

Лабораторная работа №7

Выполнила работу:
Рапп Ксения Александровна
Группа: 6204-010302D

Целью лабораторной работы является изучение и практическое применение паттернов проектирования в Java:

- **Паттерн «Итератор»** для последовательного доступа к элементам коллекции
- **Паттерн «Фабричный метод»** для создания объектов без указания конкретных классов
- **Рефлексия** для динамического создания объектов во время выполнения программы

Задание 1: Паттерн «Итератор»

Нужно было сделать объекты типа TabulatedFunction итерируемыми, чтобы их можно было использовать в улучшенном цикле for-each для последовательного доступа к точкам функции.

Реализация

1. Модификация интерфейса TabulatedFunction:

В файле TabulatedFunction.java расширила интерфейс, добавив наследование от Iterable<FunctionPoint>:

```
public interface TabulatedFunction extends Function, Cloneable, Iterable<FunctionPoint> {  
    Iterator<FunctionPoint> iterator();  
    ...  
}
```

2. Реализация итераторов:

В каждом классе, реализующем TabulatedFunction, добавила метод iterator(), возвращающий анонимный класс итератора:

Для ArrayTabulatedFunction

```
@Override  
public Iterator<FunctionPoint> iterator() {  
    // анонимный класс итератора  
    return new Iterator<FunctionPoint>() {  
        private int currentIndex = 0;  
        @Override  
        public boolean hasNext() {  
            // напрямую работаем с полем pointsCount для эффективности  
            return currentIndex < pointsCount;}  
        @Override  
        public FunctionPoint next() {  
            if (!hasNext()) {  
                throw new java.util.NoSuchElementException("Нет следующего элемента");}  
            // возвращаем копию точки (для инкапсуляции)  
            return new FunctionPoint(points[currentIndex++]);}  
        @Override  
        public void remove() {  
            throw new UnsupportedOperationException("Удаление не поддерживается");};};}
```

Для LinkedListTabulatedFunction

```
@Override
public Iterator<FunctionPoint> iterator() {
    // анонимный класс итератора
    return new Iterator<FunctionPoint>() {
        private FunctionNode currentNode = head.getNext();
        @Override
        public boolean hasNext() {
            // напрямую работаем с узлами списка для эффективности
            return currentNode != head;
        }
        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new java.util.NoSuchElementException("Нет следующего элемента");
            }
            // возвращаем копию точки для защиты инкапсуляции
            FunctionPoint point = new FunctionPoint(currentNode.getPoint());
            currentNode = currentNode.getNext();
            return point;
        }
        @Override
        public void remove() {
            throw new UnsupportedOperationException("Удаление не
поддерживается");}}};}
```

Особенности

- Инкапсуляция данных:** Каждый вызов next() возвращает новую копию точки, что предотвращает возможность модификации внутренних данных функции извне.
- Производительность:** Итераторы работают напрямую с внутренними структурами данных (массивом или узлами списка), не вызывая публичные методы getPoint(), что обеспечивает оптимальную производительность.
- Безопасность:** Реализована корректная обработка граничных случаев и исключительных ситуаций.

Теперь можно использовать for-each цикл для итерации по точкам любой табулированной функции:

Вывод на консоли:

```
ArrayTabulatedFunction (for-each цикл):
x = 1.0, y = 2.0
x = 2.0, y = 4.0
x = 3.0, y = 6.0
x = 4.0, y = 8.0
x = 5.0, y = 10.0

LinkedListTabulatedFunction (for-each цикл):
x = 0.0, y = 0.0
x = 1.0, y = 1.0
x = 2.0, y = 4.0
x = 3.0, y = 9.0
x = 4.0, y = 16.0

==== проверка исключений итератора ===

1. проверка NoSuchElementException:
NoSuchElementException выброшен корректно: NoSuchElementException

2. проверка UnsupportedOperationException:
UnsupportedOperationException выброшен корректно: UnsupportedOperationException
```

Задание 2: паттерн «Фабричный метод»

Нужно было обеспечить возможность динамического выбора типа создаваемых табулированных функций без изменения кода, который их использует.

