Документацията

Глава 1. Увод

* 1. Описание и идея на проекта

Програма реализираща операции с недетерминиран краен автомат с

ℇ-преходи, над азбука, състояща се от латински букви.

* 1. Цел и задачи на разработката

Целта на разработката е да създава автомати чрез стрингов низ/регекс, като всеки създаден автомат получава уникален идентификатор. След като приложението отвори даден файл, то извършва следните операции:

* **open <filename>**: Зарежда съдържанието на даден файл. Ако файлът не съществува, се създава нов с празно съдържание. Всички останали команди могат да се изпълняват само ако има успешно зареден файл. След като файлът бъде отворен и прочетен, той се затваря. Приложението не работи с него освен ако потребителят не поиска да запише направените промени (вижте командата Save). При зареждане на данните, ако приложението открие грешка, трябва да изведе съобщение за грешка и да прекрати изпълнението си.
* **close**: Затваря текущо отворения документ. Затварянето изчиства текущо заредената информация и след това програмата не може да изпълнява други команди, освен отваряне на файл (**open <filename>**).
* **save**: Записва направените промени обратно в същия файл, от който са били прочетени данните.
* **saveas <directory> <filename>**: Записва направените промени във файл, като позволява на потребителя да укаже неговия път.
* **help**: Извежда кратка информация за поддържаните от програмата команди.
* **exit**: Излиза от програмата.
* **list**: Списък с идентификаторите на всички прочетени автомати.
* **print <id>**: Извежда информация за всички преходи в автомата.
* **save <id> <filename>**: Записва автомат във файл.
* **empty <id>**: Проверява дали езикът на автомата е празен.
* **deterministic <id>**: Проверява дали автоматът е детерминиран.
* **recognize <id> <word>**: Проверява дали дадена дума е в езика на автомата.
* **union <id1> <id2>**: Намира обединението на два автомата и създава нов автомат. Отпечатва идентификатора на новия автомат.
* **concat <id1> <id2>**: Намира конкатенацията на два автомата и създава нов автомат. Отпечатва идентификатора на новия автомат.
* **un <id>**: Намира позитивна обвивка на автомат и създава нов автомат. Отпечатва идентификатора на новия автомат.
* **reg <regex>**: Създава нов автомат по даден регулярен израз (теорема на Клини). Отпечатва идентификатора на новия автомат.
* **mutator <id>**: реализира мутатор, който детерминира даден автомат.
* **finate <id>**: реализира операция, която проверяват дали езикът на даден автомат е краен.
  1. Структура на документацията

Глава 1. Увод

Глава 2. Преглед на предметната област

Глава 3. Проектиране, реализация, блок схеми

Глава 4. Тестване

Глава 2. Преглед на предметната област

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

**Дефиниции и концепции:**

* **Автомат:** Формален модел на изчисление, използван за представяне на езиците. Състои се от крайно множество състояния, начално състояние, крайни състояния и преходи.
* **Регулярен израз (Regex):** Символен низ, който описва шаблон за търсене в текст. Използва се за генериране на автомати.
* **Детерминиран автомат (DFA):** Автомат, в който за всяко състояние и всеки входен символ има точно един преход.
* **Недетерминиран автомат (NFA):** Автомат, в който за едно състояние и входен символ може да има множество преходи.
* **Преход:** Функция, която свързва състоянията в автомата при даден входен символ.

**Алгоритми:**

* **Конвертиране на регулярен израз в NFA:** Използва теоремата на Клини за създаване на автомат от регулярен израз.
* **Конвертиране на NFA в DFA:** Използва алгоритъма за детерминизация на недетерминиран автомат.
* **Проверка за детерминизъм:** Проверява дали автоматът има повече от един преход за дадено състояние и символ.

2.2. Дефиниране на проблеми и сложност на поставената задача

**Проблеми:**

* **Създаване на автомат от регулярен израз:** Генериране на състояния и преходи от символния низ на регулярния израз.
* **Детерминизация на автомат:** Конвертиране на NFA в DFA.
* **Проверка на думи:** Проверка дали дадена дума принадлежи на езика, определен от автомата.
* **Обединение и конкатенация на автомати:** Изчисляване на обединението и конкатенацията на два автомата.

**Сложност:**

* **Времева сложност:** Задачите включват множество стъпки за обработка на състояния и преходи, като сложността може да бъде линейна или квадратична в зависимост от броя на състоянията и преходите.
* **Пространствена сложност:** Представянето на автоматите изисква съхранение на състояния и преходи, което може да изисква значителни ресурси при големи регулярни изрази.

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

**Подходи и методи:**

* **Конструиране на автомати:** Използване на алгоритъма на Томпсън за конструиране на NFA от регулярен израз и алгоритъма за детерминизация за преобразуване на NFA в DFA.
* **Алгоритми за проверка на думи:** Използване на алгоритми за обход на автомати, като дълбочинно или широчинно търсене, за проверка дали дума принадлежи на езика.
* **Обединение и конкатенация:** Използване на алгоритми за комбиниране на състояния и преходи на два автомата за изчисляване на обединението или конкатенацията.
* **Модели и стандарти:** Следване на стандартни алгоритми и структури за данни при реализиране на операциите с автоматите.

2.4 Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...) и

качествени (нефункционални) изисквания (скалируемост, поддръжка, ...)

**Потребителски (функционални) изисквания:**

* **Права и роли:** Потребителят трябва да може да зарежда файлове, да създава нови автомати, да извършва операции като обединение и конкатенация, и да проверява думи.
* **Статуси и диаграми:** Приложението трябва да предоставя визуално представяне на автоматите и да показва текущото състояние на операциите.

**Качествени (нефункционални) изисквания:**

* **Скалируемост:** Приложението трябва да може да обработва голям брой състояния и преходи без значително влошаване на производителността.
* **Поддръжка:** Кодът трябва да бъде лесен за поддръжка и разширяване, като използва модулен и добре документиран подход.
* **Производителност:** Операциите върху автоматите трябва да се изпълняват ефективно, за да се осигури бърза обратна връзка към потребителя.

Глава 3. Проектиране

3.1. Обща структура на проекта пакети който ще се реализират

Структура:

project4/

├── src/

│ └── Terminal/

│ ├── App/

│ │ └── Main.java

│ ├── CommandProcessor/

│ │ ├── CommandExecutor.java

│ │ ├── CommandType.java

│ │ └── Automaton/

│ ├── Commands/

│ │ ├── Command.java

│ │ ├── FileCommands/

│ │ │ ├── CloseCommand.java

│ │ │ ├── ExitCommand.java

│ │ │ ├── HelpCommand.java

│ │ │ ├── OpenCommand.java

│ │ │ ├── SaveAsCommand.java

│ │ │ └── SaveCommand.java

│ │ ├── ProjectCommands/

│ │ │ ├── ConcatCommand.java

│ │ │ ├── DeterministicCommand.java

│ │ │ ├── EmptyCommand.java

│ │ │ ├── FiniteCommand.java

│ │ │ ├── ListCommand.java

│ │ │ ├── MutatorCommand.java

│ │ │ ├── PrintCommand.java

│ │ │ ├── RecognizeCommand.java

│ │ │ ├── RegCommand.java

│ │ │ ├── SaveaCommand.java

│ │ │ ├── UnCommand.java

│ │ │ └── UnionCommand.java

Пакети:  
-tuvarna.leten2024.grupa2a.f22621663.FiniteAutomataProject.Terminal.App

-tuvarna.leten2024.grupa2a.f22621663.FiniteAutomataProject.Terminal.CommandProcessor.\*

-tuvarna.leten2024.grupa2a.f22621663.FiniteAutomataProject.Terminal.Commands.Command

-tuvarna.

leten2024.grupa2a.f22621663.FiniteAutomataProject.Terminal.Commands.FileCommands

- tuvarna.

leten2024.grupa2a.f22621663.FiniteAutomataProject.Terminal.Commands.ProjectCommands

3.2. Най-важните извадки от кода, структура и поведение, Диаграми/Блок схеми

- *App/Main.java*

Main {  
 CommandExecutor executor = new CommandExecutor();  
   
 String input;

while (true){  
 System.*out*.print("> ");  
  
 input = scanner.nextLine().trim();  
 String[] commandParts = input.split("\\s+");  
 String command = commandParts[0];  
   
 executor.executeCommand(command, commandArgs);  
 }  
}

 **Инициализация на CommandExecutor**:

* CommandExecutor executor = new CommandExecutor();
  + Създава се обект от клас CommandExecutor, който ще отговаря за изпълнението на въведените команди.

 **Инициализация на променлива input**:

* String input;
  + Декларира се променлива input, която ще съдържа входа от потребителя.

 **Главен цикъл**:

* while (true){ ... }
  + Започва безкраен цикъл, който ще продължи да работи, докато програмата не бъде спряна.

 **Отпечатване на промпт**:

* System.out.print("> ");
  + Отпечатва промпт символа ">", който указва на потребителя, че трябва да въведе команда.

 **Четене на входа**:

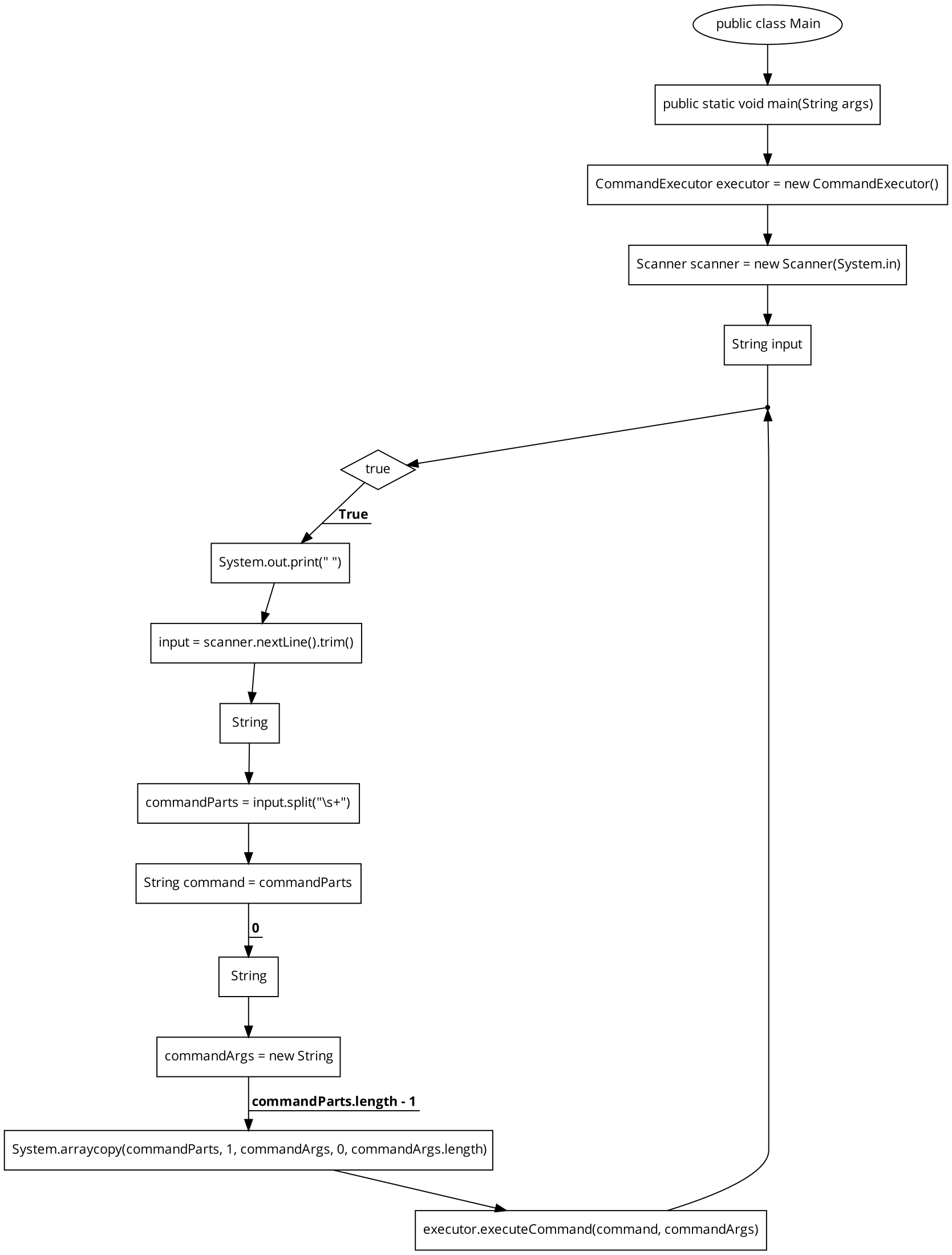
* input = scanner.nextLine().trim();
  + Чете входа от потребителя чрез обект scanner, премахва всички водещи и завършващи интервали с trim() и го запазва в променливата input.

