**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**Кафедра МОЭВМ**

# ОТЧЕТ

**по лабораторной работе №2**

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

**Тема: Наследование**

Студент гр. 7382 _____ Филиппов И.С.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

**Цель работы.**

Проектировка иерархии классов для моделирования геометрических фигур.

**Основные теоретические положения.**

Необходимо спроектировать систему классов для моделирования геометрических фигур (в соответствии с полученным индивидуальным заданием). 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;

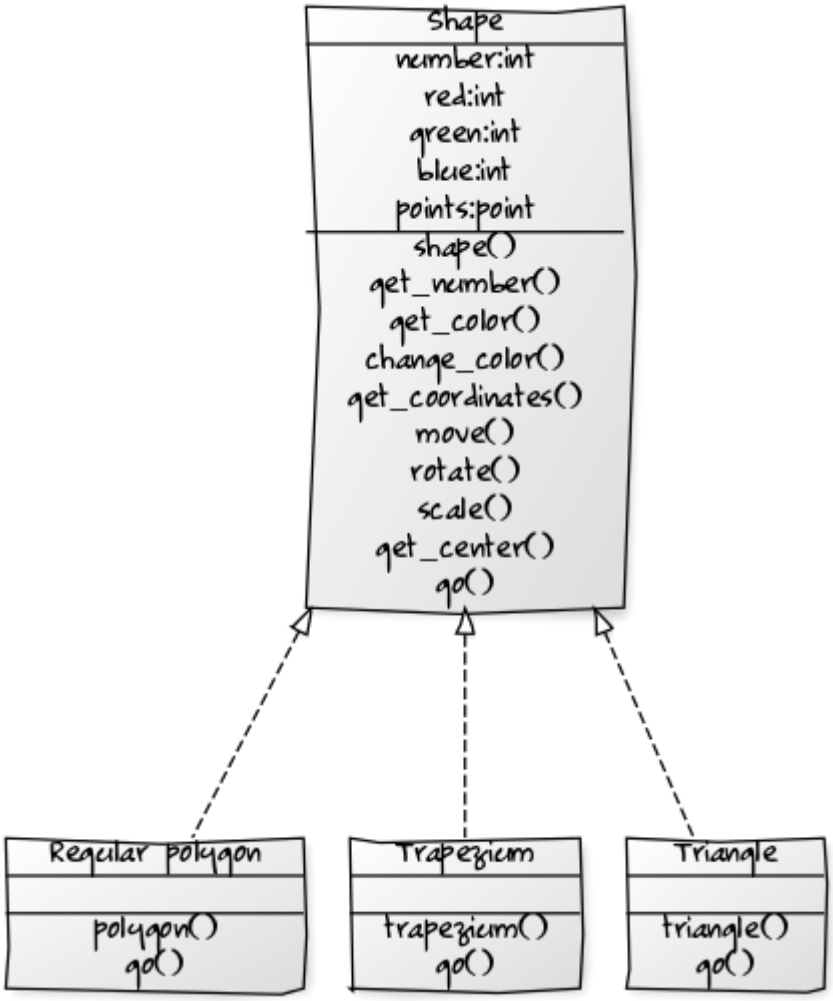
реализацию классов на языке C++.

**Выводы.**

В ходе выполнения лабораторной работы была реализованна система классов для моделирования геометрических фигур с помощью методологии объектно-ориентированного программирования.

