**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: «Наследование»**

Студент гр. 7381 Лукашев Р.С.

Преподаватель Жангиров Т.Р.

Санкт-Петербург

2019

**Цель работы.**

Ознакомиться с понятиями наследование, полиморфизм, виртуальная функция, абстрактный класс, иерархия наследования. Спроектировать систему классов для моделирования геометрических фигур в соответствии с полученным индивидуальным заданием.

**Задание.**

Необходимо спроектировать систему классов для моделирования геометрических фигур. Задание предполагает использование виртуальных функций в иерархии наследования, проектирование и использование абстрактного базового класса.  Разработанные классы должны быть наследниками абстрактного класса Shape, содержащего методы для перемещения в указанные координаты, поворота на заданный угол, масштабирования на заданный коэффициент, установки и получения цвета, а также оператор вывода в поток.

Необходимо также обеспечить однозначную идентификацию каждого объекта.

Решение должно содержать:

* условие задания;
* UML диаграмму разработанных классов;
* текстовое обоснование проектных решений;
* реализацию классов на языке С++.

﻿﻿ **Индивидуализация.**

Реализовать систему классов для фигур: сектор круга, параллелограмм, эллипс.

**Обоснование проектных решений.**

Для представления цвета была создана отдельная структура данных Color, в которой были определены конструктор, конструктор копирования и оператор вывода в поток. Цвет представляется в формате RGB.

Базовым абстрактным классом Shape определяются координата, цвет, и угол поворота фигуры. Для однозначной идентификации объектов было решено присвоить каждому новому объекту свой id, при этом для того, чтобы уследить за индивидуальным подбором id в классе Shape объявлена статическая переменная \_id\_counter, увеличивающаяся при каждом новом создании объекта класса Shape или дочерних. В соответствии с заданием в данном классе также определяются методы для перемещения фигуры в указанные координаты, поворота фигуры на заданный угол, установки и получения её цвета и оператор вывода в поток. Чисто виртуальные методы – public Scale и protected printShape. Масштабирование фигуры было решено реализовать через масштабирование ее линейных компонент, которые определяются в дочернем классе, поэтому Scale виртуальная. А поскольку в задании требуется определить оператор вывода в поток в родительском классе, было решено воспользоваться косвенной виртуализацией с использованием виртуальной функции printShape, определяемой в дочерних классах.

В классе сектора круга Sector, наследника Shape, определены конструктор, конструктор копирования, и копирующий оператор присваивания для удобной работы с объектами класса. Сектор решено было описать с помощью двух параметров – радиуса круга и угла, который охватывает сектор. Метод масштабирования фигуры в таком случае определяется масштабированием радиуса круга. Также определен метод printShape для сектора.

Аналогично организованы классы для параллелограмма и эллипса. Причем параллелограмм определяется длиной двух его сторон (линейные параметры) и углом между ними (слева-направо и снизу-вверх). А эллипс определяется двумя параметрами – длинами главной и дополнительной осей (оба линейные).

**UML-диаграмма классов.**

UML-диаграмма представлена в приложении А.

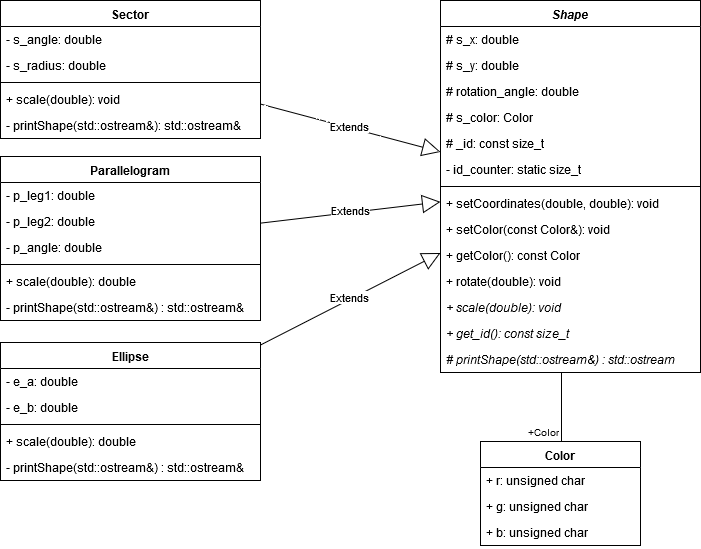
**Реализация классов на языке С++.**

Текст с реализованными классами представлен в приложении Б.

**Выводы.**

В процессе выполнения лабораторной работы была спроектирована система классов для моделирования геометрических фигур. Организована иерархия классов с одним абстрактным классом, описывающим параметры и методы присущие всем фигурам в общем, и на его основе описано три класса геометрических фигур – сектор круга, параллелограмм, и эллипс.

