Московский авиационный институт

(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа № 7

Тема: Проектирование структуры классов.

Студент: Кудинов Сергей

Преподаватель: Журавлев А.А.

Дата:

Оценка:

Москва, 2019

1. **Постановка задачи**

Спроектировать простейший графический векторный редактор, который поддерживает:

- Создание нового документа
- Импорт документа из файла
- Импорт документа в файл
- Создание графического примитива
- Удаление графического примитива
- Отображение документа на экране
- Реализация операции undo, отменяющей последнее действие

2. **Репозиторий github**

https://github.com/StormStudioAndroid2/oop_exercise_07

3. **Описание программы**

Реализован класс Figure с общим функционалом фигур, у него наследники – Cicrle, Trapeze, Rectangle, Rhombus, Polyline, Polyfigure.У каждого есть свой FigureBuilder. В классе Factory в зависимости от того, пустой builder или нет, будет производиться создание фигуры. В файле Loader.h хранится метод загрузки из файла. В файле Helper.h вспомогательные методы для определения, принадлежит точка внутренности фигуры или нет.

4. **Листинг программы**

main.cpp

```
#include<iostream>

#include <array>
#include <fstream>
#include <iostream>
#include <vector>
#include "model/Action.h"
#include "sdl.h"
#include "imgui.h"
#include "model/Rectangle.h"
#include "model/Circle.h"
#include "model/Loader.h"
#include "model/Polyfigure.h"
#include <stack>

#include "Factory.h"
void setBrush(Brush& brush) {

	ImGui::InputInt("Green", &brush.green);
```

```
        ImGui::InputInt("Blue", &brush.blue);

        ImGui::InputInt("Red", &brush.red);
        if (brush.green > 255) {
                brush.green = 255;
        }
        if (brush.blue > 255) {
                brush.blue = 255;
        }
        if (brush.red > 255) {
                brush.red = 255;
        }
}

int main() {
        Brush brush;
        Factory factory;
        std::stack<std::unique_ptr<Action>> history;

  sdl::renderer renderer("Editor");
  bool quit = false;
  std::unique_ptr<Action> action;
  std::vector<std::unique_ptr<Figure>> figures;
  std::unique_ptr<Builder> active_builder = nullptr;
  const int32_t file_name_length = 128;
  char file_name[file_name_length] = "";
  int32_t remove_id = 0;
  while(!quit){
    renderer.set_color(0,100,0);
       renderer.draw_line(20-1, 20, 20, 20+1);
       renderer.set_color(0, 0, 0);
    renderer.clear();
    sdl::event event;

    while(sdl::event::poll(event)){
      sdl::quit_event quit_event;
      sdl::mouse_button_event mouse_button_event;
      if(event.extract(quit_event)){
        quit = true;
        break;
      } else if(event.extract(mouse_button_event)) {
                if (factory.isBuilding()) {
                        std::unique_ptr<Figure> figure = facto-
ry.buildingFigure(mouse_button_event);
                        if (figure) {
                                figures.emplace_back(std::move(figure));
                                action = std::make_unique<CreateAction>();
                                history.push(std::move(action));
                                active_builder = nullptr;
                        }
                }
                else {
                        if (mouse_button_event.button() ==
sdl::mouse_button_event::right && mouse_button_event.type() ==
sdl::mouse_button_event::down) {
                                for (int i = 0; i < figures.size(); ++i) {
                                        if (figures[i]->isInside(vertex{
mouse_button_event.x(), mouse_button_event.y() })) {
                                                figures.erase(figures.begin() + i);
                                                break;
                                        }
                                }
```

```
                    }
                }
    }
}

for(const std::unique_ptr<Figure>& figure: figures){
  figure->render(renderer);
}

ImGui::Begin("Menu");
if(ImGui::Button("New canvas")){
  figures.clear();
}
ImGui::InputText("File name", file_name, file_name_length - 1);
if(ImGui::Button("Save")){
  std::ofstream os(file_name);
  if(os){
    for(const std::unique_ptr<Figure>& figure: figures){
      figure->save(os);
    }
  }
}
ImGui::SameLine();
if(ImGui::Button("Load")){
  std::ifstream is(file_name);
  if(is){
     loader loader;
     figures = loader.load(is);
  }
}
if(ImGui::Button("Rectangle")){
          factory.startBuildRectangle(brush);
}
   ImGui::SameLine();

   if (ImGui::Button("Trapeze")) {
          factory.startBuildTrapeze(brush);
   }
   ImGui::SameLine();

   if (ImGui::Button("Rhombus")) {
          factory.startBuildRhombus(brush);
   }
   ImGui::SameLine();

   if (ImGui::Button("Polyline")) {
          factory.startPolyBuildLine(brush);
   }
   ImGui::SameLine();

   if (ImGui::Button("Polyfigure")) {
          factory.startPolyBuildFigure(brush);
   }
   ImGui::SameLine();

   if (ImGui::Button("Circle")) {
          factory.startBuildCircle(brush);
   }

   if (ImGui::Button("undo")) {
          if (!history.empty()) {
                 action = std::move(history.top());
```

```
                        history.pop();
                        action->undoAction(figures);
                        action = nullptr;
                }
        }
        setBrush(brush);
        ImGui::InputInt("Remove_Id", &remove_id);

    if(ImGui::Button("Remove")) {
                if (figures.size() > remove_id) {
                std::unique_ptr<Figure> figure = std::move(figures[remove_id]);
                action = std::make_unique<DeleteAction>(std::move(figure),remove_id);
                history.push(std::move(action));
                        figures.erase(figures.begin()+remove_id);


                }
    }
    // Undo
    ImGui::End();

    renderer.present();
  }


}
```

