Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа
по курсу «ООП»

Тема:
Проектирование структуры классов.

Студент:	Игитова А.А.
Группа:	М80-208Б-18
Преподаватель:	Журавлев А.А.
Вариант:	9
Оценка:	
Дата:	

Москва
2019

1. Постановка задачи

Спроектировать простейший графический векторный редактор.
Требование к функционалу редактора:

- создание нового документа
- импорт документа из файла
- экспорт документа в файл
- создание графического примитива (согласно варианту задания)
- удаление графического примитива
- отображение документа на экране (печать перечня графических объектов и их характеристик)
- реализовать операцию undo, отменяющую последнее сделанное действие. Должно действовать для операций добавления/удаления фигур.

Требования к реализации:

- Создание графических примитивов необходимо вынести в отдельный класс – Factory.
- Сделать упор на использовании полиморфизма при работе с фигурами;
- Взаимодействие с пользователем (ввод команд) реализовать в функции main;

Вариант 9:
Треугольник, квадрат, прямоугольник.

2. Код программы на языке C++

main.cpp:

```
#include <iostream>
#include <string>
#include "editor.h"

void menu() {
	std::cout << "\nMenu\n";
	std::cout << "Create\n";
	std::cout << "Load\n";
	std::cout << "Save <fileName>\n";
	std::cout << "Add <figureType>\n";
	std::cout << "Remove <figure ID>\n";
	std::cout << "Undo\n";
	std::cout << "Print\n\n";
}

void create(Editor &editor) {
	std::string cmd;
	if (editor.DocumentExist()) {
		std::cout << "Save old document or don't save? Yes/No\n";
		std::cin >> cmd;
		if (cmd == "Yes") {
			std::string filename;
			std::cout << "Enter name of file\n";
			std::cin >> filename;
			try {
				editor.SaveDocument(filename);
				std::cout << "Document save in file " << filename << "\n";
			} catch (std::runtime_error &err) {
				std::cout << err.what() << "\n";
```

```
                }
            }
            std::cout << "Enter name of new project\n";
        }
        std::cin >> cmd;
        editor.CreateDocument(cmd);
        std::cout << "Document " << cmd << " is created\n";
}

void save(Editor &editor) {
        if (!editor.DocumentExist()) {
            throw std::runtime_error("Document does not exist");
        }
        std::string filename;
        std::cin >> filename;

        try {
            editor.SaveDocument(filename);
            std::cout << "Document save if file " << filename << "\n";
        } catch (std::runtime_error &err) {
            std::cout << err.what() << "\n";
        }
}

void load(Editor &editor) {
        std::string cmd;
        std::string filename;
        if (editor.DocumentExist()) {
            std::cout << "Save old document or don't save? Yes/No\n";
            std::cin >> cmd;
            if (cmd == "Yes") {
                std::cout << "Enter name of file\n";
                std::cin >> filename;
                try {
                    editor.SaveDocument(filename);
                    std::cout << "Document save in file " << filename << "\n";
                } catch (std::runtime_error& err) {
                    std::cout << err.what() << "\n";
                }
            }
            std::cin >> filename;
            try {
                editor.LoadDocument(filename);
                std::cout << "Document loaded from file " << filename << "\n";
            } catch (std::runtime_error& err) {
                std::cout << err.what() << "\n";
            }
        }
}

void add(Editor &editor) {
        if (!editor.DocumentExist()) {
            throw std::runtime_error("Document does not exist");
        }
        char type;
        std::cin >> type;
        if (type == 'T') {
            std::pair<double, double> *vertices = new std::pair<double, double>[3];
            for (int i = 0; i < 3; i++) {
                std::cin >> vertices[i].first >> vertices[i].second;
            }
            try {
                editor.InsertPrimitive(TRIANGLE, vertices);
            } catch (std::logic_error &err) {
                std::cout << err.what() << "\n";
                return;
            }
        }
        else if (type == 'S') {
            std::pair<double, double> *vertices = new std::pair<double, double>[4];
```

```
            for (int i = 0; i < 4; i++) {
                std::cin >> vertices[i].first >> vertices[i].second;
            }
            try {
                editor.InsertPrimitive(SQUARE, vertices);
            } catch (std::logic_error &err) {
                std::cout << err.what() << "\n";
                return;
            }
        }
        else if (type == 'R') {
            std::pair<double, double> *vertices = new std::pair<double, double>[4];
            for (int i = 0; i < 4; i++) {
                std::cin >> vertices[i].first >> vertices[i].second;
            }
            try {
                editor.InsertPrimitive(RECTANGLE, vertices);
            } catch (std::logic_error &err) {
                std::cout << err.what() << "\n";
                return;
            }
        }
        else {
            std::cout << "Primitive isn't added\n";
            std::cin.clear();
            std::cin.ignore(30000, '\n');
            return;
        }
        std::cout << "Primitive is added\n";
}

void remove(Editor &editor) {
        if (!editor.DocumentExist()) {
            throw std::runtime_error("Document does not exist");
        }
        int id;

        std::cin >> id;

        try {
            editor.RemovePrimitive(id);
        } catch (std::exception &err) {;
            return;
        }
        std::cout << "Primitive with " << id << " is removed\n";
}

int main(int argc, char **argv) {
        /*SDL_Window *gWindow = nullptr;
        SDL_Renderer *gRenderer = nullptr;*/
        Editor editor;
        std::string cmd;

        while(std::cin >> cmd) {
            if (cmd == "Menu") {
                menu();
            }
            else if (cmd == "Create") {
                create(editor);
            }
            else if (cmd == "Save") {
                try {
                    save(editor);
                } catch (std::runtime_error &err) {
                    std::cout << err.what() << "\n\n";
                }
            }
            else if (cmd == "Load") {
                try {
                    load(editor);
```

```
                } catch (std::runtime_error &err) {
                        std::cout << err.what() << "\n\n";
                }
        }
        else if (cmd == "Add") {
                try {
                        add(editor);
                } catch (std::runtime_error &err) {
                        std::cout << err.what() << "\n\n";
                }
        }
        else if (cmd == "Remove") {
                try {
                        remove(editor);
                } catch (std::exception &err) {
                        std::cout << err.what() << "\n";
                }
        }
        else if (cmd == "Undo") {
                try {
                        editor.Undo();
                        std::cout << "OK\n";
                } catch (std::logic_error &err) {
                        std::cout << err.what() << "\n\n";
                }
        }
        else if (cmd == "Print") {
                if (!editor.DocumentExist()) {
                        std::cout << "Document does not exist" << "\n\n";
                        continue;
                }
                editor.PrintDocument();
        }
        else {
                std::cin.clear();
                std::cin.ignore(30000, '\n');
        }
        std::cout << "\n";
    }

    return 0;
}
```