Реализация

1. Создание интерфейса фабрики

В файле TabulatedFunctions.java определила интерфейс:

```
public interface TabulatedFunctionFactory {
    // создает табулированную функцию по границам и количеству точек
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount);

    // создает табулированную функцию по границам и массиву значений
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values);

    // создает табулированную функцию по массиву точек
    TabulatedFunction createTabulatedFunction(FunctionPoint[] points);
}
```

2. Реализация конкретных фабрик

Создала вложенные классы в соответствующих классах табулированных функций:

ArrayTabulatedFunctionFactory:

```
// вложенный класс фабрики для Array
public static class ArrayTabulatedFunctionFactory implements TabulatedFunctions.TabulatedFunctionFactory {
    @Override
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {
        return new ArrayTabulatedFunction(leftX, rightX, pointsCount);
    }

    @Override
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values) {
        return new ArrayTabulatedFunction(leftX, rightX, values);
    }

    @Override
    public TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {
        return new ArrayTabulatedFunction(points);
    }
}
```

LinkedListTabulatedFunctionFactory реализован аналогично.

3. Интеграция фабрик в систему

В классе TabulatedFunctions добавила статическое поле для хранения текущей фабрики:

```
private static TabulatedFunctionFactory factory = new ArrayTabulatedFunction.ArrayTabulatedFunctionFactory();

// приватный конструктор
private TabulatedFunctions() {
    throw new AssertionError("нельзя создавать объекты класса TabulatedFunctions");
}

// метод для установки фабрики
public static void setTabulatedFunctionFactory(TabulatedFunctionFactory factory) {
    TabulatedFunctions.factory = factory;
}
```

4. Модификация методов создания

Все методы, которые создавали табулированные функции, теперь используют текущую фабрику:

```
public static TabulatedFunction tabulate(Function function, double leftX, double rightX, int pointsCount) {
    // ...проверки и вычисления...

    // используем фабрику вместо прямого создания Array
    return createTabulatedFunction(leftX, rightX, values);
```

ВЫВОД В КОНСОЛИ:

```
==== тестирование фабрик табулированных функций ===

1. фабрика по умолчанию (должна быть ArrayTabulatedFunction):
   создан: ArrayTabulatedFunction

2. меняем на LinkedList фабрику:
   создан: LinkedListTabulatedFunction

3. возвращаем Array фабрику:
   создан: ArrayTabulatedFunction

4. тестирование разных методов создания:
   создание по границам и количеству точек: ArrayTabulatedFunction
   создание по границам и массиву значений: ArrayTabulatedFunction
   создание по массиву точек: ArrayTabulatedFunction
```

Задание 3: Рефлексивное создание объектов

Нужно было реализовать возможность создания объектов табулированных функций с указанием конкретного класса во время выполнения программы.

Реализация

1. Реализация методов рефлексивного создания

В файле TabulatedFunctions.java добавила методы, которые позволяют создавать объекты табулированных функций **динамически во время выполнения программы**, указывая конкретный класс, который нужно использовать:

```
// рефлексивные методы создания функций
public static TabulatedFunction createTabulatedFunctionByReflection(
    Class<?> functionClass, double leftX, double rightX, int pointsCount) {

    if (!TabulatedFunction.class.isAssignableFrom(functionClass)) {
        throw new IllegalArgumentException("Класс должен реализовывать TabulatedFunction");
    }

    try {
        Constructor<?> constructor = functionClass.getConstructor(
            double.class, double.class, int.class);
        return (TabulatedFunction) constructor.newInstance(leftX, rightX, pointsCount);
    } catch (NoSuchMethodException | InstantiationException |
        IllegalAccessException | InvocationTargetException e) {
        throw new IllegalArgumentException("Не удалось создать объект: " + e.getMessage(), e);    } }
```

2. Дополнительные методы:

Аналогично реализовала методы для создания через значения и через массив точек, а также метод tabulateByReflection() для табулирования с указанием типа.

Вывод на консоли:

```
==== тестирование рефлексивного создания объектов ====

1. создание ArrayTabulatedFunction через рефлексию:
   тип: ArrayTabulatedFunction
   функция: {(0.0; 0.0), (5.0; 0.0), (10.0; 0.0)}

2. создание ArrayTabulatedFunction через значения:
   тип: ArrayTabulatedFunction
   функция: {(0.0; 0.0), (5.0; 5.0), (10.0; 10.0)}

3. создание LinkedListTabulatedFunction через массив точек:
   тип: LinkedListTabulatedFunction
   функция: {(0.0; 0.0), (5.0; 25.0), (10.0; 100.0)}

4. табулирование с рефлексивным созданием:
   тип: LinkedListTabulatedFunction
   функция: {(0.0; 0.0), (0.7853981633974483; 0.7071067811865475), (1.5707963267948966; 1.0), (2.356194490192345; 0.7071067811865476), (3.141592653589793; 1.2246467991473532E-16)}

5. тест ошибки (класс не реализует TabulatedFunction):
   ожидаемая ошибка: IllegalArgumentException
   сообщение: Класс должен реализовывать TabulatedFunction

все тесты выполнены успешно!
```