Figure.h

```
#pragma once

#include "Builder.h"
struct vertex {
        int32_t x, y;

};
struct Brush {
        int green;
        int red;
        int blue;
        Brush() {
                red = 0;
                green = 0;
                blue = 0;
        }
};
struct Figure {
public:
        virtual void render(const sdl::renderer& renderer)  = 0;
        virtual void addBrush(Brush& brush)  = 0;
        virtual bool isInside(vertex& v) = 0;
        virtual void save(std::ostream& os) const = 0;
        virtual ~Figure() = default;
};
struct Builder {
public:

        virtual std::unique_ptr<Figure> add_vertex(const vertex& v) = 0;
        virtual ~Builder() = default;

};
struct Action
```

```
{
public:
	virtual void undoAction(std::vector<std::unique_ptr<Figure>>& figures) = 0;
	virtual ~Action() = default;

};
struct CreateAction : Action
{
public:
	void undoAction(std::vector<std::unique_ptr<Figure>>& figures) override {
		figures.erase(figures.end() - 1);
	}
	CreateAction() {}
	~CreateAction() {}
};
struct DeleteAction : Action
{
public:
	void undoAction(std::vector<std::unique_ptr<Figure>>& figures) override {
		figures.insert(figures.begin() + index, std::move(figure));
	}
	DeleteAction(std::unique_ptr<Figure> figure, size_t index) {
		this->figure = (std::move(figure));
		this->index = index;
	}
	~DeleteAction() {}
private:
	std::unique_ptr<Figure> figure;
	size_t index;
};
```

Loader.h

```
#ifndef D_LOADER_H
#define D_LOADER_H

#include<vector>
#include<memory>

#include "Figure.h"
#include "Rhombus.h"
#include "Rectangle.h"
#include "Trapeze.h"
#include "Polyline.h"
#include "Polyfigure.h"

struct loader {
	std::vector<std::unique_ptr<Figure>> load(std::ifstream& is) {
		std::string figure_name;
		std::vector<std::unique_ptr<Figure>> figures;
		while (is >> figure_name) {
			vertex v;

			if (figure_name == std::string("rhombus")) {
				std::array<vertex, 4> vertices;
				for (int32_t i = 0; i < 4; ++i) {
					is >> v.x >> v.y;
					vertices[i] = v;
				}
				Brush load_clr {};
				is >> load_clr.red >> load_clr.green >> load_clr.blue;
				figures.emplace_back(std::make_unique<Rhombus>(vertices));
				(*figures[figures.size() - 1]).addBrush(load_clr);
			}
```

```
                        else if (figure_name == std::string("rectangle")) {
                                std::array<vertex, 4> vertices;
                                for (int32_t i = 0; i < 4; ++i) {
                                        is >> v.x >> v.y;
                                        vertices[i] = v;
                                }
                                 Brush load_clr {};
                                is >> load_clr.red >> load_clr.green >> load_clr.blue;
                                fig-
ures.emplace_back(std::make_unique<Rectangle>(vertices));
                                (*figures[figures.size() - 1]).addBrush(load_clr);
                        }
                        else if (figure_name == std::string("trapeze")) {
                                std::array<vertex, 4> vertices;
                                for (int32_t i = 0; i < 4; ++i) {
                                        is >> v.x >> v.y;
                                        vertices[i] = v;
                                }
                                Brush load_clr {};
                                is >> load_clr.red >> load_clr.green >> load_clr.blue;
                                figures.emplace_back(std::make_unique<Trapeze>(vertices));
                                (*figures[figures.size() - 1]).addBrush(load_clr);
                        }
                        else if (figure_name == std::string("circle")) {
                                std::array<vertex, 2> vertices;
                                for (int i = 0; i < 2; ++i) {
                                        is >> v.x >> v.y;
                                        vertices[i] = v;
                                }
                                 Brush load_clr {};
                                is >> load_clr.red >> load_clr.green >> load_clr.blue;
                                figures.emplace_back(std::make_unique<Circle>(vertices));
                                (*figures[figures.size() - 1]).addBrush(load_clr);
                        }
                        else if (figure_name == std::string("polyline")) {
                                std::vector<vertex> vertices;
                                int count_v;
                                is >> count_v;
                                for (int i = 0; i < count_v; ++i) {
                                        is >> v.x >> v.y;
                                        vertices.push_back(v);
                                }
                                Brush load_clr {};
                                is >> load_clr.red >> load_clr.green >> load_clr.blue;
                                fig-
ures.emplace_back(std::make_unique<Polyline>(vertices));
                                (*figures[figures.size() - 1]).addBrush(load_clr);
                        }
                        else if (figure_name == std::string("polyfigure")) {
                                std::vector<vertex> vertices;
                                int count_v;
                                is >> count_v;
                                for (int i = 0; i < count_v; ++i) {
                                        is >> v.x >> v.y;
                                        vertices.push_back(v);
                                }
                                Brush load_clr{};
                                is >> load_clr.red >> load_clr.green >> load_clr.blue;
                                fig-
ures.emplace_back(std::make_unique<Polyfigure>(vertices));
                                (*figures[figures.size() - 1]).addBrush(load_clr);
                        }
```

```
                }
                return figures;
        }
        ~loader() = default;
};

#endif //D_LOADER_H
```