figures.h:

```
#ifndef FIGURES_H
#define FIGURES_H 1

#include <iostream>
#include <fstream>
#include <utility>
#include <cmath>
#include <memory>

enum FigureType {
    TRIANGLE,
    SQUARE,
    RECTANGLE
};

class Figure {
public:
    virtual double Area() const = 0;
    virtual std::pair<double, double> Center() const = 0;
    virtual std::ostream &Print(std::ostream &out) const = 0;
    virtual void Serialize(std::ofstream &os) const = 0;
    virtual void Deserialize(std::ifstream &is) = 0;
    virtual int getId() const = 0;
    virtual ~Figure() = default;
```

```
};

namespace Geometry {
	using Vertex = std::pair<double, double>;
	double Product(const Vertex &v1, const Vertex &v2) {
		return v1.first * v2.first + v1.second * v2.second;
	}

	double PointDistance(const Vertex &v1, const Vertex &v2) {
		return sqrt(pow((v2.first - v1.first), 2) +
			pow((v2.second - v1.second), 2));
	}

	class Vector {
		double x, y;
	public:
		Vector(double x_cord, double y_cord) : x{x_cord}, y{y_cord} {};

		Vector(Vertex &v1, Vertex &v2) : x{v2.first - v1.first},
					y{v2.second - v1.second} {};

		double operator*(const Vector &a) const {
			return (x * a.x) + (y * a.y);
		}

		Vector &operator=(const Vector &a) {
			x = a.x;
			y = a.y;

			return *this;
		}
		friend double LengthVector(const Vector &a);
		friend bool VectorsAreParallel(const Vector &a, const Vector &b);
	};

	double LengthVector(const Vertex &v1, const Vertex &v2) {
		return PointDistance(v1, v2);
	}

	double LengthVector(const Vector &a) {
		return sqrt(pow(a.x, 2) + pow(a.y, 2));
	}

	bool VectorsAreParallel(const Vector &a, const Vector &b) {
		return (a.x * b.y) - (a.y * b.x) == 0;
	}

	double Area(const Vertex *vertices, int n) {
		double res = 0;

		for (int i = 0; i < n - 1; i++) {
			res += (vertices[i].first * vertices[i + 1].second -
				vertices[i + 1].first * vertices[i].second);
		}
		res += (vertices[n - 1].first * vertices[0].second -
				vertices[0].first * vertices[n - 1].second);

		return 0.5 * std::abs(res);
	}

	Vertex Center(const Vertex *vertices, int n) {
		double x = 0, y = 0;

		for (int i = 0; i < n; i++) {
			x += vertices[i].first;
			y += vertices[i].second;
		}

		return std::make_pair(x / n, y / n);
	}
```

```
}

std::ostream &operator<<(std::ostream &out, std::pair<double, double> v) {
	out << "(" << v.first << ", " << v.second << ")";
	return out;
}

class Triangle : public Figure {
	using Vertex = std::pair<double, double>;
	int Id;
	Vertex *vertices;
public:
	Triangle() : Id{0}, vertices{new Vertex[3]} {
		for (int i = 0; i < 3; i++) {
			vertices[i] = std::make_pair(0, 0);
		}
	}

	Triangle(Vertex a, Vertex b, Vertex c, int id) : Id{id},
	                                                 vertices{new Vertex[3]} {
		vertices[0] = a;
		vertices[1] = b;
		vertices[2] = c;
		double AB = Geometry::PointDistance(a, b), BC =
		Geometry::PointDistance(b, c), AC = Geometry::PointDistance(a, c);
		if (AB >= BC + AC || BC >= AB + AC || AC >= AB + BC) {
			throw std::logic_error("Points must not be on the same line.");
		}
	}

	double Area() const override {
		return Geometry::Area(vertices, 3);
	}

	Vertex Center() const override {
		return Geometry::Center(vertices, 3);
	}

	std::ostream &Print(std::ostream &out) const override{
		out << "Id: " << Id << "\n";
		out << "Figure: Triangle\n";
		out << "Coords:\n";
		for (int i = 0; i < 3; i++) {
			out << vertices[i] << "\n";
		}
		return out;
	}

	void Serialize(std::ofstream &os) const override{
		FigureType type = TRIANGLE;
		os.write((char *) &type, sizeof(type));
		os.write((char *) &Id, sizeof(Id));
		for (int i = 0; i < 3; i++) {
			os.write((char *) &(vertices[i].first),
