МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждения высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт компьютерных технологий и информационной безопасности

Кафедра математического обеспечения и применения ЭВМ

**ЛАБОРАТОРНАЯ РАБОТА № 3**

по дисциплине

**«Объектно-ориентированное программирование»**

на тему:

**«Динамический выбор типа объекта»**

*Вариант № 3*

Выполнил:

Студент группы

КТбо2-8

Данильченко Т. В.

Проверил:

доцент кафедры

МОП ЭВМ

Тарасов С. А.

Оценка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Таганрог 2020

# **1 Цель работы**

Ознакомление с механизмом выбора типа создаваемых объектов во время выполнения программы.

# **2 Задание, вариант № 3**

# В лабораторной работе должна быть создана программа, создающая объекты двух классов (T1, T2), выбранных из таблицы 1 согласно номеру варианта. Эти классы должны быть производными от класса Shape.

# **3 Ход работы**

**3.1 Спецификации классов**

Диаграмма используемых классов приведена на Рисунке 1.

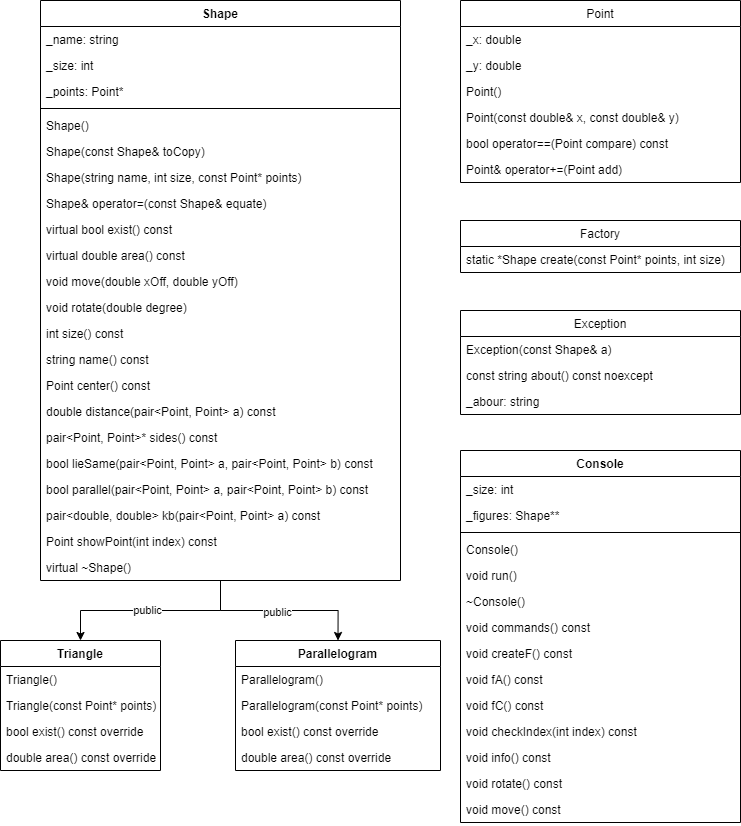


Рисунок 1 — Диаграмма классов

В классе Shape определен массив точек, описывающих фигуру. Для фигуры есть операции перемещения и вращения, получения площади и центра тяжести фигуры. Также определен фабричный метод create, позволяющий выбирать тип переменной во время выполнения программы. Класс имеет производные классы Triangle и Parallelogram.

В классе Console выполняется взаимодействие с пользователем.

**3 Листинг**

// Point.h

#pragma once

struct Point {

double \_x;

double \_y;

Point();

Point(const double& x, const double& y);

bool operator==(Point compare) const;

Point& operator+=(Point add);

};

// Shape.h

#pragma once

#define \_USE\_MATH\_DEFINES

#include "Point.h"

#include <string>

#include <cmath>

using std::string;

class Shape {

public:

Shape();

Shape(const Shape& toCopy);

Shape(string name, int size, const Point\* points);

Shape& operator=(const Shape& equate);

virtual bool exist() const = 0;

virtual double area() const = 0;

void move(double xOff, double yOff);

void rotate(double degree);

int size() const;

string name() const;

