Лабораторная работа №7

Выполнила: Джумашева Камилла Исфатовна

Студент 6203-010302D группы

Ход выполнения

Задание 1

Задание 1 В интерфейс TabulatedFunction было добавлено наследование от интерфейса Iterable<FunctionPoint>, что позволило использовать объекты табулированных функций в циклах for-each. В классах ArrayTabulatedFunction и LinkedListTabulatedFunction был реализован метод iterator(). Реализация итераторов выполнена с использованием анонимных внутренних классов:

- Для массива итератор проходит по индексам от 0 до count.
- Для связного списка итератор перемещается по узлам от головы до хвоста. Методы итераторов выбрасывают NoSuchElementException при выходе за границы и UnsupportedOperationException при попытке удаления, соблюдая контракт интерфейса. В методе Main была проведена проверка: оба типа функций успешно перебираются в улучшенном цикле for.

Задание 2

Для решения проблемы фиксированного типа создаваемых функций был применен паттерн фабричный метод. В пакете functions создан интерфейс TabulatedFunctionFactory с методами создания функций. В классах ArrayTabulatedFunction и LinkedListTabulatedFunction реализованы вложенные публичные классы-фабрики (ArrayTabulatedFunctionFactory и LinkedListTabulatedFunctionFactory), реализующие этот интерфейс. В классе TabulatedFunctions добавлено статическое поле фабрики и метод setTabulatedFunctionFactory для ее замены. Все методы, создающие функции (tabulate, readTabulatedFunction, inputTabulatedFunction), были переписаны для использования текущей установленной фабрики вместо прямого вызова конструкторов. Тестирование в Main показало, что при смене фабрики методы класса TabulatedFunctions начинают возвращать экземпляры соответствующего типа (массив или связный список).

Задание 3

Реализован механизм создания объектов через рефлексию (Reflection API), позволяющий явно указывать желаемый тип функции при вызове методов. В класс TabulatedFunctions добавлены перегруженные методы createTabulatedFunction и tabulate, принимающие параметр Class<? extends TabulatedFunction>. С помощью методов класса Class (getConstructor) и Constructor (newInstance) реализован поиск подходящего конструктора и создание экземпляра класса. Обработаны исключения рефлексии (например, NoSuchMethodException, InvocationTargetException): они перехватываются и оборачиваются в IllegalArgumentException, чтобы сохранить понятный интерфейс для

пользователя. Проверка показала успешное создание объектов обоих типов через передачу ссылки на класс .class.

```
Задание 1: Итераторы
Перебор LinkedListTabulatedFunction:
(1.0; 1.0)
(2.0; 4.0)
(3.0; 9.0)

Перебор ArrayTabulatedFunction:
(1.0; 1.0)
(2.0; 4.0)
(3.0; 9.0)

Задание 2: Фабрики
class functions.ArrayTabulatedFunction
class functions.LinkedListTabulatedFunction
class functions.ArrayTabulatedFunction

Задание 3: Рефлексия
class functions.ArrayTabulatedFunction
{(0.0; 0.0), (5.0; 0.0), (10.0; 0.0)}
class functions.ArrayTabulatedFunction
{(0.0; 0.0), (10.0; 10.0)}
class functions.LinkedListTabulatedFunction
{(0.0; 0.0), (10.0; 10.0)}
class functions.LinkedListTabulatedFunction
{(0.0; 0.0), (0.3141592653589793; 0.3090169943749474), (0.6283185307179586; 0.5877852522924731
5), (1.5707963267948966; 1.0), (1.8849555921538759; 0.9510565162951536), (2.199114857512855; 0
.3090169943749475), (3.141592653589793; 1.2246467991473532E-16)}
```

Рис. 1

Заключение

В ходе лабораторной работы были изучены и применены на практике ключевые паттерны проектирования и возможности языка Java. Паттерн «Итератор» позволил унифицировать перебор элементов функций независимо от их внутренней структуры (массив или список). Паттерн «Фабричный метод» обеспечил гибкость настройки библиотеки, позволив глобально менять тип создаваемых объектов. Использование Reflection API дало возможность динамически управлять типами создаваемых объектов во время выполнения, что расширяет возможности TabulatedFunctions без изменения его исходного кода при добавлении новых реализаций функций.