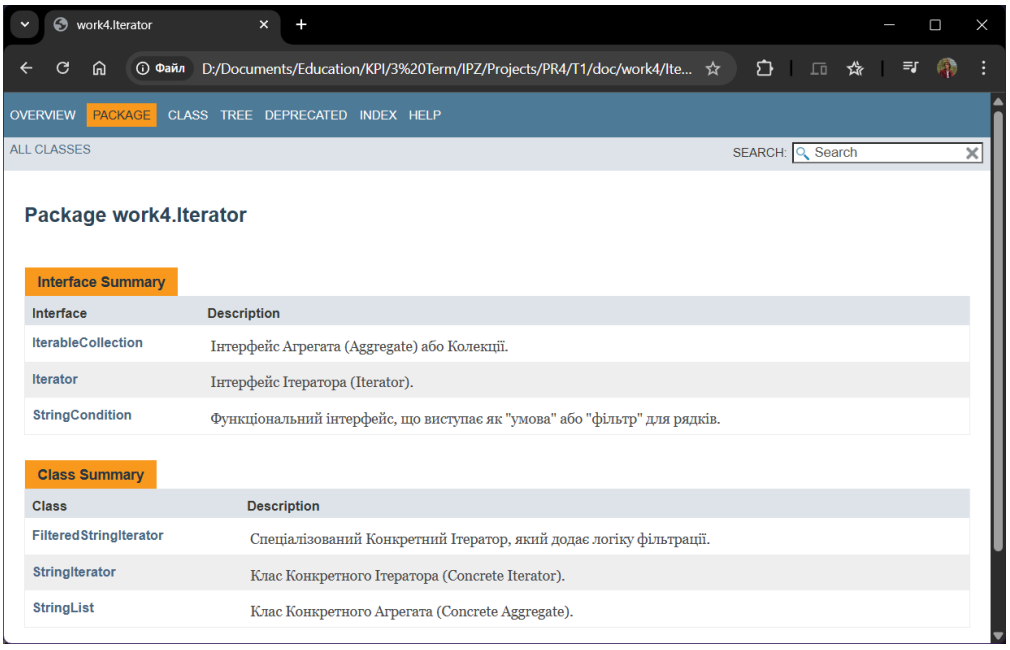

Рисунок 3 – Створена документація у форматі веб сторінки

Варіант 0 до завдання 2:

Необхідно розробити класи, які реалізують структуру для зберігання масиву цілих чисел та забезпечують можливість зміни алгоритму і напрямку сортування під час виконання програми. Конкретні алгоритми визначають спосіб сортування (наприклад, бульбашкове сортування, вибіркою, швидке сортування), параметр напрямку (зростання / спадання) задається зовнішнім чином без зміни основної логіки класу. Вимоги:

- *Клас IntArray:*
  - *містить:*
    - масив цілих чисел (int[] data);
    - посилання на поточний алгоритм сортування (SortStrategy strategy);
    - параметр напрямку сортування (boolean ascending).
  - *має методи:*
    - setStrategy(SortStrategy strategy) – встановлення алгоритму сортування;
    - setAscending(boolean ascending) – встановлення напрямку сортування;
    - sort() – виконує сортування за поточною стратегією;
    - print() – виводить масив на екран.
  - *Інтерфейс SortStrategy:*
    - визначає метод void sort(int[] data, boolean ascending)
    - який реалізують усі конкретні стратегії.
- *Конкретні стратегії сортування:*
  - BubbleSortStrategy – реалізація бульбашкового сортування;
  - SelectionSortStrategy – сортування вибіркою;
  - (додатково можна створити QuickSortStrategy – швидке сортування).
- *Напрямок сортування:*
  - передається як параметр ascending = true (за зростанням) або false (за спаданням);
  - реалізується без дублювання логіки алгоритму (через перевірку у порівняннях).
- Забезпечити можливість заміни стратегії у будь-який момент виконання програми.
  - *У методі main продемонструвати:*
    - створення об’єкта IntArray із початковим набором чисел;
    - встановлення алгоритму сортування (наприклад, BubbleSortStrategy);
    - виконання сортування за зростанням;
    - зміна алгоритму на інший (наприклад, SelectionSortStrategy) і повторне сортування;
    - демонстрацію сортування за спаданням.

