МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждения высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт компьютерных технологий и информационной безопасности

Кафедра математического обеспечения и применения ЭВМ

**ЛАБОРАТОРНАЯ РАБОТА № 5**

по дисциплине

**«Объектно-ориентированное программирование»**

на тему:

**«Язык программирования C#»**

*Вариант № 11*

Выполнил:

Студент группы

КТбо2-8

Макаров С. В.

Проверил:

Тарасов С. А.

Оценка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Таганрог 2020

# **1 ФОРМУЛИРОВКА ЗАДАНИЯ**

Необходимо создать класс Figure, который является родительским классом для других фигур. Создать виртуальные методы вычисления площади, периметра, центра.

# **2 СПЕЦИФИКАЦИЯ КЛАССОВ**

Спецификация классов находится в листинге программы.

# **3 ИСПОЛЬЗУЕМЫЕ МАТЕМАТИЧЕСКИЕ ЗАВИСИМОСТИ И АЛГОРИТМЫ**

В данной программе используются знания школьной геометрии за 5-9 классы.

# **4 ДИАГРАММА КЛАССОВ**

Uml диаграмма классов изображена на рисунке 1.

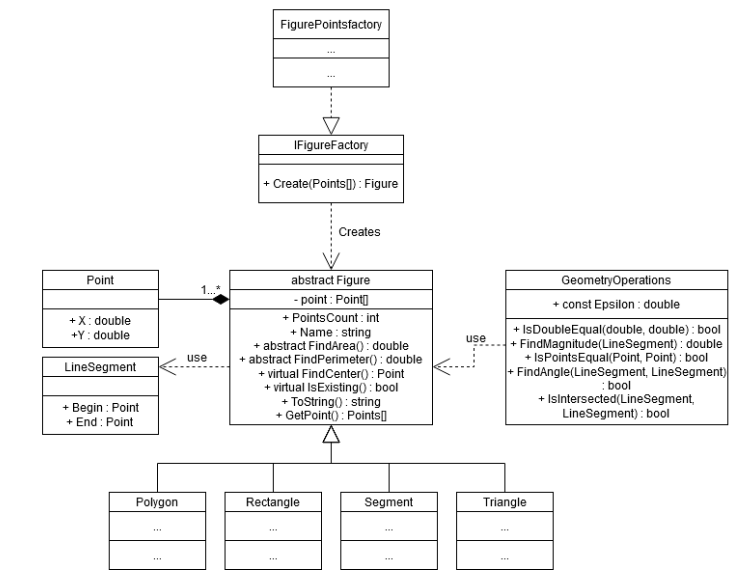


Рисунок 1 – Диаграмма классов

**ЛИСТИНГ ПРОГРАММЫ**

public abstract class Figure

{

private Point[] \_points;

public Figure(Point[] points, string name = "Figure")

{

\_points = points.Clone() as Point[];

Name = name;

}

public int PointsCount { get => \_points.Length; }

public string Name { get; set; }

abstract public double FindArea();

virtual public double FindPerimeter()

{

GeometryOperations math = new GeometryOperations();

double perimeter = 0f;

for (int i = 0; i < \_points.Length; i++)

{

perimeter += math.FindMagnitude(\_points[i], \_points[(i + 1) % \_points.Length]);

}

return perimeter;

}

virtual public Point FindCenter()

{

double x = 0, y = 0;

foreach (var point in \_points)

{

x += point.X;

y += point.Y;

}

return new Point(x / \_points.Length, y / \_points.Length);

}

public virtual bool IsExisting()

{

return \_points.Length > 2;

}

public Point[] GetPoints() => \_points;

public override string ToString()

{

return "Name: " + Name + "; Points count: " + \_points.Length.ToString();

}

public struct Point

{

public Point(double x, double y)

{

X = x;

Y = y;

}

public double X;

public double Y;

public override string ToString()

{

return string.Format("[x={0}, y={1}]", X, Y);

}

}

public struct LineSegment

{

public LineSegment(Point begin, Point end)

{

Begin = begin;

End = end;

}

public Point ConvertToVector()

{

return new Point(End.X - Begin.X, End.Y - Begin.Y);

}

public Point Begin;

public Point End;

}

public class Polygon : Figure

{

public Polygon(Point[] points) : base(points, "Polygon")

{

}

public override double FindArea()

{

double area = 0;

Point[] points = GetPoints();

int j = points.Length - 1;

for (int i = 0; i < points.Length; i++)

{

Point prev = points[j];

Point curr = points[i];

area += (prev.X + curr.X) \* (prev.Y - curr.Y);

j = i;

}

return Math.Abs(area / 2);

}

public override bool IsExisting()

{

GeometryOperations math = new GeometryOperations();

Point[] points = GetPoints();