				sizeof(vertices[i].first));
			os.write((char *) &(vertices[i].second),
				sizeof(vertices[i].second));
		}
	}

	void Deserialize(std::ifstream &is) override {
		is.read((char *) &Id, sizeof(Id));
		for (int i = 0; i < 3; i++) {
			is.read((char *) &(vertices[i].first),
				sizeof(vertices[i].first));
			is.read((char *) &(vertices[i].second),
				sizeof(vertices[i].second));
		}
	}
```

```
	int getId() const override {
		return Id;
	}
};

class Square : public Figure {
	using Vertex = std::pair<double, double>;
	int Id;
	Vertex *vertices;
public:
	Square() : Id{0}, vertices{new Vertex[4]} {
		for (int i = 0; i < 4; i++) {
			vertices[i] = std::make_pair(0, 0);
		}
	}

	Square(Vertex a, Vertex b, Vertex c, Vertex d, int id) :
					Id{id},	vertices{new Vertex[4]} {
		vertices[0] = a;
		vertices[1] = b;
		vertices[2] = c;
		vertices[3] = d;
		Geometry::Vector AB{ a, b }, BC{ b, c }, CD{ c, d }, DA{ d, a };
		if (!Geometry::VectorsAreParallel(DA, BC)) {
			std::swap(vertices[0], vertices[1]);
			AB = { vertices[0], vertices[1] };
			BC = { vertices[1], vertices[2] };
			CD = { vertices[2], vertices[3] };
			DA = { vertices[3], vertices[0] };
		}
		if (!Geometry::VectorsAreParallel(AB, CD)) {
			std::swap(vertices[1], vertices[2]);
			AB = { vertices[0], vertices[1] };
			BC = { vertices[1], vertices[2] };
			CD = { vertices[2], vertices[3] };
			DA = { vertices[3], vertices[0] };
		}

		if (AB * BC || BC * CD || CD * DA || DA * AB) {
			throw std::logic_error("The sides of the square should be perpendicular");
		}
		if (LengthVector(AB) != LengthVector(BC) || LengthVector(BC) != LengthVector(CD) || LengthVector(CD) != LengthVector(DA) || LengthVector(DA) != LengthVector(AB)) {
			throw std::logic_error("The sides of the square should be equal");
		}
		if (!LengthVector(AB) || !LengthVector(BC) || !LengthVector(CD) || !LengthVector(DA)) {
			throw std::logic_error("The sides of the square must be greater than zero");
		}
	}

	double Area() const override {
		return Geometry::Area(vertices, 4);
	}

	Vertex Center() const override {
		return Geometry::Center(vertices, 4);
	}

	std::ostream &Print(std::ostream &out) const override{
		out << "Id: " << Id << "\n";
		out << "Figure: Square\n";
		out << "Coords:\n";
		for (int i = 0; i < 4; i++) {
			out << vertices[i] << "\n";
		}
		return out;
	}

	void Serialize(std::ofstream &os) const override{
		FigureType type = SQUARE;
```

```
        os.write((char *) &type, sizeof(type));
        os.write((char *) &Id, sizeof(Id));
        for (int i = 0; i < 4; i++) {
            os.write((char *) &(vertices[i].first),
                sizeof(vertices[i].first));
            os.write((char *) &(vertices[i].second),
                sizeof(vertices[i].second));
        }
    }

    void Deserialize(std::ifstream &is) override {
        is.read((char *) &Id, sizeof(Id));
        for (int i = 0; i < 4; i++) {
            is.read((char *) &(vertices[i].first),
                sizeof(vertices[i].first));
            is.read((char *) &(vertices[i].second),
                sizeof(vertices[i].second));
        }
    }

    int getId() const override {
        return Id;
    }
};

class Rectangle : public Figure {
    using Vertex = std::pair<double, double>;
    int Id;
    Vertex *vertices;
public:
    Rectangle() : Id{0}, vertices{new Vertex[4]} {
        for (int i = 0; i < 4; i++) {
            vertices[i] = std::make_pair(0, 0);
        }
    }

    Rectangle(Vertex a, Vertex b, Vertex c, Vertex d, int id) :
                                    Id{id},      vertices{new Vertex[4]} {
        vertices[0] = a;
        vertices[1] = b;
        vertices[2] = c;