Trapeze.h

```
#pragma once
#include "Figure.h"

#include "Helper.h"


struct Trapeze : Figure {
public:

        Trapeze(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

        void render(const sdl::renderer& renderer)  override {
                renderer.set_color(brush.red, brush.green, brush.blue);
                for (int32_t i = 0; i < 4; ++i) {
                        renderer.draw_line(vertices_[i].x, vertices_[i].y,
                                vertices_[(i + 1) % 4].x, vertices_[(i + 1) % 4].y);
                }
                renderer.set_color(0, 0, 0);
        }
        void addBrush(Brush& brush)  override {
                this->brush.blue = brush.blue;
                this->brush.red = brush.red;
                this->brush.green = brush.green;
        }
        void save(std::ostream& os) const override {
                os << "trapeze\n";
                for (int32_t i = 0; i < 4; ++i) {
                        os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
                }
                os << brush.red << ' ' << brush.green << ' ' << brush.blue << '\n';

        }
        bool isInside(vertex& v) override {
        return isInside1(vertices_, v);
        }
        ~Trapeze() { }

private:
        std::array<vertex, 4> vertices_;
        Brush brush;

};



struct TrapezeBuilder : Builder {
public:
        std::unique_ptr<Figure> add_vertex(const vertex& v) {
                vertices_[n_] = v;
                n_ += 1;
                if (n_ != 4) {
                        return nullptr;
                }
```

```
                std::unique_ptr<Figure>  figure = std::make_unique<Trapeze>(vertices_);
                figure->addBrush(brush);
                return std::move(figure);
        }
        TrapezeBuilder(Brush& brush) {
                this->brush.blue = brush.blue;
                this->brush.red = brush.red;
                this->brush.green = brush.green;
        }
private:
        int32_t n_ = 0;
        std::array<vertex, 4> vertices_;
        Brush brush;
};
```

