# 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.**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 |  | Давкаева В.С. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург 2019

**Цель работы:**

Необходимо спроектировать систему классов для моделирования геометрических фигур квадрата, параллелограмма, ромба. 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++.

**Ход работы:**

В процессе выполнения работы были созданы следующие объекты:

1. Структура Point , содержащая два поля, которые описывают координаты х и у точки.
2. Структура RGB, содержащая три поля, которые характеризуют цвет фигуры.
3. Абстрактный класс Shape содержит следующие поля:

**RGB m\_color** – цвет фигуры

**unsigned int m\_ID** – номер ID фигуры

**Point m\_center** – координата центра фигуры

**vector <Point> coordinates** – вектор, хранящий координаты вершин фигур

Класс Shape содержит следующие методы:

**void set\_color(const RGB& color)** для установления заданного цвета фигуры.

**RGB& get\_color()** для получения информации об установленном

цвете фигуры.

**unsigned int get\_ID()** для получения информации об id фигуры.

**Point get\_center()** для получения координат середины фигуры

**void move(Point a)** для смещения фигуры в заданную точку. 3.5. void **rotate(double angle)** для поворота фигуры на заданный угол.

**virtual void scaling(double scale) = 0** – чисто виртуальный метод для масштабирования фигуры на заданный коэффициент.

**virtual std::ostream& print\_shape(std::ostream& stream, Shape& shape) = 0** – чисто виртуальный метод для вывода информации о фигуре на экран.

**friend std::ostream& operator << (std::ostream& stream, Shape& shape)** для переопределения оператора вывода на экран.

4. Класс Square, который наследуется от абстрактного класса Shape. Класс имеет поле, которое характеризует длину стороны квадрата. В конструкторе данного класса вычисляются все вершины квадрата, имея информацию о координатах центра квадрата, длине его стороны и угла поворота. В классе был переопределен метод scaling, который масштабирует квадрат на заданный коэффициент. А так же метод print\_shape, который выводит информацию о фигуре.

1. Класс Parallelogram, который наследуется от абстрактного класса Shape. Класс Parallelogram имеет три дополнительных поля, которые характеризуют вершины параллелограмма. В конструкторе данного класса вычисляется последняя вершина фигуры, а так же координаты ее центра. В классе был переопределен метод scaling, который масштабирует параллелограмм на заданный коэффициент. А так же метод print\_shape, который выводит информацию о фигуре.
2. Класс Rhombus который наследуется от абстрактного класса Shape. Класс Rhombus имеет два дополнительных поля, содержащих информацию о длинах диагоналей ромба. В конструкторе данного класса вычисляются координаты вершин ромба, имея информацию о координатах одной из вершин и длинах диагоналей. В классе был переопределен метод scaling, который масштабирует ромб на заданный коэффициент. А так же метод print\_shape, который выводит информацию о фигуре.

**Обоснование решения:**

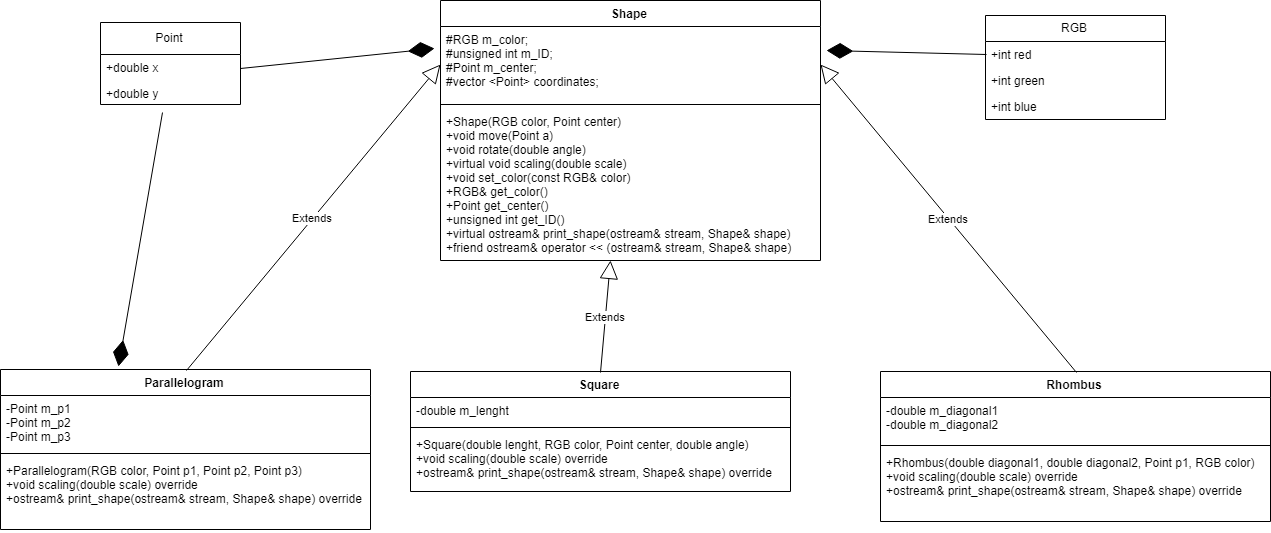
В данной лабораторной работе был реализован абстрактный класс Shape.