 **Разделяне на командата и аргументите**:

* String[] commandParts = input.split("\\s+");
  + Разделя входния низ input на части, като използва един или повече интервали като разделители. Резултатът е масив commandParts, който съдържа командата и нейните аргументи.
* String command = commandParts[0];
  + Извлича първия елемент от масива commandParts, който е самата команда.

 **Изпълнение на командата**:

* executor.executeCommand(command, commandArgs);
  + Извиква метода executeCommand на executor, като му предава командата и аргументите за изпълнение. (Забележка: В псевдокода липсва извличането на аргументите commandArgs от commandParts, което е направено в оригиналния код на Java.)



*- CommandProcessor/CommandExecutor.java*

public class CommandExecutor {  
 private final Map<CommandType, Command> commandMap;  
 private boolean fileOpened;  
 private String currentFileName;  
  
 public CommandExecutor() {  
 commandMap = new HashMap<>();  
 initializeCommands();  
 }  
  
 private void initializeCommands() {  
 commandMap.put(CommandType.*Команда*, new ИмеCommand(this));

.

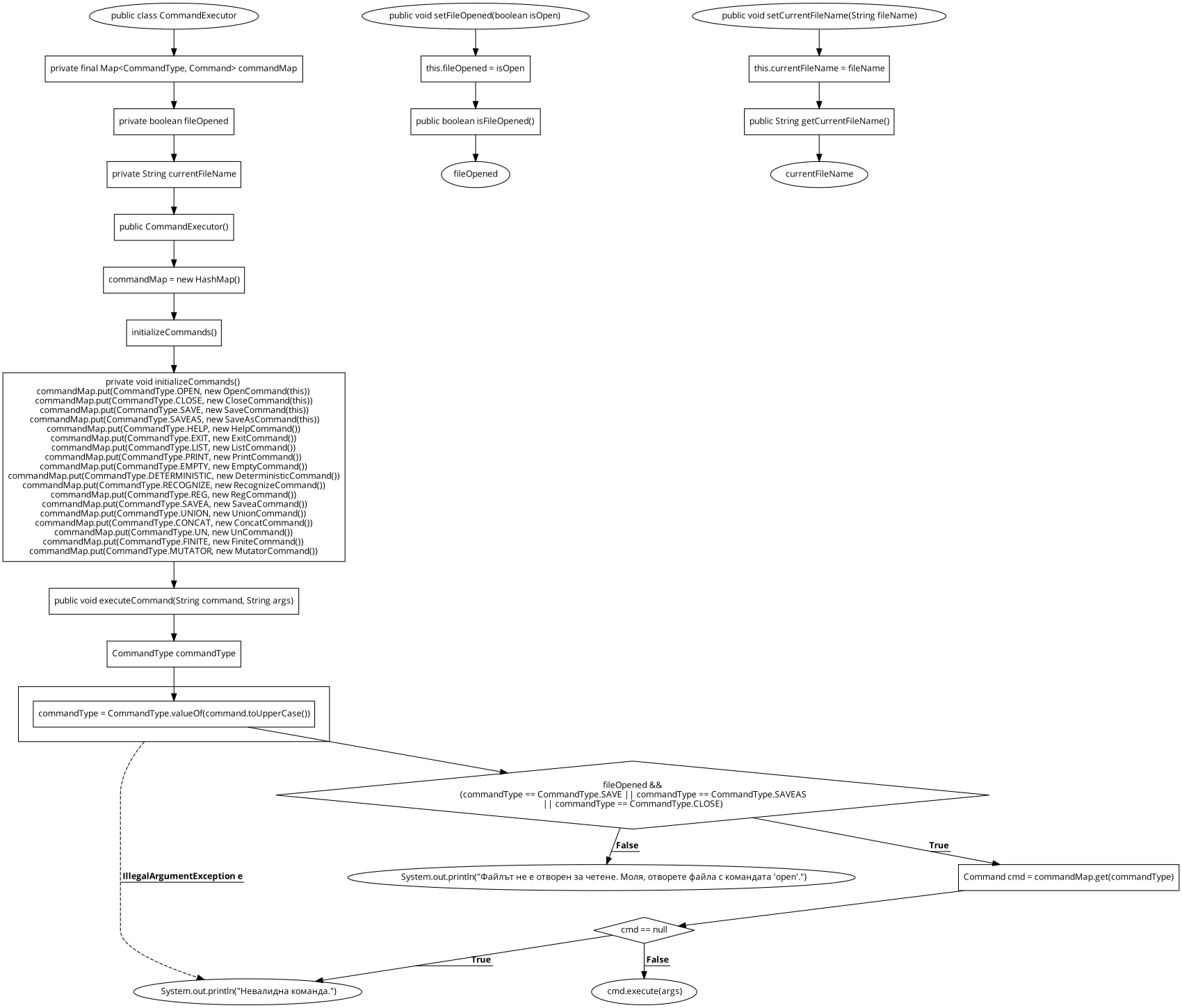
.

.  
 }  
  
 public void executeCommand(String command, String[] args) {  
 CommandType commandType;  
 try {  
 commandType = CommandType.*valueOf*(command.toUpperCase());  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Невалидна команда.");  
 return;  
 }  
  
  
  
 if (!fileOpened &&  
 (commandType == CommandType.*SAVE* || commandType == CommandType.*SAVEAS* || commandType == CommandType.*CLOSE*)) {  
 System.*out*.println("Файлът не е отворен за четене. Моля, отворете файла с командата 'open'.");  
 return;  
 }  
  
  
  
 Command cmd = commandMap.get(commandType);  
 if (cmd == null) {  
 System.*out*.println("Невалидна команда.");  
 return;  
 }  
  
  
 cmd.execute(args);  
 }

Класът CommandExecutor е ключов компонент в приложението, който управлява изпълнението на различни команди чрез използването на мапиране между типове команди (CommandType) и техните конкретни изпълнители (Command). Тази архитектура прави класа мощен инструмент за управление на операции в приложението, като позволява лесното добавяне и манипулиране на команди.

**Обяснение на структурата и функционалността на CommandExecutor:**

1. **Променливи:**
   * commandMap: Тази променлива е от тип Map<CommandType, Command> и съхранява съответствията между типовете команди и техните изпълнители.
   * fileOpened: Булева променлива, която указва дали файлът е отворен за четене или запис.
   * currentFileName: Стринг, който съдържа името на текущо отворения файл, ако такъв съществува.
2. **Конструктор (CommandExecutor()):**
   * Създава нова инстанция на CommandExecutor.
   * Инициализира commandMap като празна HashMap.
   * Извиква метода initializeCommands(), който попълва commandMap с всички поддържани команди и техните изпълнители.
3. **Метод initializeCommands():**
   * Добавя всеки тип команда (CommandType) като ключ в commandMap.
   * Създава нов обект от съответния тип Command (например OpenCommand, CloseCommand и други) като стойност и го добавя в commandMap.
4. **Метод executeCommand(String command, String[] args):**
   * Приема два параметъра: command, който е името на командата, и args, който е масив от аргументи за изпълнението на командата.
   * Преобразува името на командата command в съответния тип CommandType чрез метода CommandType.valueOf(command.toUpperCase()).
   * Хваща възможното изключение IllegalArgumentException, ако командата не е валидна за текущото приложение.
   * Проверява дали файлът е отворен, преди да изпълни команди като SAVE, SAVEAS или CLOSE. Ако файлът не е отворен, отпечатва съобщение за необходимост от отваряне на файла.
   * Извлича конкретния изпълнител (Command) от commandMap за дадения тип команда (commandType).
   * Ако изпълнителят (cmd) е null, отпечатва съобщение за невалидна команда.
   * Изпълнява командата, като предава аргументите args на метода execute() на изпълнителя (cmd).
5. **Методи за управление на файлове:**
   * setFileOpened(boolean isOpen): Задава стойността на fileOpened.
   * isFileOpened(): Проверява дали файлът е отворен.
   * setCurrentFileName(String fileName): Задава името на текущия файл.
   * getCurrentFileName(): Връща името на текущия файл.

Тази структура позволява лесно добавяне на нови команди и тяхното изпълнение в приложението, като осигурява гъвкавост и разширяемост при управлението на операции и файлове. Класът CommandExecutor играе централна роля в координацията на действията в приложението чрез мапиране на типове команди на техните изпълнители, осигурявайки ефективно и удобно управление на различни аспекти на работата с файлове и операции.

*- FileCommands/OpenCommand.java*

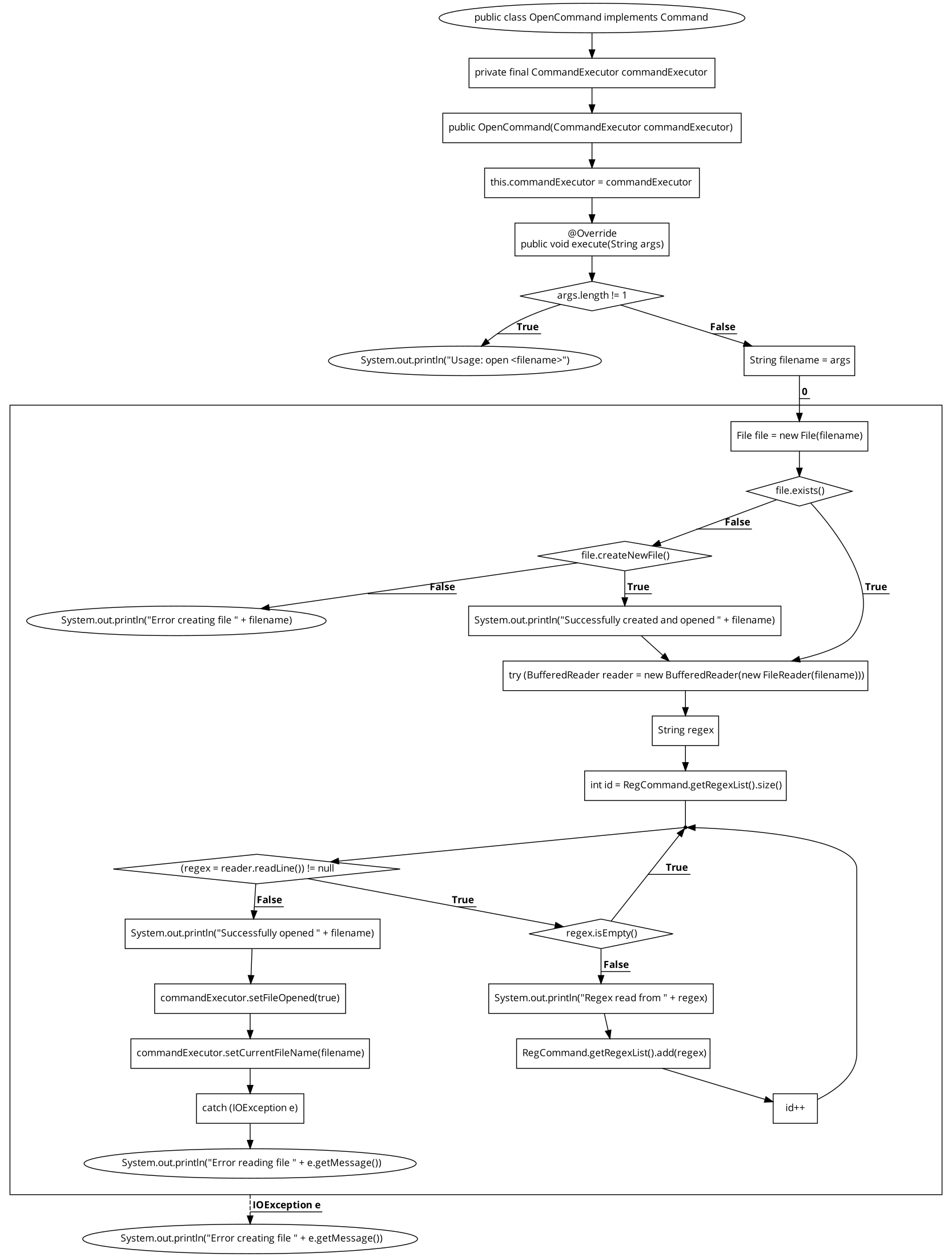
Class OpenCommand implements Command {  
 CommandExecutor commandExecutor;  
  
