**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

отчет

**по практической работе №2**

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

**Тема: Наследование**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 |  | Петрова А.С. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2019

**Цель работы.**

Ознакомиться с понятиями наследование, полиморфизм, абстрактный класс, изучить виртуальные функции, принцип их работы, способы реализации. В соответствии с индивидуальным заданием разработать систему классов

**Задание:**

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 объек-та.

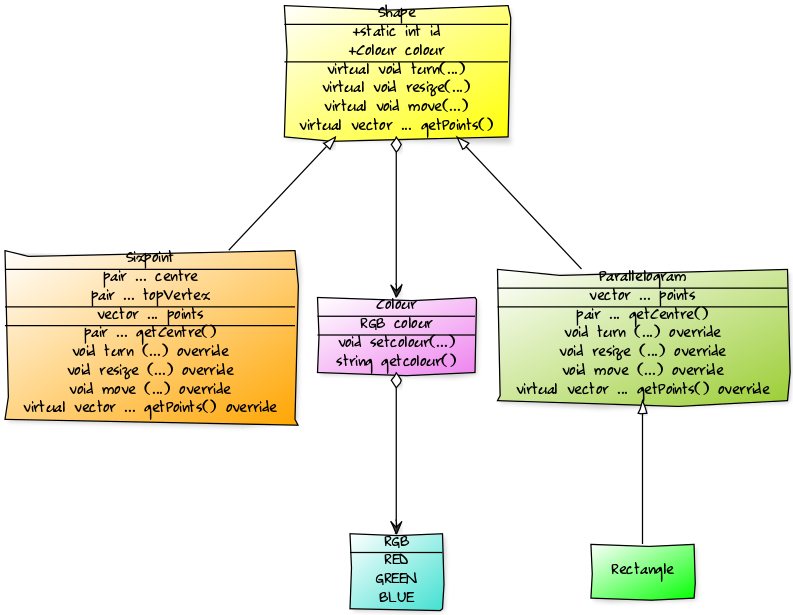
Решение должно содержать:

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

**Индивидуальное задание:**Вариант 15.

1. Параллелограмм
2. Прямоугольник
3. Правильный шестиугольник

UML-диаграмма системы классов



Исходный код программы представлен в приложении А.

**ПРИЛОЖЕНИЕ А**

**Исходный код программы**

#include "pch.h"

#include <iostream>

#include <cmath>

#include <string>

#include <vector>

#include <ctime>

#define M\_PI 3.14159265358979323846

enum RGB { RED, GREEN, BLUE };

class Colour

{

private:

RGB colour;

public:

void setcolour(RGB c)

{

switch (c)

{

case RED:

colour = RED;

break;

case GREEN:

colour = GREEN;

break;

case BLUE:

colour = BLUE;

break;

}

}

std::string getcolour()

{

switch (colour)

{

case RED:

return "Red";

case GREEN:

return "Green";

case BLUE:

return "Blue";

}

}

};

using namespace std;

class Shape

{

public:

static int id;

Shape()

{

id++;

}

Colour colour;

virtual void move(double, double) = 0;

virtual void resize(int) = 0;

virtual void turn(double) = 0;

virtual vector<pair<double, double>> getPoints() = 0;

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

};

ostream& operator<<(std::ostream& stream, Shape& obj)

{

cout << "object ID: " << obj.id << endl;

cout << "Colour: " << obj.colour.getcolour() << endl;

cout << "object place: " << endl;

for (auto i : obj.getPoints())

cout << "(" << i.first << ", " << i.second << ")" << endl;

return stream;

}

class Parallelogram : public Shape

{

protected:

vector<pair<double, double>> points;// first,second,third;

public:

Parallelogram(pair<double, double> x1, pair<double, double> x2, pair<double, double> x3) : Shape()

{

points.push\_back(x1);

points.push\_back(x2);

points.push\_back(x3);

}

//в градусах

pair<double, double> getCentre()

{

double x = abs(points[2].first + points[0].first) / 2;

double y = abs(points[1].second + points[0].second) / 2;

return { x, y };

}

void turn(double \_alpha) override

{

auto alpha = \_alpha \* M\_PI / 180;

auto centre = getCentre();

for (auto it = points.begin(); it != points.end(); it++) {

auto p1 = it->first, p2 = it->second;

it->first = centre.first + (it->first - centre.first)\*cos(alpha) - (it->second - centre.second)\*sin(alpha);

it->second = centre.second + (it->second - centre.second)\*cos(alpha) + (p1 - centre.first)\*sin(alpha);

}

}

void move(double offsetx, double offsety) override

{

for (auto it = points.begin(); it != points.end(); it++) {

it->first += offsetx;

it->second += offsety;

}

}

void resize(int k) override

{

for (auto it = points.begin(); it != points.end(); it++) {

it->first \*= k;

it->second \*= k;

}

}

vector<pair<double, double>> getPoints() override

{

auto centre = getCentre();

pair<double, double> x4 = { 2 \* centre.first - points[1].first, 2 \* centre.second - points[1].second };

return { points[0], points[1], points[2], x4 };

}

};

class Rectangle : public Parallelogram

{

public:

Rectangle(pair<double, double> x1, pair<double, double> x2, pair<double, double> x3)

: Parallelogram(x1, x2, x3)

{}

};

class Sixpoint : public Shape

{

private:

pair<double, double>centre;

pair<double, double> topVertex;

public:

Sixpoint(pair<double, double> a, pair<double, double> b) :centre(a), topVertex(b)

{}

void turn(double \_alpha) override

{

auto alpha = \_alpha \* M\_PI / 180;

auto p1 = topVertex.first;

topVertex.first = centre.first + (topVertex.first - centre.first)\*cos(alpha) - (topVertex.second - centre.second)\*sin(alpha);

topVertex.second = centre.second + (topVertex.second - centre.second)\*cos(alpha) + (p1 - centre.first)\*sin(alpha);

}

void move(double offsetx, double offsety) override

{

centre.first += offsetx;

centre.second += offsety;

topVertex.first += offsetx;

topVertex.second += offsety;

}

void resize(int k) override

{

centre.first \*= k;

centre.second \*= k;

topVertex.first \*= k;

topVertex.second \*= k;

}

vector<pair<double, double>> getPoints() override

{

auto x1 = topVertex;

turn(60);

auto x2 = topVertex;

turn(60);

auto x3 = topVertex;

turn(60);

auto x4 = topVertex;

turn(60);

auto x5 = topVertex;

turn(60);

auto x6 = topVertex;

turn(60);

return { x1, x2, x3, x4, x5, x6 };

}

};

int Shape::id = 0;

int main()

{

cout << "parallelogram" << endl;

Parallelogram par({ 2,2 }, { 0,0 }, { 4,0 });

par.colour.setcolour(RED);

cout << par;

par.turn(180);

cout << par;

par.turn(180);

cout << par;

par.move(1, 1);

cout << par;

par.resize(2);

cout << par;

cout << "rectangle" << endl;

Rectangle rec({ 0,2 }, { 0,0 }, { 4,0 });

rec.colour.setcolour(GREEN);

cout << rec;

rec.turn(180);

cout << rec;

rec.turn(180);

cout << rec;

rec.move(1, 1);

cout << rec;

rec.resize(2);

cout << rec;

cout << "regular hexagon" << endl;

Sixpoint point({ 0,0 }, { 0,2 });

point.colour.setcolour(BLUE);

cout << point;

point.turn(180);

cout << point;

point.turn(180);

cout << point;

point.move(1, 1);

cout << point;

point.resize(2);

cout << point;

return 0;

}