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 № 3**

по дисциплине

**«Объектно-ориентированное программирование»**

на тему:

**«Динамический выбор типа объектов»**

*Вариант № 2*

Выполнил:

Студент группы

КТбо2-8

Макаров С. В.

Проверил:

Тарасов С. А.

Оценка

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

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

Таганрог 2020

# **1 ФОРМУЛИРОВКА ЗАДАНИЯ**

Необходимо создать абстрактный класс Shape для создания от него объектов-фигур. Определить методы вычисления центра, площади, поворота фигуры. Создать класс операций над фигурами, в нем определить методы определения пересечения и включения фигур.

# **2 СПЕЦИФИКАЦИЯ КЛАССОВ**

struct Point

{

double x;

double y;

Point();

Point(double \_x, double \_y);

bool operator==(Point other);

Point& operator+=(Point other);

};

struct Line

{

Point begin;

Point end;

Line();

Line(Point \_begin, Point \_end);

};

class Shape

{

public:

Shape();

Shape(const Shape& other);

Shape(const Point\* points, int length, std::string \_name);

virtual ~Shape();

Shape& operator=(const Shape& other);

virtual double FindArea() const = 0;

virtual bool IsExisting() const = 0;

virtual Point FindCenter() const;

void Move(double xOffset, double yOffset);

void Rotate(int degrees);

std::string GetName() const;

int GetVerticesCount() const;

const Point\* GetVertices() const;

Line\* FindEdges() const;

private:

std::string name;

int verticesCount;

Point\* vertices;

protected:

void \_CheckExisting();

};

class Triangle : public Shape

{

public:

Triangle() = delete;

Triangle(const Point\* points);

double FindArea() const override;

bool IsExisting() const override;

};

class Rectangle : public Shape

{

public:

Rectangle() = delete;

Rectangle(const Point\* points);

double FindArea() const override;

bool IsExisting() const override;

};

class ShapeException : public std::exception

{

public:

ShapeException(const Shape& figure);

const char\* what() const noexcept;

private:

std::string \_aboutString;

};

class ShapeFactory

{

public:

static Shape\* CreateShape(const Point\* points, int length);

};

class GeometryMath

{

public:

GeometryMath();

const double pi;

const double epsilon;

double DegreesToRadian(int degrees) const;

double FindMagnitude(const Line& line) const;

double FindAngle(const Line& first, const Line& second) const;

bool IsIntersected(const Line& lFirst, const Line& lSecond) const;

bool IsIntersected(const Shape& lFirst, const Shape& lSecond) const;

// Not-strict inclusion

bool IsIncluded(Point p, const Shape& figure);

bool IsIncluded(const Shape& dest, const Shape& incl);

private:

bool \_IsIncludedDontValidate(Point p, const Shape& figure);

void \_ValidateShape(const Shape& shape) const;

};

class ConsoleInteractor

{

public:

ConsoleInteractor();

~ConsoleInteractor();

void Run();

private:

bool \_Init();

void \_PrintCommands() const;

template<class T>

void \_ReadVar(T& var) const;

bool \_ReadIndex(int& \_index);

int totalCount;

int figuresCount;

Shape\*\* figures;

};

# **3 ИСПОЛЬЗУЕМЫЕ МАТЕМАТИЧЕСКИЕ ЗАВИСИМОСТИ И АЛГОРИТМЫ**

В данной работе использовались следующие зависимости:

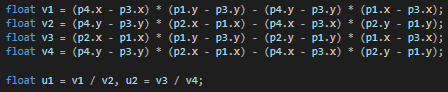
Вычисление длины вектора по формуле:

Вычисление угла между двумя векторами по формуле:

Принадлежность точки фигуре вычисляется по алгоритму суммы углов: из данной точки проводятся отрезки во все вершины фигуры, сумма углов между всеми соседними отрезками должна быть равна 2\*Pi.

Включение одной фигуры в другой определяется с помощью последовательной проверки всех точек на принадлежность.

Пересечение двух отрезков определяется по формуле, показанной на рисунке 1, которая была выведена из уравнения прямых.



Где, p1 и p2 – точки первого вектора, остальные – второго. Если коэффициенты u1 и u2 лежат на промежутке от 0 до 1 включительно, то отрезки пересекаются.

Для проверки, пересекаются ли фигуры, выполняется проверка на пересечения всех ребер со всеми возможными.

Для поворота фигур использовалась матрица поворота, которая поворачивает точку на заданный угол в радианах.

# **4 ДИАГРАММА КЛАССОВ**

Uml диаграмма классов изображена на рисунке 1.