        vertices[3] = d;
        Geometry::Vector AB{ a, b }, BC{ b, c }, CD{ c, d }, DA{ d, a };
        if (!Geometry::VectorsAreParallel(DA, BC)) {
            std::swap(vertices[0], vertices[1]);
            AB = { vertices[0], vertices[1] };
            BC = { vertices[1], vertices[2] };
            CD = { vertices[2], vertices[3] };
            DA = { vertices[3], vertices[0] };
        }
        if (!Geometry::VectorsAreParallel(AB, CD)) {
            std::swap(vertices[1], vertices[2]);
            AB = { vertices[0], vertices[1] };
            BC = { vertices[1], vertices[2] };
            CD = { vertices[2], vertices[3] };
            DA = { vertices[3], vertices[0] };
        }

        if (AB * BC || BC * CD || CD * DA || DA * AB) {
            throw std::logic_error("The sides of the square should be perpendicular");
        }
        if (!LengthVector(AB) || !LengthVector(BC) || !LengthVector(CD) || !LengthVector(DA)) {
            throw std::logic_error("The sides of the square must be greater than zero");
        }
    }

    double Area() const override {
        return Geometry::Area(vertices, 4);
    }
```

```
	Vertex Center() const override {
		return Geometry::Center(vertices, 4);
	}

	std::ostream &Print(std::ostream &out) const override{
		out << "Id: " << Id << "\n";
		out << "Figure: Rectangle\n";
		out << "Coords:\n";
		for (int i = 0; i < 4; i++) {
			out << vertices[i] << "\n";
		}
		return out;
	}

	void Serialize(std::ofstream &os) const override{
		FigureType type = RECTANGLE;
		os.write((char *) &type, sizeof(type));
		os.write((char *) &Id, sizeof(Id));
		for (int i = 0; i < 4; i++) {
			os.write((char *) &(vertices[i].first),
				sizeof(vertices[i].first));
			os.write((char *) &(vertices[i].second),
				sizeof(vertices[i].second));
		}
	}

	void Deserialize(std::ifstream &is) override {
		is.read((char *) &Id, sizeof(Id));
		for (int i = 0; i < 4; i++) {
			is.read((char *) &(vertices[i].first),
				sizeof(vertices[i].first));
			is.read((char *) &(vertices[i].second),
				sizeof(vertices[i].second));
		}
	}

	int getId() const override {
		return Id;
	}
};

class Factory {
public:
	using Vertex = std::pair<double, double>;
	virtual std::shared_ptr<Figure> FigureCreate() const = 0;
	virtual std::shared_ptr<Figure> FigureCreate(Vertex *vertices, int id)
				const = 0;
};

class TriangleFactory : public Factory {
public:
	std::shared_ptr<Figure> FigureCreate() const override {
		return std::shared_ptr<Figure>(new Triangle{});
	}

	std::shared_ptr<Figure> FigureCreate(Vertex *vertices, int id) const
                                                                override {
		return std::shared_ptr<Figure>(new Triangle{vertices[0], vertices[1],
				vertices[2], id});
	}
};

class SquareFactory : public Factory {
public:
	std::shared_ptr<Figure> FigureCreate() const override {
		return std::shared_ptr<Figure>(new Square{});
	}

	std::shared_ptr<Figure> FigureCreate(Vertex *vertices, int id) const
                                                                override {
```

```
        return std::shared_ptr<Figure>(new Square{vertices[0], vertices[1],
                vertices[2], vertices[3], id});
    }
};

class RectangleFactory : public Factory {
public:
    std::shared_ptr<Figure> FigureCreate() const override {
        return std::shared_ptr<Figure>(new Rectangle{});
    }

    std::shared_ptr<Figure> FigureCreate(Vertex *vertices, int id) const
                                                                    override {
        return std::shared_ptr<Figure>(new Rectangle{vertices[0], vertices[1],
                vertices[2], vertices[3], id});
    }
};

#endif // FIGURES_H
```

