Лабораторная работа №7
Исследование способов применения порождающих паттернов проектирования при рефакторинге ПО

Цель работы

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

Постановка задачи

1. Изучить назначение и структуру паттерна Абстрактная фабрика (выполнить в ходе самостоятельной подготовки).

2. Применительно к программному продукту, выбранному для рефакторинга, проанализировать возможность использования паттерна Абстрактная фабрика. Для этого построить диаграмму классов, на диаграмме классов выделить семейства взаимосвязанных и совместно используемых классов, которые должны инстанцироваться совместно и при этом инстанцирующий их клиент не должен быть привязан к конкретным именам классов (пример приведен в разделе 2.2.).

3. Выполнить перепроектирование системы, использовав паттерн Абстрактная фабрика, 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 порождающего паттерна был выбран паттерн «Фабричный метод», при помощи которого можно определить интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Была выбрана программа отображающая различные геометрические фигуры. Был разработан класс фабрики форм этих фигур ShapeFactory, от которого были разработаны фабрики под каждую из фигур: CircleFactory, SquareFactory, TriangleFactory. В дальнейшем, при расширении программы можно будет продолжать данную иерархию, что упрощает добавление новых фигур в код программы.

Разработанный код представлен ниже:

```
#include <iostream>
#include <memory>

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:
    std::shared_ptr<DrawingAPI> drawingAPI;

public:
    Shape(const std::shared_ptr<DrawingAPI> &drawingAPI) :
drawingAPI(drawingAPI) {}
    virtual void draw() = 0;
};

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

public:
    Circle(const std::shared_ptr<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(const std::shared_ptr<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(const std::shared_ptr<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);
        }
    };

    class ShapeFactory
    {
    public:
        virtual          std::unique_ptr<Shape>          createShape(const
std::shared_ptr<DrawingAPI> &drawingAPI) = 0;
    };

    class CircleFactory : public ShapeFactory
    {
    public:
       std::unique_ptr<Shape> createShape(const std::shared_ptr<DrawingAPI>
&drawingAPI) override
        {
            return std::make_unique<Circle>(drawingAPI, 10, 20, 15);
        }
    };

    class SquareFactory : public ShapeFactory
    {
    public:
       std::unique_ptr<Shape> createShape(const std::shared_ptr<DrawingAPI>
&drawingAPI) override
        {
            return std::make_unique<Square>(drawingAPI, 50, 60, 20);
        }
    };

    class TriangleFactory : public ShapeFactory
    {
    public:
       std::unique_ptr<Shape> createShape(const std::shared_ptr<DrawingAPI>
&drawingAPI) override
        {
          return std::make_unique<Triangle>(drawingAPI, 100, 110, 120, 130,
140, 150);
        }
    };

    int main()
    {
        std::shared_ptr<DrawingAPI>              drawingAPI              =
std::make_shared<DrawingAPI>();

        std::unique_ptr<ShapeFactory>            circleFactory           =
std::make_unique<CircleFactory>();
        std::unique_ptr<ShapeFactory>            squareFactory           =
std::make_unique<SquareFactory>();
        std::unique_ptr<ShapeFactory>           triangleFactory          =
std::make_unique<TriangleFactory>();
```

```
        std::unique_ptr<Shape> circle = circleFactory->createShape(drawingAPI);
        std::unique_ptr<Shape> square = squareFactory->createShape(drawingAPI);
        std::unique_ptr<Shape> triangle = triangleFactory->createShape(drawingAPI);

        circle->draw();
        square->draw();
        triangle->draw();

        return 0;
    }
```

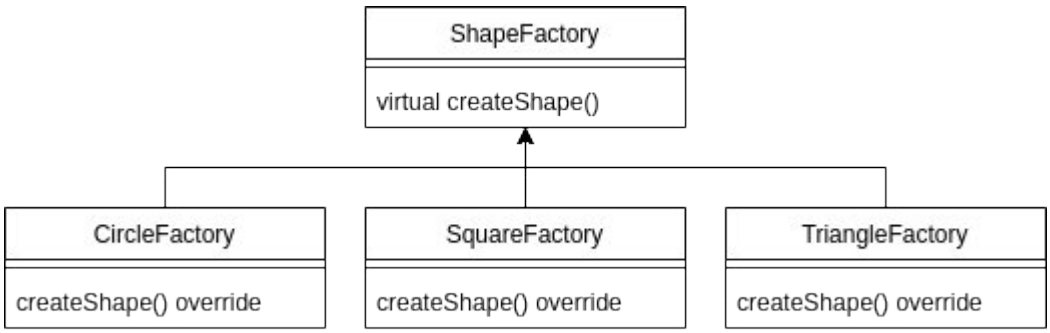

Рисунок 1 – Диаграмма классов

ВЫВОДЫ

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

Был применен паттер «Фабричный метод», при помощи которого были добавлены фабрики для создания фигур различной формы. Применение данного паттерна также облегчит работу при дальнейшем расширении программы, если понадобится добавить новые фигуры, то достаточно будет только создать соответствующие классы фабрик.