Лабораторная работа № 7

Титульный лист

Дисциплина: Объектно-ориентированное программирование
Выполнил: Мостовщиков Владимир Витальевич
Группа: 6204-010302D
Преподаватель: Борисов Дмитрий Сергеевич
Год: 2025

Содержание:

Задание 1. Реализация паттерна «Итератор»

Целью первого задания было сделать так, чтобы все реализации TabulatedFunction можно было использовать в улучшенном цикле for-each, обрабатывая все точки функции по порядку и при этом не нарушая инкапсуляцию.

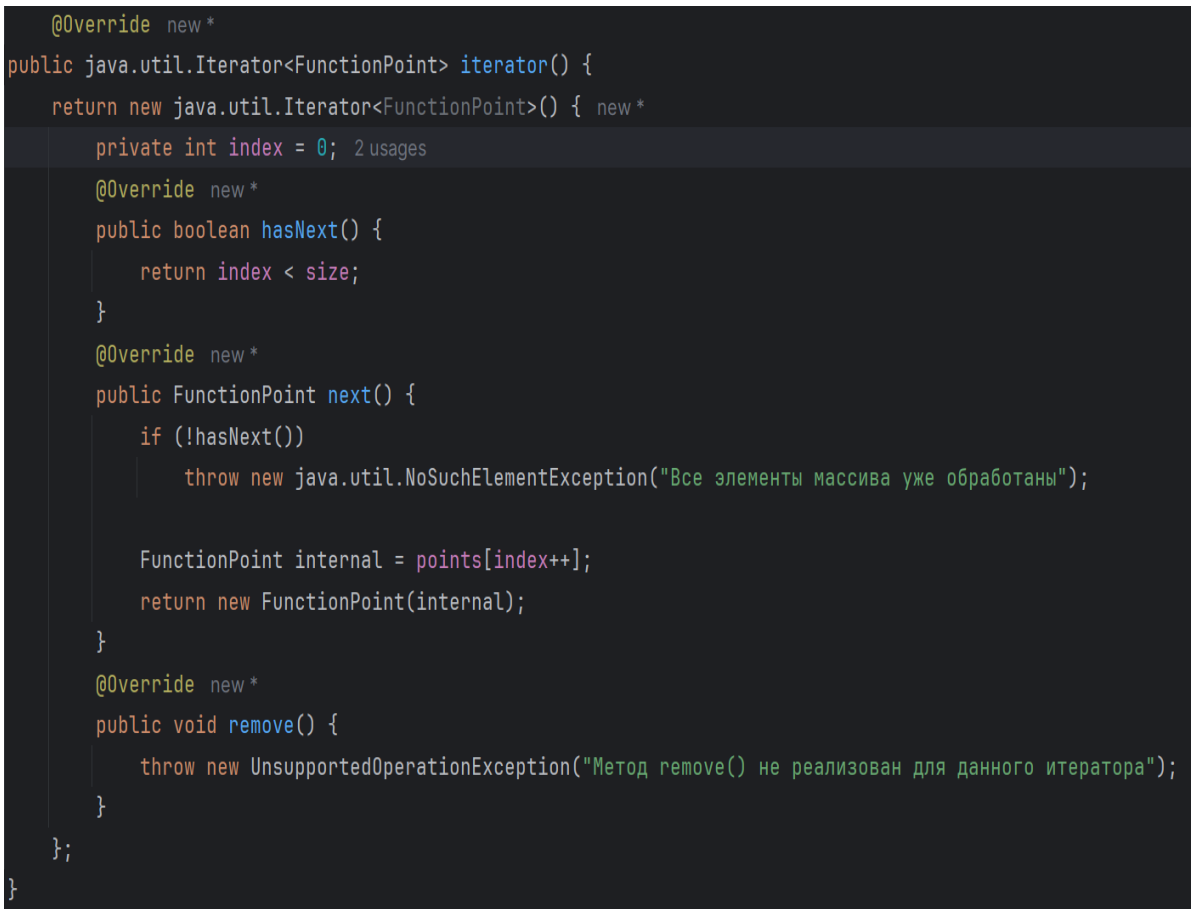
Сначала я, добавил наследование от Iterable<FunctionPoint> в интерфейс TabulatedFunction, а потом реализовал метод iterator()в ArrayTabulatedFunction, затем в LinkedListTabulatedFunction.

```
package functions;
import java.util.Iterator;
public interface TabulatedFunction extends Function, Cloneable, Iterable<FunctionPoint> {  37 us

    @Override  2 implementations  new *
    Iterator<FunctionPoint> iterator();

    TabulatedFunction clone(); // Создает и возвращает копию табулированной функции  2 implementa
    int getPointsCount();  7 usages  2 implementations  new *
    FunctionPoint getPoint(int index);  4 usages  2 implementations  new *
    void setPoint(int index, FunctionPoint point) throws InappropriateFunctionPointException;
    double getPointX(int index);  4 usages  2 implementations  new *
    void setPointX(int index, double x) throws InappropriateFunctionPointException;  no usages  2
    double getPointY(int index);  4 usages  2 implementations  new *
    void setPointY(int index, double y);  no usages  2 implementations  new *
    void deletePoint(int index);  no usages  2 implementations  new *
    void addPoint(FunctionPoint point) throws InappropriateFunctionPointException;  1 usage  2 imp
```