Код до завдання 2:

Вміст файлу main.java:

```
import work4.Strategy.*;
/**
 * Клієнтський клас для демонстрації патерну Стратегія.
 * Створює контекст {@link IntArray} і динамічно змінює його поведінку (алгоритм
сортування)
 * під час виконання програми.
 */
public class Main {
    public static void main(String[] args) {
        int[] initialData = {64, 34, 25, 12, 22, 11, 90};
        String divider = ">----------------------------------------";
        System.out.println(divider);

        IntArray array = new IntArray(initialData);

        System.out.print("Created array: ");
        array.print();
        System.out.println(divider);
        System.out.println("We dont choose any strategy, so.. Class should give us error");
        array.sort();
        System.out.println(divider);

        array.setStrategy(new BubbleSortStrategy());
        System.out.println("Bubble sorted (ascending):");
        array.sort();
        array.print();
        System.out.println(divider);

        array = new IntArray(initialData);
        System.out.print("Array restored: ");
        array.print();
        System.out.println(divider);

        array.setStrategy(new SelectionSortStrategy());
        array.setAscending(false);
        System.out.println("Choosed Selection sort strategy (descending):");
        array.sort();
        array.print();
        System.out.println(divider);

        array = new IntArray(initialData);
        System.out.print("Array restored: ");
        array.print();
        System.out.println(divider);

        array.setStrategy(new BubbleSortStrategy());
        array.setAscending(false);
        System.out.println("Choosed bubble sort (descending):");
        array.sort();
        array.print();
        System.out.println(divider);
    }
}
```

Вміст файлу SortStrategy.java:

```
package work4.Strategy;
/**
 * Інтерфейс Стратегії (Strategy).
 * Визначає загальний метод для всіх алгоритмів сортування, дозволяючи
 * контексту {@link IntArray} викликати їх однаковим чином.
 */
public interface SortStrategy {
    /**
     * Сортує масив цілих чисел.
     * @param data Масив для сортування.
     * @param ascending {@code true} для сортування за зростанням, {@code false} - за
спаданням.
     */
    void sort(int[] data, boolean ascending);
}
```

Вміст файлу BubbleSortStrategy.java:

```
package work4.Strategy;
/**
 * Клас **Конкретної Стратегії** (Concrete Strategy).
 * Реалізує алгоритм сортування "бульбашкою".
 */
public class BubbleSortStrategy implements SortStrategy {
    @Override
    public void sort(int[] data, boolean ascending) {
        int n = data.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if ((ascending && data[j] > data[j + 1]) || (!ascending && data[j] < data[j + 1])) {
                    int temp = data[j];
                    data[j] = data[j + 1];
                    data[j + 1] = temp;
                }
            }
        }
    }
}
```

Вміст файлу SelectionSortStrategy.java:

```
package work4.Strategy;
/**
 * Клас **Конкретної Стратегії** (Concrete Strategy).
 * Реалізує алгоритм сортування "вибіркою".
 */
public class SelectionSortStrategy implements SortStrategy {
    @Override
    public void sort(int[] data, boolean ascending) {
        int n = data.length;
        for (int i = 0; i < n - 1; i++) {
            int minMaxIdx = i;
            for (int j = i + 1; j < n; j++) {
                if (ascending ? (data[j] < data[minMaxIdx]) : (data[j] > data[minMaxIdx])) {
                    minMaxIdx = j;
                }
            }
            int temp = data[minMaxIdx];
```

```
            data[minMaxIdx] = data[i];
            data[i] = temp;
        }
    }
}
```

