Лабораторная работа №5
Исследование способов применения структурных паттернов проектирования при рефакторинге ПО

Цель работы

Исследовать возможность использования структурных паттернов проектирования. Получить практические навыки применения структурных паттернов при объектно-ориентированном проектировании и рефакторинге ПО.

Постановка задачи

1. Ознакомиться с основными преимуществами объектно-ориентированного проектирования на основе паттернов, изучить порядок проектирования с использованием паттернов. Изучить назначение и структуру паттерна Адаптер (выполнить в ходе самостоятельной подготовки).
2. Применительно к программному продукту, выбранному для рефакторинга, проанализировать возможность использования паттерна Адаптер. Для этого построить диаграмму классов, на диаграмме классов найти класс-клиент и адаптируемый класс, функциональностью которого должен воспользоваться клиент.
3. Выполнить перепроектирование системы, использовав паттерн Адаптер, изменения отобразить на диаграмме классов.
4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

5. На основе полученной UML-диаграммы модифицировать программный код, скомпилировать программу, выполнить ее тестирование и продемонстрировать ее работоспособность.

Ход работы

В качестве структурного паттерна для изучения был выбран паттерн «Декоратор».

Изначально имелась программа, которая позволяла «отрисовывать» различные фигуры: круг, квадрат и треугольник.

Появилась необходимость в добавлении данным фигурам цветной обводки и заливки. Без использования паттерна декоратора, чтобы достичь данной цели, необходимо было бы создавать много новых классов для каждой фигуры, например OutlineCircle и ColorCircle, что является не очень хорошим подходом к проектированию приложений.

В связи с этим было решено применить паттерн «Декоратор», который позволяет добавлять объектам новую функциональность, «оборачивая» их в обёртки.

Изначально все фигуры наследовались от базового класса Shape, поэтому можно обернуть этот класс классом ShapeDecorator, от которого в свою очередь будут наследоваться классы для добавления функциональности с обводкой OutlineDecorator и с разукрашиванием фигур ColorDecorator,

Получившийся в результате применения паттерна «Декоратор» код:

```
#include <iostream>
#include <string>

class DrawingAPI
{
public:
    void drawCircle(int x, int y, int radius)
    {
      std::cout << " API.circle at " << x << "," << y << " with radius
" << radius << "\n";
    };
```

```
        virtual void drawSquare(int x, int y, int side)
        {
            std::cout << " API.square at " << x << "," << y << " with side
" << side << "\n";
        };
       virtual void drawTriangle(int x1, int y1, int x2, int y2, int x3, int
y3)
        {
           std::cout << " API.triangle with points (" << x1 << "," << y1 <<
"), (" << x2 << "," << y2 << "), (" << x3 << "," << y3 << ")\n";
        };
    };

    class Shape
    {
    protected:
        DrawingAPI *drawingAPI;

    public:
        Shape(DrawingAPI *drawingAPI) : drawingAPI(drawingAPI) {}
        virtual void draw() = 0;
        virtual ~Shape() {}
    };

    class ShapeDecorator : public Shape
    {
    protected:
        Shape *decoratedShape;

    public:
        ShapeDecorator(DrawingAPI  *drawingAPI,  Shape  *decoratedShape)  :
Shape(drawingAPI), decoratedShape(decoratedShape) {}
        virtual void draw() override
        {
            decoratedShape->draw();
        }
    };

    class OutlineDecorator : public ShapeDecorator
    {
    private:
        std::string outlineColor;

    public:
        OutlineDecorator(DrawingAPI  *drawingAPI,  Shape  *decoratedShape,
std::string outlineColor)
            :          ShapeDecorator(drawingAPI,          decoratedShape),
outlineColor(outlineColor) {}

        void draw() override
        {
            std::cout << " Outline color: " << outlineColor;
            decoratedShape->draw();
        }
    };
```

```
class ColorDecorator : public ShapeDecorator
{
private:
    std::string fillColor;

public:
    ColorDecorator(DrawingAPI *drawingAPI, Shape *decoratedShape,
std::string fillColor)
      : ShapeDecorator(drawingAPI, decoratedShape), fillColor(fillColor)
{}

    void draw() override
    {
        std::cout << "Fill color: " << fillColor;
        decoratedShape->draw();
    }
};

class Circle : public Shape
{
private:
    int x, y, radius;

public:
    Circle(DrawingAPI *drawingAPI, int x, int y, int radius) :
Shape(drawingAPI), x(x), y(y), radius(radius) {}

    void draw() override
    {
        drawingAPI->drawCircle(x, y, radius);
    }
};

class Square : public Shape
{
private:
    int x, y, side;

public:
    Square(DrawingAPI *drawingAPI, int x, int y, int side) :
Shape(drawingAPI), x(x), y(y), side(side) {}

    void draw() override
    {
        drawingAPI->drawSquare(x, y, side);
    }
};

class Triangle : public Shape
{
private:
    int x1, y1, x2, y2, x3, y3;

public:
```

```
        Triangle(DrawingAPI *drawingAPI, int x1, int y1, int x2, int y2, int
x3, int y3) : Shape(drawingAPI), x1(x1), y1(y1), x2(x2), y2(y2), x3(x3), y3(y3)
{}

        void draw() override
        {
            drawingAPI->drawTriangle(x1, y1, x2, y2, x3, y3);
        }
    };

    int main()
    {
        DrawingAPI *drawingAPI = new DrawingAPI();

        Circle *circle = new Circle(drawingAPI, 10, 20, 15);
        Square *square = new Square(drawingAPI, 50, 60, 20);
       Triangle *triangle = new Triangle(drawingAPI, 100, 110, 120, 130, 140,
150);

        OutlineDecorator *circleOutline = new OutlineDecorator(drawingAPI,
circle, "red");
        ColorDecorator   *circleColor   =   new   ColorDecorator(drawingAPI,
circleOutline, "blue");

        OutlineDecorator *squareOutline = new OutlineDecorator(drawingAPI,
square, "green");
        ColorDecorator   *squareColor   =   new   ColorDecorator(drawingAPI,
squareOutline, "yellow");

       OutlineDecorator *triangleOutline = new OutlineDecorator(drawingAPI,
triangle, "purple");
        ColorDecorator  *triangleColor  =  new  ColorDecorator(drawingAPI,
triangleOutline, "orange");

        circleColor->draw();
        squareColor->draw();
        triangleColor->draw();

        delete circle;
        delete square;
        delete triangle;
        delete circleOutline;
        delete squareOutline;
        delete triangleOutline;
        delete circleColor;
        delete squareColor;
        delete triangleColor;
        delete drawingAPI;

        return 0;
    }
```