 Constructor OpenCommand(CommandExecutor commandExecutor) {  
 this.commandExecutor = commandExecutor;  
 }  
  
 Method execute(String[] args) {  
 If (args.length != 1) {  
 Print "Usage: open <filename>"  
 Return  
 }  
  
 String filename = args[0];  
 Try {  
 File file = new File(filename);  
 If (!file.exists()) {  
 If (file.createNewFile()) {  
 Print "Successfully created and opened " + filename  
 } Else {  
 Print "Error creating file " + filename  
 Return  
 }  
 }  
  
 Try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {  
 String regex;  
 Integer id = RegCommand.getRegexList().size();  
  
 While ((regex = reader.readLine()) != null) {  
 If (!regex.isEmpty()) {  
 Print "Regex read from " + regex  
 RegCommand.getRegexList().add(regex);  
 id++;  
 }  
 }  
  
 Print "Successfully opened " + filename  
 commandExecutor.setFileOpened(true)  
 commandExecutor.setCurrentFileName(filename)  
  
 } Catch (IOException e) {  
 Print "Error reading file " + e.getMessage()  
 }  
  
 } Catch (IOException e) {  
 Print "Error creating file " + e.getMessage()  
 }  
 }  
 }

 **Променливи и конструктор**:

* commandExecutor: Променлива, която държи референция към CommandExecutor, отговорен за изпълнението на команди.
* Конструкторът инициализира commandExecutor със съответния подаден обект от тип CommandExecutor.

 **Метод execute**:

* Методът execute е отговорен за изпълнението на командата "open", която служи за отваряне на файл.
* Приема масив от аргументи args, очакващ само един аргумент - името на файла за отваряне.
* Проверява дали е подаден точно един аргумент. Ако не е, извежда съобщение за правилна употреба и приключва изпълнението на метода.
* Извлича името на файла от първия елемент на масива args.
* Опитва се да създаде нов файл с даденото име, ако такъв не съществува. В случай на успешно създаване, извежда съобщение за успешно създаден и отворен файл.
* Ако създаването на файла не е успешно, извежда съобщение за грешка и приключва изпълнението.
* Чете съдържанието на файла чрез BufferedReader и добавя прочетените регулярни изрази в списъка (предполагаемо RegCommand.getRegexList()).
* Накрая, отбелязва файлът като успешно отворен в commandExecutor, като задава fileOpened на true и записва текущото име на файла.



*- FileCommands/CloseCommand.java*

Class CloseCommand implements Command {  
 CommandExecutor commandExecutor;  
  
 Constructor CloseCommand(CommandExecutor commandExecutor) {  
 this.commandExecutor = commandExecutor;  
 }  
  
 Method execute(String[] args) {  
 If (commandExecutor.isFileOpened()) {  
 Print "Файлът е затворен."  
 commandExecutor.setFileOpened(false)  
 } Else {  
 Print "Няма отворен файл, който да затворите."  
 }  
 }  
 }

**Обяснение:**

**Променливи и конструктор:**

* commandExecutor: Тази променлива е от тип CommandExecutor и се използва за достъп до методите и състоянието на CommandExecutor.
* Конструкторът CloseCommand приема един аргумент от тип CommandExecutor и инициализира полето commandExecutor с него. Това позволява командата CloseCommand да взаимодейства с CommandExecutor.

**Метод execute:**

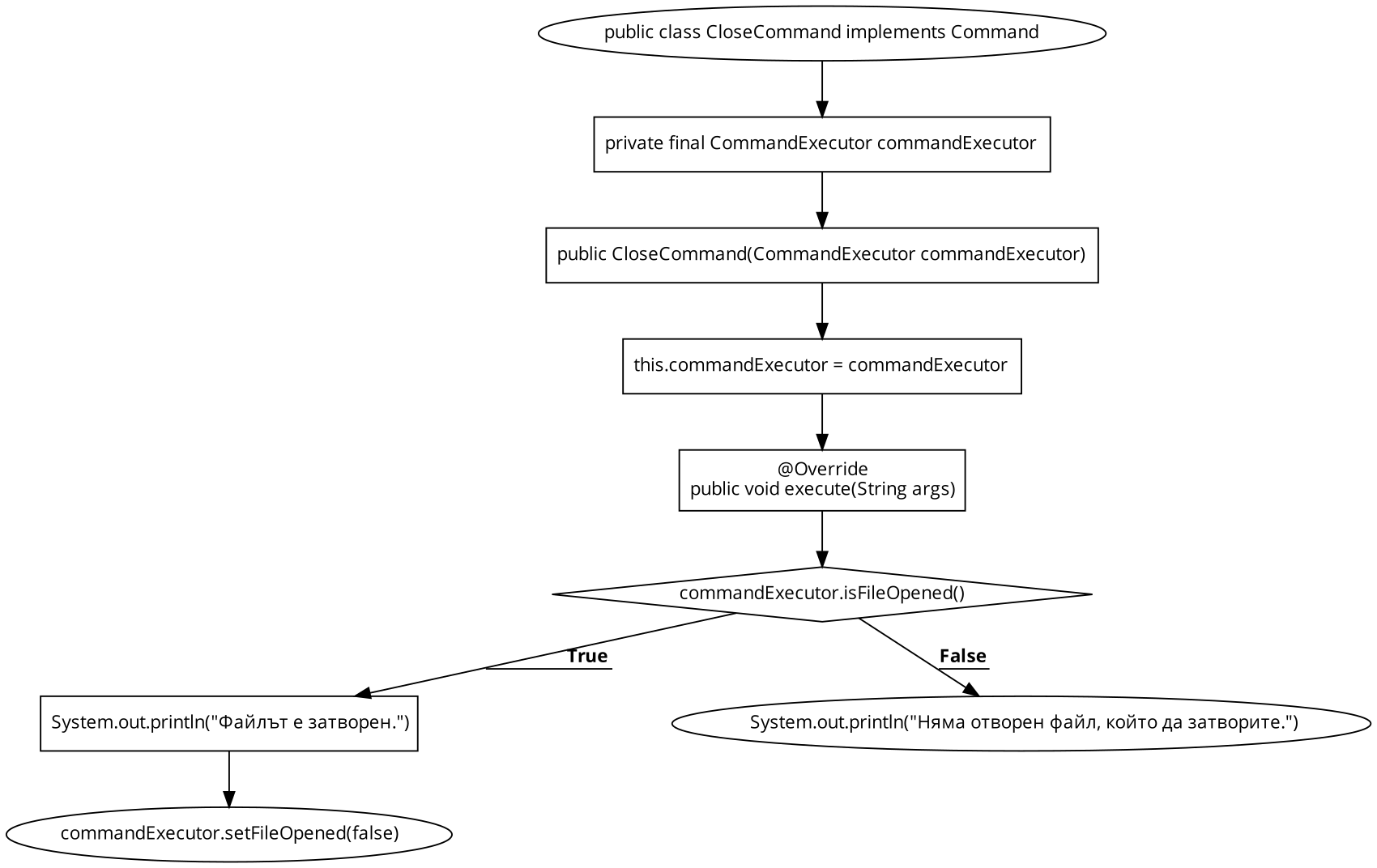
* Методът execute се изпълнява, когато командата CloseCommand бъде извикана.
* Приема масив от аргументи args, въпреки че в този случай не се използва, защото командата CloseCommand не изисква допълнителни аргументи.

**Проверка и затваряне на файла:**

* Първо се проверява дали има отворен файл чрез метода isFileOpened() на commandExecutor.
* Ако резултатът от проверката е true, т.е. има отворен файл:
  + Извежда съобщение "Файлът е затворен."
  + Задава стойността на fileOpened на false чрез метода setFileOpened(false), което означава, че няма отворен файл в момента.
* Ако резултатът от проверката е false, т.е. няма отворен файл:
  + Извежда съобщение "Няма отворен файл, който да затворите."

**Заключение:**

Този псевдо код и обяснението му показват как командата CloseCommand работи за затваряне на отворен файл в контекста на приложение, което използва команди. Командата проверява състоянието на CommandExecutor и изпълнява подходящи действия в зависимост от това дали има отворен файл или не.

**

*- FileCommands/HelpCommand.java*

public class HelpCommand implements Command {  
 @Override  
 public void execute(String[] args) {  
 if (args.length != 0) {  
 System.*out*.println("ERROR! Expected command: help");  
 return;  
 }  
  
 StringBuilder helpMessage = new StringBuilder("The following commands are supported:\n\n");

.

.

.  
  
 helpMessage.append("open <file> opens <file>\n");  
 System.*out*.println(helpMessage);  
 }  
}