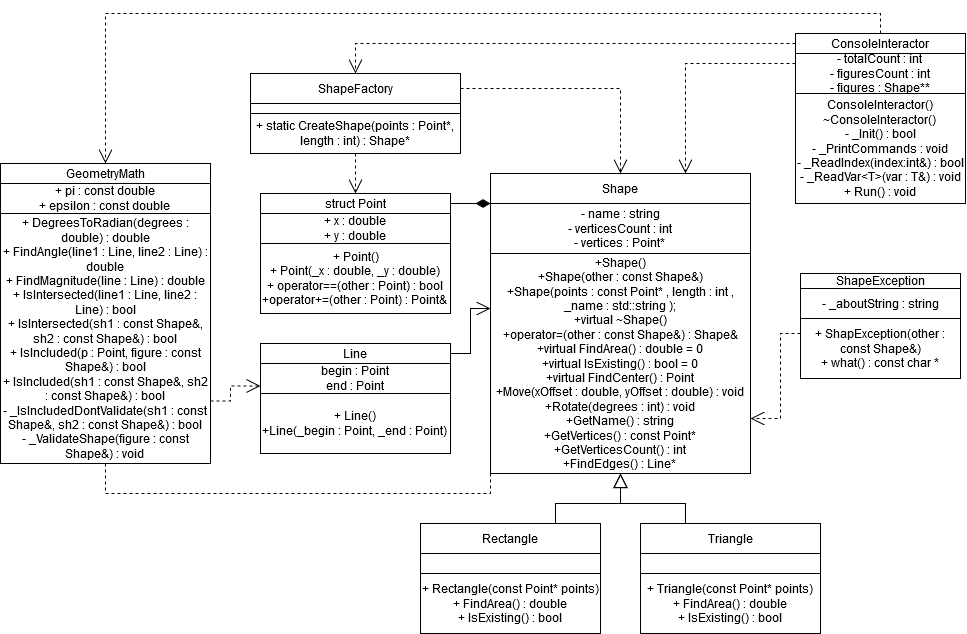


Рисунок 1 – Диаграмма классов

**ЛИСТИНГ ПРОГРАММЫ**

Point::Point()

: x(0), y(0)

{}

Point::Point(double \_x, double \_y)

: x(\_x), y(\_y)

{}

bool Point::operator==(Point other)

{

if (x == other.x && y == other.y)

{

return true;

}

return false;

}

Point& Point::operator+=(Point other)

{

x += other.x;

y += other.y;

return \*this;

}

Line::Line()

: begin(), end()

{}

Line::Line(Point \_begin, Point \_end)

: begin(\_begin), end(\_end)

{}

Shape::Shape()

: vertices(nullptr),

verticesCount(0),

name("Abstract shape")

{

}

Shape::Shape(const Shape& other)

: Shape(other.vertices, other.verticesCount, other.name)

{}

Shape::Shape(const Point\* points, int length, std::string \_name)

: verticesCount(length),

vertices(new Point[verticesCount]),

name(\_name)

{

std::copy(points, points + verticesCount, vertices);

}

Shape::~Shape()

{

if (vertices != nullptr)

{

delete[] vertices;

}

}

Shape& Shape::operator=(const Shape& other)

{

if (this == &other)

{

return \*this;

}

if (vertices != nullptr)

{

delete[] vertices;

}

verticesCount = other.verticesCount;

vertices = new Point[verticesCount];

std::copy(other.vertices, other.vertices + verticesCount, vertices);

return \*this;

}

Point Shape::FindCenter() const

{

double \_x = 0, \_y = 0;

for (int i = 0; i < verticesCount; i++)

{

\_x += vertices[i].x;

\_y += vertices[i].y;

}

return Point(\_x / verticesCount, \_y / verticesCount);

}

void Shape::Move(double xOffset, double yOffset)

{

Point \_offset(xOffset, yOffset);

for (int i = 0; i < verticesCount; i++)

{

vertices[i] += \_offset;

}

}

void Shape::Rotate(int degrees)

{

GeometryMath math;

double \_angle = math.DegreesToRadian(degrees);

Point \_old;

for (int i = 0; i < verticesCount; i++)

{

\_old = vertices[i];

vertices[i] = Point(

\_old.x \* cos(\_angle) - \_old.y \* sin(\_angle),

\_old.y \* sin(\_angle) + \_old.y \* cos(\_angle)

);

}

}

std::string Shape::GetName() const

{

return name;

}

int Shape::GetVerticesCount() const

{

return verticesCount;

}

const Point\* Shape::GetVertices() const

{

return vertices;