**Приложение А. UML-диаграмма классов.**



**Приложение Б. Реализация классов на языке С++.**

typedef struct Color {

unsigned char r;

unsigned char g;

unsigned char b;

Color(unsigned char red = 0, unsigned char green = 0, unsigned char blue = 0) :

r(red),

g(green),

b(blue)

{

}

Color(const Color& other) :

r(other.r),

b(other.b),

g(other.g)

{

}

friend inline std::ostream& operator<<(std::ostream& stream, const Color& color) {

stream << '(' << (int)color.r << ',' << (int)color.g << ',' << (int)color.b << ')';

return stream;

}

} Color;

class Shape abstract{

public:

void setCoordinates(double newX, double newY) {

s\_x = newX;

s\_y = newY;

}

Shape(double xCoordinate = 0.0, double yCoordinate = 0.0, double r\_angle = 0.0, const Color& color = Color()) :

s\_x(xCoordinate),

s\_y(yCoordinate),

s\_color(color),

rotation\_angle(r\_angle),

\_id (\_id\_counter++)

{

}

~Shape() = default;

void setColor(const Color& newColor)

{

s\_color = newColor;

}

const Color getColor() const noexcept{

return s\_color;

}

void rotate(double angle) {

rotation\_angle += angle;

}

virtual void scale(double) = 0;

friend inline std::ostream& operator<<(std::ostream& stream, Shape& shape) {

return shape.printShape(stream);

}

const size\_t get\_id() const noexcept{

return \_id;

}

protected:

virtual std::ostream& printShape(std::ostream&) = 0;

const size\_t \_id;

double s\_x;

double s\_y;

double rotation\_angle;

Color s\_color;

private:

static size\_t \_id\_counter;

};

size\_t Shape::\_id\_counter = 0;

class Sector : public Shape {

public:

Sector(double x = 0, double y = 0, double r\_angle = 0.0, Color color = Color(), double radius = 1.0, double angle = 360.0) :

Shape(x, y, r\_angle, color),

s\_radius(radius),

s\_angle(angle)

{

}

Sector(const Sector& other):

Shape(other.s\_x, other.s\_y, other.rotation\_angle, other.s\_color),

s\_angle(other.s\_angle),

s\_radius(other.s\_radius)

{

}

Sector& operator=(Sector& other) {

s\_angle = other.s\_angle;

s\_color = other.s\_color;

rotation\_angle = other.rotation\_angle;

s\_x = other.s\_x;

s\_y = other.s\_y;

return \*this;

}

void scale(double coefficient) override {

s\_radius \*= coefficient;

}

~Sector() = default;

private:

std::ostream& printShape(std::ostream& stream = std::cout) override{

stream << "Sector of " << s\_angle << " degrees of a circle with radius " << s\_radius << " on position (" << this->s\_x << ',' << this->s\_y << ")" << " rotated by " << this->rotation\_angle << " degrees" << ". Shape's color is " << this->s\_color << " id - " << get\_id() << std::endl;

return stream;

}

double s\_angle;

double s\_radius;

};

class Parallelogram : public Shape {

public:

Parallelogram(double x = 0, double y = 0, double r\_angle = 0.0, Color color = Color(), double leg1 = 1.0, double leg2 = 1.0, double angle = 90.0) :

Shape(x, y, r\_angle, color),

p\_leg1(leg1),

p\_leg2(leg2),

p\_angle(angle)

{

}

Parallelogram(const Parallelogram& other) :

Shape(other.s\_x, other.s\_y, other.rotation\_angle, other.s\_color),

p\_leg1(other.p\_leg1),

p\_leg2(other.p\_leg2),

p\_angle(other.p\_angle)

{

}

Parallelogram& operator=(Parallelogram& other) {

p\_leg1 = other.p\_leg1;

p\_leg2 = other.p\_leg2;

p\_angle = other.p\_angle;

s\_color = other.s\_color;

rotation\_angle = other.rotation\_angle;

s\_x = other.s\_x;

s\_y = other.s\_y;

return \*this;

}

void scale(double coefficient) override {

p\_leg1 \*= coefficient;

p\_leg2 \*= coefficient;

}

~Parallelogram() = default;

private:

std::ostream& printShape(std::ostream& stream = std::cout) override {

stream << "Parallelogram with legs " << p\_leg1 << " and " << p\_leg2 << " with base angle of " << p\_angle << " degrees on position (" << this->s\_x << "," << this->s\_y << ")" << " rotated by " << this->rotation\_angle << " degrees" << ". Shape's color is " << this->s\_color << " id - " << get\_id() << std::endl;

return stream;

}

double p\_leg1;

double p\_leg2;

double p\_angle;

};

class Ellipse : public Shape {

public:

Ellipse(double x = 0, double y = 0, double r\_angle = 0.0, const Color& color = Color(), double a = 1.0, double b = 1.0) :

Shape(x, y, r\_angle, color),

e\_a(a),

e\_b(b)

{

}

Ellipse(const Ellipse& other) :

Shape(other.s\_x, other.s\_y, other.rotation\_angle, other.s\_color),

e\_a(other.e\_a),

e\_b(other.e\_b)

{

rotation\_angle = other.rotation\_angle;

}

Ellipse& operator=(Ellipse& other) {

e\_a = other.e\_a;

e\_b = other.e\_b;

rotation\_angle = other.rotation\_angle;

s\_x = other.s\_x;

s\_y = other.s\_y;

return \*this;

}

void scale(double coefficient) override {

e\_a \*= coefficient;

e\_b \*= coefficient;

}

~Ellipse() = default;

private:

std::ostream& printShape(std::ostream& stream = std::cout) override {

stream << "Ellipse with parameters " << e\_a << " and " << e\_b << " on position (" << this->s\_x << "," << this->s\_y << ")" << " rotated by " << this->rotation\_angle << " degrees" << ". Shape's color is " << this->s\_color << " id - " << get\_id() << std::endl;

return stream;

}

double e\_a;

double e\_b;

};