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 <array>

#include <vector>
#include "sdl.h"
#include "imgui.h"
#include "Loader.h"
#include <stack>
#include "Program.h"
void setBrush(Program& program) {

	ImGui::InputInt("Blue", &(program.getBrush()->blue));

	ImGui::InputInt("Green", &(program.getBrush()->green));

	ImGui::InputInt("Red", &(program.getBrush()->red));
	if (program.getBrush()->red > 255) {
		program.getBrush()->red = 255;
	}
	if (program.getBrush()->blue > 255) {
```

```
                program.getBrush()->blue = 255;
        }
        if (program.getBrush()->green > 255) {
                program.getBrush()->green = 255;
        }
        if (program.getBrush()->red < 0) {
                program.getBrush()->red = 0;
        }
        if (program.getBrush()->blue < 0) {
                program.getBrush()->blue = 0;
        }
        if (program.getBrush()->green < 0) {
                program.getBrush()->green = 0;
        }
}

int main() {
        Brush brush;
        Program program;
  sdl::renderer renderer("Editor");
  bool quit = false;
  std::unique_ptr<Action> action;
  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)) {
                program.mouseClickListener(mouse_button_event);
      }
    }

       program.render(renderer);

    ImGui::Begin("Menu");
    if(ImGui::Button("New canvas")){
                program.clear();

       }
    ImGui::InputText("File name", file_name, file_name_length - 1);
    if(ImGui::Button("Save")) {
                program.saveFile(file_name);
    }
    ImGui::SameLine();
    if(ImGui::Button("Load")){
                program.loadFile(file_name);
    }
    if(ImGui::Button("Rectangle")){
                program.addRectangle();
    }
```

```
        ImGui::SameLine();

        if (ImGui::Button("Trapeze")) {
                program.addTrapeze();
        }
        ImGui::SameLine();

        if (ImGui::Button("Rhombus")) {
                program.addRhombus();
        }
        ImGui::SameLine();

        if (ImGui::Button("Polyline")) {
                program.addPolyline();
        }
        ImGui::SameLine();

        if (ImGui::Button("Polyfigure")) {
                program.addPolyfigure();
        }
        ImGui::SameLine();

        if (ImGui::Button("Circle")) {
                program.addCircle();
        }

        if (ImGui::Button("undo")) {
                program.undo();
        }
        setBrush(program);
        ImGui::InputInt("Remove_Id", &remove_id);

    if(ImGui::Button("Remove")) {
                program.remove(remove_id);
    }
    // Undo
    ImGui::End();

    renderer.present();
  }
  return 0;

}
```

Figure.h

```
pragma once

#include <memory>
#include "sdl.h"
#include <array>
#include <vector>
#include <math.h>
#define INF INT16_MAX;
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;
};
struct DeleteAction : Action
{
public:
	void undoAction(std::vector<std::unique_ptr<Figure>>& figures) override;
	explicit DeleteAction(std::unique_ptr<Figure> figure, size_t index);
private:
	std::unique_ptr<Figure> figure;
	size_t index;
};

bool onSegment(vertex p, vertex q, vertex r);

// 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);

bool doIntersect(vertex p1, vertex q1, vertex p2, vertex q2);


bool isInside1(std::vector<vertex>& polygon, vertex p);
```

Figure.cpp

```
#include "Figure.h"
```

```
	void CreateAction::undoAction(std::vector<std::unique_ptr<Figure>>& figures)  {
		figures.erase(figures.end() - 1);
	}


	void DeleteAction::undoAction(std::vector<std::unique_ptr<Figure>>& figures)  {
		figures.insert(figures.begin() + index, std::move(figure));
	}
	DeleteAction::DeleteAction(std::unique_ptr<Figure> figure, size_t index) {
		this->figure = (std::move(figure));
		this->index = index;
	}


bool onSegment(vertex p, vertex q, vertex r)
{
	if (q.x <= std::max(p.x, r.x) && q.x >= std::min(p.x, r.x) &&
		q.y <= std::max(p.y, r.y) && q.y >= std::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


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)
}
```

Loader.h

```
#ifndef D_LOADER_H
#define D_LOADER_H
#pragma once
#include<vector>
#include<memory>
#include "Circle.h"
#include "Figure.h"
#include "Rhombus.h"
#include "Rectangle.h"
#include "Trapeze.h"
#include "Polyline.h"
#include "Polyfigure.h"
#include <iostream>
#include <fstream>
struct loader {
	std::vector<std::unique_ptr<Figure>> load(std::ifstream& is);
	~loader() = default;
};
```