}

Line\* Shape::FindEdges() const

{

int \_count = GetVerticesCount();

Line\* \_lines = new Line[\_count];

for (int i = 0; i < \_count - 1; i++)

{

\_lines[i] = Line(vertices[i], vertices[i + 1]);

}

\_lines[\_count - 1] = Line(vertices[\_count - 1], vertices[0]);

return \_lines;

}

void Shape::\_CheckExisting()

{

if (!IsExisting())

{

throw ShapeException(\*this);

}

}

Triangle::Triangle(const Point\* points)

: Shape(points, 3, "Triangle")

{}

double Triangle::FindArea() const

{

GeometryMath math;

Line\* \_lines = FindEdges();

double A = math.FindMagnitude(\_lines[0]);

double B = math.FindMagnitude(\_lines[1]);

double C = math.FindMagnitude(\_lines[2]);

double p = (A + B + C) / 2;

delete[] \_lines;

return sqrt(p \* (p - A) \* (p - B) \* (p - C));

}

bool Triangle::IsExisting() const

{

GeometryMath math;

Line\* \_lines = FindEdges();

double A = math.FindMagnitude(\_lines[0]);

double B = math.FindMagnitude(\_lines[1]);

double C = math.FindMagnitude(\_lines[2]);

delete[] \_lines;

if (A + B > C && A + C > B && B + C > A)

{

return true;

}

return false;

}

Rectangle::Rectangle(const Point\* points)

: Shape(points, 4, "Rectangle")

{}

double Rectangle::FindArea() const

{

GeometryMath math;

const Point\* \_vertices = GetVertices();

double w = math.FindMagnitude({ \_vertices[0], \_vertices[1] });

double h = math.FindMagnitude({ \_vertices[1], \_vertices[2] });

return h \* w;

}

bool Rectangle::IsExisting() const

{

GeometryMath math;

const Point\* \_vertices = GetVertices();

int vCount = 4;

int eCount = vCount + 1;

const Line\* \_originLines = FindEdges();

Line\* \_lines = new Line[eCount];

// Getting all edges in Rectangle + copy of first

std::copy(\_originLines, \_originLines + vCount, \_lines);

\_lines[eCount - 1] = \_lines[0];

bool \_ans = true;

// for all angles in Rectangle

for (int i = 0; i < eCount - 1; i++)

{

double \_angle = math.FindAngle(\_lines[i], \_lines[i + 1]);

if (abs(\_angle - math.pi / 2) > DBL\_EPSILON) // if angle not equals 90 degrees

{

\_ans = false;

break;

}

}

Point p1, p2;

for (int i = 0; i < vCount - 1; i++)

{

p1 = \_vertices[i];

p2 = \_vertices[i + 1];

if (p1 == p2)

{

\_ans = false;

break;

}

}

delete[] \_originLines;

delete[] \_lines;

return \_ans;

}

ShapeException::ShapeException(const Shape& figure)

{

std::ostringstream sstr;

sstr << "Shape [" << figure.GetName() << "] with " << figure.GetVerticesCount() << " can't be used";

\_aboutString = sstr.str();

}

const char\* ShapeException::what() const noexcept

{

return \_aboutString.c\_str();

}

Shape\* ShapeFactory::CreateShape(const Point\* points, int length)

{

Shape\* res = nullptr;

if (length == 3)

{

Triangle t(points);

if (t.IsExisting())

{

res = new Triangle(points);

}

}

else if (length == 4)

{

Rectangle r(points);

if (r.IsExisting())

{

res = new Rectangle(points);

}

}

return res;

}

ConsoleInteractor::ConsoleInteractor()

: totalCount(0),

figuresCount(0),

figures(nullptr)

{}

ConsoleInteractor::~ConsoleInteractor()

{

for (int i = 0; i < figuresCount; i++)

{

if (figures[i] != nullptr)

{

delete figures[i];

}

}

delete[] figures;

}

void ConsoleInteractor::Run()

{

if (!\_Init())

{

return;

}

int \_switch;

\_PrintCommands();

while (true)

{

try

{

cout << "\n enter command: ";

\_ReadVar(\_switch);

switch (\_switch)

{

case 0:

return;

case 1:

{

if (totalCount == figuresCount)

{

cout << "failed. array of figures is full\n";

break;

}

int \_ptsAmount, \_index;

cout << "enter index (starts from 0) of new figure: ";

\_ReadVar(\_index);

if (\_index < 0 || \_index >= figuresCount)

{

cout << "failed. invalid index\n";

break;