Circle.h

```
#pragma once
#include "Figure.h"
#include "math.h"
#include "sdl.h"

struct Circle : Figure {
public:

        Circle(const std::array<vertex, 2>& vertices) : vertices_(vertices) {}

        void render(const sdl::renderer& renderer)  override {
                renderer.set_color(brush.red, brush.green, brush.blue);
                ;
                DrawCircle(renderer, vertices_[0].x, vertices_[0].y, this-
>getLength(vertices_[0], vertices_[1]));
                renderer.set_color(0, 0, 0);
        }
        void addBrush(Brush& brush)  override {
                this->brush.blue = brush.blue;
                this->brush.red = brush.red;
                this->brush.green = brush.green;
        }
        void save(std::ostream& os) const override {
                os << "circle\n";
                for (int32_t i = 0; i < 2; ++i) {
                        os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
                }
                os << brush.red << ' ' << brush.green << ' ' << brush.blue << '\n';
        }

        bool isInside(vertex& v) override {
                int32_t radius = (vertices_[0].x - vertices_[1].x)*(vertices_[0].x -
vertices_[1].x) + (vertices_[0].y - vertices_[1].y)*(vertices_[0].y - vertices_[1].y);
                return ((vertices_[0].x - v.x)*(vertices_[0].x - v.x) + (vertices_[0].y
- v.y)*(vertices_[0].y - v.y) < radius);
        }
        ~Circle() { }

private:
        void DrawCircle(const sdl::renderer&renderer, int32_t centreX, int32_t centreY,
int32_t radius)
        {

int
points_count
= 999;
```

```
                double step = 2*M_PI / points_count;
                vertex first_point;
                first_point.x= centreX+radius;
                first_point.y= centreY;

                for (int i = 1; i < 1000; ++i) {
                    vertex new_point;
                    new_point.x = centreX+radius * cos(step * i);
                    new_point.y = centreY+radius * sin(step * i);
                    renderer.draw_line(first_point.x, first_point.y, new_point.x, new_point.y);
                    first_point = new_point;
                }
        int32_t getLength(const vertex& v1, const vertex& v2) {
                return (int32_t)sqrt((v1.x - v2.x)*(v1.x - v2.x) + (v1.y - v2.y)*(v1.y -
v2.y));
        }
        std::array<vertex, 2> vertices_;
        Brush brush;

};



struct CircleBuilder : Builder {
public:
        std::unique_ptr<Figure> add_vertex(const vertex& v) {
                vertices_[n_] = v;
                n_ += 1;
                if (n_ != 2) {
                        return nullptr;
                }
                std::unique_ptr<Figure>  figure = std::make_unique<Circle>(vertices_);
                figure->addBrush(brush);
                return std::move(figure);
        }
        CircleBuilder(Brush& brush) {
                this->brush.blue = brush.blue;
                this->brush.red = brush.red;
                this->brush.green = brush.green;
        }
private:
        int32_t n_ = 0;
        std::array<vertex, 2> vertices_;
        Brush brush;
};
Factory.h
#pragma once
#include "sdl.h"
#include "model/Rectangle.h"
#include "model/Rhombus.h"
#include "model/Trapeze.h"
#include "model/Circle.h"
#include "model/Polyline.h"
#include "model/Polyfigure.h"

struct Factory {
private:
         std::unique_ptr<Builder> builder;
         std::unique_ptr<Builder> polyBuilder;
```

```
public:
		std::unique_ptr<Figure> buildingFigure(sdl::mouse_button_event
mouse_button_event) {
			if (builder && mouse_button_event.button() ==
sdl::mouse_button_event::left && mouse_button_event.type() ==
sdl::mouse_button_event::down) {
				std::unique_ptr<Figure> figure =  builder-
>add_vertex(vertex{ mouse_button_event.x(), mouse_button_event.y() });
				if (figure) {
					this->builder = nullptr;
					return std::move(figure);
				}
			}
			if (polyBuilder && mouse_button_event.button() ==
sdl::mouse_button_event::left && mouse_button_event.type() ==
sdl::mouse_button_event::down) {
				std::unique_ptr<Figure> figure = polyBuilder-
>add_vertex(vertex{ mouse_button_event.x(), mouse_button_event.y() });
				figure = nullptr;
				return nullptr;
			}
			if (polyBuilder && mouse_button_event.button() ==
sdl::mouse_button_event::right && mouse_button_event.type() ==
sdl::mouse_button_event::down) {
				std::unique_ptr<Figure> figure = polyBuilder-
>add_vertex(vertex{ mouse_button_event.x(), mouse_button_event.y() });
				polyBuilder = nullptr;
				return std::move(figure);

			}
			return nullptr;
		}
		void startBuildFigure() {}
		void startBuildTrapeze(Brush& brush) {
			builder = std::make_unique<RectangleBuilder>(brush);


		}
		void startBuildRectangle(Brush& brush) {
			builder = std::make_unique<TrapezeBuilder>(brush);


		}
		void startBuildRhombus(Brush& brush) {
			builder = std::make_unique<RhombusBuilder>(brush);


		}
		void startBuildCircle(Brush& brush) {
			builder = std::make_unique<CircleBuilder>(brush);


		}
		void startPolyBuildLine(Brush& brush) {
			polyBuilder = std::make_unique<PolylineBuilder>(brush);


		}
		void startPolyBuildFigure(Brush& brush) {
			polyBuilder = std::make_unique<PolyfigureBuilder>(brush);
```

```
		}
		bool isBuilding() {
			return builder != nullptr || polyBuilder!=nullptr;
		}

};
```