// Checking for repeating points

for (int i = 0; i < points.Length; i++)

{

for (int j = 0; j < points.Length; j++)

{

if (i != j && math.IsPointsEqual(points[i], points[j]))

{

return false;

}

}

}

// Checking for self-intersections;

for (int i = 0; i < points.Length; i++)

{

for (int j = 0; j < points.Length; j++)

{

LineSegment a = new LineSegment(points[i], points[(i + 1) % points.Length]);

LineSegment b = new LineSegment(points[j], points[(j + 1) % points.Length]);

if (i != j && math.IsIntersected(a, b))

{

return false;

}

}

}

return true;

}

}

public class Rectangle : Figure

{

public Rectangle(Point[] points) : base(points, "Rectangle")

{

}

public override double FindArea()

{

double h, w;

FindParameters(out h, out w);

return h \* w;

}

public override bool IsExisting()

{

GeometryOperations math = new GeometryOperations();

Point[] points = GetPoints();

for (int i = 0; i < points.Length; i++)

{

double angle = math.FindAngle(

new LineSegment(points[i], points[(i + 1) % points.Length]),

new LineSegment(points[(i + 1) % points.Length], points[(i + 2) % points.Length])

);

if (math.IsDoubleEqual(angle, System.Math.PI / 2) == false)

{

return false;

}

}

return true;

}

private void FindParameters(out double height, out double width)

{

GeometryOperations math = new GeometryOperations();

Point[] pts = GetPoints();

height = math.FindMagnitude(new LineSegment(pts[0], pts[1]));

width = math.FindMagnitude(new LineSegment(pts[1], pts[2]));

}

}

public class Triangle : Figure

{

public Triangle(Point[] points) : base(points, "Triangle")

{

}

public override double FindArea()

{

GeometryOperations math = new GeometryOperations();

var lines = GetLines();

double p = FindPerimeter() / 2;

double a = math.FindMagnitude(lines[0]);

double b = math.FindMagnitude(lines[1]);

double c = math.FindMagnitude(lines[2]);

return System.Math.Sqrt(p \* (p - a) \* (p - b) \* (p - c));

}

public override bool IsExisting()

{

GeometryOperations math = new GeometryOperations();

var lines = GetLines();

double a = math.FindMagnitude(lines[0]);

double b = math.FindMagnitude(lines[1]);

double c = math.FindMagnitude(lines[2]);

return a + b > c && a + c > b && b + c > a;

}

private LineSegment[] GetLines()

{

LineSegment[] lines = new LineSegment[3];

Point[] points = GetPoints();

for (int i = 0; i < lines.Length; i++)

{

lines[i] = new LineSegment(points[i], points[(i + 1) % lines.Length]);

}

return lines;

}

}

public class GeometryOperations

{

public const double Epsilon = 1e-9;

public double FindMagnitude(Point from, Point to)

{

double x = to.X - from.X;

double y = to.Y - from.Y;

return Math.Sqrt(x \* x + y \* y);

}

public bool IsDoubleEqual(double left, double right)

{

return Math.Abs(left - right) <= Epsilon;

}

public double FindMagnitude(LineSegment line)

{

return FindMagnitude(line.Begin, line.End);

}

public bool IsPointsEqual(Point left, Point right)

{

return Math.Abs(left.X - right.X) <= Epsilon

&& Math.Abs(left.Y - right.Y) <= Epsilon;

}

public double FindAngle(LineSegment fLine, LineSegment sLine)

{

Point v1 = fLine.ConvertToVector();

Point v2 = sLine.ConvertToVector();

double scalar = v1.X \* v2.X + v1.Y \* v2.Y;

double m1 = FindMagnitude(fLine);

double m2 = FindMagnitude(sLine);

return Math.Acos(scalar / m1 \* m2);

}

public bool IsIntersected(LineSegment fLine, LineSegment sLine)

{

Point p1 = fLine.Begin;

Point p2 = fLine.End;

Point p3 = sLine.Begin;

Point p4 = sLine.End;

double v1 = (p4.X - p3.X) \* (p1.Y - p3.Y) - (p4.Y - p3.Y) \* (p1.X - p3.X);

double v2 = (p4.Y - p3.Y) \* (p2.X - p1.X) - (p4.X - p3.X) \* (p2.Y - p1.Y);

double v3 = (p2.X - p1.X) \* (p1.Y - p3.Y) - (p2.Y - p1.Y) \* (p1.X - p3.X);

double v4 = (p4.Y - p3.Y) \* (p2.X - p1.X) - (p4.X - p3.X) \* (p2.Y - p1.Y);

double u1 = v1 / v2, u2 = v3 / v4;

return u1 > 0 && u1 < 1 && u2 > 0 && u2 < 1;

}

}