SVG Files

от Александър Илиев Девинизов

ФН: 2MI0800684, група 3, курс 1

1.Увод

1.1. Описание и идея на проекта

Проектът представлява конзолна програма, която позволява на потребителя да въвежда и обработва SVG файлове (Scalable Vector Graphics). Програмата предоставя възможности за създаване, изтриване, манипулиране и извеждане на данни от файлове, съдържащи базови фигури като правоъгълник, кръг и елипса, описани във формат SVG (вж. Пример 1). Използван е команден шаблон (*Command Pattern*), който разделя командите и капсулира тяхната логика. Програмата е написана в обектно-ориентиран стил.

(*Пример 1: <rect x="1" y="2" width="10" height="30" fill="red" />*).

1.2. Цел и задади на разработката

Проектът трябва да може да приема данни от файл с разширение .svg, като ги прочита и запазва в подходяща структура. Освен това трябва да поддържа операции за създаване и манипулиране на фигури: create, erase, print, translate и within. Програмата трябва също да предоставя възможност за работа с файлове – отваряне, затваряне, запазване в текущ файл и запазване като нов файл. Тя трябва да бъде реализирана с обектно-ориентиран подход и да изпълнява тестове с помощта на библиотеката doctest.

1.3. Структура на документацията

- 1.Увод

    1.1. Описание и идея на проекта

    1.2. Цел и задачи на разработка

    1.3. Структура на документация

- 2. Преглед на предметната област

    2.1. Основни дефиниции, концепции и алгоритми, които ще бъдат използвани

    2.2. Дефиниране на проблеми и сложност на поставената задача

2.3. Подходи, методи (евентуално модели и стандарти) за решаването на поставените проблеми

2.4. Потребителски (функционални) изисквания (права, роли, статуси, диаграми...) и качествени (нефункционални) изисквания (скалируемост, поддръжка…)

- 3. Проектиране

3.1. Обща архитектура - ООП дизайн

3.2. Диаграми (на структура и поведение - по обекти, слоеве с най-важните извадки от кода)

- 4. Реализация, тестване

4.1. Реализация на класове (включва важни моменти от реализацията на класовете и малки фрагменти от кода)

4.2. Управление на паметта и алгоритми. Оптимизация

4.3. Планиране, описание и създаване на тестови сценарии (създаване на примери)

- 5. Заключение

5.1. Обобщение на изпълнението на началните цели

5.2. Насоки за бъдещо развитие и усъвършенстване

2. Преглед на предметната област

2.1. Основни дефиниции, концепции и алгоритми, които ще бъдат използвани

SVG (или Scalable Vector Graphics) е език, базиран на XML, предназначен за описание на двумерна векторна графика. Всеки SVG елемент представлява графичен обект с определени свойства като начална позиция, размер, цвят и други визуални атрибути. В рамките на този проект се използват трите базови фигури — правоъгълник (rect), кръг (circle) и елипса (ellipse), като ще използваме най-важните атрибути — дефиниране на фигурата и цвят.

За обработка на потребителски команди се използва команден шаблон (*Command Pattern*). Всяка команда се реализира чрез отделен клас, наследяващ “*Command*”, който инкапсулира логиката на съответната операция. Освен това се използва фабричен метод (*factory*), който създава подходящ обект в зависимост от типа, въведен от потребителя.

Въведените команди се анализират и разделят на части (*tokens*) чрез помощна функция. Първата част указва типа фигура, която се предава на фабричния метод, а останалите съдържат нейните атрибути. Те се подават на метода за parse-ване на части, който ги валидира и съхранява.

В допълнение, за по-добро визуално представяне в терминала се използват ANSI escape кодове, които позволяват оцветяване на текст (например цветовете на фигурите). Това прави интерфейса по-лесен за възприемане от потребителя.

2.2. Дефиниране на проблеми и сложност на поставената задача

При реализацията на операцията за отваряне на файл (*open*) възникна проблем, който включва четене на даден SVG файл, взимане на типа на фигурата и спрямо нейният тип, взима атрибутите на фигурата и запазване на самата фигура в структура (във вектор), която да позволява лесна обработка и извеждане на данните от нея. Проблемът се изразяваше в неправилно четене от файловия поток, което затрудняваше правилното създаване на обектите от входните данни.

2.3. Подходи, методи (евентуално модели и стандарти) за решаването на поставените проблеми

Програмата използва команден шаблон (Command Pattern), при който всяка операция е реализирана като клас с метод “*execute”*. Командите се съхраняват в “`std::map`”, който свързва име на команда със съответния клас (ключ-стойност).