Рисунок 1 – расширяем интерфейс TabulatedFunction



```
    @Override  new *
public java.util.Iterator<FunctionPoint> iterator() {
    return new java.util.Iterator<FunctionPoint>() {  new *
        private int index = 0;  2 usages
        @Override  new *
        public boolean hasNext() {
            return index < size;
        }
        @Override  new *
        public FunctionPoint next() {
            if (!hasNext())
                throw new java.util.NoSuchElementException("Все элементы массива уже обработаны");

            FunctionPoint internal = points[index++];
            return new FunctionPoint(internal);
        }
        @Override  new *
        public void remove() {
            throw new UnsupportedOperationException("Метод remove() не реализован для данного итератора");
        }
    };
}
```

Рисунок 2 – реализация метода iterator() в ArrayTabulatedFunction

```
@Override  new *
public java.util.Iterator<FunctionPoint> iterator() {
    return new java.util.Iterator<FunctionPoint>() {  new *
        private FunctionNode current = head.next;  3 usages
        private int passed = 0;  2 usages
        @Override  new *
        public boolean hasNext() {
            return passed < size;
        }
        @Override  new *
        public FunctionPoint next() {
            if (!hasNext())
                throw new java.util.NoSuchElementException("Нет следующего элемента");
            FunctionPoint p = current.point;
            current = current.next;
            passed++;
            return new FunctionPoint(p.getX(), p.getY());   // Возвращаем копию точки
        }
        @Override  new *
        public void remove() {
            throw new UnsupportedOperationException("Удаление через итератор не поддерживается");
        }
    };
```

Рисунок 3 – реализация метода iterator() в LinkedListTabulatedFunction

Задание 2. Реализация паттерна «Фабричный метод»

Во втором задании требовалось сделать так, чтобы тип табулированной функции можно было выбирать динамически в ходе работы программы. Для этого нужно было избавиться от жёсткой привязки к ArrayTabulatedFunction внутри класса TabulatedFunctions и перенести создание объектов в отдельные фабрики. Сначала в пакете functions я добавил интерфейс фабрики

```java
package functions;

public interface TabulatedFunctionFactory {  4 usages  2 implementations  new *
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount);
    TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values);
    TabulatedFunction createTabulatedFunction(FunctionPoint[] points);  1 usage  2 implementations
}
```

Рисунок 4 - интерфейс фабрики

Тем самым я зафиксировал единый контракт для всех фабрик табулированных функций: любая фабрика обязана уметь создавать объект TabulatedFunction тремя способами: по границам и количеству точек, по границам и массиву y-значений и по массиву точек FunctionPoint[].