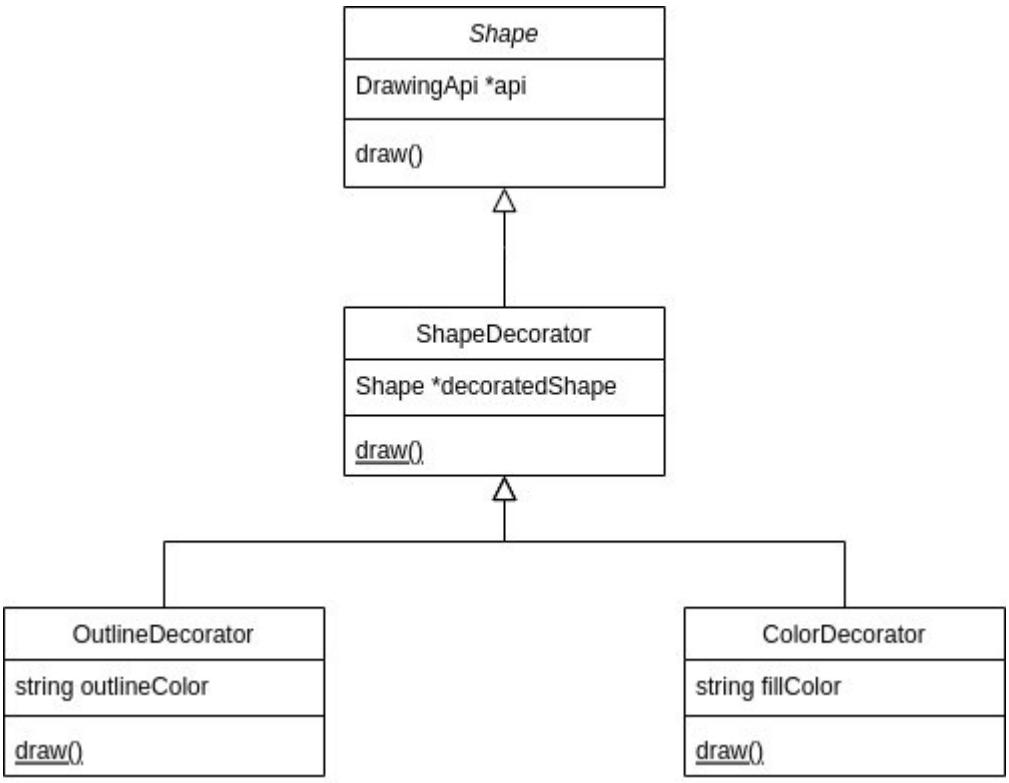

Рисунок 1 – Диаграмма классов после применения паттерна Декоратор

ВЫВОДЫ

В ходе выполнения лабораторной работы были исследованы возможности использования структурных паттернов проектирования. Получены практические навыки применения структурных паттернов при объектно-ориентированном проектировании и рефакторинге ПО.

Был применен структурный паттерн декоратор, который позволил добавить в программу новую функциональность, обернув базовый класс в новые, с необходимой функциональностью. Применение такого паттерна помогло избежать разрастания количества классов в программе, также получившаяся структура стала понятной и легко читаемой.