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

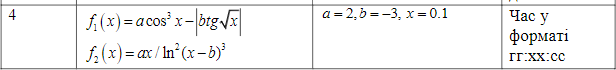
Дніпро

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:

Інтерфейс Function визначає метод calculate(double x), який повертає подвійне значення для заданого вхідного значення x, і інший метод derivative(), який повертає похідну функції як інший об’єкт Function.

Клас Variable представляє змінну в математичному виразі.

Клас Const представляє постійне значення.

Класи Sum, Difference, Product і Fraction представляють арифметичні операції між двома об’єктами Function.

Класи Ln, Log, Exp, Sin, Cos, Tan, Cot, Sec і Csc представляють різні математичні функції.

Клас Abs представляє функцію абсолютного значення.

Клас Negation представляє заперечення функції.

Клас Power представляє функцію степеня.

Клас Composite — це абстрактний клас, який надає список об’єктів Function і спосіб додавання нових об’єктів Function до списку.

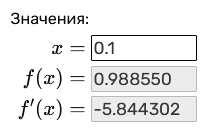
Об’єкти Function у списку можна використовувати для створення складених функцій.

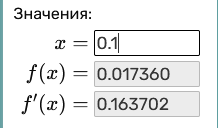
Клас Main демонструє використання різних класів шляхом створення двох об’єктів Function f1 і f2, обчислення їхніх значень і похідних і друку результатів на консолі.

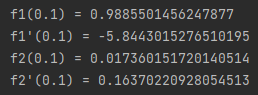
У наданому коді використовується кілька шаблонів проектування:

1. Composite pattern - клас Composite є абстрактним класом, що представляє набір функцій, які можна комбінувати для створення більш складних функцій. Інтерфейс функції та його конкретні реалізації, такі як сума, різниця, добуток і частка, реалізують цей шаблон.
2. Шаблон декоратора – класи Abs, Cos, Ln і Negation реалізують шаблон декоратора. Вони обертаються навколо іншого об’єкта Function і надають додаткові функції, такі як обчислення абсолютного значення, косинуса, натурального логарифма та заперечення виходу функції.
3. Шаблон фабричного методу – клас Variable надає фабричний метод, який створює власні екземпляри на основі заданого параметра рядка. Це дозволяє клієнтському коду створювати екземпляри об’єктів Variable без необхідності знати подробиці його реалізації.
4. Шаблон ланцюга відповідальності – метод derivative() кожної конкретної реалізації функції відповідає за обчислення похідної цієї функції. Ланцюг відповідальності формується шляхом виклику методу derivative() для підвиразів функції, що дозволяє рекурсивно обчислювати похідну всієї функції.
5. Патерн стратегії – клас Power дозволяє експоненті степеневої функції бути будь-яким об’єктом Function, а не лише постійним значенням. Це забезпечує більшу гнучкість у визначенні математичних виразів, оскільки експонента може бути будь-яким дійсним виразом, а не лише постійним значенням.

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.Мої 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:

package edu.mde.lab2;  
  
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 Product(arg.derivative(), new Sign(arg));  
 }  
}

package edu.mde.lab2;  
  
import java.util.ArrayList;  
  
  
  
public abstract class Composite {  
 private final ArrayList<Function> terms = new ArrayList<>();  
  
 public Composite() {  
 }  
  
 public Composite(Function... terms) {  
 for (Function term : terms) {  
 this.terms.add(term);  
 }  
 }  
  
 public Composite(ArrayList<Function> terms) {  
 this.terms.addAll(terms);  
 }  
  
 public ArrayList<Function> terms() {  
 return terms;  
 }  
  
 public void add(Function function) {  
 terms.add(function);  
 }  
}

public class Const implements Function {  
 private final double value;  
  
 public Const(double value) {  
 this.value = value;  
 }  
  
 @Override  
 public double calculate(double x) {  
 return value;  
 }  
  
 @Override  
 public Function derivative() {  
 return new Const(0);  
 }  
}

package edu.mde.lab2;  
  
  
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()  
 );  
 }  
}

package edu.mde.lab2;  
  
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());  
 }  
}

package edu.mde.lab2;  
  
// 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))  
 );  
 }  
  
}

package edu.mde.lab2;  
  
public interface Function {  
 double calculate(double x);  
 Function derivative();  
}

package edu.mde.lab2;  
  
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);  
 }  
}

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;  
  
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());  
 }  
}

package edu.mde.lab2;  
  
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(  
 exponent,  
 new Power(base, new Difference(exponent, new Const(1))),  
 base.derivative()  
 );  
 }  
}

package edu.mde.lab2;  
  
import java.util.ArrayList;  
  
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);  
 }  
}

package edu.mde.lab2;  
  
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);  
 }  
}

package edu.mde.lab2;  
  
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(  
 arg.derivative(),  
 new Cos(arg)  
 );  
 }  
}

package edu.mde.lab2;  
import java.util.ArrayList;  
public class Sum extends Composite implements Function {  
 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 int size = terms().size();  
 ArrayList<Function> derivatives = new ArrayList<>(size);  
 for (Function function : terms()) {  
 derivatives.add(function.derivative());  
 }  
 return new Sum(derivatives);  
 }  
}

package edu.mde.lab2;  
  
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)));  
 }  
}

package edu.mde.lab2;  
  
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);  
 }  
}