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 «ООП»
Тема: Наследование

Студент гр. 6304 _____ Рыбин А.С.

Преподаватель _____ Терентьев А. О.

Санкт-Петербург
2018

Цель работы.

Изучения понятия наследования. Разработка абстрактного класса. Виртуальные методы и полиморфизм.

Постановка задачи.

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++.

Ход работы.

1. Сначала определим необходимые для удобства работы константы и перечисления.

```
struct Point
{
    double m_x;
    double m_y;
};
```

Рисунок 1 – структура точка

```
const std::string default_color = "black";
const Point default_point = { 0.0, 0.0 };
const double default_size = 1.0;

enum what { CIRCLE, SQUARE, TRIANGLE };
enum rot_angle { NIL, RIGHT90, RIGHT180, LEFT90, LEFT180 };
```

Рисунок 2 – константы и перечисления

Для удобства работы все точки будут иметь класс Point. Все объекты будут иметь цвет, размер и будут привязаны к какой-либо точки, так что определим константы по умолчанию точка, цвет, и размер.

2. Перегрузим операторы вывода в поток точки, вектора точек и перечислений тип фигуры и угол.

```
std::ostream & operator<<(std::ostream & out, const Point& point);
std::ostream & operator<<(std::ostream & out, const std::vector<Point>& points);
```

Рисунок 3 – перегруженные операторы

```
std::ostream & operator<<(std::ostream & out, const what& type);
std::ostream & operator<<(std::ostream & out, const rot_angle& angle);
```

Рисунок 4 – перегруженные операторы

3. Рассмотрим базовый класс Shape

```
class Shape
{
public:
    Shape(std::string color = default_color, double size = default_size);
    virtual ~Shape() = default;

    std::string get_color() const;

    void set_color(std::string color);

    virtual what id() const = 0;
    virtual void move(Point to_move) = 0;
    virtual void rotate(rot_angle angle) = 0;
    virtual void zoom(double coef) = 0;
    virtual std::ostream& operator<<(std::ostream& out) const = 0;

protected:
    std::string m_color;
    double m_size;
};
```

Рисунок 5 – базовый класс

Т.к. в программе будет реализована полиморфная логика, то деструктор объявляется виртуальным. Все объекты будут иметь цвет и размер, так что

они объявлены в базовом классе. Так же есть методы установки и получения цвета. Остальные методы являются чистыми виртуальными; их реализация будет своя для каждого из классов, но семантика одна и та же. Приведём её описание:

- ```
  virtual what id() const = 0;
  ```

  Возвращает тип объекта в виде перечисляемого типа для однозначной идентификации объекта.

- ```
  virtual void move(Point to_move) = 0;
  ```

  Перемещает фигуру в заданную точку.

- ```
  virtual void rotate(rot_angle angle) = 0;
  ```

  Поворачивает фигуру на заданный перечисляемым типом угол.

- ```
  virtual void zoom(double coef) = 0;
  ```

  Увеличивает фигуру на заданный положительный коэффициент.

- ```
  virtual std::ostream& operator<<(std::ostream& out) const = 0;
  ```

  Перегруженный оператор вывода в поток.

4. Рассмотрим дочерние классы

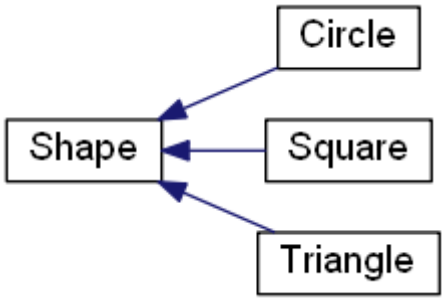

Рисунок 6 – Диаграмма иерархии классов

В программе будет три дочерних класс: круг, квадрат и правильный треугольник.

- Класс круг будет дополнительно иметь поле центр, а поле размер из базового класса будет интерпретироваться как радиус.

  ```
  Point m_center;
  ```

- Класс квадрат будет дополнительно иметь вектор из четырёх точек, определяющих его, а поле размер базового класса будет означать его сторону.

  ```
  std::vector<Point> m_points;
  ```
  ```

- Класс круг будет дополнительно иметь вектор из трёх точек, определяющих его и поле угол, которое задаёт угол, на который в данный момент повёрнут треугольник, т.к. это необходимо для корректного перемещения его в заданную точку ненарущающего инвариант класса.