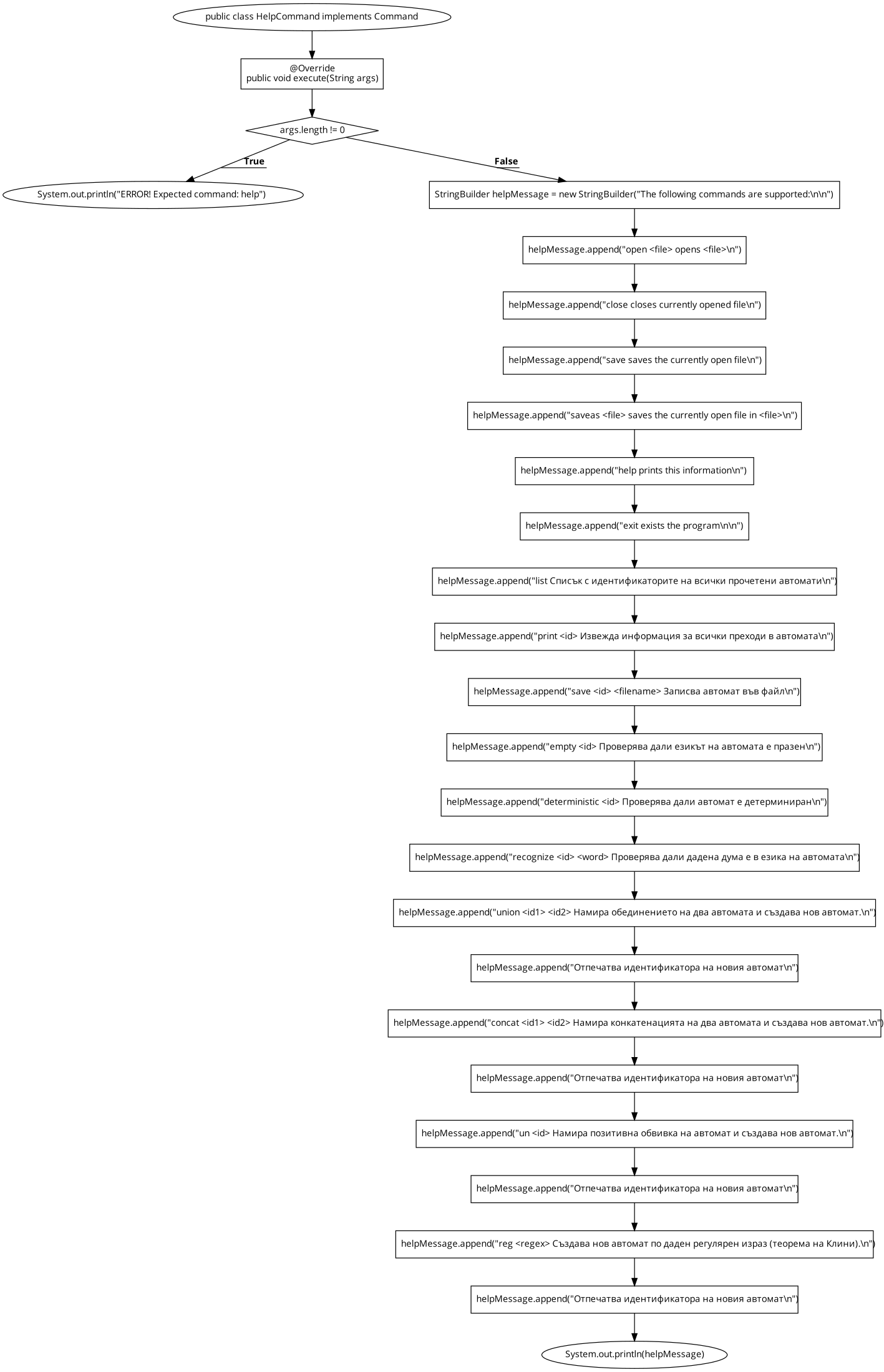
Този код дефинира клас HelpCommand, който имплементира интерфейса Command. Класът има за цел да предостави помощна информация за поддържаните команди в приложението.

**Променливи и конструктор:**

Класът HelpCommand няма член-данни и конструктор, защото неговата функционалност е дефинирана изцяло в метода execute.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args, които могат да бъдат подадени от потребителя.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е нула (т.е. потребителят е подал някакви аргументи), извежда се съобщение за грешка: "ERROR! Expected command: help".
  + След това методът се връща, като не изпълнява останалата част от кода.
* **Създаване на помощно съобщение:**
  + Създава се обект StringBuilder с начално съдържание "The following commands are supported:\n\n", който ще съдържа помощното съобщение.
  + В останалата част от кода (заместена с "...") се добавят описания на поддържаните команди към helpMessage.
  + Добавя се описанието на командата open към helpMessage чрез helpMessage.append("open <file> opens <file>\n");.
  + Накрая, помощното съобщение се извежда на конзолата чрез System.out.println(helpMessage);.

**

*- FileCommands/ExitCommand.java*

Class ExitCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length > 1) {  
 Print "ERROR! Unexpected argument: " + args[1]  
 Return  
 }  
  
 If (args.length == 1 AND NOT args[0].trim().equals("exit")) {  
 Print "ERROR! Unexpected argument: " + args[0]  
 Return  
 }  
  
 Print "Exiting the program..."  
 System.exit(0)  
 }  
 }

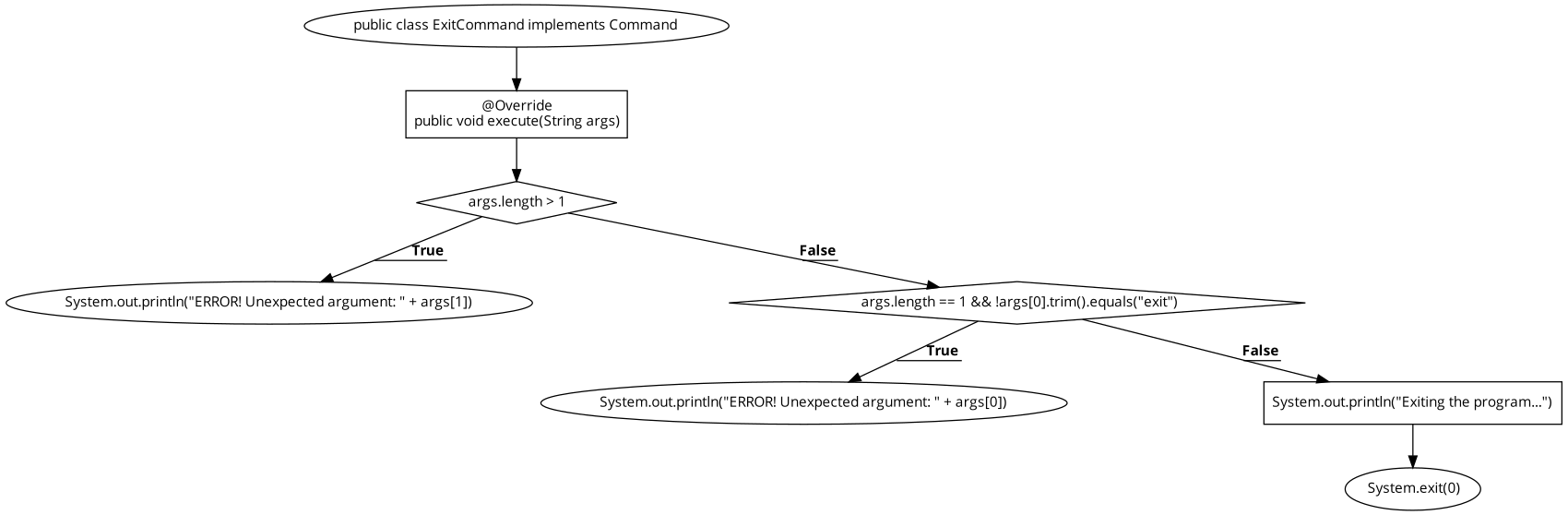
Класът ExitCommand имплементира интерфейса Command. Този клас е предназначен да изпълнява командата за изход от програмата.

**Променливи и конструктор:**

Класът ExitCommand няма член-данни и конструктор, защото неговата функционалност е дефинирана изцяло в метода execute.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args, които могат да бъдат подадени от потребителя.
* **Проверка на аргументите:**
  + Ако дължината на масива args е повече от 1 (т.е. потребителят е подал повече от един аргумент), извежда се съобщение за грешка: "ERROR! Unexpected argument: " + args[1] и методът се връща.
  + Ако дължината на масива args е 1 и аргументът не е равен на "exit", извежда се съобщение за грешка: "ERROR! Unexpected argument: " + args[0] и методът се връща.
* **Изпълнение на командата за изход:**
  + Ако няма грешки в аргументите, се извежда съобщение: "Exiting the program..." и програмата приключва изпълнението си с System.exit(0).

**

*- FileCommands/SaveCommand.java*

Class SaveCommand implements Command {  
 CommandExecutor commandExecutor  
  
 Constructor SaveCommand(CommandExecutor commandExecutor) {  
 this.commandExecutor = commandExecutor  
 }  
  
 Method execute(String[] args) {  
 If (args.length != 0) {  
 Print "Usage: save"  
 Return  
 }  
  
 String filename = commandExecutor.getCurrentFileName()  
  
 If (filename == null) {  
 Print "No file opened. Please open a file first."  
 Return  
 }  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 Try (PrintWriter writer = new PrintWriter(new FileWriter(filename))) {  
 For (String regex : regexList) {  
 writer.println(regex)  
 }  
 Print "Automatons saved successfully to " + filename  
 } Catch (IOException e) {  
 Print "Error saving automatons to file: " + e.getMessage()  
 }  
 }  
 }

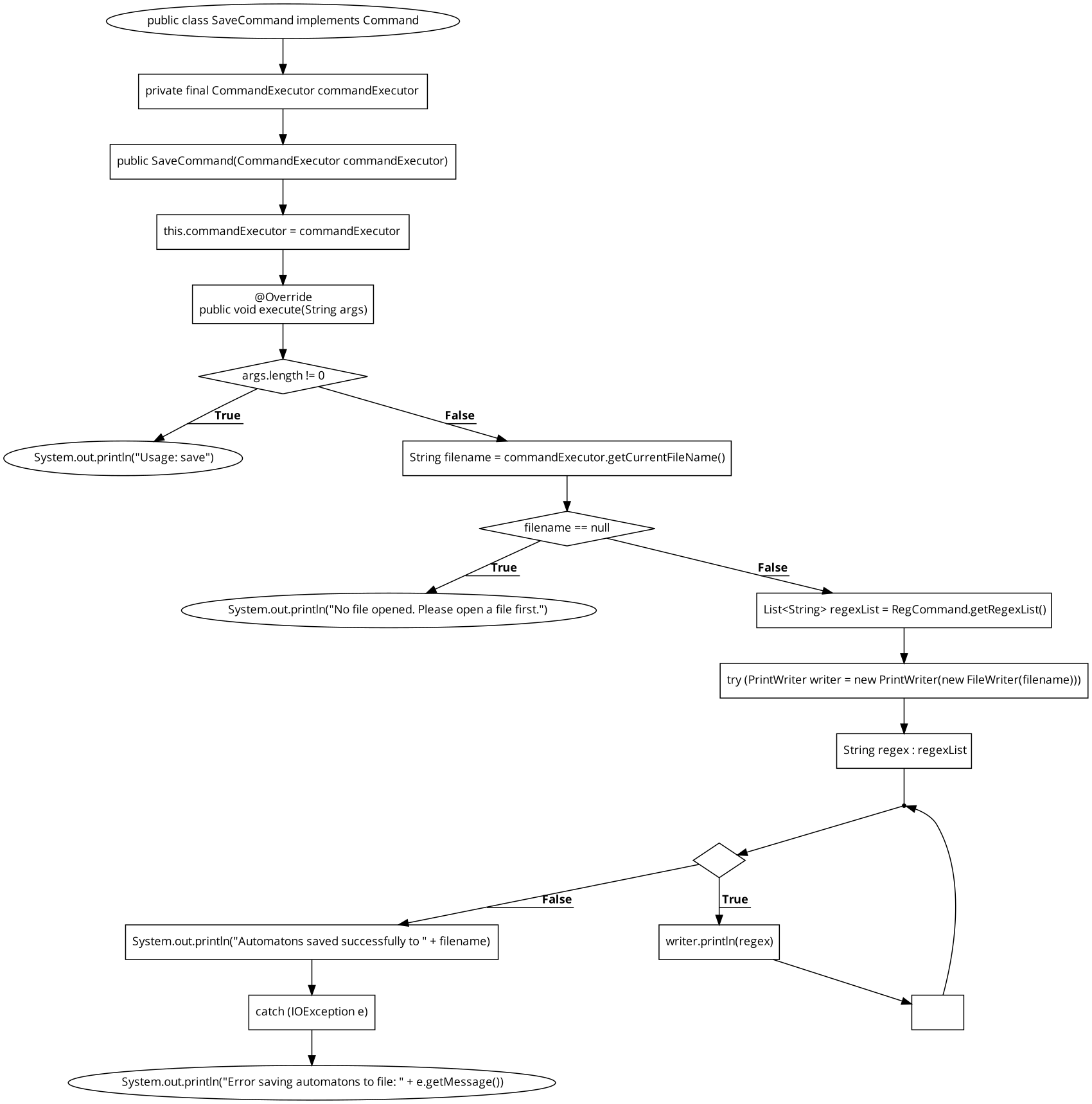
Класът SaveCommand имплементира интерфейса Command и е предназначен да изпълнява командата за запазване на текущо отворения файл.

