Отчет по лабораторной работе №7

Паттерны Итератор и Фабричный метод

Выполнила: Андреяшкина Мария
Группа: 6204-010302D

Оглавление

Задание 1

Итератор

Цель: Сделать объекты типа TabulatedFunction итерируемыми для использования в цикле for-each.

Реализация

1. Изменения в интерфейсе TabulatedFunction:

```
public interface TabulatedFunction extends Function,
Iterable<FunctionPoint> {
    // существующие методы
}
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private int currentIndex = 0;

        @Override
        public boolean hasNext() {
            return currentIndex < pointsCount;
        }

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new java.util.NoSuchElementException("Нет
следующего элемента");
            }
            return new FunctionPoint(points[currentIndex++]);
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Удаление не
поддерживается");
        }
    };
```

```
}
```

2. Реализация в LinkedListTabulatedFunction:

```
@Override
public Iterator<FunctionPoint> iterator() {
    return new Iterator<FunctionPoint>() {
        private FunctionNode currentNode = head.next;

        @Override
        public boolean hasNext() {
            return currentNode != head;
        }

        @Override
        public FunctionPoint next() {
            if (!hasNext()) {
                throw new java.util.NoSuchElementException("Нет
следующего элемента");
            }
            FunctionPoint point = new
FunctionPoint(currentNode.point);
            currentNode = currentNode.next;
            return point;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Удаление не
поддерживается");
        }
    };
}
```

Результат тестирования

```
ArrayTabulatedFunction:
(0.0; 0.0)
(1.0; 1.0)
(2.0; 4.0)
(3.0; 9.0)

LinkedListTabulatedFunction:
(0.0; 0.0)
(1.0; 1.0)
(2.0; 4.0)
(3.0; 9.0)
```

Вывод: Итераторы работают корректно для обоих типов табулированных функций.

Задание 2

Фабричный метод

Цель: Реализовать возможность динамического выбора типа создаваемых объектов табулированных функций.

Реализация

1. Интерфейс фабрики:

```
public interface TabulatedFunctionFactory {
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount);
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values);
    TabulatedFunction createTabulatedFunction(FunctionPoint[] points);
}
```

2. Фабрика для ArrayTabulatedFunction:

```
public static class ArrayTabulatedFunctionFactory implements
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);

    }
}
```

3. Фабрика для LinkedListTabulatedFunction:

```
public static class LinkedListTabulatedFunctionFactory implements
TabulatedFunctionFactory {

    @Override

    public TabulatedFunction createTabulatedFunction(double leftX,
double rightX, int pointsCount) {

        return new LinkedListTabulatedFunction(leftX, rightX,
pointsCount);

    }


    @Override
```

```
    public TabulatedFunction createTabulatedFunction(double leftX,
double rightX, double[] values) {
        return new LinkedListTabulatedFunction(leftX, rightX,
values);
    }

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

4. Изменения в TabulatedFunctions:

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

public static void
setTabulatedFunctionFactory(TabulatedFunctionFactory factory) {
    TabulatedFunctions.factory = factory;
}

public static TabulatedFunction createTabulatedFunction(double
leftX, double rightX, int pointsCount) {
    return factory.createTabulatedFunction(leftX, rightX,
pointsCount);
}
```

Результат тестирования

```
Фабрика по умолчанию: class functions.ArrayTabulatedFunction
LinkedList фабрика: class functions.LinkedListTabulatedFunction
Array фабрика: class functions.ArrayTabulatedFunction
```

Вывод: Фабричный метод позволяет динамически изменять тип создаваемых объектов.

Задание 3

Рефлексия

Цель: Реализовать создание объектов через рефлексию по имени класса.

Реализация

1. Методы с рефлексией в TabulatedFunctions:

```
public static TabulatedFunction createTabulatedFunction(Class<?>
clazz, double leftX, double rightX, int pointsCount) {
    try {
        if (!TabulatedFunction.class.isAssignableFrom(clazz)) {
            throw new IllegalArgumentException("Класс должен
реализовывать TabulatedFunction");
        }

        java.lang.reflect.Constructor<?> constructor =
clazz.getConstructor(
            double.class, double.class, int.class);
        return (TabulatedFunction) constructor.newInstance(leftX,
rightX, pointsCount);
    } catch (Exception e) {
        throw new IllegalArgumentException("Ошибка при создании
объекта", e);
    }
}

public static TabulatedFunction tabulate(Class<?> clazz, Function
function,
                                         double leftX, double
rightX, int pointsCount) {
    // вычисление значений
    double[] values = new double[pointsCount];
```

```
    // ... вычисление

    return createTabulatedFunction(clazz, leftX, rightX, values);
}
```

Результат тестирования

```
1. Создание ArrayTabulatedFunction через рефлексию:
Класс: class functions.ArrayTabulatedFunction
Функция: {(0.0; 0.0), (5.0; 0.0), (10.0; 0.0)}

2. Создание ArrayTabulatedFunction с массивом значений:
Класс: class functions.ArrayTabulatedFunction
Функция: {(0.0; 0.0), (5.0; 10.0), (10.0; 20.0)}

3. Создание LinkedListTabulatedFunction через рефлексию:
Класс: class functions.LinkedListTabulatedFunction
Функция: {(0.0; 0.0), (5.0; 25.0), (10.0; 100.0)}

4. Табулирование через рефлексию:
Класс: class functions.LinkedListTabulatedFunction
Количество точек: 11
Первые 5 точек:
(0.0; 0.0)
(0.3141592653589793; 0.3090169943749474)
(0.6283185307179586; 0.5877852522924731)
(0.9424777960769379; 0.8090169943749475)
(1.2566370614359172; 0.9510565162951535)
```

Вывод: Рефлексия позволяет создавать объекты по имени класса в runtime.

Выводы

1. **Успешно реализован паттерн "Итератор"**:
    - Объекты TabulatedFunction стали итерируемыми
    - Поддерживается цикл for-each
    - Итераторы корректно работают для обеих реализаций
2. **Успешно реализован паттерн "Фабричный метод"**:
    - Создан гибкий механизм создания объектов
    - Возможность динамической смены фабрики
    - Упрощена модификация кода при добавлении новых типов функций
3. **Успешно реализовано создание через рефлексию**:
    - Объекты создаются по имени класса
    - Проверка соответствия интерфейсу
    - Обработка исключений при создании

Итог: Все требования лабораторной работы выполнены. Реализованы три паттерна проектирования, улучшающие гибкость и расширяемость системы функций.