editor.h:

```
#ifndef EDITOR_H
#define EDITOR_H

#include "document.h"
#include "command.h"
#include <stack>

class Editor {
public:
    Editor() : Doc(nullptr), History() {};

    void CreateDocument(const std::string &name) {
        Doc = std::make_shared<Document>(name);
    }

    void InsertPrimitive(FigureType type, std::pair<double, double> *vertices) {
        std::shared_ptr<Command> command = std::shared_ptr<Command>(
            new InsertCommand(type, vertices));
        command->SetDocument(Doc);
        command->Execute();
        History.push(command);
    }

    void RemovePrimitive(int id) {
        try {
            std::shared_ptr<Command> command = std::shared_ptr<Command>(new
RemoveCommand(id));
            command->SetDocument(Doc);
            command->Execute();
            History.push(command);
        } catch (std::exception &err) {
            std::cout << err.what() << "\n";
            throw;
        }
    }
```

```
    void SaveDocument(const std::string &filename) {
        Doc->Save(filename);
    }

    void LoadDocument(const std::string &filename) {
        Doc = std::make_shared<Document>(filename);
        Doc->Load(filename);
    }

    void Undo() {
        if (History.empty()) {
            throw std::logic_error("History is empty");
        }
        std::shared_ptr<Command> lastCommand = History.top();
        lastCommand->UnExecute();
        History.pop();
    }

    void PrintDocument() {
        Doc->Print();
    }

    bool DocumentExist() {
        return Doc != nullptr;
    }

    ~Editor() = default;

private:
    std::shared_ptr<Document> Doc;
    std::stack<std::shared_ptr<Command>> History;
};

#endif //EDITOR_H
```