За решаването на проблемът с четенето от файлов поток при отваряне на файл, в базовия клас Figure бе добавен метод “*deserializeAll*”, който използва “`std::istringstream`”, за да третира входния ред като поток. След това се извиква виртуалният метод “*deserialize*” за съответния тип фигура, който използва помощни функции от “*deserializeHelper.hpp*” за прочитане на числови стойности и цветове. Това осигурява правилно създаване на фигури от входния файл.

2.4 Потребителски (функционални) изисквания (права, роли, статуси, диаграми...) и качествени (нефункционални) изисквания (скалируемост, поддръжка…)

За работа с програмата не е нужен администраторски достъп или други специални права, но програмата трябва да може да има достъп до четене на файлове и създаването им. Всички команди се изпълняват при стартиране на програмата, но командите, които изпълняват манипулация на фигури или запазване и затваряне на файл могат да бъдат използвани коректно само след отваряне на файл.

Добавени са съобщения, които информират потребителя за успеха или грешки при изпълнение на командите. При неправилно въведена команда или атрибути, се извежда съобщение за грешката, без да прекъсва изпълнението на програмата.

3. Проектиране

3.1 Обща архитектура - ООП дизайн

Има абстрактен клас “*Figure*”, който дефинира интерфейс за всички фигури, включително методи като “*containsPoint*”, “*within*”, “*translate*”, “*clone*”, “*print*” и “*deserialize*”. От този абстрактен родителски клас произлизат базовите фигури (*basic shapes*): Rectangle, Circle и Ellipse, които имплементират виртуалните методи спрямо спецификите на съответната геометрична форма и специалните си атрибути. Използва се също така абстрактен клас “*Command*”, които дефинира виртуален метод “execute()” и виртуален деструктор. От него произлизат командни класове (напр. CreateCommand, EraseCommand и др.) които изолират всяка операция да работи със свои параметри и логика.

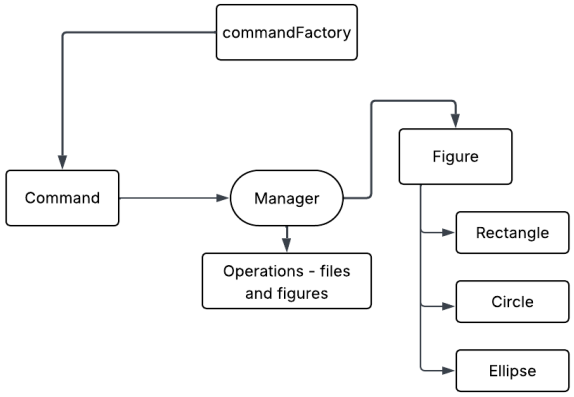
За управление на състоянието и операциите върху фигурите се използва класът “*Manager*”, който съдържа и управлява вектор от указатели към фигури (`std::vector<Figure*>`). Този клас е отговорен за обработката на логиката, без външните класове да имат достъп данните му.

Създаването на команди се осъществява чрез функция “*createCommandFactory*”, която връща “`std::map`”, която свързва низ (име на команда) с функция (key-value), която създава съответния обект от тип “*Command*”. Това позволява лесно добавяне на нови команди без промяна в основния контролен поток. По този начин по-лесно се добавя нова фигура или промяна по съществуващия код.

3.2 Диаграми (на структура и поведение - по обекти, слоеве с най-важните извадки от кода)

На диаграмата (Диаграма №1) е показана опростена структура на проекта, както и основните връзки между класовете. Процесът започва от “*commandFactory*”, който създава съответния обект от тип “*Command*” на база подадената команда. Оттам се извикват методи от класа “*Manager*”, който управлява операциите с файлове и фигури, наследени от абстрактния клас “*Figure*”.