Далее я реализовал конкретную фабрику для массивной реализации. Внутри класса ArrayTabulatedFunction был добавлен вложенный публичный статический класс

```java
    public static class ArrayTabulatedFunctionFactory implements TabulatedFunctionFactory {  1 usage  new

    @Override  1 usage  new *
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {
        return new ArrayTabulatedFunction(leftX, rightX, pointsCount);
    }

    @Override  1 usage  new *
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values) {
        return new ArrayTabulatedFunction(leftX, rightX, values);
    }

    @Override  1 usage  new *
    public TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {
        return new ArrayTabulatedFunction(points);
    }
}
```

Рисунок 5 - Вложенная в ArrayTabulatedFunction фабрика

Аналогично я добавил вложенный класс LinkedListTabulatedFunctionFactory внутри LinkedListTabulatedFunction. Он также реализует интерфейс TabulatedFunctionFactory, но внутри вызывает уже конструкторы связного

списка.

```
    public static class LinkedListTabulatedFunctionFactory implements TabulatedFunctionFactory {  1 usage

    @Override  1 usage  new *
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {
        return new LinkedListTabulatedFunction(leftX, rightX, pointsCount);
    }

    @Override  1 usage  new *
    public TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values) {
        return new LinkedListTabulatedFunction(leftX, rightX, values);
    }

    @Override  1 usage  new *
    public TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {
        return new LinkedListTabulatedFunction(points);
    }
}
```

Рисунок 6 - Вложенная в LinkedListTabulatedFunction фабрика

После этого я перенёс точку принятия решения о типе функции в класс TabulatedFunctions. Я добавил:
«private static TabulatedFunctionFactory factory = new ArrayTabulatedFunction.ArrayTabulatedFunctionFactory();»

Задание 3. Создание объектов через рефлексию

В третьем задании я добавил возможность создания объектов табулированных функций через рефлексию, передавая класс объекта в качестве параметра.

В классе TabulatedFunctions я добавил три перегруженных метода createTabulatedFunction(), которые принимают первым параметром объект типа Class<? extends TabulatedFunction>

```
        // перегруженные методы создания функции
        public static TabulatedFunction createTabulatedFunction(double leftX, double rightX, int pointsCount) {
            return factory.createTabulatedFunction(leftX, rightX, pointsCount);
    }
    public static TabulatedFunction createTabulatedFunction(double leftX, double rightX, double[] values) {  no u
        return factory.createTabulatedFunction(leftX, rightX, values);
}



    public static TabulatedFunction createTabulatedFunction(FunctionPoint[] points) {  3 usages  new *
        return factory.createTabulatedFunction(points);
    }

public static TabulatedFunction createTabulatedFunction(  1 usage  new *
            Class<? extends TabulatedFunction> functionClass,
            double leftX, double rightX, int pointsCount) {

        try {
            // Ищем конструктор с параметрами (double, double, int)
            Constructor<? extends TabulatedFunction> constructor =
                functionClass.getConstructor(double.class, double.class, int.class);
```

```
public static TabulatedFunction createTabulatedFunction(  1 usage  new *
            Class<? extends TabulatedFunction> functionClass,
            double leftX, double rightX, int pointsCount) {

        try {
            // Ищем конструктор с параметрами (double, double, int)
            Constructor<? extends TabulatedFunction> constructor =
                functionClass.getConstructor(double.class, double.class, int.class);
            // Создаем объект с помощью найденного конструктора
            return constructor.newInstance(leftX, rightX, pointsCount);
        } catch (Exception e) {
            // Отлавливаем любые исключения рефлексии
            throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
        }
    }


    public static TabulatedFunction createTabulatedFunction(  no usages  new *
            Class<? extends TabulatedFunction> functionClass,
            double leftX, double rightX, double[] values) {
```

```java
public static TabulatedFunction createTabulatedFunction( no usages  new *
        Class<? extends TabulatedFunction> functionClass,
        double leftX, double rightX, double[] values) {

    try {
        // Ищем конструктор с параметрами (double, double, double[])
        Constructor<? extends TabulatedFunction> constructor =
            functionClass.getConstructor(double.class, double.class, double[].class);
        // Создаем объект с помощью найденного конструктора
        return constructor.newInstance(leftX, rightX, values);
    } catch (Exception e) {
        throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
    }
}


public static TabulatedFunction createTabulatedFunction( 1 usage  new *
        Class<? extends TabulatedFunction> functionClass,
        FunctionPoint[] points) {

    try {
```

```java
public static TabulatedFunction createTabulatedFunction( 1 usage  new *
        Class<? extends TabulatedFunction> functionClass,
        FunctionPoint[] points) {

    try {
        // Ищем конструктор с параметрами (FunctionPoint[])
        Constructor<? extends TabulatedFunction> constructor =
            functionClass.getConstructor(FunctionPoint[].class);
        // Создаем объект с помощью найденного конструктора
        return constructor.newInstance((Object) points);
    } catch (Exception e) {
        throw new IllegalArgumentException("Ошибка при создании объекта через рефлексию", e);
    }
}

public static TabulatedFunction tabulate( 1 usage  new *
        Class<? extends TabulatedFunction> functionClass,
        Function function, double leftX, double rightX, int pointsCount) {

    // Проверяем входные параметры
    if (function == null) {
```

