ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ОЛЕСЯ ГОНЧАРА

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ МАТЕМАТИКИ ТА МАТЕМАТИЧНОЇ КІБЕРНЕТИКИ

ЗВІТ ПРО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ № 2

З дисципліни «Сучасні середовища програмування»

Перший (бакалаврський) рівень вищої освіти

Спеціальність 124 Системний аналіз

Освітня програма  Комп’ютерне моделювання

  та технології програмування

Виконавець:

Студент групи ПС–21–1

Кравченко Юлія

Варіант № 4

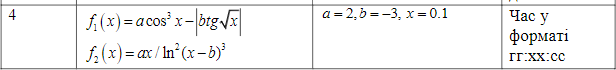
Дніпро

2023

1. Постановка завдання

Застосувати патерн Компонувальник (Composite) для задачі обчислення похідної довільної складеної функції. Використати дані табл.1(я обрала за приклад функцію №2)

Варіант індивідуального завдання 4:



1. Опис логічної структури Java-програми.

Даний код містить наступні класи:

1. Abs - клас, що реалізує функцію модуля.
2. Composite - абстрактний клас, який визначає спільний інтерфейс для складових функцій.
3. Const - клас, що визначає фіксоване значення (константу).
4. Cos - клас, що реалізує функцію косинуса.
5. Difference - клас, що визначає різницю між двома функціями.
6. Fraction - клас, що реалізує функцію ділення.
7. Ln - клас, що реалізує функцію натурального логарифму.
8. Main - головний клас програми, який містить точку входу до програми.
9. Negation - клас, що реалізує функцію зміни знаку.
10. Power - клас, що реалізує функцію піднесення до степеню.
11. Product - клас, що визначає добуток декількох функцій.
12. Sign - клас, що визначає знак функції.

Кожен клас реалізує інтерфейс Function, який містить три методи: calculate, derivative, toPrettyString, що дозволяє їх використовувати як функції.

Також в коді є імпорти java.text.NumberFormat та java.util.ArrayList. Перший використовується для форматування чисел, а другий - для роботи з динамічними масивами.

1. Патерн "Компонувальник" (Composite)

Опис: Цей патерн використовується для створення ієрархії об'єктів, де кожен об'єкт може містити інші об'єкти, які в свою чергу можуть містити ще інші об'єкти і так далі. Всі об'єкти повинні мати спільний інтерфейс, що дозволяє їх взаємозамінювати, тобто кожен об'єкт повинен мати методи, які мають однаковий інтерфейс, незалежно від того, який об'єкт він представляє.

У нашій програмі патерн "Компонувальник" використовується для створення складних математичних виразів, які можуть містити в собі різні функції. Кожна функція є окремим об'єктом, який містить методи обчислення значення та похідної функції, а також може містити інші функції.

Параметри конструкторів:

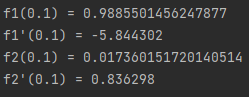
* Клас Function містить параметри конструктора без параметрів, оскільки всі об'єкти функцій створюються за допомогою фабричного методу createFunction() класу FunctionFactory.
* Класи-спадкоємці, наприклад, AddFunction, містять параметри конструктора, які необхідні для створення складних виразів. Наприклад, AddFunction містить два параметри - лівий та правий операнд, які додаються.

Паттерн "Factory Method" був використаний для створення різних функцій без необхідності вказувати конкретний клас функції. Фабричний метод (Factory Method) - це породжуючий паттерн проектування, який визначає загальний інтерфейс для створення об'єктів у суперкласі, дозволяючи підкласам змінювати тип створюваних об'єктів.

Паттерн "Decorator" був використаний для додавання додаткової функціональності до існуючих функцій. Декоратор (Decorator) - це структурний паттерн проектування, який дозволяє динамічно додавати до об'єктів нову функціональність, обгортаючи їх в об'єкти декораторів.

Паттерн "Strategy" був використаний для визначення різних алгоритмів обчислення похідної. Стратегія (Strategy) - це поведінковий паттерн проектування, який визначає сімейство алгоритмів, інкапсулює кожен з них і забезпечує їх взаємозамінність.

3.Результати виконання:



4.Висновок:

За результатами виконання даної роботи можна зробити висновок, що програмування на Java вимагає від програмістів доброго розуміння концепцій об'єктно-орієнтованого програмування та патернів проектування. Використання патернів дозволяє створювати гнучкі та розширювані програми, які легко змінювати та модифікувати у випадку потреби.

