Технически университет – Варна

Факлутет по изчислитена техника и автоматизация
Софтуерни и интернет технологии
(СИТ)

Курсов проект по Обектно-ориентирано програмиране част I

Стивън Ивов Иванов

Факултетен №: 21621523

Група: 1а

Проект 2: Работа със SVG файлове

Contents

Глава 1. Увод

1.1. Описание и идея на проекта

Проектът представлява приложение за работа със Scalable Vector Graphics (SVG) файлове. Целта е да се реализира система, която може да зарежда фигури от SVG файл, да извършва различни операции върху тях и след това да записва промените обратно във файла. Програмата ще работи само с основните геометрични фигури, като поне три от тях трябва да бъдат поддържани. Примери за такива фигури са линия, кръг и правоъгълник.

За да се улесни разбирането и обработката на координатите, ще се използва стандартната координатна система, където положителната ос X сочи надясно, а положителната ос Y сочи надолу.

Дизайнът на приложението ще бъде модулен и разширяем, така че при бъдещи промени да може лесно да се добавят нови фигури и функционалности.

1.2. Цел и задачи на разработката

Основната цел на проекта е да предостави на потребителя лесен начин за манипулиране на SVG файлове чрез команден интерфейс. За всяка фигура ще се поддържат основни операции, като:

- Зареждане на фигури от SVG файл
- Преглед и извеждане на списък с наличните фигури
- Добавяне на нови фигури
- Изтриване на фигури по индекс
- Транслиране на определена фигура или всички фигури
- Проверка дали фигура попада в даден регион (кръг или правоъгълник)
- Записване на модифицираните фигури обратно в SVG файл

1.3. Структура на документацията

Документацията е разделена на 5 глави, представящи различните стъпки от решението на задачата. Първа глава представлява увод и описва идеята и целите на проекта. Втора глава представя основните дефиниции и концепции, които ще се използват и дефинирането на основните проблеми в задачата и подходите за тяхното решение. Трета глава има за цел да дефинира общата структура на и да визуализира различните слоеве и обекти на проекта. В четвърта глава се вниква в различните алгоритми и малки фрагменти код използвани за реализацията на проекта, също се представят тестови сценарии. Пета глава обобщава проекта и представя идеи за подобряване и развитие на проекта.

Глава 2. Преглед на предметната област

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

Проектът използва **Scalable Vector Graphics (SVG)** – XML-базиран формат за векторна графика, поддържан от уеб браузъри и графични редактори. В рамките на този проект ще работим с три основни фигури:

- **Линия (<line>)** – представена чрез начална и крайна точка.
- **Кръг (<circle>)** – описан с централни координати и радиус.
- **Правоъгълник (<rect>)** – дефиниран чрез координати на горния ляв ъгъл, ширина и височина.

Основните операции, които се прилагат върху фигурите, включват **създаване**, **изтриване**, **транслация** и **проверка дали фигура попада в даден регион**. Всички тези операции ще бъдат реализирани чрез команден интерфейс (CLI), като всяка команда ще бъде свързана с конкретен клас, който я обработва.

Програмата използва **обектно-ориентиран подход**, като всяка фигура е представена чрез клас, наследяващ интерфейса Shape. Това позволява полиморфизъм и лесно разширяване на функционалността в бъдеще.

2.2. Дефиниране на проблеми и сложност на поставената задача

Основните предизвикателства при разработката на системата включват:

1. **Четене и парсване на SVG файлове** – Извличане на релевантните фигури и игнориране на неподдържаните SVG елементи.
2. **Коректно манипулиране на фигури** – Осигуряване на правилно съхранение на координати и атрибути за всяка фигура.

3. **Извършване на операции върху фигурите** – Реализиране на трансформации като преместване и проверка за съдържание в дадена област.

4. **Записване на SVG файлове** – Коректно генериране на SVG структура, съвместима със стандарта.

Сложността на задачата произтича от факта, че **SVG е XML-формат**, което изисква работа със **DOM (Document Object Model)** за четене и запис на файлове. Освен това, командите трябва да бъдат обработвани динамично и правилно да манипулират съхраняваните фигури.

2.3. Подходи, методи и модели за решаване на поставените проблеми

За справяне с изброените проблеми използваме:

- **Модулна архитектура** – разделяне на проекта на независими компоненти:
    - **SvgFileHandler** – отговаря за четене и запис на SVG файлове.
    - **SvgRepository** – съхранява в паметта всички заредени фигури.
    - **CommandProcessor** – обработва въведените от потребителя команди.
    - **ShapeFactory** – създава фигури по подадени параметри.
- **Обектно-ориентирано програмиране (ООП)** – използване на абстрактен клас Shape и негови конкретни имплементации (RectangleShape, CircleShape). Това позволява разширяемост и повторна употреба на код.
- **Алгоритми за пространствено съдържание** – за проверка дали фигура попада в даден регион, използваме:
    - Проверка на **ограничаващи правоъгълници (Bounding Box)**.
    - Формули за **разстояние между точки** при работа с кръгове.
- **Обработка на грешки** – осигуряване на механизъм за хендлиране на грешни входни данни и невалидни команди.