Поля цвет, координаты центра фигуры и координаты вершин фигуры являются общими, поэтому они содержатся в абстрактном классе Shape. Для реализации квадрата необходима информация о вершинах квадрата. Для получения данной информации достаточно знать центр квадрата, угол и длину его стороны. Для реализации параллелограмма необходима информация о вершинах параллелограмма. Для получения данной информации достаточно знать информацию о трех точках. Для реализации ромба необходима информация о вершинах ромба. Для получения данной информации достаточно знать одну из точек ромба и длины двух диагоналей ромба.

Move - это перемещение в заданную точку. Для любой фигуры можно найти расстояние между новой точкой и текущим центром фигуры. Перемещение фигуры – это смещение каждой из вершин этой фигура на полученное расстояние.

Rotate – это поворот на заданный угол. Для каждой фигуры поворот на заданный угол можно получить, умножив все координаты фигуры на матрицу поворота.

**UML диаграмма классов:**

****

**Заключение :**

В ходе выполнения данной лабораторной работы была изучена тема наследование. Была спроектирована система классов для моделирования геометрических фигур квадрат, параллелограмм, ромб. Были использованы виртуальные функции в иерархии наследования. Были разработаны классы, которые являются наследниками абстрактного класса Shape.

**Приложение А**

**Исходный код программы**

#include "pch.h"

#include <iostream>

#include <cmath>

#include <vector>

#define PI 3.14

struct RGB

{

int red;

int green;

int blue;

};

struct Point

{

double x;

double y;

};

class Shape

{

protected:

RGB m\_color;

unsigned int m\_ID;

Point m\_center;

std::vector <Point> coordinates;

public:

Shape(RGB color, Point center) :m\_color(color), m\_center(center)

{

static int ID = 0;

m\_ID = ID;

ID++;

}

//перемещение

void move(Point a)

{

double moving\_x = a.x - m\_center.x;

double moving\_y = a.y - m\_center.y;

m\_center = a;

for (size\_t i = 0; i < coordinates.size(); i++)

{

coordinates[i].x += moving\_x;

coordinates[i].y += moving\_y;

}

}

//поворот

void rotate(double angle)

{

//перевод в радианы

double angle\_rad = angle \* PI / 180.0;

for (size\_t i = 0; i < coordinates.size(); i++)

{

coordinates[i].x = m\_center.x + (coordinates[i].x - m\_center.x)\*cos(angle\_rad) - (coordinates[i].y - m\_center.y)\*sin(angle\_rad);

coordinates[i].y = m\_center.y + (coordinates[i].x - m\_center.x)\*sin(angle\_rad) + (coordinates[i].y - m\_center.y)\*cos(angle\_rad);

}

}

//масштабирование

virtual void scaling(double scale) = 0;

//установка цвета

void set\_color(const RGB& color)

{

m\_color = color;

}

//получение цвета

RGB& get\_color()

{

return m\_color;

}

Point get\_center()

{

return m\_center;

}

unsigned int get\_ID()

{

return m\_ID;

}

virtual std::ostream& print\_shape(std::ostream& stream, Shape& shape) = 0;

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

{

return shape.print\_shape(stream, shape);

}

};

class Parallelogram :public Shape {

Point m\_p1;

Point m\_p2;

Point m\_p3;

public:

Parallelogram(RGB color, Point p1, Point p2, Point p3) :Shape(color, m\_center) {

m\_p1 = p1;

m\_p2 = p2;

m\_p3 = p3;

coordinates.push\_back(p1);

coordinates.push\_back(p2);

coordinates.push\_back(p3);

coordinates.push\_back(Point{ coordinates[0].x + (coordinates[2].x - coordinates[1].x), coordinates[2].y - (coordinates[1].y - coordinates[0].y) });

m\_center.x = (coordinates[0].x + coordinates[2].x) / 2;

m\_center.y = (coordinates[0].y + coordinates[2].y) / 2;

}

void scaling(double scale) override {

for (size\_t i = 0; i < coordinates.size(); i++)

{

coordinates[i].x= m\_center.x + (coordinates[i].x - m\_center.x)\*scale;

coordinates[i].y = m\_center.y + (coordinates[i].y - m\_center.y)\*scale;