```
std::vector<Point> m_points;
rot_angle m_angle;
```

Реализация представлена далее в приложениях.

Выводы.

Способность к наследованию встроена в язык C++, что позволяет максимизировать многократное переиспользование кода и даёт возможность производить моделирование предметной области естественным образом. Виртуальные функции дают возможность полиморфической обработки связанных объектов, а абстрактные классы реализацию общего интерфейса.

ПРИЛОЖЕНИЕ А

Shape.hpp

```
#pragma once

#include <string>
#include <vector>
#include <ostream>

struct Point
{
	double m_x;
	double m_y;
};

std::ostream & operator<<(std::ostream & out, const Point& point);
std::ostream & operator<<(std::ostream & out, const std::vector<Point>& points);
bool operator==(const Point& lhs, const Point& rhs);

const std::string default_color = "black";
const Point default_point = { 0.0, 0.0 };
const double default_size = 1.0;

enum what { CIRCLE, SQUARE, TRIANGLE };
enum rot_angle { NIL, RIGHT90, RIGHT180, LEFT90, LEFT180 };

std::ostream & operator<<(std::ostream & out, const what& type);
std::ostream & operator<<(std::ostream & out, const rot_angle& angle);

class Shape
{
public:
	Shape(std::string color = default_color, double size = default_size);
	virtual ~Shape() = default;

	std::string get_color() const;

	void set_color(std::string color);

	virtual what id() const = 0;
	virtual void move(Point to_move) = 0;
	virtual void rotate(rot_angle angle) = 0;
	virtual void zoom(double coef) = 0;
	virtual std::ostream& operator<<(std::ostream& out) const = 0;

protected:
	std::string m_color;
	double m_size;
};

class Circle : public Shape
{
public:
	Circle(std::string color = default_color, double radius = default_size, Point
center = default_point);
	what id() const override;
	void move(Point to_move) override;
	void rotate(rot_angle angle) override;
	void zoom(double coef) override;
	std::ostream& operator<<(std::ostream& out) const override;
protected:
	Point m_center;
};
```

```
class Square : public Shape
{
public:
	Square(std::string color = default_color, double size = default_size, Point
bottom_left = default_point);
	what id() const override;
	void move(Point to_move) override;
	void rotate(rot_angle angle) override;
	void zoom(double coef) override;
	std::ostream& operator<<(std::ostream& out) const override;
protected:
	std::vector<Point> m_points;
};

class Triangle : public Shape
{
public:
	Triangle(std::string color = default_color, double size = default_size, Point
bottom_left = default_point);
	what id() const override;
	void move(Point to_move) override;
	void rotate(rot_angle angle) override;
	void zoom(double coef) override;
	std::ostream& operator<<(std::ostream& out) const override;
protected:
	std::vector<Point> m_points;
	rot_angle m_angle;
	};
```

ПРИЛОЖЕНИЕ Б

Shape_Point.cpp

```
#include "Shape.hpp"

std::ostream & operator<<(std::ostream & out, const Point& point)
{
	out << "(" << point.m_x << ","
		<< point.m_y << ")";

	return out;
}

std::ostream & operator<<(std::ostream & out, const std::vector<Point>& points)
{
	for (const auto& point : points)
	{
		out << point << ", ";
	}

	return out;
}

bool operator==(const Point& lhs, const Point& rhs)
{
	return lhs.m_x == rhs.m_x && lhs.m_y == rhs.m_y;
}

std::ostream & operator<<(std::ostream & out, const what & type)
{
	switch (type)
	{
	case CIRCLE:
		out << "Circle";
		break;
	case SQUARE:
		out << "Square";
		break;
	case TRIANGLE:
		out << "Triangle";
		break;
	default:
		break;
	}

	return out;
}

std::ostream & operator<<(std::ostream & out, const rot_angle & angle)
{
	switch (angle)
	{
	case NIL:
		out << "0 gradus";
		break;
	case RIGHT90:
		out << "90 gradus on right";
		break;
	case RIGHT180:
	case LEFT180:
		out << "180 gradus on right or left";
		break;
	case LEFT90:
		out << "90 gradus on left";
```

```
            break;
      default:
            break;
      }
      return out;
}

Shape::Shape(std::string color, double size)
      : m_color(color)
      , m_size(size)
{
}

std::string Shape::get_color() const
{
      return m_color;
}

void Shape::set_color(std::string color)
{
      m_color = color;
      }
```

ПРИЛОЖЕНИЕ В

Circle.cpp

```
#include "Shape.hpp"

Circle::Circle(std::string color, double radius, Point center)
	: Shape(color, radius)
	, m_center(center)
{
}

what Circle::id() const
{
	return what(CIRCLE);
}

void Circle::move(Point to_move)
{
	if (m_center == to_move)
	{
		return;
	}
	else
	{
		m_center = to_move;
	}
}

void Circle::rotate(rot_angle angle)
{
	;
}

void Circle::zoom(double coef)
{
	if (coef > 0)
	{
		m_size *= coef;
	}
	else
	{
		return;
	}
}

std::ostream & Circle::operator<<(std::ostream & out) const
{
	out << "type: " << id() << "\n"
		<< "color: " << m_color << "\n"
		<< "center: " << m_center << "\n"
		<< "radius: " << m_size << "\n\n";

	return out;
}
```

ПРИЛОЖЕНИЕ Г

Square.cpp

```
#include "Shape.hpp"