2.4. Потребителски (функционални) изисквания и качествени (нефункционални) изисквания

2.4.1. Функционални изисквания

1. Потребителят трябва да може да отваря и затваря SVG файлове.
2. Приложението трябва да поддържа поне три основни фигури – линия, кръг и правоъгълник.
3. Трябва да има команди за **създаване, изтриване, транслация и проверка** на съдържанието в дадена област.
4. Фигурите трябва да могат да бъдат извеждани в списък с техните параметри.
5. Програмата трябва да записва SVG файловете коректно след направените промени.

2.4.2. Нефункционални изисквания

1. **Разширяемост** – трябва да може лесно да се добавят нови фигури и функционалности.
2. **Стабилност** – приложението трябва да се справя с грешки при въвеждане на команди и невалидни файлове.
3. **Скорост** – операциите трябва да се изпълняват ефективно дори при голям брой фигури.
4. **Съвместимост** – програмата трябва да работи на различни операционни системи (Windows, Linux, macOS).
5. **Лесна употреба** – командният интерфейс трябва да бъде интуитивен и удобен за работа.

Глава 3. Проектиране

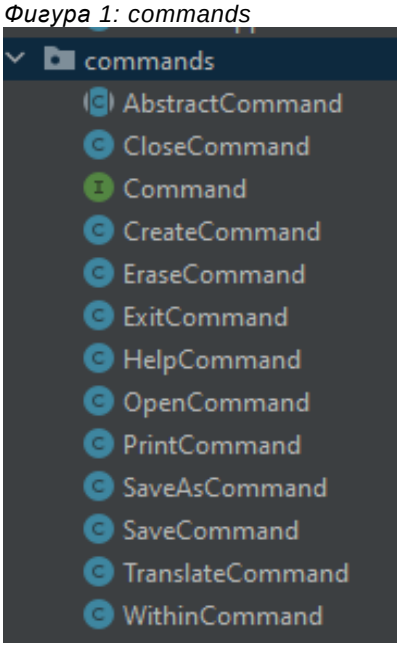
3.1. Обща структура на проекта, пакети които ще се реализират