Рисунок 7-10 – реализация методов createTabulatedFunction()

Следующим шагом была перегрузка метода табуляции:

```
public static TabulatedFunction tabulate( 1 usage  new *
        Class<? extends TabulatedFunction> functionClass,
        Function function, double leftX, double rightX, int pointsCount) {
    // Проверяем входные параметры
    if (function == null)
        throw new IllegalArgumentException("функция не должна быть null");
    if (pointsCount < 2)
        throw new IllegalArgumentException("количество точек должно быть >= 2");
    if (!(leftX < rightX))
        throw new IllegalArgumentException("левая граница должна быть меньше правой");
    // Проверяем, что отрезок внутри области определения функции
    if (!ge(leftX, function.getLeftDomainBorder()) || !le(rightX, function.getRightDomainBorder()))
        throw new IllegalArgumentException("отрезок табуляции выходит за область определения функции");
    // Создаем массив точек
    double step = (rightX - leftX) / (pointsCount - 1);
    FunctionPoint[] pts = new FunctionPoint[pointsCount];
    for (int i = 0; i < pointsCount; ++i) {
        double x = leftX + i * step;
        double y = function.getFunctionValue(x);
        pts[i] = new FunctionPoint(x, y);}
    return createTabulatedFunction(functionClass, pts); // Создаем табулированную функцию через рефлексию
```

Рисунок 11 – реализация tabulate

Так же я добавил перегруженные методы чтения через Рефлексию:

```
    // Бинарный ввод с указанием класса табулированной функции Реализованный через рефлексию
    public static TabulatedFunction inputTabulatedFunction( no usages  new *
            Class<? extends TabulatedFunction> functionClass,
            InputStream in) {
        try {
            DataInputStream dis = new DataInputStream(new BufferedInputStream(in));
            int n = dis.readInt();
            FunctionPoint[] pts = new FunctionPoint[n];
            for (int i = 0; i < n; ++i) {
                double x = dis.readDouble();
                double y = dis.readDouble();
                pts[i] = new FunctionPoint(x, y);
            }
            // Здесь используем рефлексивный createTabulatedFunction
            return createTabulatedFunction(functionClass, pts);
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }
```