Також було добре виконано роботу зі змінними та функціями, які є ключовими концепціями в багатьох наукових та технічних обчисленнях. Використання об'єктно-орієнтованого підходу дозволило легко створювати та маніпулювати об'єктами, що представляють змінні та функції.

Також було показано використання різних патернів проектування, таких як фабричний метод, декоратор та стратегія. Вони дозволяють створювати гнучкі та розширювані програми, які можна легко модифікувати та розширювати у випадку потреби.

5.Питання:

1.

**Як можна застосувати патерни проектування для оптимізації швидкості та ефективності виконання коду?**

Патерни проектування можуть бути застосовані для оптимізації швидкості та ефективності виконання коду шляхом використання підходів, які дозволяють створювати гнучкі та легкі для розширення системи.

Наприклад, патерн "Стратегія" може бути застосований для оптимізації алгоритмів, дозволяючи замінювати різні стратегії обчислення на льоту. Патерн "Команда" може допомогти оптимізувати швидкість виконання коду, зберігаючи його у вигляді команд, які можна виконувати асинхронно. Патерн "Декоратор" може бути використаний для динамічного додавання функціональності до об'єктів, що дозволяє досягти більшої ефективності коду та зменшити його складність.

Крім того, патерни проектування можуть допомогти уникнути дублювання коду та розширити функціональність за допомогою розробки систем, які є легкими для розширення та підтримки.

2.

**Як можна застосувати патерни проектування для забезпечення безпеки та стабільності програмного забезпечення?**

Існує кілька патернів проектування, які можна використовувати для забезпечення безпеки та стабільності програмного забезпечення. Ось кілька прикладів:

1. Патерн "Singleton" можна використовувати для забезпечення безпеки даних ініціалізації об'єктів, тому що цей патерн дозволяє гарантувати, що лише один екземпляр класу буде створений і використовуваний в програмі. Це зменшує ризик витоку даних та забезпечує стабільність роботи програми.
2. Патерн "Decorator" можна використовувати для забезпечення безпеки при обробці даних. Цей патерн дозволяє додавати додаткову функціональність до об'єктів, не змінюючи їх основний функціонал. Це зменшує ризик помилок при обробці даних та забезпечує стабільність роботи програми.
3. Патерн "Observer" можна використовувати для забезпечення безпеки при сповіщенні про зміни. Цей патерн дозволяє створювати залежності між об'єктами, так що коли стан одного об'єкта змінюється, інші об'єкти, які на нього підписалися, автоматично отримують сповіщення про зміни. Це зменшує ризик втрати даних та забезпечує стабільність роботи програми.

Ці патерни проектування є лише деякими прикладами того, як можна використовувати патерни для забезпечення безпеки та стабільності програмного забезпечення. Однак, використання патернів проектування не гарантує 100% безпеки та стабільності програмного забезпечення

6.Додаток:

ь в коді є помилка при знаходженні похідної функцій що маємо в классі Мейн, будь ласка знайди та виправ її точно вказавши ділянку кода

package edu.mde.lab2;  
import java.text.NumberFormat;  
  
public class Abs implements Function {  
 private final Function arg;  
 public Abs(Function arg) {  
 this.arg = arg;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return Math.*abs*(arg.calculate(x));  
 }  
  
 @Override  
 public Function derivative() {  
 return new Fraction(  
 new Product(arg, new Sign(arg)),  
 new Abs(arg)  
 );  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat format) {  
 return "|" + arg.toPrettyString(format) + "|";  
 }}

package edu.mde.lab2;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
  
public abstract class Composite implements Function {  
 private final ArrayList<Function> terms;  
  
 public ArrayList<Function> terms() {  
 return terms;  
 }  
  
 public Composite() {  
 terms = new ArrayList<>();  
 }  
  
 public Composite(Function... terms) {  
 this.terms = new ArrayList<>(Arrays.*asList*(terms));  
 }  
  
 public Composite(ArrayList<Function> terms) {  
 this.terms = terms;  
 }  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
public class Const extends Number implements Function {  
 public static final Const *ZERO* = new Const(0);  
 public static final Const *ONE* = new Const(1);  
 public static final Const *NEGATIVE\_ONE* = new Const(-1);  
 private final double value;  
 public Const(double value) {  
 this.value = value;  
 }  
 @Override  
 public double calculate(double x) {  
 return value;  
 }  
 @Override  
 public Function derivative() {  
 return *ZERO*;  
 }  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return nf.format(value);  
 }  
 @Override  
 public int intValue() {  
 return (int) value;  
 }  
 @Override  
 public long longValue() {  
 return (long) value;  
 }  
 @Override  
 public float floatValue() {  
 return (float) value;  
 }  
 @Override  
 public double doubleValue() {  
 return value;  
 }  
 public static Const of(double value) {  
 return new Const(value);  
 }  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
public class Cos implements Function {  
 private final Function arg;  
 public Cos(Function arg) {  
 this.arg = arg;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return Math.*cos*(arg.calculate(x));  
 }  
  