Point center() const;

double distance(std::pair<Point, Point> a) const;

std::pair<Point, Point>\* sides() const;

bool lieSame(std::pair<Point, Point> a, std::pair<Point, Point> b) const;

bool parallel(std::pair<Point, Point> a, std::pair<Point, Point> b) const;

std::pair<double, double> kb(std::pair<Point, Point> a) const;

Point showPoint(int index) const;

virtual ~Shape();

private:

string \_name;

int \_size;

Point\* \_points;

protected:

void existCheck() const ;

};

// Triangle.h

#pragma once

#include "Shape.h"

class Triangle:public Shape {

public:

Triangle() = delete;

Triangle(const Point\* points);

bool exist() const override;

double area() const override;

};

// Parallelogram.h

#pragma once

#include "Shape.h"

class Parallelogram :public Shape {

public:

Parallelogram() = delete;

Parallelogram(const Point\* points);

bool exist() const override;

double area() const override;

};

// Fabric.h

#pragma once

#include "Shape.h"

#include "Triangle.h"

#include "Parallelogram.h"

class Factory {

public:

static Shape\* create(const Point\* points, int size);

};

// Exception.h

#pragma once

#include <exception>

#include "Shape.h"

class Exception : public std::exception {

public:

Exception(const Shape& a);

const std::string about() const noexcept;

private:

string \_aboutS;

};

// Console.h

#pragma once

#include "Triangle.h"

#include "Parallelogram.h"

class Console {

public:

Console();

void run();

~Console();

private:

int \_size;

Shape\*\* \_figures;

void commands() const;

void createF();

void fA() const;

void fC() const;

bool checkIndex(int index) const;

void info() const;

void rotate() const;

void move() const;

};

// Point.cpp

#include "Point.h"

Point::Point()

: \_x(0.0), \_y(0.0) {}

Point::Point(const double& x, const double& y)

: \_x(x), \_y(y) {}

bool Point::operator==(Point compare) const {

return (\_x == compare.\_x && \_y == compare.\_y);

}

Point& Point::operator+=(Point add) {

\_x += add.\_x;

\_y += add.\_y;

return \*this;

}

// Shape.cpp

#include "Shape.h"

#include "Exception.h"

#include <algorithm>

Shape::Shape()

: \_name(""), \_size(0), \_points(new Point[\_size]){}

Shape::Shape(string name, int size, const Point\* points)

: \_name(name), \_size(size), \_points(new Point[\_size]) {

std::copy(points, points + size, \_points);

}

Shape::Shape(const Shape& toCopy)

: Shape(toCopy.\_name, toCopy.\_size, toCopy.\_points){}

Shape& Shape::operator=(const Shape& equate) {

if (this == &equate) return \*this;

if (\_points != nullptr) {

delete[] \_points;

}

\_size = equate.\_size;

\_points = new Point[\_size];

std::copy(equate.\_points, equate.\_points + \_size, \_points);

return \*this;

}

void Shape::move(double xOff, double yOff){

Point off(xOff, yOff);

for (int i = 0; i < \_size; ++i) {

\_points[i] += off;

}

}

void Shape::rotate(double degree) {

double angle = ((degree) \* M\_PI / 180.0);

for (int i = 0; i < \_size; ++i) {

double EPS = 1e-6;

double x = \_points[i].\_x \* cos(angle) - \_points[i].\_y \* sin(angle);

double y = \_points[i].\_x \* sin(angle) + \_points[i].\_y \* cos(angle);

if (x > -EPS && x < EPS) x = 0.0;

if (y > -EPS && y < EPS) y = 0.0;

\_points[i] = Point(x, y);

}

}

int Shape::size() const {

return \_size;

}

string Shape::name() const {

return \_name;

}

Point Shape::center() const {

double \_xB = 0.0,

\_yB = 0.0;

for (int i = 0; i < \_size; ++i) {

\_xB += \_points[i].\_x;

\_yB += \_points[i].\_y;

}

return Point(\_xB / \_size, \_yB / \_size);

}

double Shape::distance(std::pair<Point, Point> a) const {

return sqrt((a.first.\_x - a.second.\_x) \* (a.first.\_x - a.second.\_x) + (a.first.\_y - a.second.\_y) \* (a.first.\_y - a.second.\_y));