3.1.1. Обща структура

Проектът е разделен на няколко пакета: commands, app, cli, shapes, storage, svg

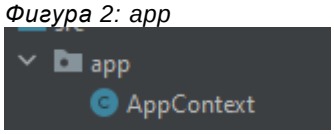
3.1.2. Пакет commands

Фигура 1: commands



3.1.3. Пакет app

Фигура 2: app

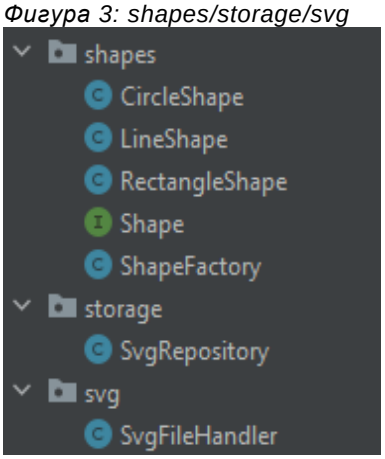


3.1.4. Пакет cli

Фигура 3: cli

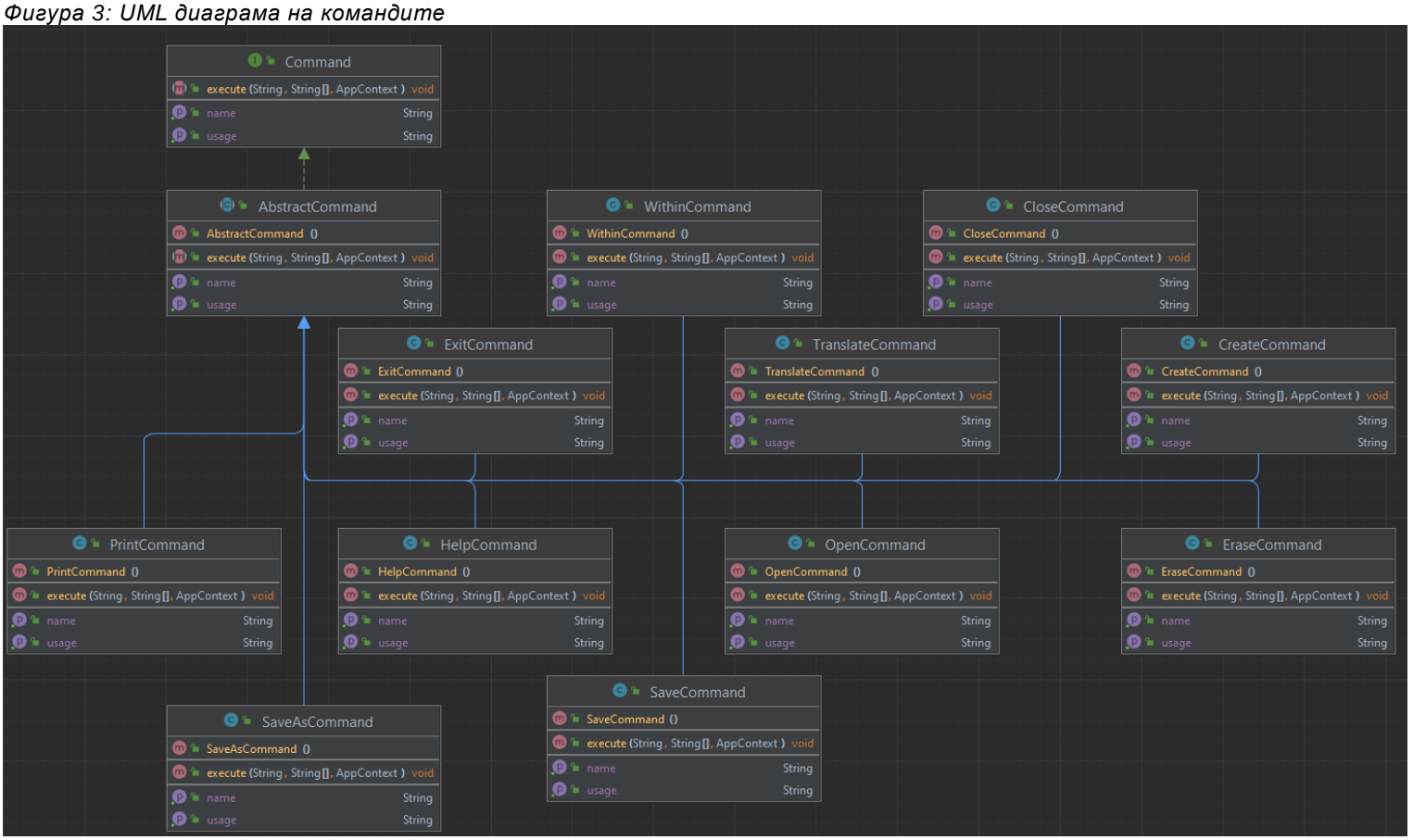
3.1.5. Пакети shapes/storage/svg

Фигура 3: shapes/storage/svg



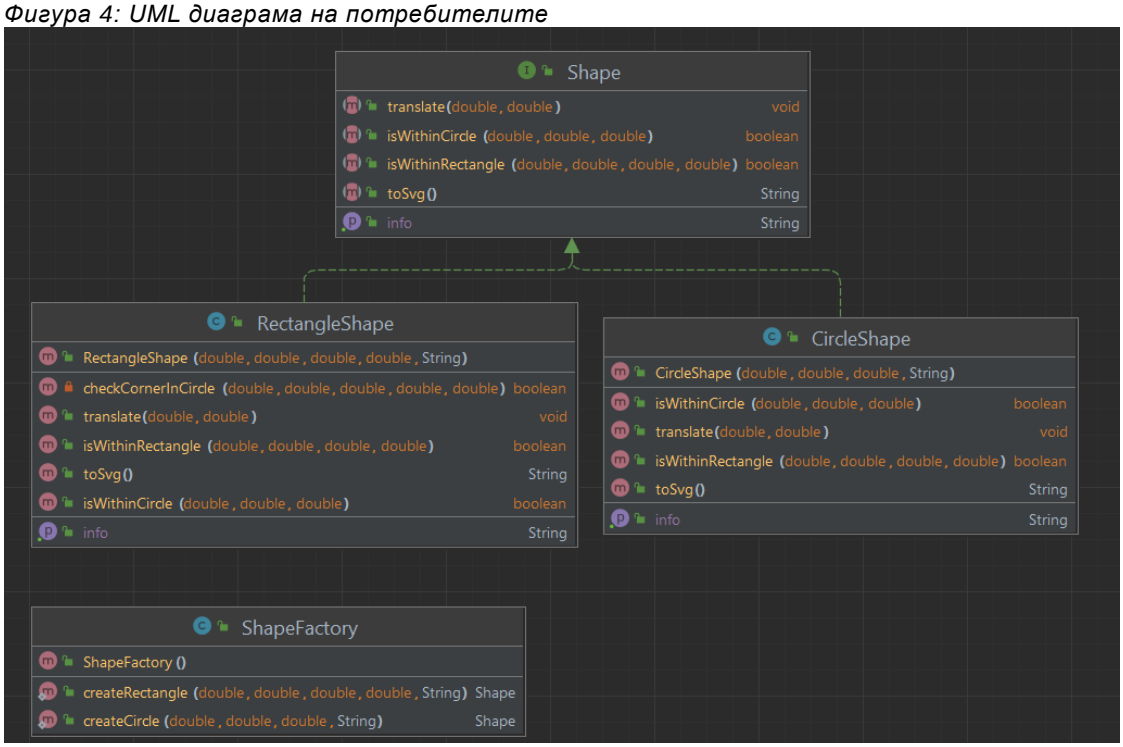
3.2. Диаграми/Блок схеми (на структура и поведение – по обекти, слоеве с най-важните извадки от кода)

Фигура 3: UML диаграма на командите



На **Error! Reference source not found.** е представена диаграма на класовете, отговарящи за извършването на желаните от потребителя операции. Тази дървовидна структура представлява шаблона *command*. Всяка конкретна команда имплементира интерфейса *Command* и неговите методи. Методът *execute* извършва действието на командата.