**Променливи и конструктор:**

* commandExecutor: Това е инстанция на CommandExecutor, която се използва за управление на състоянието на текущо отворения файл и извършване на операции върху него.
* **Конструктор:** Конструкторът приема CommandExecutor като параметър и го присвоява на член-данната commandExecutor.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е равна на 0, се извежда съобщение за грешка: "Usage: save" и методът се връща.
* **Извличане на името на текущия файл:**
  + filename се извлича чрез метода commandExecutor.getCurrentFileName().
  + Ако няма отворен файл (filename е null), се извежда съобщение: "No file opened. Please open a file first." и методът се връща.
* **Записване на данни:**
  + Създава се списък regexList, който съдържа всички регулярни изрази от RegCommand.getRegexList().
  + Използва се PrintWriter за записване на съдържанието на списъка в текущия файл.
  + Всеки ред от списъка с регулярни изрази се записва във файла.
  + Ако записът е успешен, се извежда съобщение: "Automatons saved successfully to " + filename.
  + Ако възникне грешка по време на записването, се извежда съобщение за грешка.

**

*- FileCommands/SaveAsCommand.java*

Class SaveAsCommand implements Command {  
 CommandExecutor commandExecutor  
  
 Constructor SaveAsCommand(CommandExecutor commandExecutor) {  
 this.commandExecutor = commandExecutor  
 }  
  
 Method execute(String[] args) {  
 If (args.length != 2) {  
 Print "Usage: saveas <directory> <filename>"  
 Return  
 }  
  
 String directoryPath = args[0]  
 String fileName = args[1]  
  
 If (!commandExecutor.isFileOpened()) {  
 Print "No file opened. Please open a file first."  
 Return  
 }  
  
 Path directory = Paths.get(directoryPath)  
 String currentFileName = commandExecutor.getCurrentFileName()  
 Path currentFilePath = Paths.get(currentFileName)  
 Path newFilePath = directory.resolve(fileName)  
  
 Try (PrintWriter writer = new PrintWriter(Files.newBufferedWriter(newFilePath))) {  
 For (String regex : RegCommand.getRegexList()) {  
 writer.println(regex)  
 }  
 Print "Automatons saved successfully to " + newFilePath  
 } Catch (IOException e) {  
 Print "Error saving automatons to file: " + e.getMessage()  
 }  
 }

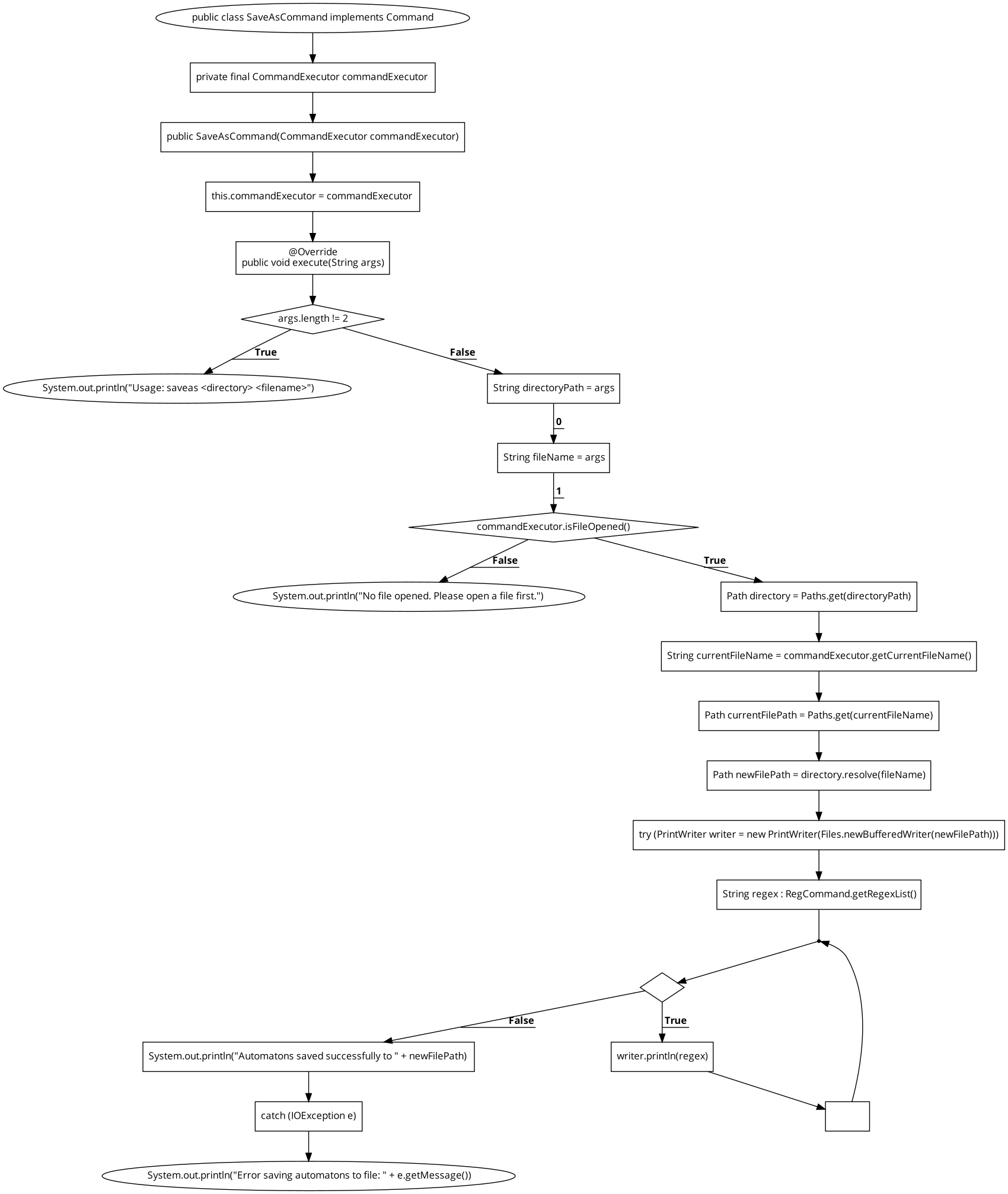
Класът SaveAsCommand имплементира интерфейса Command и е предназначен да изпълнява командата за запазване на текущия файл в нова директория и с ново име.

**Променливи и конструктор:**

* commandExecutor: Това е инстанция на CommandExecutor, която се използва за управление на състоянието на текущо отворения файл и извършване на операции върху него.
* **Конструктор:** Конструкторът приема CommandExecutor като параметър и го присвоява на член-данната commandExecutor.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args, които се очаква да съдържат директория и име на файл.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е равна на 2, се извежда съобщение за грешка: "Usage: saveas <directory> <filename>" и методът се връща.
* **Извличане на параметрите:**
  + directoryPath и fileName се извличат от масива args.
* **Проверка дали файлът е отворен:**
  + Ако няма отворен файл, се извежда съобщение: "No file opened. Please open a file first." и методът се връща.
* **Създаване на пътища:**
  + Създават се пътища за текущия файл и за новия файл в указаната директория.
* **Записване на данни:**
  + Използва се PrintWriter за записване на съдържанието от текущия файл в новия файл.
  + Всеки ред от списъка с регулярни изрази се записва в новия файл.
  + Ако записът е успешен, се извежда съобщение: "Automatons saved successfully to " + newFilePath.
  + Ако възникне грешка по време на записването, се извежда съобщение за грешка.

**

*- ProjectCommands/ConcatCommand*

Class ConcatCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length != 2) {  
 Print "ERROR! Expected command: concat <id1> <id2>"  
 Return  
 }  
  
 Try {  
 int id1 = Integer.parseInt(args[0])  
 int id2 = Integer.parseInt(args[1])  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 If (id1 < 0 OR id1 >= regexList.size() OR id2 < 0 OR id2 >= regexList.size()) {  
 Print "Invalid IDs."  
 Return  
 }  
  
 String regex1 = regexList.get(id1)  
 String regex2 = regexList.get(id2)  
  
 String concatRegex = regex1 + regex2  
 RegCommand regCommand = new RegCommand()  
 regCommand.execute([concatRegex])  
  
 Print "Concatenation of regexes with IDs " + id1 + " and " + id2 + " created with ID: " + (regexList.size() - 1)  
 } Catch (NumberFormatException e) {  
 Print "Invalid IDs."  
 }  
 }  
 }

Класът ConcatCommand имплементира интерфейса Command и е предназначен за конкатенация на два регулярни израза, съхранявани в списъка с регулярни изрази.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е равна на 2, се извежда съобщение за грешка: "ERROR! Expected command: concat <id1> <id2>" и методът се връща.
* **Опит за преобразуване на аргументите в цели числа:**
  + Методът се опитва да преобразува първия и втория аргумент в цели числа (id1 и id2).
  + Ако възникне изключение NumberFormatException, се извежда съобщение за грешка: "Invalid IDs." и методът се връща.
* **Проверка на валидността на идентификаторите:**
  + Създава се списък regexList, който съдържа всички регулярни изрази от RegCommand.getRegexList().
  + Проверява се дали id1 и id2 са валидни индекси за списъка regexList. Ако не са валидни, се извежда съобщение: "Invalid IDs." и методът се връща.
* **Конкатенация на регулярните изрази:**
  + Извличат се регулярните изрази от списъка regexList съответно на позициите id1 и id2.
  + Създава се нов редовен израз чрез конкатенация на двата израза.
  + Създава се нов обект от тип RegCommand и се извиква неговият метод execute с новия конкатениран редовен израз като аргумент.
* **Извеждане на резултат:**
  + *Картина, която съдържа черен, тъмнина

    Описанието е генерирано автоматично*Извежда се съобщение за успешна конкатенация: "Concatenation of regexes with IDs " + id1 + " and " + id2 + " created with ID: " + (regexList.size() - 1).

*- ProjectCommands/DeterministicCommand*

Class DeterministicCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length != 1) {  
 Print "Usage: deterministic <id>"  
 Return  
 }  
  
 Try {  
 int id = Integer.parseInt(args[0])  
 List<String> regexList = RegCommand.getRegexList()  
  
 If (id < 0 OR id >= regexList.size()) {  
 Print "Invalid ID."  
 Return  
 }  
  
 String regex = regexList.get(id)  
 boolean deterministic = isDeterministic(regex)  
  
 If (deterministic) {  
 Print "The automaton with ID " + id + " is deterministic."  
 } Else {  
 Print "The automaton with ID " + id + " is not deterministic."  
 }  
 } Catch (NumberFormatException e) {  
 Print "Invalid ID."  
 }  
 }  
  
 Private Method isDeterministic(String regex) {  
 int length = regex.length()  
 For (int i = 0; i < length; i++) {  
 char currentChar = regex.charAt(i)  
 If (currentChar == '\*') {  
 If (i + 1 < length) {  
 char nextChar = regex.charAt(i + 1)  
 If (nextChar == regex.charAt(i - 1)) {  
 Return false  
 }  
 }  
 }  
 If (currentChar == '+') {  
 If (i + 1 < length) {  
 char nextChar = regex.charAt(i + 1)  
 If (nextChar == regex.charAt(0)) {  
 Return false  
 }  
 }  
 }  
 }  
 Return true  
 }  
 }