Кратка структура на програмата



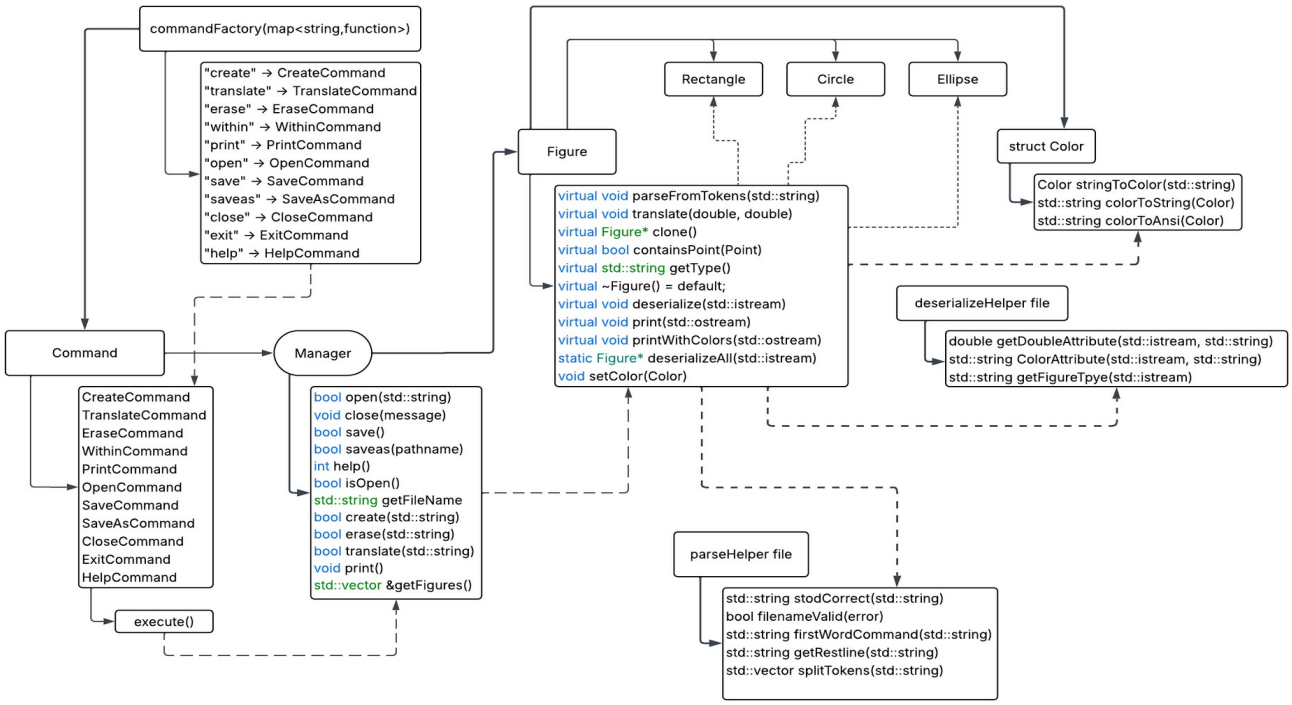
Диаграма №1

```cpp
struct Point{
    double x, y;
};

class Figure{
protected:
    Color color;
public:
    Figure();
    Figure(Color);
    virtual ~Figure() = default;

    virtual void parseFromTokens(const std::vector<std::string> &) = 0;
    virtual void translate(double, double) = 0;
    virtual Figure* clone() const = 0;
    virtual bool within(const Figure&) const = 0;
    virtual bool containsPoint(const Point&) const = 0;
    virtual std::string getType() const = 0;
    virtual void deserialize(std::istream &) = 0;
    virtual void print(std::ostream &) const = 0;
    virtual void printToTerminalWithColors(std::ostream &) const = 0;
    static Figure* deserializeAll(std::istream &);
    void setColor(const Color&);
};
```

Снимка №1

Абстрактен и родителски клас “Figure”

Голямата диаграма по-долу (Диаграма №2) илюстрира цялостното взаимодействие между класовете и допълнителните файлове с помощни функции. Стрелките обозначават връзките между класовете, а пунктираните стрелки показват използването и извикването на функции от други функции. Частно за пунктираните стрелки: класовете “*Rect*”, “*Circle*” и “*Ellipse*” имплементират виртуалните методи от класа “*Figure*“.



Диаграма №2

4. Реализация, тестване

4.1. Реализация на класове (включва важни моменти от реализацията на класовете и малки фрагменти от кода)

Един от важните елементи в програмата е реализацията на класа “*Manager*”, в който се съдържат всички операции с които програмата трябва да може да работи. Класът използва вектор от фигури, булева променлива за отворен файл и низ за името на текущия файл. Класът “*Manager*” служи като връзка за командния шаблон и съхранението на фигурите. Използват се деструктор за освобождаване на всички фигури, за да се избегне изтичане на памет.