## ПРИЛОЖЕНИЕ

## UML-диаграмма классов




## Исходный код

```
#include <vector>
#include <cmath>
#include <cstdint>
#include <iostream>
#include <tuple>
#include <cstring>

namespace figures
{
   using namespace std;

   struct point
   {
      double y;
      double x;
   };
```

```
class shape
{
public:
   uint32_t get_figure_number() const;
   virtual tuple<int16_t, int16_t, int16_t> get_color() const;
   virtual void change_color(int16_t red, int16_t green, int16_t blue);
   virtual vector<point> get_coordinates() const;
   virtual void move(pair<double, double> delta);
   virtual void rotate(int16_t angle);
   virtual void scale(int8_t coef);
   virtual ~shape() = 0;
   friend ostream& operator<<(ostream& stream, shape& obj);
   virtual void go() = 0;
protected:
   shape()
      : number_(create_figure_number())
      , red_(255), green_(255), blue_(255), points()
   {}

   static uint32_t create_figure_number()
   {
      static uint32_t number = 1;

      return number++;
   }

   virtual point get_centre() const;

protected:
   uint32_t number_;
   int16_t red_;
   int16_t green_;
   int16_t blue_;
   vector<point> points;
};

shape::~shape() {}

inline uint32_t shape::get_figure_number() const
{
   return number_;
}

inline tuple<int16_t, int16_t, int16_t> shape::get_color() const
{
   return {red_, green_, blue_};
}

inline void shape::change_color(int16_t red, int16_t green, int16_t blue)
{
   red_ = red;
   green_ = green;
```

```
    blue_ = blue;
}

inline vector<point> shape::get_coordinates() const
{
    return points;
}

inline void shape::move(pair<double, double> delta)
{
    for (auto& p : points)
    {
        p.y += delta.first;
        p.x += delta.second;
    }
}

inline void shape::rotate(int16_t angle)
{
    auto [y_0, x_0] = get_centre();

    angle %= 360;
    double radian = angle * acos(-1) / 180;

    // Rotation
    for (auto& p : points)
    {
        p.x = x_0 + (p.x - x_0) * cos(radian) - (p.y - y_0) * sin(radian);
        p.x = y_0 + (p.y - y_0) * cos(radian) - (p.x - x_0) * sin(radian);
    }
}

inline void shape::scale(int8_t coef)
{
    auto [y_0, x_0] = get_centre();

    for (auto& p : points)
    {
        p.x *= coef;
        p.y *= coef;
    }

    auto [y_1, x_1] = get_centre();

    pair<double, double> delta = {abs(y_0 - y_1), abs(x_0 - x_1)};

    if (y_1 > y_0)
        delta.first = -delta.first;

    if (x_1 > x_0)
        delta.second = -delta.second;
```

```
    move(delta);
}

inline point shape::get_centre() const
{
    double y_0 = 0;
    for (auto p : points)
        y_0 += p.y;

    double x_0 = 0;
    for (auto p : points)
        x_0 += p.x;

    return {y_0, x_0};
}

ostream& operator<<(ostream& stream, shape& obj)
{
    auto [red, green, blue] = obj.get_color();
    stream << "Figure number is " << obj.get_figure_number() << endl
           << "Color is " << red << ' ' << green << ' ' << blue << endl
           << "Coordinates are ";
    for (auto e : obj.get_coordinates())
        stream << e.y << "," << e.x << ' ';
    stream << endl;

    return stream;
}

class triangle final : public shape
{
public:
    triangle(point a, point b, point c);
    void go() override
    {
        move({10, 10});
        cout << "Triangle was moved" << endl;
    }
    ~triangle() override = default;
};

inline triangle::triangle(point a, point b, point c)
{
    points.push_back(a);
    points.push_back(b);
    points.push_back(c);
}

class trapezium : public shape
{
public:
    trapezium(point a, point b, point c, point d);
```

```
    void go() override
    {
        move({10, 10});
        cout << "Trapezium was moved" << endl;
    }
    ~trapezium() override = default;
};

inline trapezium::trapezium(point a, point b, point c, point d)
{
    points.push_back(a);
    points.push_back(b);
    points.push_back(c);
    points.push_back(d);
}

class regular_polygon : public shape
{
public:
    regular_polygon(point a, point b, point c, point d, point e);
    void go() override
    {
        scale(2);
        cout << "Polygon was scaled" << endl;
    }
    ~regular_polygon() override = default;
};

inline regular_polygon::regular_polygon(point a, point b, point c, point d, point e)
{
    points.push_back(a);
    points.push_back(b);
    points.push_back(c);
    points.push_back(d);
    points.push_back(e);

    auto get_distance = [](point a, point b) -> double
    {
        return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
    };

    auto is_equal = [](double a, double b)
    {
        return fabs(a - b) < 0.5;
    };

    auto distance = get_distance(points.front(), points.back());

    for (auto it = points.begin(); it != prev(points.end()); it++)
    {
        if (!is_equal(distance, get_distance(*it, *(next(it)))))
            throw runtime_error("The polygon is not regular! Abort");
```

```
        }
    }
}

using std::vector;

void go(const vector<figures::shape*>& arg)
{
    using std::iostream;

    for (auto e : arg)
    {
        //if (typeid(e).name() == "triangle")
        if (typeid(figures::triangle) == typeid(*e))
        {
            e->move({10, 10});
            std::cout << "Triangle was moved" << std::endl;

        }
        else if (typeid(figures::trapezium) == typeid(*e))
        {
            e->move({10, 10});
            std::cout << "Trapezium was moved" << std::endl;
        }
        else if (typeid(figures::regular_polygon) == typeid(*e))
        {
            e->scale(2);
            std::cout << "Polygon was scaled" << std::endl;
        }
        else
        {
            throw std::runtime_error("Unknown figure");
        }
    }
}

int main()
{
    vector<figures::shape*> arg;
    figures::triangle tr({1, 1}, {2, 2}, {3, 3});
    arg.push_back(&tr);

    figures::trapezium trap({1, 1}, {2, 2}, {3, 3}, {4, 4});
    arg.push_back(&trap);

    vector<figures::point> points_for_polygon;
    long R = 20;
    long X = 50;
    long Y = 50;
    for (auto i = 0; i < 5; i++)
    {
        double angle = 2 * M_PI * i / 5;
```

```
            points_for_polygon.push_back({R * sin(angle)  + Y, R * cos(angle) + X});
        }

        figures::regular_polygon pol(points_for_polygon[0], points_for_polygon[1],
points_for_polygon[2], points_for_polygon[3], points_for_polygon[4]);
        arg.push_back(&pol);

        go(arg);

        return 0;
    }
```