```
#endif //D_LOADER_H
Loader.cpp
#include "Loader.h"

std::vector<std::unique_ptr<Figure>> loader::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;
			figures.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;
                        figures.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;
                        figures.emplace_back(std::make_unique<Polyfigure>(vertices));
                        (*figures[figures.size() - 1]).addBrush(load_clr);
                }
        }
        return figures;
}
```

Program.h

```
#pragma once
#include "Figure.h"
#include "Factory.h"
#include <vector>
#include <fstream>
#include <iostream>
#include "Loader.h"
#include <stack>
struct Program {
private:
        std::stack<std::unique_ptr<Action>> history;
        Brush brush;
        std::vector<std::unique_ptr<Figure>> figures;
    Factory factory;
public:

         void undo();
         void clear();
         void render(sdl::renderer& renderer);
         void addRectangle();
         void addTrapeze();
         void addRhombus();
         void addPolyline();
         void addPolyfigure();
         void addCircle();
         void mouseClickListener(sdl::mouse_button_event mouse_button_event);
         void remove(int remove_id);
         void saveFile(char file_name[128] );
         void loadFile(char file_name[128]);
         Brush* getBrush();
};
```

Program.cpp

```
#include "Program.h"

void Program::undo() {
        if (!history.empty()) {
                std::unique_ptr<Action> action = std::move(history.top());
                history.pop();
                action->undoAction(figures);
                action = nullptr;
```

```
	}
 }
void Program::clear() {
	figures.clear();
	factory.clear();
	while (!history.empty()) {
		history.pop();
	}
}
void Program::remove(int remove_id) {
	if (figures.size() > remove_id && remove_id>=0) {
		std::unique_ptr<Figure> figure = std::move(figures[remove_id]);
		std::unique_ptr<Action> action =
std::make_unique<DeleteAction>(std::move(figure), remove_id);
		history.push(std::move(action));
		figures.erase(figures.begin() + remove_id);


	}
}
 void Program::render(sdl::renderer& renderer) {
	 for (const std::unique_ptr<Figure>& figure : figures) {
		figure->render(renderer);
	 }
}
 void Program::mouseClickListener(sdl::mouse_button_event 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));
			std::unique_ptr<Action> action =
std::make_unique<CreateAction>();
			history.push(std::move(action));
		}
	 }
	 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() })) {
					std::unique_ptr<Action> action =
std::make_unique<DeleteAction>(std::move(figures[i]),i);
					history.push(std::move(action));
					figures.erase(figures.begin() + i);
					break;
				}
			}
		}
	 }
 }

 void Program::saveFile(char file_name[128]) {
	 std::ofstream os(file_name);
	 if (os) {
		for (const std::unique_ptr<Figure>& figure : figures) {
			figure->save(os);
		}
	 }
 }
 void Program::loadFile(char file_name[128]) {
```

```
        std::ifstream is(file_name);
        if (is) {
                loader loader;
                figures = loader.load(is);
        }
}
Brush* Program::getBrush() {
        return &(this->brush);

}
void Program::addRectangle() {
        factory.startBuildRectangle(brush);
        }
void Program::addTrapeze() {
        factory.startBuildTrapeze(brush);
        }
void Program::addRhombus() {
        factory.startBuildRhombus(brush);
        }
void Program::addPolyline() {
        factory.startPolyBuildLine(brush);
        }
void Program::addPolyfigure() {
        factory.startPolyBuildFigure(brush);
        }
void Program::addCircle() {
        factory.startBuildCircle(brush);
        }
```

Trapeze.h

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



struct Trapeze : Figure {
public:

        explicit Trapeze(const std::array<vertex, 4>& vertices);

        void render(const sdl::renderer& renderer);
        void addBrush(Brush& brush)  override;
        void save(std::ostream& os) const override;
        bool isInside(vertex& v);



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

};



struct TrapezeBuilder : Builder {
public:
        std::unique_ptr<Figure> add_vertex(const vertex& v);
        explicit TrapezeBuilder(Brush& brush);
private:
        int32_t n_ = 0;
```

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

Trapeze.cpp

```
#include "Trapeze.h"



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

	void Trapeze::render(const sdl::renderer& renderer)   {
		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 Trapeze::addBrush(Brush& brush)   {
		this->brush.blue = brush.blue;
		this->brush.red = brush.red;
		this->brush.green = brush.green;
	}
	void Trapeze::save(std::ostream& os) const  {
		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 Trapeze::isInside(vertex& v)  {

		std::vector<vertex> vect(this->vertices_.begin(), this-
>vertices_.end());;

		return isInside1(vect, v);
	}
	std::unique_ptr<Figure> TrapezeBuilder::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::TrapezeBuilder(Brush& brush) {
		this->brush.blue = brush.blue;
		this->brush.red = brush.red;
		this->brush.green = brush.green;
	}
```