Вміст файлу IntArray.java:

```
package work4.Strategy;
import java.util.Arrays;
/**
 * Клас Контексту (Context).
 * Зберігає дані (масив) та посилання на поточну {@link SortStrategy}.
 * Делегує виконання операції сортування об'єкту-стратегії.
 */
public class IntArray {
    private int[] data;
    private SortStrategy strategy;
    private boolean ascending = true;
    /**
     * Конструктор, що ініціалізує масив.
     * @param data Вхідний масив.
     */
    public IntArray(int[] data) {
        this.data = Arrays.copyOf(data, data.length);
    }
    /**
     * Встановлює (або змінює) поточну стратегію сортування.
     * @param strategy Об'єкт-стратегія для використання.
     */
    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    /**
     * Встановлює напрямок сортування.
     * @param ascending {@code true} для сортування за зростанням, {@code false} - за
спаданням.
     */
    public void setAscending(boolean ascending) {
        this.ascending = ascending;
    }
    /**
     * Виконує сортування, викликаючи метод {@code sort} поточної стратегії.
     */
    public void sort() {
        if (strategy == null) {
            System.out.println("[!] Error: You don't have a Strategy!");
            return;
        }
        strategy.sort(this.data, this.ascending);
    }
    /**
     * Виводить поточний стан масиву на екран.
     */
    public void print() {
        System.out.println(Arrays.toString(data));
    }
}
```

Результат виконання програми:

```
>---------------------------------------
Created array: [64, 34, 25, 12, 22, 11, 90]
>---------------------------------------
We dont choose any strategy, so.. Class should give us error
[!] Error: You don't have a Strategy!
>---------------------------------------
Bubble sorted (ascending):
[11, 12, 22, 25, 34, 64, 90]
>---------------------------------------
Array restored: [64, 34, 25, 12, 22, 11, 90]
>---------------------------------------
Choosed Selection sort strategy (descending):
[90, 64, 34, 25, 22, 12, 11]
>---------------------------------------
Array restored: [64, 34, 25, 12, 22, 11, 90]
>---------------------------------------
Choosed bubble sort (descending):
[90, 64, 34, 25, 22, 12, 11]
>---------------------------------------
```