Рисунок 12 – реализация бинарного чтения с указанием класса через рефлексию

```java
// Ввод табулированной функции из символьного потока с указанием класса Реализованный через рефлексию
public static TabulatedFunction readTabulatedFunction(  no usages  new *
        Class<? extends TabulatedFunction> functionClass,
        Reader in) {
    try {
        StreamTokenizer st = new StreamTokenizer(in);
        st.parseNumbers();
        int t = st.nextToken();
        if (t != StreamTokenizer.TT_NUMBER)
            throw new IOException("Expected points count");
        int n = (int) st.nval; // Считываем количество точек
        List<FunctionPoint> list = new ArrayList<>(n);
        for (int i = 0; i < n; ++i) {
            if (st.nextToken() != StreamTokenizer.TT_NUMBER)
                throw new IOException("Expected x");
            double x = st.nval;
            if (st.nextToken() != StreamTokenizer.TT_NUMBER)
                throw new IOException("Expected y");
            double y = st.nval;
            list.add(new FunctionPoint(x, y));}
        FunctionPoint[] pts = list.toArray(new FunctionPoint[0]); // Собираем функцию из считанных точек
```

```java
            int t = st.nextToken();
            if (t != StreamTokenizer.TT_NUMBER)
                throw new IOException("Expected points count");
            int n = (int) st.nval; // Считываем количество точек
            List<FunctionPoint> list = new ArrayList<>(n);
            for (int i = 0; i < n; ++i) {
                if (st.nextToken() != StreamTokenizer.TT_NUMBER)
                    throw new IOException("Expected x");
                double x = st.nval;
                if (st.nextToken() != StreamTokenizer.TT_NUMBER)
                    throw new IOException("Expected y");
                double y = st.nval;
                list.add(new FunctionPoint(x, y));}
            FunctionPoint[] pts = list.toArray(new FunctionPoint[0]); // Собираем функцию из считанных точек
            return createTabulatedFunction(functionClass, pts); // Создаём объект нужного класса через рефлексию
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    }
```

Рисунок 13,14 – реализация метода чтения табулированной функции из символьного потока через рефлексию