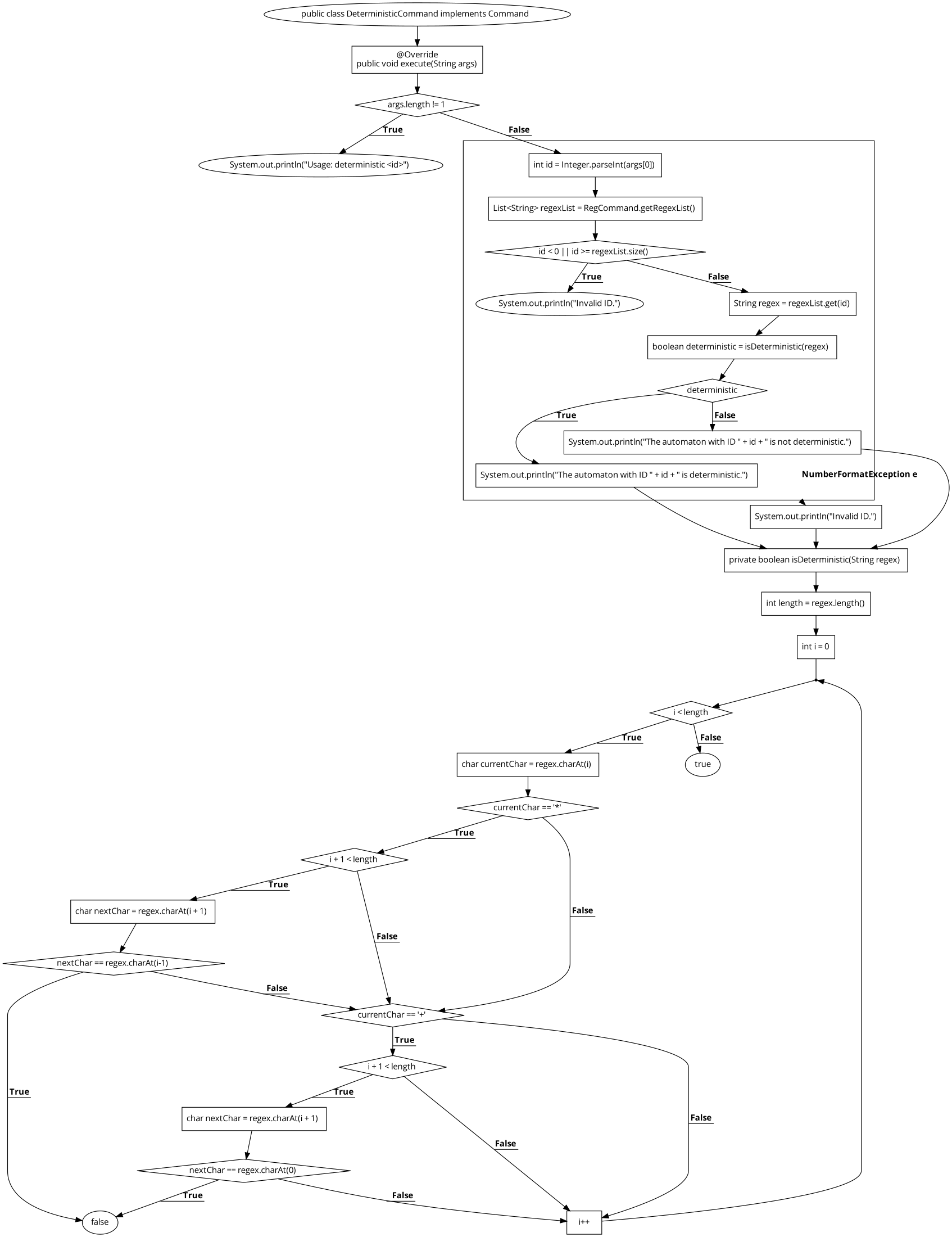
Класът DeterministicCommand имплементира интерфейса Command и проверява дали даден регулярен израз представлява детерминистичен автомат.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е равна на 1, се извежда съобщение за грешка: "Usage: deterministic <id>" и методът се връща.
* **Опит за преобразуване на аргумента в цяло число:**
  + Методът се опитва да преобразува първия аргумент в цяло число (id).
  + Ако възникне изключение NumberFormatException, се извежда съобщение за грешка: "Invalid ID." и методът се връща.
* **Проверка на валидността на идентификатора:**
  + Създава се списък regexList, който съдържа всички регулярни изрази от RegCommand.getRegexList().
  + Проверява се дали id е валиден индекс за списъка regexList. Ако не е валиден, се извежда съобщение: "Invalid ID." и методът се връща.
* **Проверка за детерминистичност:**
  + Извлича се регулярният израз от списъка regexList на позиция id.
  + Методът isDeterministic се извиква с този регулярен израз, за да провери дали изразът е детерминистичен.
  + Въз основа на резултата от isDeterministic се извежда съответното съобщение: дали автоматът е детерминистичен или не.

**Метод isDeterministic:**

* **Параметри:** Методът isDeterministic приема стринг regex.
* **Проверка за детерминистичност:**
  + Методът обхожда всеки символ в регулярния израз.
  + Проверява се дали символът '\*' или '+' следва символ, който се повтаря.
  + Ако се намери такъв символ, методът връща false, означавайки че изразът не е детерминистичен.
  + Ако няма такива символи, методът връща true, означавайки че изразът е детерминистичен.

**

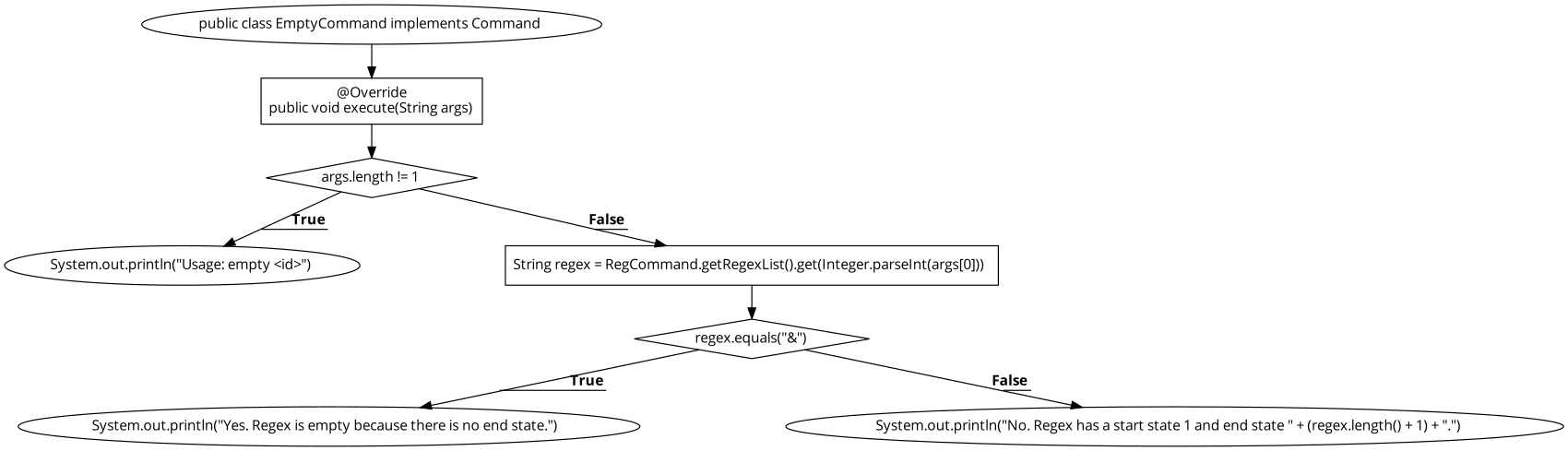
*- ProjectCommands/EmptyCommand*

Class EmptyCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length != 1) {  
 Print "Usage: empty <id>"  
 Return  
 }  
  
 String regex = RegCommand.getRegexList().get(Integer.parseInt(args[0]))  
  
 If (regex.equals("&")) {  
 Print "Yes. Regex is empty because there is no end state."  
 } Else {  
 Print "No. Regex has a start state 1 and end state " + (regex.length() + 1) + "."  
 }  
 }  
 }

Класът EmptyCommand имплементира интерфейса Command и проверява дали даден регулярен израз е празен, което означава, че няма крайно състояние.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е равна на 1, се извежда съобщение за грешка: "Usage: empty <id>" и методът се връща.
* **Извличане на регулярния израз:**
  + Методът извлича регулярния израз от списъка RegCommand.getRegexList() на позиция, определена от първия аргумент (args[0]), преобразуван в цяло число.
* **Проверка за празен редовен израз:**
  + Ако редовният израз е равен на символа '&', това означава, че няма крайно състояние, и се извежда съобщение: "Yes. Regex is empty because there is no end state."
  + Ако редовният израз не е равен на '&', се извежда съобщение: "No. Regex has a start state 1 and end state " + (regex.length() + 1) + ".", което указва, че редовният израз има начално състояние 1 и крайно състояние, равно на дължината на редовния израз плюс 1.

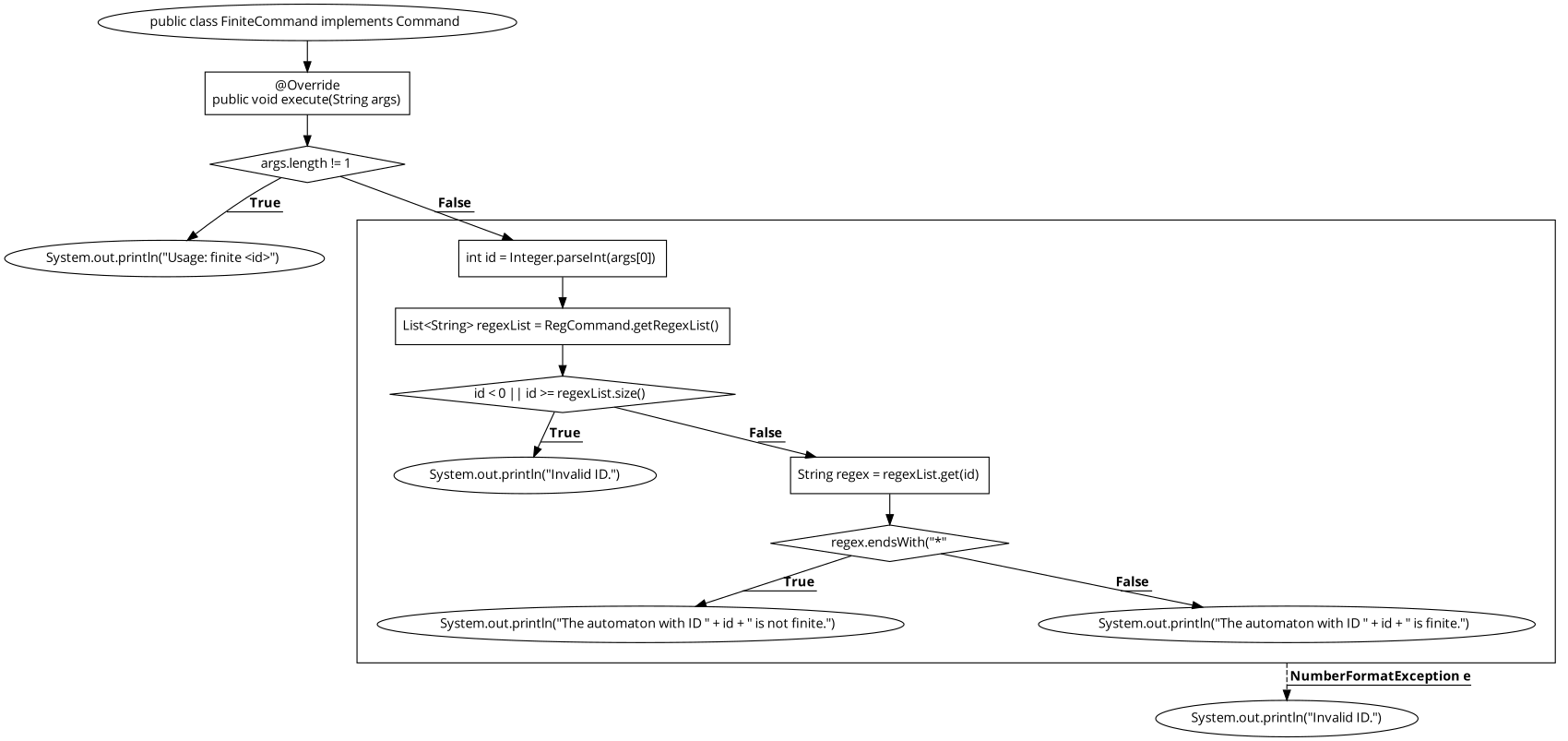
**

*- ProjectCommands/FiniteCommand*

Class FiniteCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length != 1) {  
 Print "Usage: finite <id>"  
 Return  
 }  
  
 Try {  
 Integer id = Integer.parseInt(args[0])  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 If (id < 0 OR id >= regexList.size()) {  
 Print "Invalid ID."  
 Return  
 }  
  
 String regex = regexList.get(id)  
  
 If (regex.endsWith("\*")) {  
 Print "The automaton with ID " + id + " is not finite."  
 } Else {  
 Print "The automaton with ID " + id + " is finite."  
 }  
  
 } Catch (NumberFormatException e) {  
 Print "Invalid ID."  
 }  
 }  
 }

Класът FiniteCommand имплементира интерфейса Command и проверява дали автоматът, представен чрез даден регулярен израз, е краен или не.