Фигура 4: UML диаграма на потребителите



Интерфейсът Shape (Абстрактно ниво)

- **Роля:** *Интерфейсът Shape дефинира общ набор от методи, които всяка фигура трябва да имплементира. Това гарантира, че независимо от конкретния тип фигура (например правоъгълник или кръг), може да се работи с нея чрез общи операции.*

- **Методи:**

  - **getInfo()** *Връща кратко, човеко-разбираемо описание на фигурата (например "rectangle 5 5 10 20 green").*

  - **translate(double dx, double dy)** *Премества фигурата с дадените хоризонтални и вертикални офсети.*

- **isWithinRectangle(double rx, double ry, double rw, double rh)** *Проверява дали фигурата е напълно съдържана в даден правоъгълник.*
- **isWithinCircle(double cx, double cy, double r)** *Проверява дали фигурата е напълно съдържана в даден кръг.*
- **toSvg()** *Генерира SVG представяне на фигурата (например <rect ... /> или <circle ... />).*

Конкретните класове, имплементиращи Shape:

- **RectangleShape:**
  - **Описание:** *Представлява правоъгълник с определени координати (x, y), размери (width, height) и цвят (fillColor).*
  - **Имплементация:** *Класът имплементира всички методи от интерфейса Shape според логиката, специфична за правоъгълника:*
    - **getInfo()** *връща описание във формат "rectangle x y width height color".*
    - **translate()** *актуализира координатите x и y.*
    - **isWithinRectangle()** *проверява дали правоъгълникът е напълно в даден правоъгълник.*
    - **isWithinCircle()** *проверява дали всички четири ъгъла на правоъгълника попадат в даден кръг.*
    - **toSvg()** *генерира SVG низ, представящ правоъгълника.*
- **CircleShape:**
  - **Описание:** *Представлява кръг с център (cx, cy), радиус r и цвят (fillColor).*
  - **Имплементация:** *Класът имплементира методите от интерфейса Shape с логика, специфична за кръга:*
    - **getInfo()** *връща описание във формат "circle cx cy r color".*

        - **translate()** *премества центъра на кръга.*

        - **isWithinRectangle()** *проверява дали целият кръг е съдържан в даден правоъгълник (като се проверяват границите на кръга спрямо правоъгълника).*

        - **isWithinCircle()** *проверява дали целият кръг е съдържан в друг кръг (като се сравнява разстоянието между центровете и радиусите).*

        - **toSvg()** *генерира SVG низ, представящ кръга.*

- **LIneShape:**

    - **Описание:** *линията се дефинира чрез начална точка (x, y), размери (width, height) – като вторият край се получава като (x + width, y + height) – и цвят (fillColor):*

    - **Имплементация:** *Класът имплементира методите от интерфейса Shape с логика, специфична за линията:*

        - **getInfo()** *Използва String.format за форматиране на стойностите с две десетични места.*
        - **translate()** *Добавя подадените измествания dx и dy към координатите на двата края на линията.*
        - **isWithinRectangle()** *За всяка от двете точки се проверява дали лежи в интервала по x и y, дефиниран от правоъгълника (началната точка (rx, ry) и размерите (rw, rh)).*
        - **isWithinCircle()** *Използва помощния метод isPointInCircle() за да провери дали разстоянието от всяка точка до центъра на кръга е по-малко или равно на радиуса.*
        - **toSvg()** *Създава SVG елемент <line> със зададените координати, цвят и дебелина.*