Helper.h

```
#pragma once
#include "Figure.h"
#define INF 1000000;

int32_t max(int32_t a, int32_t b) {
	if (a > b) {
		return a;
	}
	return b;
}
int32_t min(int32_t a, int32_t b) {
	if (a < b) {
		return a;
	}
	return b;
}
bool onSegment(vertex p, vertex q, vertex r)
{
	if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
		q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
		return true;
	return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(vertex p, vertex q, vertex r)
{
	int val = (q.y - p.y) * (r.x - q.x) -
		(q.x - p.x) * (r.y - q.y);

	if (val == 0) return 0;  // colinear
	return (val > 0) ? 1 : 2; // clock or counterclock wise
}

// The function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool doIntersect(vertex p1, vertex q1, vertex p2, vertex q2)
{
	// Find the four orientations needed for general and
	// special cases
	int o1 = orientation(p1, q1, p2);
	int o2 = orientation(p1, q1, q2);
	int o3 = orientation(p2, q2, p1);
	int o4 = orientation(p2, q2, q1);

	// General case
	if (o1 != o2 && o3 != o4)
		return true;

	// Special Cases
	// p1, q1 and p2 are colinear and p2 lies on segment p1q1
```

```
        if (o1 == 0 && onSegment(p1, p2, q1)) return true;

        // p1, q1 and p2 are colinear and q2 lies on segment p1q1
        if (o2 == 0 && onSegment(p1, q2, q1)) return true;

        // p2, q2 and p1 are colinear and p1 lies on segment p2q2
        if (o3 == 0 && onSegment(p2, p1, q2)) return true;

        // p2, q2 and q1 are colinear and q1 lies on segment p2q2
        if (o4 == 0 && onSegment(p2, q1, q2)) return true;

        return false; // Doesn't fall in any of the above cases
}

// Returns true if the vertex p lies inside the polygon[] with n vertices
template<std::size_t SIZE>

bool isInside1(std::array<vertex, SIZE> polygon, vertex p)
{
        // There must be at least 3 vertices in polygon[]
        int n = polygon.size();
        if (n < 3)  return false;

        // Create a vertex for line segment from p to infinite
        vertex extreme;
        extreme.x = INF;
        extreme.y = p.y;

        // Count intersections of the above line with sides of polygon
        int count = 0, i = 0;
        do      {
                int next = (i + 1) % n;

                // Check if the line segment from 'p' to 'extreme' intersects
                // with the line segment from 'polygon[i]' to 'polygon[next]'
                if (doIntersect(polygon[i], polygon[next], p, extreme))
                {
                        // If the vertex 'p' is colinear with line segment 'i-next',
                        // then check if it lies on segment. If it lies, return true,
                        // otherwise false
                        if (orientation(polygon[i], p, polygon[next]) == 0)
                                return onSegment(polygon[i], p, polygon[next]);

                        count++;
                }
                i = next;
        } while (i != 0);

        // Return true if count is odd, false otherwise
        return count & 1;  // Same as (count%2 == 1)
}
bool isInside1(std::vector<vertex>& polygon, vertex p)
{
        // There must be at least 3 vertices in polygon[]
        int n = polygon.size();
        if (n < 3)  return false;

        // Create a vertex for line segment from p to infinite
        vertex extreme;
        extreme.x = INF;
        extreme.y = p.y;
```

```
        // Count intersections of the above line with sides of polygon
        int count = 0, i = 0;
        do {
                int next = (i + 1) % n;

                // Check if the line segment from 'p' to 'extreme' intersects
                // with the line segment from 'polygon[i]' to 'polygon[next]'
                if (doIntersect(polygon[i], polygon[next], p, extreme))
                {
                        // If the vertex 'p' is colinear with line segment 'i-next',
                        // then check if it lies on segment. If it lies, return true,
                        // otherwise false
                        if (orientation(polygon[i], p, polygon[next]) == 0)
                                return onSegment(polygon[i], p, polygon[next]);

                        count++;
                }
                i = next;
        } while (i != 0);

        // Return true if count is odd, false otherwise
        return count & 1;  // Same as (count%2 == 1)
}
```

Вывод

Я изучил создание проектов со сложной архитектурой, также создал Gui с помощью библиотеки и использовал принципы ООП для построения проекта

Список литературы

1. Шилдт, Герберт. C++: базовый курс, 3-е изд. : Пер. с англ. - М. : ООО “И.Д. Вильямс”, 2018. - 624 с. : ил. - Парал. тит. англ.