command.h:

```
#ifndef COMMAND_H
#define COMMAND_H 1

#include "document.h"
#include <stack>

class Command {
protected:
    std::shared_ptr<Document> Doc;
public:
    virtual void Execute() = 0;
    virtual void UnExecute() = 0;
    virtual ~Command() = default;

    void SetDocument(std::shared_ptr<Document> doc) {
        Doc = doc;
```

```
    }
};

class InsertCommand : public Command {
public:
    InsertCommand(FigureType type, std::pair<double, double> *vertices) :
        Type{type}, Vertices{vertices} {};

    void Execute() override {
        Doc->InsertPrimitive(Type, Vertices);
    }

    void UnExecute() override {
        Doc->RemoveLastPrimitive();
    }

private:
    FigureType Type;
    std::pair<double, double> *Vertices;
};

class RemoveCommand : public Command {
public:
    RemoveCommand(int id) : Id(id), Pos(0), figure(nullptr) {};

    void Execute() override {
        if (Id > Doc->Id || Id < 1 || (Id == Doc->Id && Id== 1)) {
            throw std::out_of_range("Invalid id");
        }
        figure = Doc->GetFigure(Id);
        Pos = Doc->GetPos(Id);
        Doc->RemovePrimitive(Id);
    }

    void UnExecute() override {
        Doc->InsertPrimitive(Pos, figure);
    }

private:
    int Id;
    int Pos;
    std::shared_ptr<Figure> figure;
};

#endif // COMMAND_H
```