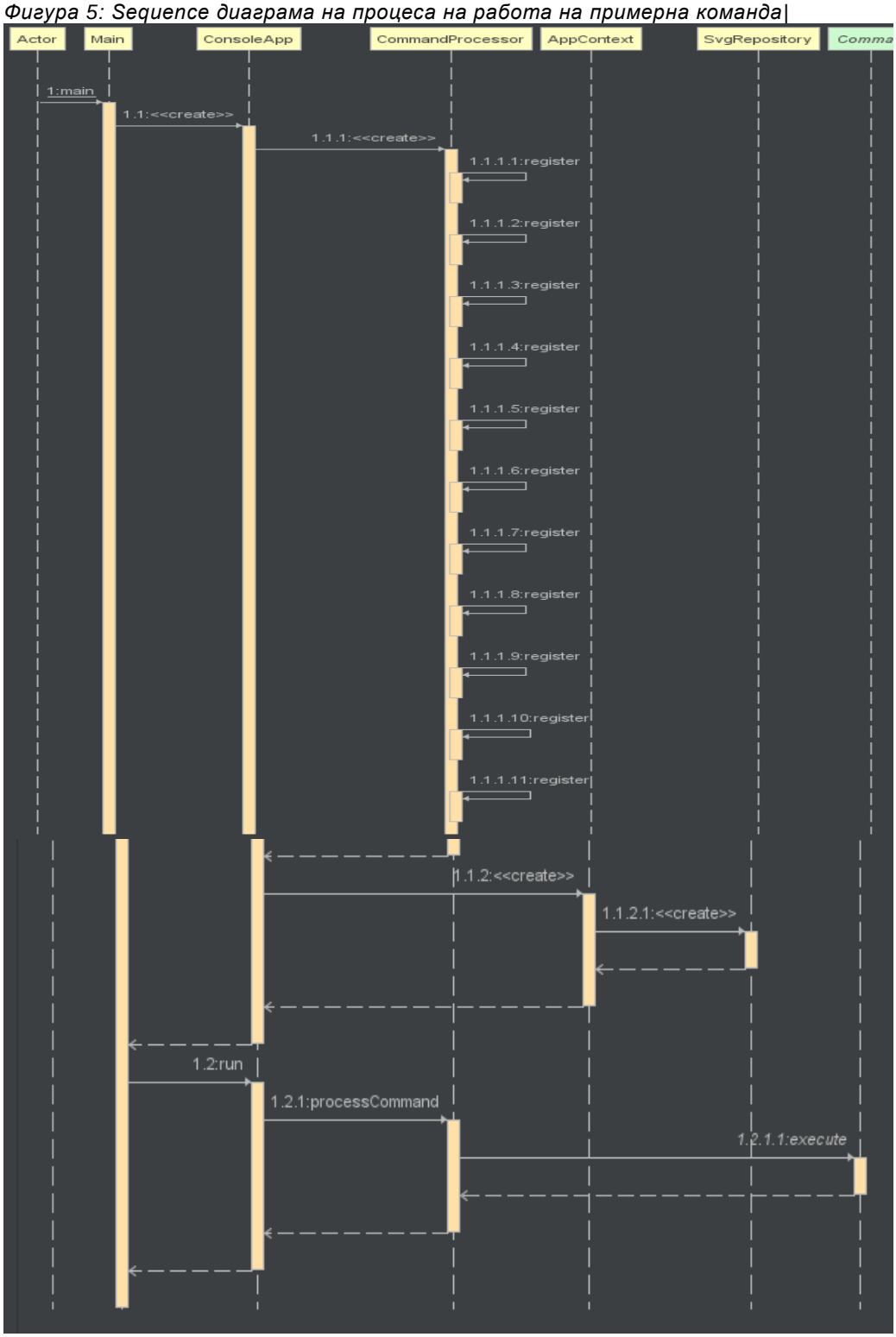
Фабрика за създаване на фигури – ShapeFactory:

- **Роля:** *Класът ShapeFactory предоставя статични методи за създаване на конкретни обекти, реализиращи интерфейса Shape. Това позволява на останалата част от*

програмата да създава фигури без да се занимава с детайлите на конкретната имплементация.

- **Методи:**
    - **createRectangle(double x, double y, double width, double height, String color)** *Създава и връща нов обект от тип RectangleShape.*
    - **createCircle(double cx, double cy, double radius, String color***) Създава и връща нов обект от тип CircleShape.*

Фигура 5: Sequence диаграма на процеса на работа на примерна команда|



На Фигура 5 са представени последователните действия за изпълнението на команда.

Глава 4. Реализация, тестване

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

4.1.1. Реализация на командни класове и интерфейси

Интерфейс *Command*:

```
public interface Command {

    String getName();

    void execute(String input, String[] args, AppContext context);

    default String getUsage() {
        return "No usage provided for this command.";
    }
}
```

Това е интерфейсът Command, който дефинира общия контракт за всички команди, изпълними от CLI (Command-Line Interface). Ето кратко описание:

- **getName()**
  Връща името на командата (например "open", "close", "print" и т.н.), което се използва за идентификация и разпознаване на командата.

- **execute(String input, String[] args, AppContext context)**
  Изпълнява командата с дадения потребителски вход.

  - input съдържа пълния текст, въведен от потребителя.

  - args съдържа токените (аргументите), които следват името на командата.

  - context предоставя глобалния контекст на приложението, който може да съдържа общи данни или ресурси.

- **default String getUsage()**

  Предоставя кратко описание или помощен текст за командата. Този метод е дефолтен, което означава, че ако не бъде презаписан от конкретната команда, по подразбиране ще върне съобщението "No usage provided for this command."

Така интерфейсът Command осигурява стандартизирана структура за добавяне и изпълнение на команди в CLI приложението.