Circle.h

```
#pragma once
#include "Figure.h"
#include "math.h"
struct Circle : Figure {
```

```
public:

	explicit Circle(const std::array<vertex, 2>& vertices);

	void render(const sdl::renderer& renderer)  override;
	void addBrush(Brush& brush)  override;
	void save(std::ostream& os) const override;

	bool isInside(vertex& v) override;

private:
	void DrawCircle(const sdl::renderer&renderer, int32_t centreX, int32_t centreY,
int32_t radius);
	int32_t getLength(const vertex& v1, const vertex& v2);
	std::array<vertex, 2> vertices_;
	Brush brush;

};



struct CircleBuilder : Builder {
public:
	std::unique_ptr<Figure> add_vertex(const vertex& v);
	explicit CircleBuilder(Brush& brush);
private:
	int32_t n_ = 0;
	std::array<vertex, 2> vertices_;
	Brush brush;
};
Circle.cpp
#include "Circle.h"
#include "math.h"

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

	void Circle::render(const sdl::renderer& renderer)   {
		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 Circle::addBrush(Brush& brush)   {
		this->brush.blue = brush.blue;
		this->brush.red = brush.red;
		this->brush.green = brush.green;
	}
	void Circle::save(std::ostream& os) const  {
		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 Circle::isInside(vertex& v)  {
		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);
	}
```

```
	void Circle::DrawCircle(const sdl::renderer&renderer, int32_t centreX, int32_t
centreY, int32_t radius)
	{
		const int32_t diameter = (radius * 2);

		int32_t x = (radius - 1);
		int32_t y = 0;
		int32_t tx = 1;
		int32_t ty = 1;
		int32_t error = (tx - diameter);

		while (x >= y)
		{
			//  Each of the following renders an octant of the circle
			renderer.draw_line(centreX + x, centreY - y, centreX + x + 1,
centreY - y + 1);
			renderer.draw_line(centreX + x, centreY + y, centreX + x + 1,
centreY + y + 1);
			renderer.draw_line(centreX - x, centreY - y, centreX - x + 1,
centreY - y + 1);
			renderer.draw_line(centreX - x, centreY + y, centreX - x + 1,
centreY + y + 1);
			renderer.draw_line(centreX + y, centreY - x, centreX + y + 1,
centreY - x + 1);
			renderer.draw_line(centreX + y, centreY + x, centreX + y + 1,
centreY + x + 1);
			renderer.draw_line(centreX - y, centreY - x, centreX - y + 1,
centreY - x + 1);
			renderer.draw_line(centreX - y, centreY + x, centreX - y + 1,
centreY + x + 1);

			if (error <= 0)
			{
				++y;
				error += ty;
				ty += 2;
			}

			if (error > 0)
			{
				--x;
				tx += 2;
				error += (tx - diameter);
			}
		}
	}
	int32_t Circle::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::unique_ptr<Figure> CircleBuilder::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::CircleBuilder(Brush& brush) {
		this->brush.blue = brush.blue;
		this->brush.red = brush.red;
		this->brush.green = brush.green;
	}
```

Factory.h

```
#pragma once
#include "sdl.h"
#include "Rectangle.h"
#include "Rhombus.h"
#include "Trapeze.h"
#include "Circle.h"
#include "Polyline.h"
#include "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);
		void startBuildFigure() {}
		void startBuildTrapeze(Brush& brush);
		void startBuildRectangle(Brush& brush);
		void startBuildRhombus(Brush& brush);
		void startBuildCircle(Brush& brush);
		void startPolyBuildLine(Brush& brush);
		void startPolyBuildFigure(Brush& brush);
		bool isBuilding();
		void clear();

};
```

Factory.cpp

```
#include "Factory.h"


	std::unique_ptr<Figure> Factory::buildingFigure(sdl::mouse_button_event
mouse_button_event) {
		vertex v1;
		v1.x = mouse_button_event.x();
		v1.y = mouse_button_event.y();
		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(v1);
			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(v1);
			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(v1);
                        polyBuilder = nullptr;
                        return std::move(figure);

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


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


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


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


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


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


        }
        bool Factory::isBuilding() {
                return builder != nullptr || polyBuilder != nullptr;
        }
        void Factory::clear() {
                builder = nullptr;
                polyBuilder = nullptr;
        }
```

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.