}

}

std::ostream& print\_shape(std::ostream& stream, Shape& shape) override {

stream << "Figure: Parallelogram" << std::endl;

stream << "ID: " << shape.get\_ID() << std::endl;

stream << "Coordinates: " << std::endl;

stream << "Center: (" << shape.get\_center().x << ";" << shape.get\_center().y << ")" << std::endl;

stream << "Vertexes:" << std::endl;

for (size\_t i = 0; i < coordinates.size(); i++)

{

stream << "(" << coordinates[i].x << ";" << coordinates[i].y << ")" << std::endl;

}

stream << "Color: " << shape.get\_color().red << " " << shape.get\_color().green << " " << shape.get\_color().blue << std::endl;

stream << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << std::endl;

return stream;

}

};

class Square : public Shape

{

double m\_lenght;

public:

Square(double lenght, RGB color, Point center, double angle) :Shape(color, center)

{

m\_lenght = lenght;

coordinates.push\_back(Point{center.x-lenght/2,center.y-lenght/2});

coordinates.push\_back(Point{ center.x - lenght / 2,center.y + lenght / 2 });

coordinates.push\_back(Point{ center.x + lenght / 2,center.y + lenght / 2 });

coordinates.push\_back(Point{ center.x + lenght / 2,center.y - lenght / 2 });

rotate(angle);

}

void scaling(double scale) override

{

for (size\_t i = 0; i < coordinates.size(); i++)

{

coordinates[i].x = m\_center.x + (coordinates[i].x - m\_center.x)\*scale;

coordinates[i].y = m\_center.y + (coordinates[i].y - m\_center.y)\*scale;

}

}

std::ostream& print\_shape(std::ostream& stream, Shape& shape) override {

stream << "Figure: Square" << std::endl;

stream << "ID: " << shape.get\_ID() << std::endl;

stream << "Coordinates: " << std::endl;

stream << "Center: (" << shape.get\_center().x << ";"<< shape.get\_center().y<< ")"<< std::endl;

stream << "Vertexes:" << std::endl;

for (size\_t i = 0; i < coordinates.size(); i++) {

stream << "(" << coordinates[i].x << ";" << coordinates[i].y << ")" << std::endl;

}

stream << "Color: " << shape.get\_color().red << " " << shape.get\_color().green << " " << shape.get\_color().blue << std::endl;

stream << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << std::endl;

return stream;

}

};

class Rhombus : public Shape {

double m\_diagonal1;

double m\_diagonal2;

public:

Rhombus(double diagonal1, double diagonal2, Point p1, RGB color) :Shape(color, m\_center)

{

m\_diagonal1 = diagonal1;

m\_diagonal2 = diagonal2;

coordinates.push\_back(p1);

coordinates.push\_back(Point{ p1.x - diagonal2 / 2, p1.y + diagonal1 / 2 });

coordinates.push\_back(Point{ p1.x, p1.y + diagonal1 });

coordinates.push\_back(Point{ p1.x + diagonal2 / 2, p1.y + diagonal1 / 2 });

m\_center.x = (coordinates[1].x + coordinates[3].x) / 2;

m\_center.y = (coordinates[0].y + coordinates[2].y) / 2;

}

void scaling(double scale) override {

for (size\_t i = 0; i < coordinates.size(); i++)

{

coordinates[i].x = m\_center.x + (coordinates[i].x - m\_center.x)\*scale;

coordinates[i].y = m\_center.y + (coordinates[i].y - m\_center.y)\*scale;

}

}

std::ostream& print\_shape(std::ostream& stream, Shape& shape) override

{

stream << "Figure: Rhombus" << std::endl;

stream << "ID: " << shape.get\_ID() << std::endl;

stream << "Coordinates: " << std::endl;

stream << "Center: (" << shape.get\_center().x << ";" << shape.get\_center().y << ")" << std::endl;

stream << "Vertexes:" << std::endl;

for (size\_t i = 0; i < coordinates.size(); i++)

{

stream << "(" << coordinates[i].x << ";" << coordinates[i].y << ")" << std::endl;

}

stream << "Color: " << shape.get\_color().red << " " << shape.get\_color().green << " " << shape.get\_color().blue << std::endl;

stream << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << std::endl;

return stream;

}

};

int main() {

Square test1(2, { 255, 255, 255 }, { 1, 2 },30);

std::cout << test1;

test1.rotate(30);

std::cout << test1;

Parallelogram test2({ 128,35,127 }, { 1, 5 }, { -1,9 }, { 7,9 });

std::cout << test2;

test2.scaling(20);

std::cout << test2;

Rhombus test3(7, 3, { 0,0 }, { 123, 23, 77 });

std::cout << test3;

test3.move({ 3, 6 });

std::cout << test3;

return 0;

}