document.h:

```
#ifndef DOCUMENT_H
#define DOCUMENT_H 1

#include <fstream>
#include <list>
#include <stdexcept>
#include <string>
```

```
#include <algorithm>
#include <utility>
#include "figures.h"

class Document {
public:
	Document() : Id(1), Name(""), Buffer(0), triangleFactory(),
		squareFactory(), rectangleFactory() {};

	Document(std::string name) : Id(1), Name(std::move(name)), Buffer(0),
		triangleFactory(), squareFactory(), rectangleFactory() {};

	~Document() = default;

	void Rename(const std::string &newName) {
		Name = newName;
	}

	void Save(const std::string &filename) {
		SerializeImpl(filename);
	}

	void Load(const std::string &filename) {
		DeserializeImpl(filename);
	}

	void Print() {
		std::for_each(Buffer.begin(), Buffer.end(), [](std::shared_ptr<Figure>
			shape) {
			shape->Print(std::cout) << "\n";
		});
	}

	void RemovePrimitive(int id) {
		auto it = std::find_if(Buffer.begin(), Buffer.end(),
			[id](std::shared_ptr<Figure> shape) -> bool {
				return id == shape->getId();
			});

		if (it == Buffer.end()) {
			throw std::logic_error("Figure with this id doesn't exist");
		}

		Buffer.erase(it);
	}

	void InsertPrimitive(FigureType type, std::pair<double, double> *
                                                                    vertices) {
		switch (type) {
			case TRIANGLE:
				Buffer.push_back(triangleFactory.FigureCreate(vertices,
                                                                        Id++));
				break;
			case SQUARE:
				Buffer.push_back(squareFactory.FigureCreate(vertices,
                                                                        Id++));
				break;
			case RECTANGLE:
				Buffer.push_back(rectangleFactory.FigureCreate(vertices,
                                                                        Id++));
				break;
		}
	}

private:
	int Id;
	std::string Name;
	std::list<std::shared_ptr<Figure>> Buffer;
	TriangleFactory triangleFactory;
	SquareFactory squareFactory;
```

```
RectangleFactory rectangleFactory;

friend class InsertCommand;
friend class RemoveCommand;

void SerializeImpl(const std::string &filename) const {
	std::ofstream os(filename, std::ios::binary | std::ios::out);
	if (!os) {
		throw std::runtime_error("File is not opened");
	}
	size_t nameLen = Name.size();
	os.write((char *) &nameLen, sizeof(nameLen));
	os.write((char *) Name.c_str(), nameLen);
	for (const auto &shape : Buffer) {
		shape->Serialize(os);
	}
}

void DeserializeImpl(const std::string &filename) {
	std::ifstream is(filename, std::ios::binary | std::ios::in);
	if (!is) {
		throw std::runtime_error("File is not opened");
	}
	size_t nameLen;
	is.read((char *) &nameLen, sizeof(nameLen));
	char *name = new char[nameLen + 1];
	name[nameLen] = 0;
	is.read(name, nameLen);
	Name = std::string(name);
	delete [] name;
	FigureType type;
	while (true) {
		is.read((char *) &type, sizeof(type));
		if (is.eof()) {
			break;
		}
		switch (type) {
			case TRIANGLE:
				Buffer.push_back(triangleFactory.FigureCreate());
				break;
			case SQUARE:
				Buffer.push_back(squareFactory.FigureCreate());
				break;
			case RECTANGLE:
				Buffer.push_back(rectangleFactory.FigureCreate());
				break;
		}
		Buffer.back()->Deserialize(is);
	}
	Id = Buffer.size();
}

std::shared_ptr<Figure> GetFigure(int id) {
	/*if (id > Id || id == 0) {
		throw std::runtime_error("Invalid id");
	}*/
	auto it = std::find_if(Buffer.begin(), Buffer.end(),
		[id](std::shared_ptr<Figure> shape) -> bool {
		return id == shape->getId();
	});
	return *it;
}

int GetPos(int id) {
	auto it = std::find_if(Buffer.begin(), Buffer.end(),
		[id](std::shared_ptr<Figure> shape) -> bool {
			return id == shape->getId();
		});
	return std::distance(Buffer.begin(), it);
}
```

```
        void InsertPrimitive(int pos, std::shared_ptr<Figure> figure) {
                auto it = Buffer.begin();
                std::advance(it, pos);
                Buffer.insert(it, figure);
        }

        void RemoveLastPrimitive() {
                if (Buffer.empty()) {
                        throw std::logic_error("Document is empty");
                }
                Buffer.pop_back();
        }
};

#endif //DOCUMENT_H
```