**Метод execute:**

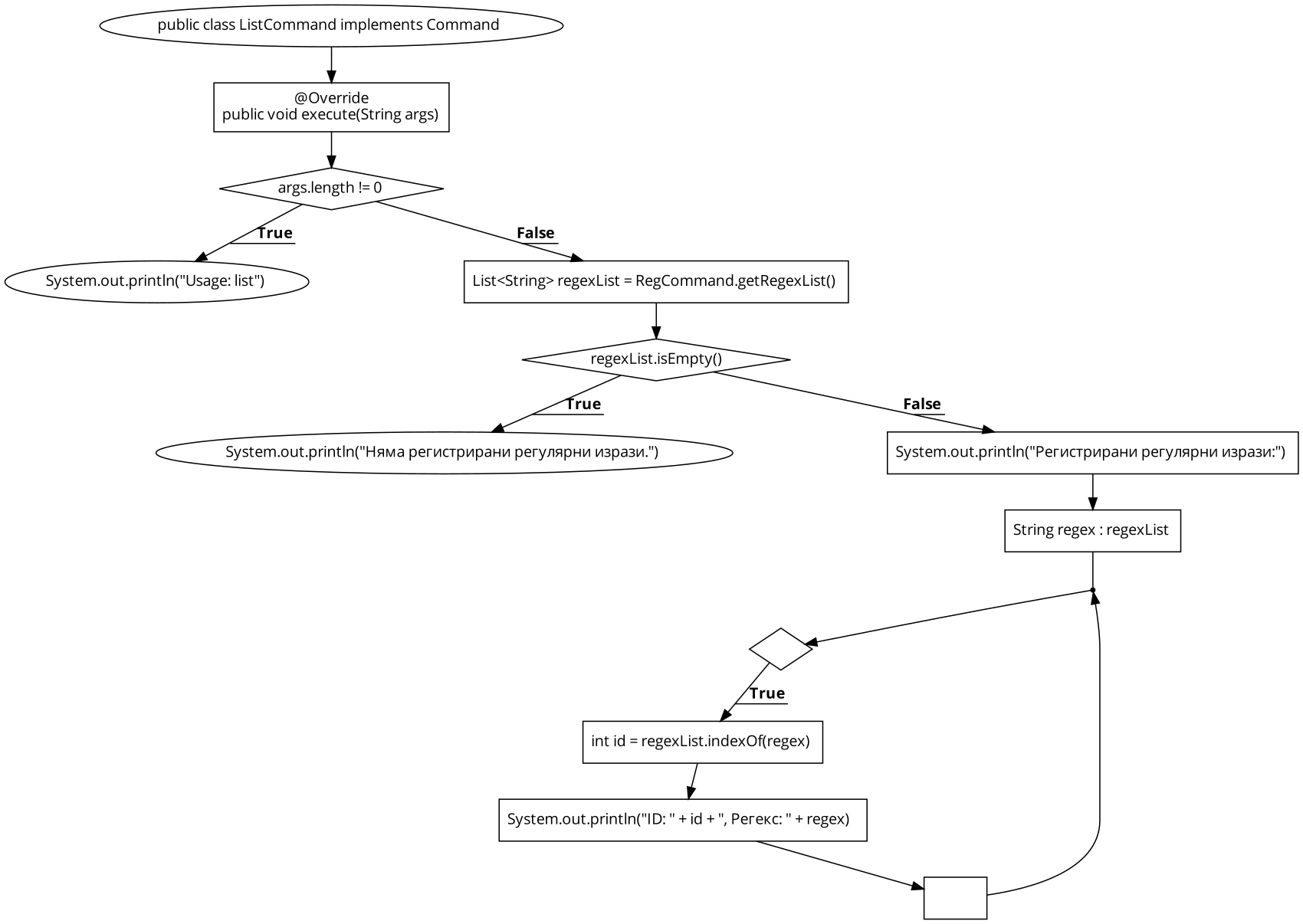
* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е равна на 1, се извежда съобщение за грешка: "Usage: finite <id>" и методът се връща.
* **Проверка и извличане на идентификатора:**
  + Методът се опитва да преобразува първия аргумент (args[0]) в цяло число id. Ако възникне изключение NumberFormatException, се извежда съобщение за грешка: "Invalid ID." и методът се връща.
* **Проверка на валидността на идентификатора:**
  + Ако id е по-малък от 0 или по-голям или равен на размера на списъка regexList, се извежда съобщение за грешка: "Invalid ID." и методът се връща.
* **Извличане на регулярния израз:**
  + Методът извлича редовния израз от списъка RegCommand.getRegexList() на позиция, определена от id.
* **Проверка за крайност на автомата:**
  + Ако редовният израз завършва със символа '\*', автоматът не е краен и се извежда съобщение: "The automaton with ID " + id + " is not finite."
  + **Ако редовният израз не завършва със символа '\*', автоматът е краен и се извежда съобщение: "The automaton with ID " + id + " is finite."

*- ProjectCommands/ListCommand*

Class ListCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length != 0) {  
 Print "Usage: list"  
 Return  
 }  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 If (regexList.isEmpty()) {  
 Print "Няма регистрирани регулярни изрази."  
 Return  
 }  
  
 Print "Регистрирани регулярни изрази:"  
 For each regex in regexList {  
 Integer id = regexList.indexOf(regex)  
 Print "ID: " + id + ", Регекс: " + regex  
 }  
 }  
 }

Класът ListCommand имплементира интерфейса Command и се използва за изброяване на всички регистрирани регулярни изрази в приложението. Методът execute се грижи за извеждането на тези изрази на екрана.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е 0, се извежда съобщение за грешка: "Usage: list" и методът се връща.
* **Извличане на списъка с регулярни изрази:**
  + Списъкът с регулярни изрази се извлича от RegCommand.getRegexList().
* **Проверка дали списъкът е празен:**
  + Ако списъкът е празен, се извежда съобщение: "Няма регистрирани регулярни изрази." и методът се връща.
* **Извеждане на регистрираните регулярни изрази:**
  + Ако списъкът не е празен, се извежда съобщение: "Регистрирани регулярни изрази:".
  + **За всеки регулярен израз в списъка се извлича неговият индекс и се извежда на екрана във формат: "ID: <индекс>, Регекс: <регулярният израз>".

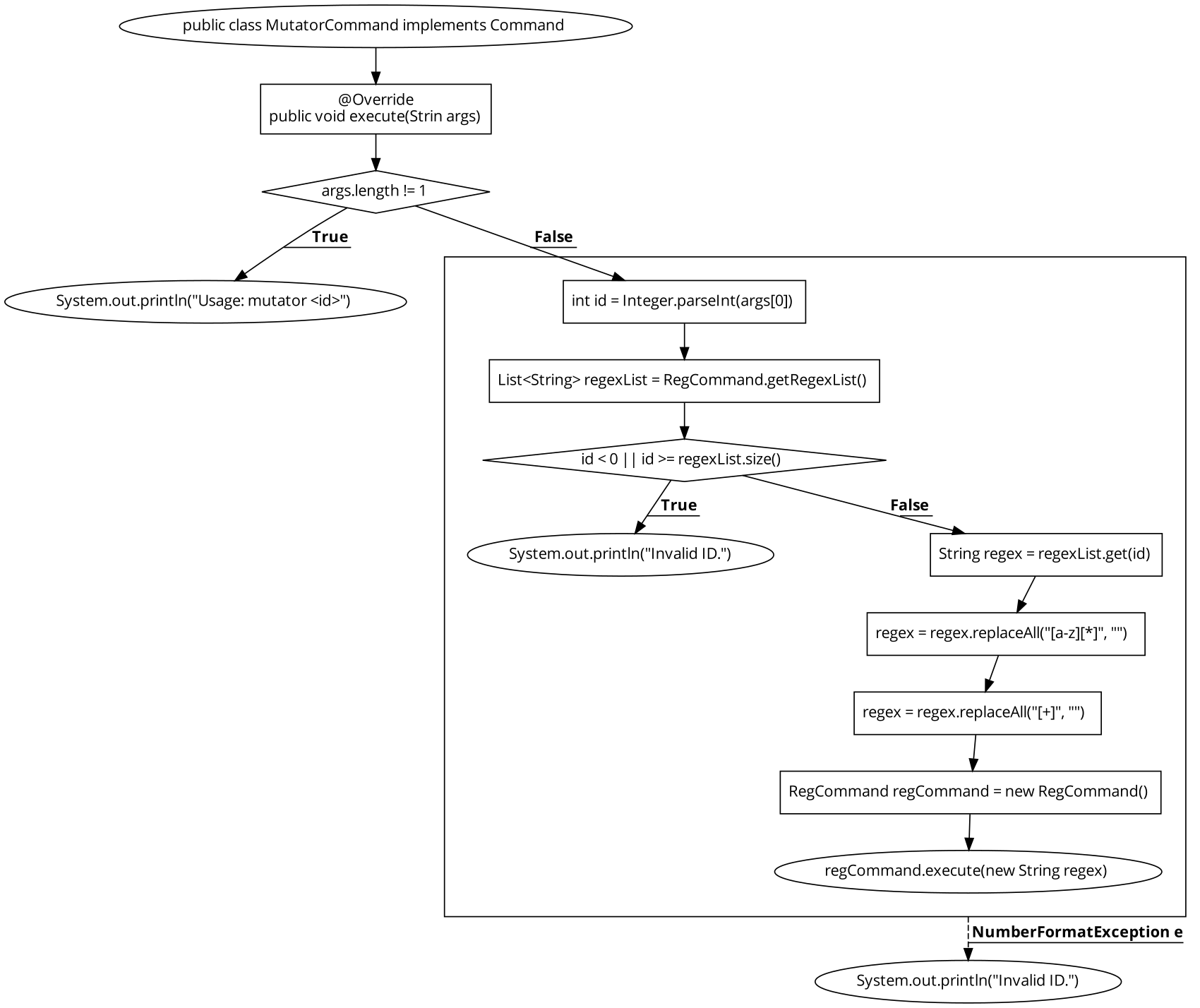
*- ProjectCommands/MutatorCommand*

Class MutatorCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length != 1) {  
 Print "Usage: mutator <id>"  
 Return  
 }  
  
 Try {  
 Integer id = Parse args[0] to Integer  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 If (id < 0 || id >= regexList.size()) {  
 Print "Invalid ID."  
 Return  
 }  
  
 String regex = regexList.get(id)  
  
 regex = regex.replaceAll("[a-z][\*]", "")  
 regex = regex.replaceAll("[+]", "")  
  
 RegCommand regCommand = new RegCommand()  
 regCommand.execute(new String[]{regex})  
  
 } Catch (NumberFormatException e) {  
 Print "Invalid ID."  
 }  
 }  
 }

Класът MutatorCommand имплементира интерфейса Command и се използва за модифициране на регулярни изрази в приложението. Методът execute приема идентификатор (ID) на регулярен израз, извлича го от списъка с регулярни изрази, прави определени модификации на израза и след това го записва обратно в списъка.