Далее я перешел к тестированию.

```
public class Main {  new *
    public static void main(String[] args) {  new *

        // Проверка итераторов
        System.out.println("1. Проверка итераторов:");
        System.out.println("ArrayTabulatedFunction:");
        TabulatedFunction func1 = new ArrayTabulatedFunction( leftX: 0,  rightX: 5,  pointsCount: 3);
        for (FunctionPoint p : func1) {
            System.out.println(p);
        }

        System.out.println("LinkedListTabulatedFunction:");
        TabulatedFunction func2 = new LinkedListTabulatedFunction(
            new FunctionPoint[]{new FunctionPoint( x: 1,  y: 1), new FunctionPoint( x: 2,  y: 4), new FunctionPoint( x: 3,  y: 9)});
        for (FunctionPoint p : func2) {
            System.out.println(p);
        }

        // Проверка фабрик
        System.out.println("\n2. Проверка фабрик:");
        TabulatedFunction tf;

        // Сначала фабрика по умолчанию (должна быть Array)
        tf = TabulatedFunctions.tabulate(new Sin(),  leftX: 0, Math.PI,  pointsCount: 4);
```

```
        // Сначала фабрика по умолчанию (должна быть Array)
        tf = TabulatedFunctions.tabulate(new Sin(),  leftX: 0, Math.PI,  pointsCount: 4);
        System.out.println("По умолчанию: " + tf.getClass().getSimpleName());

        // Меняем на LinkedList
        TabulatedFunctions.setTabulatedFunctionFactory(
            new LinkedListTabulatedFunction.LinkedListTabulatedFunctionFactory());
        tf = TabulatedFunctions.tabulate(new Sin(),  leftX: 0, Math.PI,  pointsCount: 4);
        System.out.println("После смены: " + tf.getClass().getSimpleName());

        // Обратно на Array
        TabulatedFunctions.setTabulatedFunctionFactory(
            new ArrayTabulatedFunction.ArrayTabulatedFunctionFactory());
        tf = TabulatedFunctions.tabulate(new Sin(),  leftX: 0, Math.PI,  pointsCount: 4);
        System.out.println("Обратно: " + tf.getClass().getSimpleName());

        // Проверка рефлексии
        System.out.println("\n3. Проверка рефлексии:");
        TabulatedFunction rf;
```

```java
// Проверка рефлексии
System.out.println("\n3. Проверка рефлексии:");
TabulatedFunction rf;

rf = TabulatedFunctions.createTabulatedFunction(
    ArrayTabulatedFunction.class, leftX: 0, rightX: 10, pointsCount: 4);
System.out.println("Array через рефлексию: " + rf.getClass().getSimpleName());

rf = TabulatedFunctions.createTabulatedFunction(
    LinkedListTabulatedFunction.class,
    new FunctionPoint[]{new FunctionPoint( x: 0, y: 0), new FunctionPoint( x: 1, y: 1)});
System.out.println("LinkedList через рефлексию: " + rf.getClass().getSimpleName());

rf = TabulatedFunctions.tabulate(
    LinkedListTabulatedFunction.class, new Cos(), leftX: 0, Math.PI, pointsCount: 5);
System.out.println("Табулирование через рефлексию: " + rf.getClass().getSimpleName());
```

```java
// Проверка ошибок
System.out.println("\n4. Проверка ошибок:");
try {
    TabulatedFunctions.createTabulatedFunction(
        ArrayTabulatedFunction.class, leftX: 10, rightX: 0, pointsCount: 3);
} catch (Exception e) {
    System.out.println("Поймали ошибку: " + e.getMessage());
}
```

```
1. Проверка итераторов:
ArrayTabulatedFunction:
(0.0; 0.0)
(2.5; 0.0)
(5.0; 0.0)
LinkedListTabulatedFunction:
(1.0; 1.0)
(2.0; 4.0)
(3.0; 9.0)

2. Проверка фабрик:
По умолчанию: ArrayTabulatedFunction
После смены: LinkedListTabulatedFunction
Обратно: ArrayTabulatedFunction

3. Проверка рефлексии:
Array через рефлексию: ArrayTabulatedFunction
LinkedList через рефлексию: LinkedListTabulatedFunction
Табулирование через рефлексию: LinkedListTabulatedFunction
```

```
3. Проверка рефлексии:
Array через рефлексию: ArrayTabulatedFunction
LinkedList через рефлексию: LinkedListTabulatedFunction
Табулирование через рефлексию: LinkedListTabulatedFunction


4. Проверка ошибок:
Поймали ошибку: Ошибка при создании объекта через рефлексию
```

```
5. Проверка бинарного чтения через фабрику и через рефлексию:
Исходная функция ArrayTabulatedFunction:
(0.0; 0.0)
(1.0; 1.0)
(2.0; 4.0)

Прочитано обычным бинарным методом через фабрику: ArrayTabulatedFunction
(0.0; 0.0)
(1.0; 1.0)
(2.0; 4.0)

Прочитано бинарным методом через рефлексию LinkedListTabulatedFunction: LinkedListTabulatedFunction
(0.0; 0.0)
(1.0; 1.0)
(2.0; 4.0)
```

```
6. Проверка символьного чтения через фабрику и через рефлексию:
Исходная функция ArrayTabulatedFunction:
(0.0; 0.0)
(1.0; 1.0)
(2.0; 4.0)
(3.0; 9.0)

Строковое представление функции:
4 0.0 0.0 1.0 1.0 2.0 4.0 3.0 9.0

Прочитано обычным символьным методом через фабрику: ArrayTabulatedFunction
(0.0; 0.0)
(1.0; 1.0)
(2.0; 4.0)
(3.0; 9.0)

Прочитано символьным методом через рефлексию LinkedListTabulatedFunction: LinkedListTabulatedFunction
(0.0; 0.0)
(1.0; 1.0)
(2.0; 4.0)
(3.0; 9.0)
```