(вж. снимки №1, №2 и №3)

```
class Manager{
std::vector<Figure*> figures;
std::string currentFilename;
bool isFileOpen;
public:
    Manager();
    ~Manager();

    //Working with files
    bool open(const std::string &path);
    void close(bool outputMessage);
    bool save() const;
    bool saveAs(const std::string &path);
    int help() const;
    bool isOpen() const;
    std::string getFilename() const;

    //Working with figures
    void withinCommandCall(const Figure &);
    bool createFigure(const std::string &);
    bool erase(const std::string &);
    bool translateFigures(const std::string &);
    void printFigures();

    std::vector<Figure*> &getFigures();

    const std::vector<Figure*> &getFigures() const;
};
```

Снимка №1

Главни методи на клас Manager, с които програма трябва да може да работи.

```
// close the file, delete all figures and print a message
void Manager::close(bool outputMessage){
    if(!isFileOpen){
        if(!outputMessage){
            std::cout << "No file is opened." << std::endl;
        }
        return;
    }
    // delete all figures before closing
    for (Figure *fig : figures){
        delete fig;
    }
    figures.clear();
    if(!outputMessage){
        std::cout << "Successfully closed: " << currentFilename << std::endl;
    }
    currentFilename.clear();
    isFileOpen = false;
}
```

Снимка №2

Код на метода close, който освобождава всички досега запазени фигури и затваря досега отворения файл.

4.2 Управление на паметта и алгоритми. Оптимизация

Всеки обект от тип “*Figure*” се съхранява в структурата вектор от указатели към фигура (`std::vector<Figure*>`). Затова имаме виртуални деструктори и ръчно освобождаваме динамично заделените обекти от паметта. За по-ниска употреба на памет, части от кода са преместени в помощни функции за десериализация и парсване - намиращи се във файл с име: *deserializeHelper.cpp*. Също така използваме “`std::map`” в “*commandFactory*”, което позволява по-лесно търсене на командата по име, като всяка стойност е ламбда функция, която връща съответен обект от тип “*Command*”. Разделянето на логиката на командите чрез команден шаблон позволява всяка една операция да бъде самостоятелна и да не зависи от друг код. Вместо да използваме дълъг код от проверки (if-else), използваме фабриката за команди, което улеснява тяхното извикване.

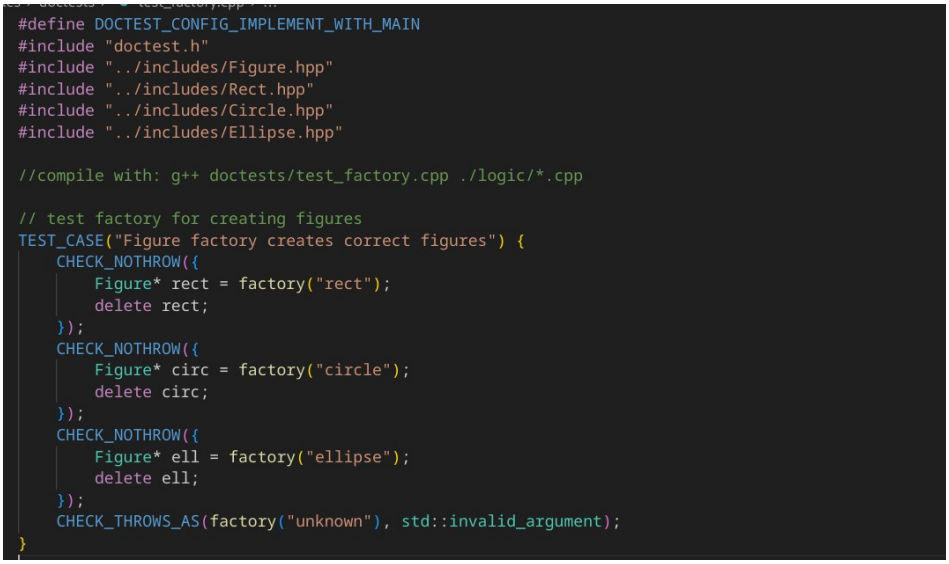
4.3. Планиране, описание и създаване на тестови сценарии (създаване на примери)

Тестовете, които бяха направени, имат за цел да проверят дали операциите за файлове и за фигури работят и дали фабриката за команди (*commandFactory*) - командите дали съществуват и дали биват изпълнени. Дали фабриката за връщане на тип фигура връща правилен тип.

```cpp
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"
#include "../includes/Figure.hpp"
#include "../includes/Rect.hpp"
#include "../includes/Circle.hpp"
#include "../includes/Ellipse.hpp"
#include <sstream> // for istringstream

//compile with: g++ doctests/test_helper.cpp ./login/*.cpp


// Use istringstream to wrap the string as istream so we can use it with deserializeAll
TEST_CASE("Deserialize Rect from svg string") {
    std::string rectSvg = "(<rect x=\"10\" y=\"20\" width=\"30\" height=\"40\" fill=\"red\"/>)";
    std::istringstream ss(rectSvg);
    Figure* fig = Figure::deserializeAll(ss);
    CHECK(fig != nullptr);
    // check if the figure is of type Rect
    CHECK(dynamic_cast<Rect*>(fig) != nullptr);
    delete fig;
}
```