}

cout << "enter amount of points: ";

\_ReadVar(\_ptsAmount);

auto \_points = make\_unique<Point[]>(\_ptsAmount);

if (\_ptsAmount <= 0)

{

cout << "invalid amount number\n";

break;

}

double x, y;

for (int i = 0; i < \_ptsAmount; i++)

{

cout << "point " << i + 1 << ": ";

\_ReadVar(x);

\_ReadVar(y);

\_points[i] = Point(x, y);

}

figures[\_index] = ShapeFactory::CreateShape(\_points.get(), \_ptsAmount);

if (figures[\_index] == nullptr)

{

cout << "failed. factory can't create figure from enteret points\n";

break;

}

totalCount++;

cout << figures[\_index]->GetName() << " was created\n";

}

break;

case 2:

{

if (totalCount == 0)

{

cout << "already empty\n";

}

int \_index;

if (!\_ReadIndex(\_index))

{

break;

}

delete figures[\_index];

figures[\_index] = nullptr;

totalCount--;

cout << "success\n";

}

break;

case 3:

{

int \_index;

if (!\_ReadIndex(\_index))

{

break;

}

cout << "Shape : " << figures[\_index]->GetName() << "\n";

const Point\* \_vertices = figures[\_index]->GetVertices();

for (int i = 0; i < figures[\_index]->GetVerticesCount(); i++)

{

Point p = \_vertices[i];

cout << "Point " << i + 1 << ": " << "x - " << p.x << ", y - " << p.y << "\n";

}

}

break;

case 4:

{

int \_index;

if (!\_ReadIndex(\_index))

{

break;

}

Point c = figures[\_index]->FindCenter();

cout << "Center: x - " << c.x << ", y - " << c.y << "\n";

}

break;

case 5:

{

int \_index;

if (!\_ReadIndex(\_index))

{

break;

}

double \_area = figures[\_index]->FindArea();

cout << "area = " << \_area << "\n";

}

break;

case 6:

{

int \_index;

if (!\_ReadIndex(\_index))

{

break;

}

int \_degrees;

\_ReadVar(\_degrees);

figures[\_index]->Rotate(\_degrees);

cout << "success\n";

}

break;

case 7:

{

cout << "index of first figure: ";

int \_firstInd;

if (!\_ReadIndex(\_firstInd))

{

break;

}

cout << "index of second figure: ";

int \_secondInd;

if (!\_ReadIndex(\_secondInd))

{

break;

}

GeometryMath math;

bool ans = math.IsIntersected(\*figures[\_firstInd], \*figures[\_secondInd]);

cout << "is figures intersected? " << (ans ? "yes" : "no") << "\n";

}

break;

case 8:

{

cout << "index of first figure: ";

int \_firstInd;

if (!\_ReadIndex(\_firstInd))

{

break;

}

cout << "index of second figure: ";

int \_secondInd;

if (!\_ReadIndex(\_secondInd))

{

break;

}

GeometryMath math;

bool ans = math.IsIncluded(\*figures[\_firstInd], \*figures[\_secondInd]);

cout << "is first figure includes second? " << (ans ? "yes" : "no") << "\n";

}

break;

case 10:

\_PrintCommands();

break;

default:

cout << "invalid command\n";

break;

}

}

catch (const std::exception & e)

{

cout << e.what() << "\n";

}

}

}

bool ConsoleInteractor::\_Init()

{

int \_count;

cout << "Enter a maximum amount of figures or enter 0 to quit\n";

while (true)

{

try

{

\_ReadVar(\_count);

if (\_count == 0)

{

return false;

}

else if (\_count < 0)

{

cout << "Invalid count\n";

}

else

{

cout << "Success\n";

figuresCount = \_count;

figures = new Shape \* [figuresCount];

std::fill(figures, figures + figuresCount, nullptr);

return true;

}

}

catch (const std::exception & e)

{

cout << e.what() << "\n";

}

}

}

void ConsoleInteractor::\_PrintCommands() const

{

system("cls");

cout << "Commands: \n";

cout << " 1: create shape, 2: delete shape, 3: print shape\n";

cout << " 4: find center, 5: find area, 6 rotate shape\n";

cout << " 7: is figures intersected, 8: is figures included\n";

cout << " 9: move shape, 10: clear the console, 0: exit\n\n";

cout << " Maximum amount of figures = " << figuresCount << "\n";

}

bool ConsoleInteractor::\_ReadIndex(int& \_index)

{

cout << "enter index (starts from 0) of figure: ";

\_ReadVar(\_index);

if (\_index < 0 || \_index >= figuresCount)