 @Override  
 public Function derivative() {  
 return new Product(new Negation(new Sin(arg)), arg.derivative());  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat format) {  
 return "cos(" + arg.toPrettyString(format) + ")";  
 }  
}

package edu.mde.lab2;  
  
  
import java.text.NumberFormat;  
  
public class Difference implements Function {  
 private final Function left;  
 private final Function right;  
  
 public Difference(Function left, Function right) {  
 this.left = left;  
 this.right = right;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return left.calculate(x) - right.calculate(x);  
 }  
  
 @Override  
 public Function derivative() {  
 return new Difference(left.derivative(), right.derivative());  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return null;  
 }  
  
}

package edu.mde.lab2;  
import java.text.NumberFormat;  
  
// Деление  
public class Fraction implements Function {  
 private final Function numerator;  
 private final Function denominator;  
  
 public Fraction(Function numerator, Function denominator) {  
 this.numerator = numerator;  
 this.denominator = denominator;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return numerator.calculate(x) / denominator.calculate(x);  
 }  
  
 @Override  
 public Function derivative() {  
 return new Fraction(  
 new Difference(  
 new Product(numerator.derivative(), denominator),  
 new Product(numerator, denominator.derivative())  
 ),  
 new Power(denominator, new Const(2))  
 );  
 }  
 @Override  
 public String toPrettyString(NumberFormat format) {  
 return "(" + numerator.toPrettyString(format) + " / " + denominator.toPrettyString(format) + ")";  
 }  
  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
public interface Function {  
 double calculate(double x);  
  
 Function derivative();  
  
 String toPrettyString(NumberFormat nf);  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
public class Ln implements Function {  
  
 private final Function arg;  
 public Ln(Function arg) {  
 this.arg = arg;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return Math.*log*(arg.calculate(x));  
 }  
  
 @Override  
 public Function derivative() {  
 return new Fraction(arg.derivative(), arg);  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat format) {  
 return "ln(" + arg.toPrettyString(format) + ")";  
 }  
}

package edu.mde.lab2;  
  
  
public class Main {  
 public static void main(String[] args) {  
 // ф-ция f1(x)= 2\*cos(x)^3-|-3\*tg√x|  
 Function f1 = new Difference(  
 new Product(  
 new Const(2),  
 new Power(  
 new Cos(new Variable("x")),  
 new Const(3)  
 )  
 ),  
 new Abs(  
 new Negation(  
 new Product(  
 new Const(3),  
 new Tan(  
 new Power(  
 new Variable("x"),  
 new Const(0.5)  
 )  
 )  
 )  
 )  
 )  
 );  
  
 double x = 0.1;  
 System.*out*.println("f1(" + x + ") = " + f1.calculate(x));  
 System.*out*.println("f1'(" + x + ") = " + f1.derivative().calculate(x));  
  
 // f2(x)=2\*x/(ln((x+3)^3))^2  
 Function f2 = new Fraction(  
 new Product(  
 new Const(2),  
 new Variable("x")  
 ),  
 new Power(  
 new Ln(  
 new Power(  
 new Sum(  
 new Variable("x"),  
 new Const(3)  
 ),  
 new Const(3)  
 )  
 ),  
 new Const(2)  
 )  
 );  
 System.*out*.println("f2(" + x + ") = " + f2.calculate(x));  
 System.*out*.println("f2'(" + x + ") = " + f2.derivative().calculate(x));  
 }  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
public class Negation implements Function {  
 private final Function arg;  
 public Negation(Function arg) {  
 this.arg = arg;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return -arg.calculate(x);  
 }  
  
 @Override  
 public Function derivative() {  
 return new Negation(arg.derivative());  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat format) {  
 return "-(" + arg.toPrettyString(format) + ")";  
 }  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
public class Power implements Function {  
 private final Function base;  
 private final Function exponent;  
 public Power(Function base, Function exponent) {  
 this.base = base;  
 this.exponent = exponent;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return Math.*pow*(base.calculate(x), exponent.calculate(x));  
 }  
  