}

std::pair<Point, Point>\* Shape::sides() const {

std::pair<Point, Point>\* sides = new std::pair<Point, Point>[\_size];

for (int i = 0; i < \_size - 1; ++i) {

sides[i] = { \_points[i], \_points[i + 1] };

}

sides[\_size - 1] = { \_points[\_size - 1], \_points[0] };

return sides;

}

std::pair<double, double> Shape::kb(std::pair<Point, Point> a) const {

double k;

if ((a.first.\_x - a.second.\_x) != 0) {

k = (a.first.\_y - a.second.\_y) / (a.first.\_x - a.second.\_x);

}

else {

k = INFINITY;

}

double b;

if (k == INFINITY) {

b = 0;

}

else {

b = a.first.\_y - a.first.\_x \* k;

}

return { k, b };

}

bool Shape::lieSame(std::pair<Point, Point> a, std::pair<Point, Point> b) const {

double EPS = -1e9;

std::pair<double, double> koef1 = kb(a);

std::pair<double, double> koef2 = kb(b);

if (std::abs(koef1.first - koef2.first) < EPS && std::abs(koef1.second - koef2.second < EPS)) {

return true;

}

return false;

}

bool Shape::parallel(std::pair<Point, Point> a, std::pair<Point, Point> b) const {

double EPS = 1e-9;

std::pair<double, double> koef1 = kb(a);

std::pair<double, double> koef2 = kb(b);

if (koef1.first == INFINITY && koef2.first == INFINITY) return true;

if (std::abs(koef1.first - koef2.first) < EPS && std::abs(koef1.second - koef2.second) > EPS) {

return true;

}

return false;

}

Point Shape::showPoint(int index) const{

return \_points[index];

}

Shape::~Shape() {

if (\_points != nullptr) {

delete[] \_points;

}

}

void Shape::existCheck() const {

if (!exist()) {

throw Exception(\*this);

}

}

// Triangle.cpp

#include "Triangle.h"

Triangle::Triangle(const Point\* points)

: Shape("Triangle", 3, points) {}

bool Triangle::exist() const {

std::pair<Point, Point>\* s = sides();

double A = distance(s[0]);

double B = distance(s[1]);

double C = distance(s[2]);

delete[] s;

return (A + B > C && C + A > B && B + C > A);

}

double Triangle::area() const {

std::pair<Point, Point>\* s = sides();

double A = distance(s[0]);

double B = distance(s[1]);

double C = distance(s[2]);

delete[] s;

double p = (A + B + C) / 2;

return sqrt(p \* (p - A) \* (p - B) \* (p - C));

}

// Parallelogram.cpp

#include "Parallelogram.h"

Parallelogram::Parallelogram(const Point\* points)

: Shape("Parallelogram", 4, points) {}

bool Parallelogram::exist() const {

std::pair<Point, Point>\* s = sides();

bool res = true;

if ((!(parallel(s[0], s[2]) && parallel(s[1], s[3])))

|| (!(distance(s[0]) == distance(s[2]) && distance(s[1]) == distance(s[3])))) {

res = false;

}

delete[] s;

return res;

}

double Parallelogram::area() const {

std::pair<Point, Point>\* s = sides();

Point a = s[0].first;

Point b = s[0].second;

Point c = s[1].second;

double ab = distance({ a, b });

double bc = distance({ b, c });

double ca = distance({ c, a });

double angle = acos(sqrt((ca \* ca - ab \* ab - bc \* bc) / (2 \* ab \* bc)));

double area = ab \* bc \* sin(angle);

delete[] s;

return area;

}

// Fabric.cpp

#include "Fabric.h"

Shape\* Factory::create(const Point\* points, int size) {

Shape\* f = nullptr;

if (size == 3) {

Triangle t(points);

if (t.exist()) {

f = new Triangle(points);

}

}

else if (size == 4) {

Parallelogram p(points);

if (p.exist()) {

f = new Parallelogram(points);

}

}

return f;

}

// Exception.cpp

#include "Exception.h"

#include <sstream>

Exception::Exception(const Shape& a) {

std::ostringstream sstr;

sstr << "Shape [" << a.name() << "] with " << a.size() << " can't be used";

\_aboutS = sstr.str();

}

const string Exception::about() const noexcept {

return \_aboutS.c\_str();