Рисунок 4 – результат виконання програми згідно другого завдання

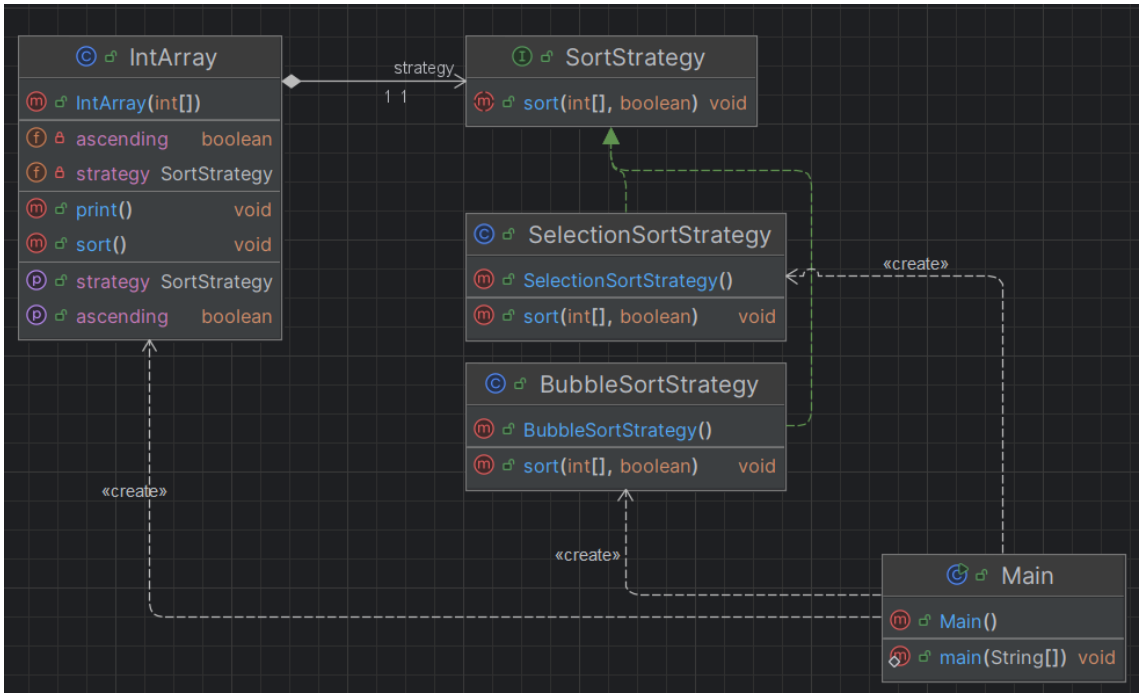

Рисунок 5 – Побудована в IntelliJ IDEA UML діаграма для другого завдання

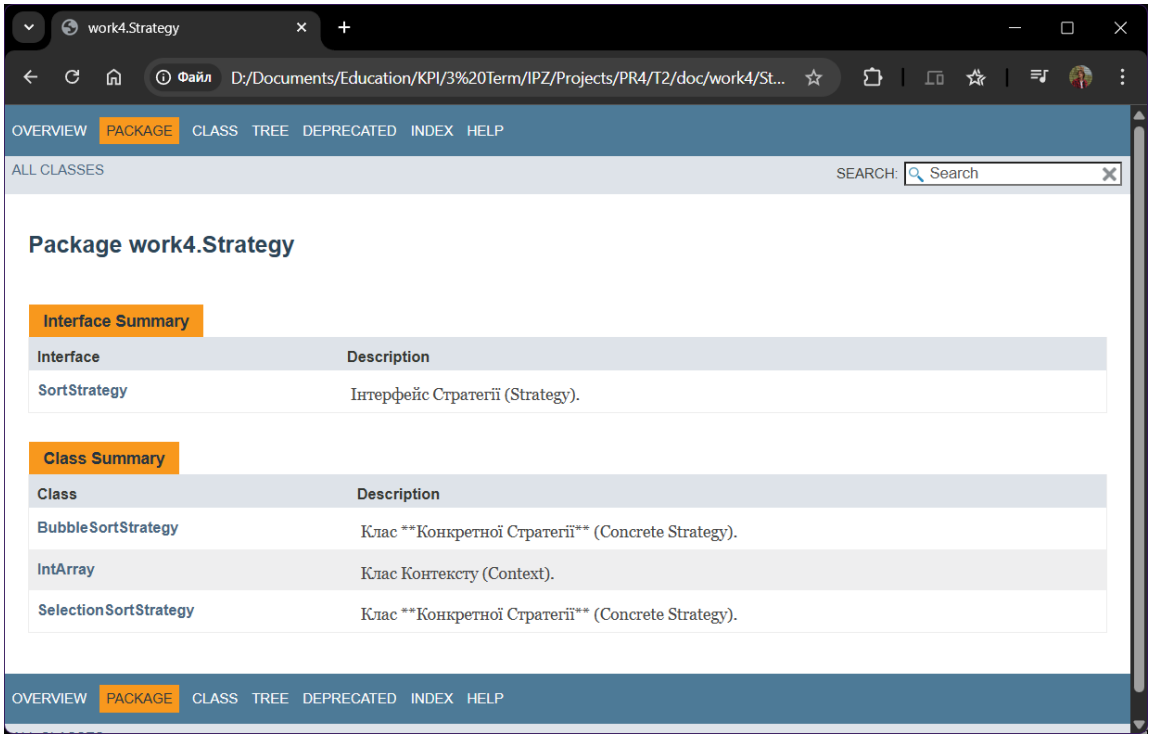

Рисунок 6 – Створена документація у форматі веб сторінки

Висновки:

Отже, під час виконання практичної роботи було створено програми, що реалізують поведінкові патерни проектування, а саме Strategy та Iterator. Патерн Strategy забезпечив можливість зміни алгоритму та напрямку сортування під час виконання програми, а в свою чергу, Iterator, послідовний доступ до елементів колекції без розкриття її внутрішньої структури. Використання цих шаблонів підвищило гнучкість, модульність і зручність супроводу коду. Під час виконання практичної роботи проблем не виникло. Увесь код міститься на даному GitHub репозиторії: https://github.com/BronievM/KPI-IPZ-2025/tree/main/PR4