{

cout << "failed. invalid index\n";

return false;

}

if (figures[\_index] == nullptr)

{

cout << "failed. index is empty\n";

return false;

}

return true;

}

template void ConsoleInteractor::\_ReadVar(int& var) const;

template void ConsoleInteractor::\_ReadVar(double& var) const;

template<class T>

void ConsoleInteractor::\_ReadVar(T& var) const

{

string input;

cin >> input;

istringstream sin(input);

char c;

if (!(sin >> var) || (sin >> c))

{

throw std::runtime\_error("Input error. Failed to enter a variable");

}

}

GeometryMath::GeometryMath()

: pi(2 \* acos(0)),

epsilon(DBL\_EPSILON)

{}

double GeometryMath::DegreesToRadian(int degrees) const

{

return (degrees % 360) \* pi / 180;

}

double GeometryMath::FindMagnitude(const Line& line) const

{

double \_x = line.end.x - line.begin.x;

double \_y = line.end.y - line.begin.y;

return sqrt(\_x \* \_x + \_y \* \_y);

}

double GeometryMath::FindAngle(const Line& first, const Line& second) const

{

Point \_v1(first.end.x - first.begin.x, first.end.y - first.begin.y);

Point \_v2(second.end.x - second.begin.x, second.end.y - second.begin.y);

double \_scalar = \_v1.x \* \_v2.x + \_v1.y \* \_v2.y;

double \_v1Magnitude = FindMagnitude(first);

double \_v2Magnitude = FindMagnitude(second);

return acos(\_scalar / (\_v1Magnitude \* \_v2Magnitude));

}

bool GeometryMath::IsIntersected(const Line& lFirst, const Line& lSecond) const

{

Point p1 = lFirst.begin;

Point p2 = lFirst.end;

Point p3 = lSecond.begin;

Point p4 = lSecond.end;

if (p1 == p3 || p2 == p4)

{

return true;

}

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;

}

bool GeometryMath::IsIntersected(const Shape& fFirst, const Shape& fSecond) const

{

\_ValidateShape(fFirst);

\_ValidateShape(fSecond);

Line\* \_firstEdgs = fFirst.FindEdges();

Line\* \_secondEdgs = fSecond.FindEdges();

int \_firstCnt = fFirst.GetVerticesCount();

int \_secondCnt = fSecond.GetVerticesCount();

bool ans = false;

Line l1, l2;

for (int i = 0; i < \_firstCnt && !ans; i++)

{

l1 = \_firstEdgs[i];

for (int j = 0; j < \_secondCnt; j++)

{

l2 = \_secondEdgs[j];

if (IsIntersected(l1, l2))

{

ans = true;

break;

}

}

}

delete[] \_firstEdgs;

delete[] \_secondEdgs;

return ans;

}

bool GeometryMath::IsIncluded(Point p, const Shape& figure)

{

\_ValidateShape(figure);

return \_IsIncludedDontValidate(p, figure);

}

bool GeometryMath::IsIncluded(const Shape& dest, const Shape& incl)

{

\_ValidateShape(dest);

\_ValidateShape(incl);

bool ans = true;

const Point\* \_vertices = incl.GetVertices();

for (int i = 0; i < incl.GetVerticesCount() && !ans; i++)

{

if (\_IsIncludedDontValidate(\_vertices[i], dest) == false)

{

ans = false;

break;

}

}

return ans;

}

bool GeometryMath::\_IsIncludedDontValidate(Point p, const Shape& figure)

{

const Point\* \_vertices = figure.GetVertices();

for (int i = 0; i < figure.GetVerticesCount(); i++)

{

Point figureP = \_vertices[i];

if (figureP == p)

{

return true;

}

}

int \_checkingCount = figure.GetVerticesCount() + 1;

Line\* \_checkingLines = new Line[\_checkingCount];

for (int i = 0; i < \_checkingCount - 1; i++)

{

\_checkingLines[i] = Line(p, \_vertices[i]);

}

\_checkingLines[\_checkingCount - 1] = \_checkingLines[0];

double angle = 0;

for (int i = 0; i < \_checkingCount - 1; i++)

{

angle += FindAngle(\_checkingLines[i], \_checkingLines[i + 1]);

}

bool ans = false;

if (abs(angle - 2 \* pi) <= epsilon)

{

ans = true;

}

delete[] \_checkingLines;

return ans;

}

void GeometryMath::\_ValidateShape(const Shape& shape) const

{

if (!shape.IsExisting() || shape.GetVerticesCount() == 0)

{

throw ShapeException(shape);

}

}