3. Ссылка на репозиторий на GitHub.

https://github.com/SandraIgitova/oop_exercise_07/tree/master

4. Набор testcases.

test_01.test:

Menu
Create newDoc
Add T 0 0 0 1 1 1
Add S 0 0 1 1 0 1 1 0
Add R 2 2 0 0 0 2 2 0
Print
Remove 1
Print
Remove 2
Print
Remove 1
Print

test_02.test:

Create p
Add T -1 -1 0 0 0 -1
Add S 0 0 0 1 1 1 1 0
Add R 0 0 0 2 4 0 4 2
Print
Save newFile
Undo
Undo
Print
load
No
newFile

5. Результаты выполнения тестов.

Test_01.test:

```
Menu
Create
Load
Save <fileName>
Add <figureType>
Remove <figure ID>
Undo
Print


Document newDoc is created

Primitive is added

Primitive is added

Primitive is added

Id: 1
Figure: Triangle
Coords:
(0, 0)
(0, 1)
(1, 1)

Id: 2
Figure: Square
Coords:
(0, 0)
(0, 1)
(1, 1)
(1, 0)

Id: 3
Figure: Rectangle
Coords:
(2, 2)
(0, 2)
(0, 0)
(2, 0)


ERROR
Add primitive at id: 1

Id: 1
Figure: Triangle
Coords:
(0, 0)
(0, 1)
(1, 1)

Id: 2
Figure: Square
Coords:
(0, 0)
(0, 1)
(1, 1)
(1, 0)

Id: 3
Figure: Rectangle
Coords:
(2, 2)
```

(0, 2)
(0, 0)
(2, 0)

Add primitive at id: 2

Id: 1
Figure: Triangle
Coords:
(0, 0)
(0, 1)
(1, 1)

Id: 3
Figure: Rectangle
Coords:
(2, 2)
(0, 2)
(0, 0)
(2, 0)

ERROR
Add primitive at id: 1

Id: 1
Figure: Triangle
Coords:
(0, 0)
(0, 1)
(1, 1)

Id: 3
Figure: Rectangle
Coords:
(2, 2)
(0, 2)
(0, 0)
(2, 0)

test_02.txt:

Document p is created

Primitive is added

Primitive is added

Primitive is added

Id: 1
Figure: Triangle
Coords:
(-1, -1)
(0, 0)
(0, -1)

Id: 2
Figure: Square
Coords:
(0, 0)
(0, 1)
(1, 1)
(1, 0)

Id: 3
Figure: Rectangle
Coords:
(0, 2)

(0, 0)
(4, 0)
(4, 2)

Document save if file newFile

OK

OK

Id: 1
Figure: Triangle
Coords:
(-1, -1)
(0, 0)
(0, -1)

6. Объяснение результатов работы программы.

В проекте есть 6 файлов. Файл document.h, в котором реализован класс Document, содержащий следующие методы-члены:

- Конструкторы
- Деструктор
- Переименование
- Сохранение файла в бинарном виде
- Загрузка бинарного файла
- Добавление примитива в документ
- Удаление примитива из документа

И следующие переменные:

- Id документа. Нужен для удаления примитивов.
- Name. Имя документа.
- Buffer. Буфер для хранения указателей на фигуры.

Файл figures.h используется для представления фигур.
Файл factory.h необходим для создания графических примитивов.
Файл editor.h содержит основной функционал редактора.
Файл command.h содержит команды добавления и удаления.
Файл main.cpp основной файл, в котором находится функция main.

7. Вывод.

Выполняя данную лабораторную, я получила практические навыки в проектировании структуры классов приложения. На мой взгляд умение правильно проектировать классы приложения — это очень нужный навык, т. к. правильно структированные классы, на мой взгляд, добавляют гибкости программе, её гораздо легче будет исправлять.