 @Override  
 public Function derivative() {  
 return new Product(  
 new Power(base, new Difference(exponent, new Const(1))),  
 new Sum(  
 new Product(exponent, base.derivative(), new Ln(base)),  
 new Product(exponent.derivative(), new Ln(base), base)  
 )  
 );  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return null;  
 }  
  
  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
import java.util.ArrayList;  
import java.util.StringJoiner;  
  
public class Product extends Composite implements Function{  
 public Product() {  
 super();  
 }  
  
 public Product(Function... terms) {  
 super(terms);  
 }  
  
 public Product(ArrayList<Function> terms) {  
 super(terms);  
 }  
  
 @Override  
 public double calculate(double x) {  
 double result = 1.0;  
 for (Function function : terms()) {  
 result \*= function.calculate(x);  
 }  
 return result;  
 }  
  
 @Override  
 public Function derivative() {  
 final int size = terms().size();  
 ArrayList<Function> sumTerms = new ArrayList<>(size);  
 ArrayList<Function> productTerms = new ArrayList<>(size);  
 for (int i = 0; i < size; i++) {  
 for (int j = 0; j < size; j++) {  
 Function function = terms().get(j);  
 productTerms.add(j == i ? function.derivative() : function);  
 }  
 sumTerms.add(new Product(productTerms));  
 productTerms.clear();  
 }  
 return new Sum(sumTerms);  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 final StringJoiner joiner = new StringJoiner("\*");  
 for (Function function : terms()) {  
 joiner.add(function.toPrettyString(nf));  
 }  
 return joiner.toString();  
 }  
}

public class Sign implements Function {  
 private final Function arg;  
 public Sign(Function arg) {  
 this.arg = arg;  
 }  
  
 @Override  
 public double calculate(double x) {  
 double argValue = arg.calculate(x);  
 if (argValue > 0) {  
 return 1;  
 } else if (argValue < 0) {  
 return -1;  
 } else {  
 return 0;  
 }  
 }  
  
 @Override  
 public Function derivative() {  
 return new Const(0);  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat format) {  
 return "sgn(" + arg.toPrettyString(format) + ")";  
 }  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
public class Sin implements Function {  
 private final Function arg;  
  
 public Sin(Function arg) {  
 this.arg = arg;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return Math.*sin*(arg.calculate(x));  
 }  
  
 @Override  
 public Function derivative() {  
 return new Product(new Cos(arg), arg.derivative());  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat format) {  
 return "sin(" + arg.toPrettyString(format) + ")";  
 }  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
import java.util.ArrayList;  
import java.util.StringJoiner;  
  
public class Sum extends Composite {  
 public Sum() {super();}  
 public Sum(Function... terms) {super(terms);}  
 public Sum(ArrayList<Function> terms) {super(terms);}  
 @Override  
 public double calculate(double x) {  
 double result = 0.0;  
 for (Function function : terms()) {  
 result += function.calculate(x);  
 }  
 return result;  
 }  
 @Override  
 public Function derivative() {  
 final ArrayList<Function> derivativeTerms = new ArrayList<>(terms().size());  
 for (Function function : terms()) {  
 derivativeTerms.add(function.derivative());  
 }  
 return new Sum(derivativeTerms);  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 final StringJoiner joiner = new StringJoiner("+");  
 for (Function function : terms()) {  
 joiner.add(function.toPrettyString(nf));  
 }  
 return String.*format*("(%s)", joiner.toString()).replace("+-", "-");  
 }  
 public static Sum of(Function... terms) {return new Sum(terms);}  
 }

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
public class Tan implements Function {  
 private final Function arg;  
 public Tan(Function arg) {  
 this.arg = arg;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return Math.*tan*(arg.calculate(x));  
 }  
  
 @Override  
 public Function derivative() {  
 return new Fraction(arg.derivative(), new Power(new Cos(arg), new Const(2)));  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat format) {  
 return "tan(" + arg.toPrettyString(format) + ")";  
 }  
}

package edu.mde.lab2;  
  
import java.text.NumberFormat;  
  
  
public class Variable implements Function {  
 private final String name;  
  
 public Variable(String name) {  
 this.name = name;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return x;  
 }  
  
 @Override  
 public Function derivative() {  
 return new Const(1);  
 }  
  
 @Override  
 public String toPrettyString(NumberFormat nf) {  
 return null;  
 }  
  
  
}