}

// Console.cpp

#include "Console.h"

#include <iostream>

#include <algorithm>

#include "Fabric.h"

using std::cin;

using std::cout;

using std::copy;

enum \_commands\_ {

\_exit\_, \_create\_, \_area\_, \_center\_, \_rotate\_, \_move\_, \_info\_, \_commands\_

};

Console::Console()

: \_size(0), \_figures(new Shape\*[\_size]) {}

Console::~Console() {

delete[] \_figures;

}

void Console::commands() const {

cout << "Command list: \n";

cout << \_exit\_ <<": exit \n";

cout << \_create\_ <<": create figure\n";

cout << \_area\_ << ": find area of figure\n";

cout << \_center\_ << ": find center of figure\n";

cout << \_rotate\_ << ": rotate figure\n";

cout << \_move\_ << ": move figure\n";

cout << \_info\_ << ": show information about figure\n";

cout << \_commands\_ << ": commands list\n";

}

void Console::createF() {

Shape\*\* buffer = new Shape\*[\_size + 1];

copy(\_figures, \_figures + \_size, buffer);

delete[] \_figures;

\_figures = buffer;

int select;

cout << "1)Triangle\n2)Parallelogram\nSelect figure:";

cin >> select;

Point\* points = nullptr;

int size;

if (select == 1) size = 3;

else if (select == 2) size = 4;

else {

cout << "No such figures.\n";

return;

}

points = new Point[size];

cout << "Enter info: \n";

for (int i = 0; i < size; ++i) {

cout << "x" << i + 1 << ": ";

cin >> points[i].\_x;

cout << "y" << i + 1 << ": ";

cin >> points[i].\_y;

cout << "\n";

}

Shape\* new\_f = Factory::create(points, size);

if (new\_f != nullptr) {

\_figures[\_size++] = new\_f;

}

else {

cout << "Figure hasn't been created. Figure doesn't exist.\n";

}

}

bool Console::checkIndex(int index) const {

return index < \_size;

}

void Console::fA() const {

cout << "Enter index of figure: ";

int index;

cin >> index;

if (checkIndex(--index)) {

cout << \_figures[index]->area() << "\n";

}

else {

cout << "Index more than size.\n";

}

}

void Console::fC() const {

cout << "Enter index of figure: ";

int index;

cin >> index;

if (checkIndex(--index)) {

Point c = \_figures[index]->center();

cout << "x: "<< c.\_x << "\ny: " << c.\_y << "\n";

}

else {

cout << "Index more than size.\n";

}

}

void Console::info() const {

cout << "Enter index of figure: ";

int index;

cin >> index;

if (checkIndex(--index)) {

cout << \_figures[index]->name() << "\n";

cout << "Points: \n";

for (int i = 0; i < \_figures[index]->size(); ++i) {

Point current = \_figures[index]->showPoint(i);

cout << i + 1 << ": x" << i + 1 << ": "<< current.\_x << " " << "|y" << i + 1 << ": "<< current.\_y << "\n";

}

}

else {

cout << "Index more than size.\n";

}

}

void Console::rotate() const {

cout << "Enter index of figure: ";

int index;

cin >> index;

if (checkIndex(--index)) {

double degree;

cout << "Enter degree to rotate: ";

cin >> degree;

\_figures[index]->rotate(degree);

}

else {

cout << "Index more than size.\n";

}

}

void Console::move() const {

int index;

cout << "Enter index io figure: ";

cin >> index;

if(!checkIndex(--index)) cout << "Index more than size.\n";

cout << "Enter offset\n";

double x, y;

cout << "x: ";

cin >> x;

cout << "y: ";

cin >> y;

\_figures[index]->move(x, y);

}

void Console::run() {

commands();

int request = 1;

while (request) {

cout << "Enter command number: ";

cin >> request;

switch (request) {

case \_exit\_:

cout << "Goodbye.\n";

break;

case \_create\_:

createF();

break;

case \_area\_:

fA();

break;

case \_center\_:

fC();

break;

case \_info\_:

info();

break;

case \_rotate\_:

rotate();

break;

case \_move\_:

move();

break;

case \_commands\_:

commands();

break;

default:

cout << "No such command.\n";

break;

}

}

}