Square::Square(std::string color, double size, Point bottom_left)
	: Shape(color, size)
{
	m_points.resize(4);
	move(bottom_left);
}

what Square::id() const
{
	return what(SQUARE);
}

void Square::move(Point to_move)
{
	m_points[0] = to_move;

	m_points[1].m_x = m_points[0].m_x;
	m_points[1].m_y = m_points[0].m_y + m_size;

	m_points[2].m_x = m_points[0].m_x + m_size;
	m_points[2].m_y = m_points[0].m_y + m_size;

	m_points[3].m_x = m_points[0].m_x + m_size;
	m_points[3].m_y = m_points[0].m_y;
}

void Square::rotate(rot_angle angle)
{
	switch (angle)
	{
	case RIGHT90:
		move({ m_points[0].m_x, m_points[0].m_y - m_size });
		break;
	case RIGHT180:
		move({ m_points[0].m_x - m_size, m_points[0].m_y - m_size });
		break;
	case LEFT90:
		move({ m_points[0].m_x - m_size, m_points[0].m_y });
		break;
	case LEFT180:
		move({ m_points[0].m_x - m_size, m_points[0].m_y - m_size });
		break;
	case NIL:
		break;
	default:
		break;
	}
}

void Square::zoom(double coef)
{
	if (coef > 0)
	{
		m_size *= coef;
		move(m_points[0]);
	}
	else
	{
		return;
```

```
	}
}

std::ostream & Square::operator<<(std::ostream & out) const
{
	out << "type: " << id() << "\n"
		<< "color: " << m_color << "\n"
		<< "points: " << m_points << "\n"
		<< "size:" << m_size << "\n\n";

	return out;
	}
```

ПРИЛОЖЕНИЕ Д

Triangle.cpp

```
#include "Shape.hpp"
#include <cmath>

what Triangle::id() const
{
	return what(TRIANGLE);
}

Triangle::Triangle(std::string color, double size, Point bottom_left)
	: Shape(color, size)
	, m_angle(NIL)
{
	m_points.resize(3);
	move(bottom_left);
}

void Triangle::move(Point to_move)
{

	switch (m_angle)
	{
	case NIL:
		m_points[0] = to_move;

		m_points[1].m_x = m_points[0].m_x + m_size * 1.0 / 2.0;
		m_points[1].m_y = m_points[0].m_y + m_size * sqrt(3.0) / 2.0;

		m_points[2].m_x = m_points[0].m_x + m_size;
		m_points[2].m_y = m_points[0].m_y;

		break;
	case RIGHT90:
		m_points[0] = to_move;

		m_points[1].m_x = m_points[0].m_x;
		m_points[1].m_y = m_points[0].m_y + m_size;

		m_points[2].m_x = m_points[0].m_x + m_size * sqrt(3.0) / 2.0;
		m_points[2].m_y = m_points[0].m_y + m_size / 2.0;

		break;
	case RIGHT180:
	case LEFT180:
		m_points[0] = to_move;

		m_points[1].m_x = m_points[0].m_x - m_size / 2.0;
		m_points[1].m_y = m_points[0].m_y + m_size * sqrt(3.0) / 2.0;

		m_points[2].m_x = m_points[0].m_x + m_size / 2.0;
		m_points[2].m_y = m_points[0].m_y + m_size * sqrt(3.0) / 2.0;

		break;
	case LEFT90:
		m_points[0] = to_move;

		m_points[1].m_x = m_points[0].m_x - m_size * sqrt(3.0) / 2.0;
		m_points[1].m_y = m_points[0].m_y + m_size / 2.0;

		m_points[2].m_x = m_points[0].m_x;
		m_points[2].m_y = m_points[0].m_y + m_size;
```

```
            break;
        default:
            break;
        }
}

void Triangle::zoom(double coef)
{
        if (coef > 0)
        {
            m_size *= coef;
            move(m_points[0]);
        }
        else
        {
            return;
        }
}

void Triangle::rotate(rot_angle angle)
{
        m_angle = angle;

        switch (m_angle)
        {
        case NIL:
            break;
        case RIGHT90:
            move({ m_points[0].m_x, m_points[0].m_y - m_size });
            break;
        case RIGHT180:
        case LEFT180:
            move({ m_points[0].m_x - m_size / 2.0, m_points[0].m_y - m_size });
            break;
        case LEFT90:
            move(m_points[0]);
        default:
            break;
        }
}

std::ostream & Triangle::operator<<(std::ostream & out) const
{
        out << "type: " << id() << "\n"
            << "color: " << m_color << "\n"
            << "angle: " << m_angle << "\n"
            << "points: " << m_points << "\n"
            << "size:" << m_size << "\n\n";

        return out;
        }
```