Снимка №1

Снимка 1: част от кода която проверява дали десереализира низа и дали типа на фигурата е от същия тип.



```cpp
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"
#include "../includes/Figure.hpp"
#include "../includes/Rect.hpp"
#include "../includes/Circle.hpp"
#include "../includes/Ellipse.hpp"

//compile with: g++ doctests/test_factory.cpp ./logic/*.cpp

// test factory for creating figures
TEST_CASE("Figure factory creates correct figures") {
    CHECK_NOTHROW({
        Figure* rect = factory("rect");
        delete rect;
    });
    CHECK_NOTHROW({
        Figure* circ = factory("circle");
        delete circ;
    });
    CHECK_NOTHROW({
        Figure* ell = factory("ellipse");
        delete ell;
    });
    CHECK_THROWS_AS(factory("unknown"), std::invalid_argument);
}
```

Снимка №2

Снимка 2 – проверява за правилно създаване на фигури от въведения тип

5. Заключение

5.1. Обобщение на изпълнението на началните цели

Има нужните класове за базови фигури, клас за работа с всички операции - както и работещи операции за работа с файлове и създаване и манипулация на фигури. Проблемът с четенето при отварянето на файл беше решен и сега правилно прочита и записва фигурите във вектор от фигури. Беше направена "козметична" вариационна функция print, която чрез ANSI escape кодове добавя цвят към терминала при принтиране на всички фигури и фигурите, които се съдържат в някоя друга фигура (“*within*”). Добавен беше валидатор за въведено име при отваряне и записване на файл.

5.2. Насоки за бъдещо развитие и усъвършенстване

Идеи за бъдещо развитие:

1. Добавяне на други базови фигури и потенциално по-сложни фигури

2. Добавяне на допълнителни атрибути - fill-opacity, fill-rule, rotate, scale и други

3. Подобряване на информацията при извеждане на грешки

4. Поддръжка за undo механизъм на командния шаблон, който може да направи работата с фигурите при грешно въведени стойности по-лесна

Информация за github repository и компилация

Github: https://github.com/AdamS839/SVG-Files.

Компилацията на проекта става чрез: g++ main.cpp logic/*.cpp

Компилацията на тестовете става чрез: g++ test/(name_of_test_file).cpp logic/*.cpp

Използвана литература:

"Colorizing Text in the Console with C++." *Stack Overflow*, https://stackoverflow.com/questions/4053837/colorizing-text-in-the-console-with-c.

"How to Replace All Occurrences of a Character in String." *Stack Overflow*, https://stackoverflow.com/questions/2896600/how-to-replace-all-occurrences-of-a-character-in-string.

"How Does string::npos Know Which String I Am Referring To?" *Stack Overflow*, https://stackoverflow.com/questions/21654609/how-does-stringnpos-know-which-string-i-am-referring-to.

"string::npos in C++ with Examples." *GeeksforGeeks*, https://www.geeksforgeeks.org/stringnpos-in-c-with-examples/.

"Inserting Member Functions in a std::map of..." *Reddit*, 12 March 2021, https://www.reddit.com/r/cpp_questions/comments/mbbg58/inserting_member_functions_in_a_stdmap_of/.

The Cherno. "C++ Tutorial: Using std::map." *YouTube*, 15 Dec. 2019, https://www.youtube.com/watch?v=KiB0vRi2wlc&t=1361s.

"istringstream." *cplusplus.com*, https://cplusplus.com/reference/sstream/istringstream/.

"Istringstream: How to Do This." *Stack Overflow*, https://stackoverflow.com/questions/2323929/istringstream-how-to-do-this.

"Command Pattern in C++." *GeeksforGeeks*, https://www.geeksforgeeks.org/command-pattern-c-design-patterns/.

“std::remove.” *cppreference.com*, 28 May 2024, https://en.cppreference.com/w/cpp/io/c/remove.

Базови фигури: Ferraiolo, Jon, et al. *Scalable Vector Graphics (SVG) 1.1 (Second Edition) - Shapes.* W3C, 16 Aug. 2011, https://www.w3.org/TR/SVG/shapes.html.

Разглеждане на данните в онлайн viewer: *SVG Viewer Dev*, https://www.svgviewer.dev/.

Диаграмите са направени с: *LucidChart Diagram*, https://www.lucidchart.com/pages.

Проверка за коректно написани тестове с библиотеката “doctest.h” и граматична и правописна проверка на документацията беше извършена от ChatGPT.