Примерна реализация на конкретна команда (*Close*):

```
public class CloseCommand extends AbstractCommand {

    @Override
    public String getName() {
        return "close";
    }

    @Override
    public String getUsage() {
        return "close - Closes the currently opened file.";
    }

    @Override
    public void execute(String input, String[] args, AppContext context) {
        if (!context.isFileOpened()) {
            System.out.println("No file is currently opened.");
            return;
        }
        String fileName = context.getCurrentFile();
        context.closeFile();
        System.out.println("Successfully closed " + fileName);
    }
}
```

Класът CloseCommand разширява (extends) абстрактния клас AbstractCommand и реализира необходимите методи, за да се използва като команда в CLI приложението. Ето кратко описание на неговата функционалност:

- **Методът getName()**

  Връща името на командата – в случая, "close". Това име се използва за разпознаване и извикване на командата от потребителя.

- **Методът getUsage()**

  Връща кратко описание на начина на използване на командата: "close - Closes the currently opened file." Това дава на потребителя информация за това какво прави командата.

- **Методът execute(String input, String[] args, AppContext context)**
  Това е мястото, където се дефинира логиката на командата:

1. **Проверка дали е отворен файл:**

   Извиква се context.isFileOpened(). Ако няма отворен файл, се извежда съобщение: "No file is currently opened." и изпълнението на командата приключва.

2. **Затваряне на файла:**

   Ако файлът е отворен, първо се взема името му чрез context.getCurrentFile(). След това се извиква context.closeFile(), за да се затвори файлът.

3. **Извеждане на резултат:**

   След успешно затваряне се отпечатва съобщение, съдържащо името на затворения файл, например:"Successfully closed <fileName>".

С други думи, CloseCommand предоставя функционалността за затваряне на текущо отворения файл в приложението, като изпълнява подходящи проверки и информира потребителя за резултата.

Обработване на команди – Command Processor:

```
public class CommandProcessor {
    private final Map<String, Command> commands = new HashMap<>();

    public CommandProcessor() {
        register(new OpenCommand());
        register(new CloseCommand());
        register(new SaveCommand());
        register(new SaveAsCommand());
        register(new PrintCommand());
        register(new CreateCommand());
        register(new EraseCommand());
        register(new TranslateCommand());
        register(new WithinCommand());
        register(new HelpCommand());
        register(new ExitCommand());
    }

    private void register(Command cmd) {
        commands.put(cmd.getName().toLowerCase(), cmd);
    }

    public void processCommand(String inputLine, AppContext context) {
        if (inputLine == null || inputLine.trim().isEmpty()) {
            return;
```

```
        }

        String[] tokens = inputLine.trim().split("\\s+");
        String cmdName = tokens[0].toLowerCase();
        String[] args = new String[tokens.length - 1];
        System.arraycopy(tokens, 1, args, 0, tokens.length - 1);

        Command cmd = commands.get(cmdName);
        if (cmd == null) {
            System.out.println("Unknown command. Type 'help' for available
commands.");
            return;
        }

        cmd.execute(inputLine, args, context);
    }
}
```

Класът CommandProcessor отговаря за обработката на потребителския вход, като:

- **Регистрация на команди:**

  В конструктора се регистрират множество команди (например OpenCommand, CloseCommand, SaveCommand и др.) в HashMap, където ключът е името на командата в малки букви.

- **Обработка на входа:**

  Методът processCommand разделя входния низ на токени, взема първия като име на командата и останалите като аргументи.

- **Изпълнение на командата:**

  Ако намери съответната команда в регистрацията, извиква нейния метод execute с входния низ, аргументите и контекста. При непозната команда се извежда съобщение за грешка.

4.1.2. Фигури, групиране на фигури в хранилище и работа с SVG

Клас Shape:

```
public interface Shape {

    String getInfo();
```

```
    void translate(double dx, double dy);

    boolean isWithinRectangle(double rx, double ry, double rw, double rh);

    boolean isWithinCircle(double cx, double cy, double r);

    String toSvg();
}
```

Интерфейсът Shape дефинира основните операции за всяка геометрична фигура:

- **getInfo()** – Връща кратко описание (напр. "rectangle 5 5 10 20 green").
- **translate(dx, dy)** – Премества фигурата с дадените хоризонтален и вертикален офсет.
- **isWithinRectangle(rx, ry, rw, rh)** – Проверява дали фигурата е напълно съдържана в посочения правоъгълник.
- **isWithinCircle(cx, cy, r)** – Проверява дали фигурата е напълно съдържана в посочения кръг.
- **toSvg()** – Генерира SVG представяне на фигурата.

Circle Shape

```
public class CircleShape implements Shape {

    private double cx;
    private double cy;
    private double r;
    private String fillColor;

    public CircleShape(double cx, double cy, double r, String fillColor) {
        this.cx = cx;
        this.cy = cy;
        this.r = r;
        this.fillColor = fillColor;
    }

    @Override
    public String getInfo() {
        // e.g. "circle 5 5 10 blue"
        return String.format("circle %.2f %.2f %.2f %s",
                cx, cy, r, fillColor);
    }
```

```
    @Override
    public void translate(double dx, double dy) {
        cx += dx;
        cy += dy;
    }

    @Override
    public boolean isWithinRectangle(double rx, double ry, double rw, double
rh) {
        // The circle must fit entirely within the rectangle:
        // (cx - r) >= rx, (cy - r) >= ry
        // (cx + r) <= (rx + rw), (cy + r) <= (ry + rh)
        if ((cx - r) < rx) return false;
        if ((cy - r) < ry) return false;
        if ((cx + r) > (rx + rw)) return false;
        if ((cy + r) > (ry + rh)) return false;
        return true;
    }

    @Override
    public boolean isWithinCircle(double centerX, double centerY, double
bigR) {
        double distCentersSq = Math.pow(cx - centerX, 2) + Math.pow(cy -
centerY, 2);
        double bigRSq = Math.pow(bigR, 2);
        double distCenters = Math.sqrt(distCentersSq);
        return distCenters + r <= bigR;
    }

    @Override
    public String toSvg() {
        return String.format(Locale.US,
                "<circle cx=\"%.2f\" cy=\"%.2f\" r=\"%.2f\" fill=\"%s\"/>",
                cx, cy, r, fillColor
        );
    }
}
```

Класът CircleShape представлява кръг и имплементира интерфейса Shape. Основни точки:

- **Полетата:**
  Съдържа координати на центъра (cx, cy), радиус (r) и цвят (fillColor).

- **getInfo():**
  Връща описание във формат "circle cx cy r color".

- **translate(dx, dy):**
  Премества центъра на кръга с посочените офсети.

- **isWithinRectangle(rx, ry, rw, rh):**
  Проверява дали целият кръг (с неговия радиус) се съдържа в даден правоъгълник.

- **isWithinCircle(centerX, centerY, bigR):**
  Проверява дали кръгът се съдържа напълно в друг кръг, като сравнява разстоянието между центровете плюс радиуса.

- **toSvg():**
  Генерира SVG представяне на кръга във формат <circle ... />.

Клас SvgRepository:

```
public class SvgRepository {

    private final List<Shape> shapes;

    public SvgRepository() {
        this.shapes = new ArrayList<>();
    }

    public void addShape(Shape shape) {
        shapes.add(shape);
    }

    public Shape getShape(int index) {
        if (index < 0 || index >= shapes.size()) {
            return null;
        }
        return shapes.get(index);
    }

    public boolean removeShape(int index) {
        if (index < 0 || index >= shapes.size()) {
            return false;
        }
        shapes.remove(index);
        return true;
    }

    public List<Shape> getAllShapes() {
        return shapes;
    }

    public int size() {
        return shapes.size();
    }
```

```
    public void translateAll(double dx, double dy) {
        for (Shape shape : shapes) {
            shape.translate(dx, dy);
        }
    }

    public void clear() {
        shapes.clear();
    }
}
```

Класът SvgRepository управлява колекция от обекти, които имплементират интерфейса Shape, използвайки списък. Основни функционалности:

- **Добавяне и достъп:**
    - addShape(Shape shape) – Добавя нова фигура.
    - getShape(int index) – Връща фигура по индекс (или null, ако индексът е невалиден).
    - removeShape(int index) – Премахва фигурата по индекс, връщайки true ако операцията е успешна.
- **Общи операции:**
    - getAllShapes() – Връща всички фигури.
    - size() – Връща броя на съхранените фигури.
    - translateAll(double dx, double dy) – Премества всички фигури с посочените офсети.
    - clear() – Изчиства всички фигури от хранилището.

Клас SvgFileHandler:

```
public class SvgFileHandler {

    public static void loadFromFile(String filePath, SvgRepository
repository) throws Exception {
        File file = new File(filePath);
```

```
        if (!file.exists()) {
            // Create empty file if not existing
            file.createNewFile();
            return;
        }

        DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
        dbFactory.setIgnoringComments(true);
        dbFactory.setIgnoringElementContentWhitespace(true);
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(file);

        doc.getDocumentElement().normalize();

        // Parse all <rect> elements
        NodeList rectNodes = doc.getElementsByTagName("rect");
        for (int i = 0; i < rectNodes.getLength(); i++) {
            Node node = rectNodes.item(i);
            if (node.getNodeType() == Node.ELEMENT_NODE) {
                Element elem = (Element) node;
                double x = getDoubleAttribute(elem, "x", 0);
                double y = getDoubleAttribute(elem, "y", 0);
                double w = getDoubleAttribute(elem, "width", 0);
                double h = getDoubleAttribute(elem, "height", 0);
                String fill = elem.hasAttribute("fill") ?
elem.getAttribute("fill") : "black";

                repository.addShape(new RectangleShape(x, y, w, h, fill));
            }
        }

        // Parse all <circle> elements
        NodeList circleNodes = doc.getElementsByTagName("circle");
        for (int i = 0; i < circleNodes.getLength(); i++) {
            Node node = circleNodes.item(i);
            if (node.getNodeType() == Node.ELEMENT_NODE) {
                Element elem = (Element) node;
                double cx = getDoubleAttribute(elem, "cx", 0);
                double cy = getDoubleAttribute(elem, "cy", 0);
                double r = getDoubleAttribute(elem, "r", 0);
                String fill = elem.hasAttribute("fill") ?
elem.getAttribute("fill") : "black";

                repository.addShape(new CircleShape(cx, cy, r, fill));
            }
        }
    }

    private static double getDoubleAttribute(Element elem, String attrName,
double defaultVal) {
        if (elem.hasAttribute(attrName)) {
            String val = elem.getAttribute(attrName);
            // Replace commas with dots:
            val = val.replace(',', '.');
```

```
            try {
                return Double.parseDouble(val);
            } catch (NumberFormatException e) {
                // If parsing fails, we return defaultVal
                return defaultVal;
            }
        }
        return defaultVal;
    }
    public static void saveToFile(String filePath, SvgRepository repository)
throws IOException {
        try (PrintWriter pw = new PrintWriter(new FileWriter(filePath))) {
            pw.println("<?xml version=\"1.0\" standalone=\"no\"?>");
            pw.println("<!DOCTYPE svg PUBLIC \"-//W3C//DTD SVG 1.1//EN\" " +
                    "\"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd\">");
            pw.println("<svg>");

            // Each Shape's toSvg() uses dot decimals (due to Locale.US in
shape classes)
            for (Shape shape : repository.getAllShapes()) {
                pw.println("  " + shape.toSvg());
            }

            pw.println("</svg>");
        }
    }
}
```

Класът SvgFileHandler отговаря за зареждане и запис на фигури в SVG формат:

- **loadFromFile:**
    - Проверява дали файлът съществува и ако не – създава празен файл.
    - Използва XML парсър за четене на SVG съдържанието.
    - Обхожда <rect> и <circle> елементите, като извлича атрибутите (с помощта на помощния метод getDoubleAttribute) и създава съответните обекти (RectangleShape и CircleShape), които добавя към хранилището.
- **saveToFile:**
    - Записва SVG документ с валиден XML и SVG хедър.
    - Извиква метода toSvg() на всяка фигура от хранилището, за да генерира SVG елементите.
    - Използва try-with-resources за автоматично затваряне на потока за запис.

Така този клас служи като мост между вътрешното представяне на фигурите и външния SVG файл.

4.2. Планиране, описание и създаване на тестови сценарии

4.2.1. Примерни файлове с данни

За тестването ще се използва *SVG* файл съдържащ различни фигурки с примерни данни. На Фигура 6 е показана част от файла *example.svg*, където са съхранени примерните данни.

Фигура 6: Файл example.svg

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN">
<svg>
  <rect x="30.00" y="30.00" width="100.00" height="50.00" fill="blue"/>
  <rect x="60.00" y="20.00" width="200.00" height="60.00" fill="yellow"/>
  <circle cx="200.00" cy="200.00" r="50.00" fill="red"/>
```

4.2.2. Тестови резултати

Команда save:

Правилно използване: save

Резултат: Данните за книги в програмата се запазват в отворения файл и на екрана се извежда съобщение за успешна операция: „Successfully saved example.svg!“

Команда close:

Правилно използване: close

Резултат: Данните в програмата не се съхраняват и на екрана се извежда съобщение за успешна операция: „Successfully closed example.svg!“

Команда exit:

Правилно използване: exit

Резултат: Извежда се съобщението „Exiting the program...“ на екрана и се излиза от програмата.

Команда help:

Правилно използване: help

Резултат: На екрана се отпечатва съобщение съдържащо информация за начина на използване на поддържаните от програмата команди.

```
> help
The following commands are supported:
open <file>         - Opens an SVG file
close               - Closes the currently opened file
save                - Saves the current file
saveas <file>       - Saves the current file under a new path
print               - Prints all loaded shapes
create ...          - Creates a new shape (rectangle/circle)
erase <n>           - Erases shape #n (1-based index)
translate ...       - Translates shape(s)
within ...          - Lists shapes fully within a region
help                - Displays this help message
exit                - Exits the program
```

Глава 5. Заключение

Проектът предоставя функционално CLI приложение за създаване, управление и визуализация на геометрични фигури във формат SVG. Основните операции включват добавяне, модифициране (преизчисляване на позиция) и премахване на фигури, както и проверка на тяхното съдържание в зададени области. Фигурите се съхраняват в паметта чрез репозитори, а данните се записват и зареждат от SVG файлове, което улеснява обмена и визуализацията им.

5.1. Насоки за бъдещо развитие и усъвършенстване

5.1.1. Поддръжка на допълнителни типове фигури (елипси, линии, полигони)

5.1.2. Разширяване на функционалността за манипулация на фигури (завъртане, скалиране)

5.1.3. Разработване на графичен потребителски интерфейс (GUI) за по-интуитивно взаимодействие

5.1.4. Добавяне на тестови сценарии (unit, integration и e2e) за гарантиране на стабилността на приложението