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е 1, се извежда съобщение за грешка: "Usage: mutator <id>" и методът се връща.
* **Обработка на аргумента:**
  + Опитва се да преобразува аргумента в цяло число id. Ако преобразуването не успее, се извежда съобщение за грешка: "Invalid ID." и методът се връща.
* **Извличане на списъка с регулярни изрази:**
  + Списъкът с регулярни изрази се извлича от RegCommand.getRegexList().
* **Проверка на валидността на ID:**
  + Ако id е по-малко от 0 или по-голямо или равно на размера на списъка, се извежда съобщение за грешка: "Invalid ID." и методът се връща.
* **Извличане и модификация на регулярния израз:**
  + Регулярният израз с дадения id се извлича от списъка.
  + Всички съвпадения на малки латински букви, следвани от \*, се премахват.
  + Всички съвпадения на + се премахват.
* **Записване на модифицирания регулярен израз:**
  + Създава нов обект RegCommand и изпълнява неговия метод execute, като му предава модифицирания регулярен израз.

**

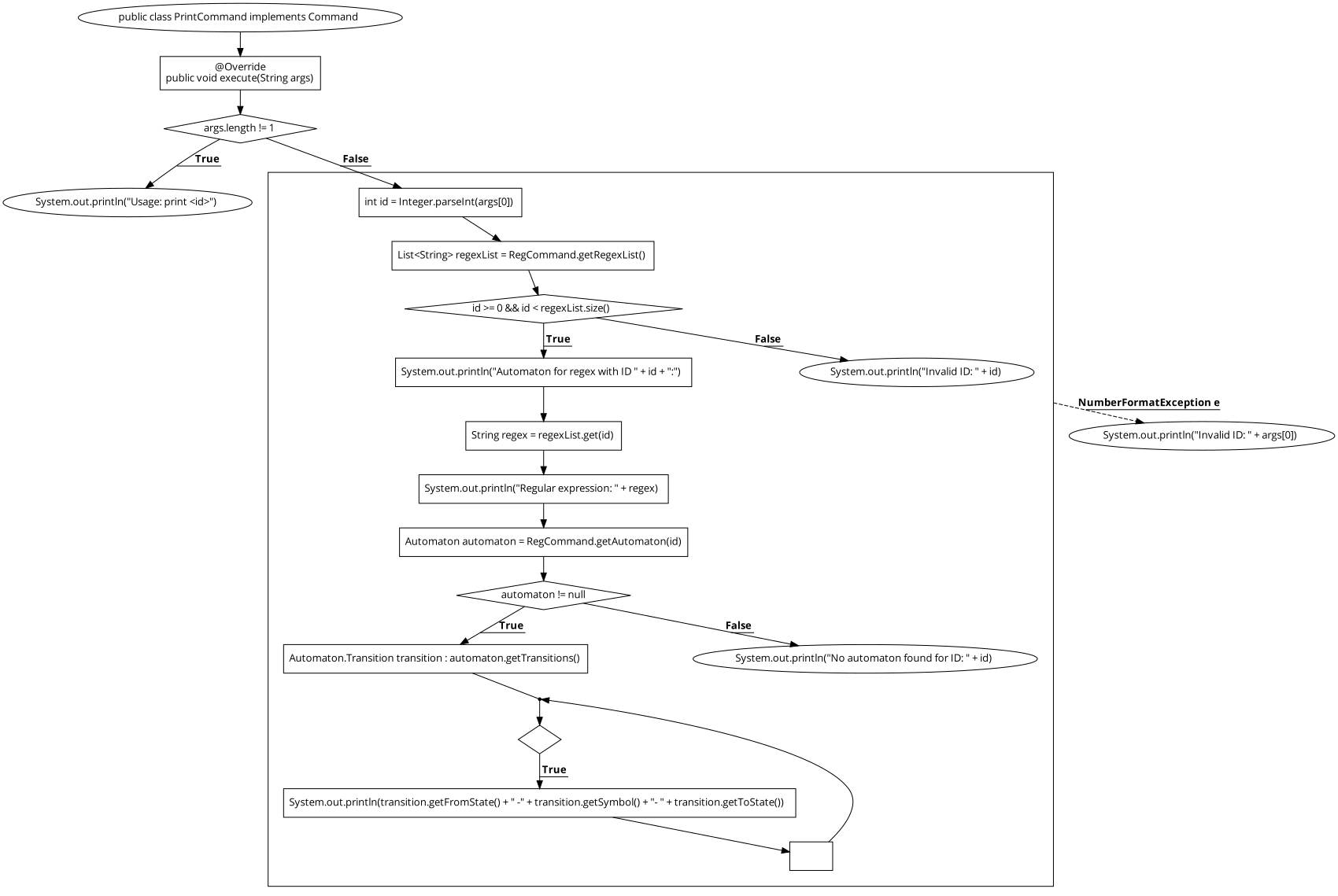
*- ProjectCommands/PrintCommand*

Class PrintCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length != 1) {  
 Print "Usage: print <id>"  
 Return  
 }  
  
 Try {  
 Integer id = Parse args[0] to Integer  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 If (id >= 0 and id < regexList.size()) {  
 Print "Automaton for regex with ID " + id + ":"  
  
 String regex = regexList.get(id)  
 Print "Regular expression: " + regex  
  
 Automaton automaton = RegCommand.getAutomaton(id)  
  
 If (automaton != null) {  
 For (Automaton.Transition transition : automaton.getTransitions()) {  
 Print transition.getFromState() + " -" + transition.getSymbol() + "-> " + transition.getToState()  
 }  
 } Else {  
 Print "No automaton found for ID: " + id  
 }  
 } Else {  
 Print "Invalid ID: " + id  
 }  
 } Catch (NumberFormatException e) {  
 Print "Invalid ID: " + args[0]  
 }  
 }  
 }

Класът PrintCommand имплементира интерфейса Command и се използва за извеждане на информация за регулярен израз и свързания с него автомат (если съществува) въз основа на даден идентификатор (ID).

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е 1, се извежда съобщение за грешка: "Usage: print <id>" и методът се връща.
* **Обработка на аргумента:**
  + Опитва се да преобразува аргумента в цяло число id. Ако преобразуването не успее, се извежда съобщение за грешка: "Invalid ID: " + args[0] и методът се връща.
* **Извличане на списъка с регулярни изрази:**
  + Списъкът с регулярни изрази се извлича от RegCommand.getRegexList().
* **Проверка на валидността на ID:**
  + Ако id е по-малко от 0 или по-голямо или равно на размера на списъка, се извежда съобщение за грешка: "Invalid ID: " + id и методът се връща.
* **Извличане и извеждане на регулярния израз:**
  + Регулярният израз с дадения id се извлича от списъка и се извежда на екрана.
* **Извличане и извеждане на автомата:**
  + Автоматът, свързан с дадения id, се извлича чрез RegCommand.getAutomaton(id).
  + Ако автоматът не е null, се извеждат преходите му.
  + Ако автоматът е null, се извежда съобщение: "No automaton found for ID: " + id.

**

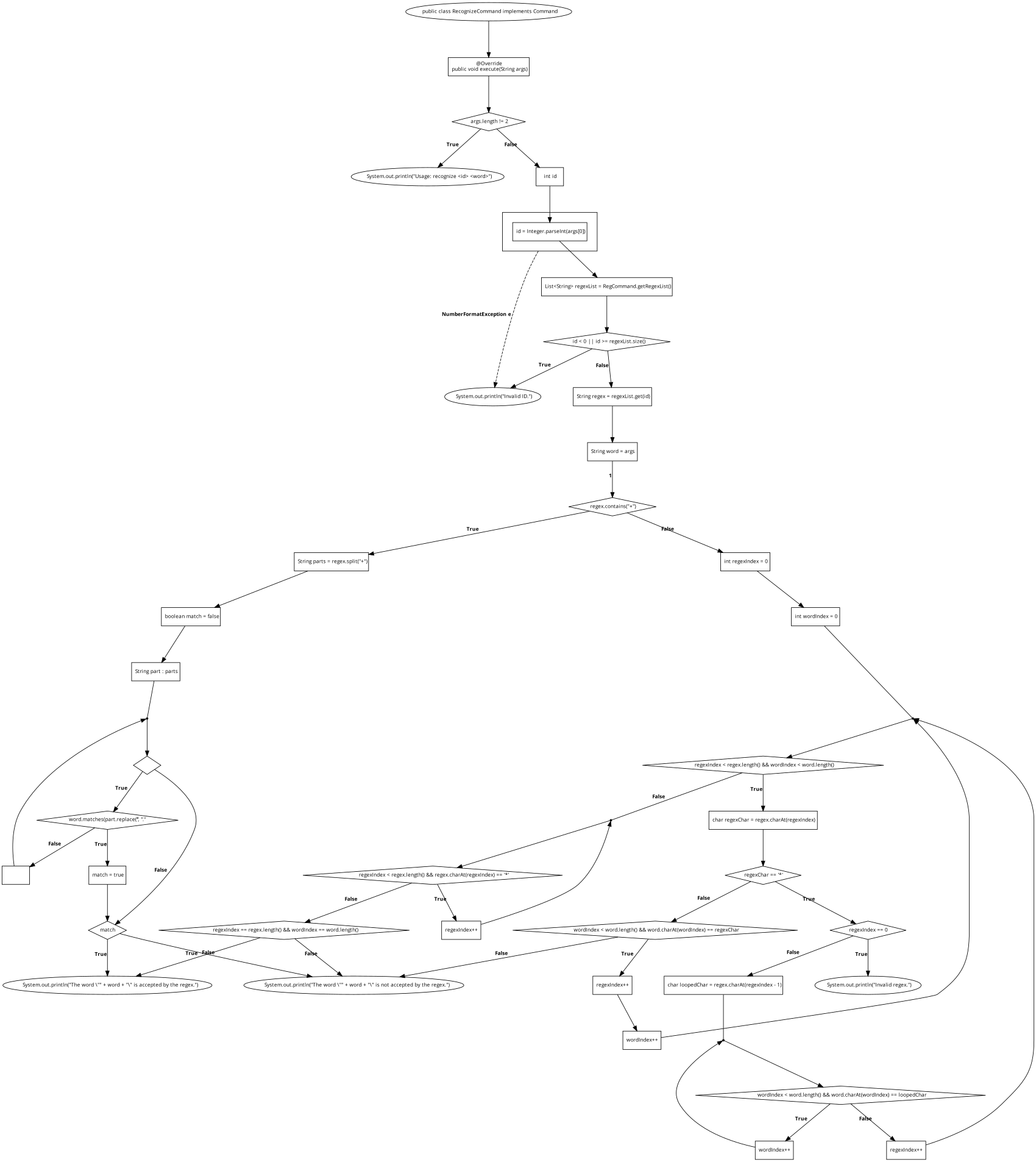
*- ProjectCommands/RecognizeCommand*

Class RecognizeCommand implements Command {  
 Method execute(String[] args) {  
 If (args.length != 2) {  
 Print "Usage: recognize <id> <word>"  
 Return  
 }  
  
 Try {  
 Integer id = Parse args[0] to Integer  
 } Catch (NumberFormatException e) {  
 Print "Invalid ID."  
 Return  
 }  
  
 List<String> regexList = RegCommand.getRegexList()  
 If (id < 0 or id >= regexList.size()) {  
 Print "Invalid ID."  
 Return  
 }  
  
 String regex = regexList.get(id)  
 String word = args[1]  
  
 If (regex.contains("+")) {  
 String[] parts = regex.split("\\+")  
 Boolean match = False  
  
 For each part in parts {  
 If (word matches part.replace("\*", ".\*")) {  
 match = True  
 Break  
 }  
 }  
  
 If (match) {  
 Print "The word \"" + word + "\" is accepted by the regex."  
 } Else {  
 Print "The word \"" + word + "\" is not accepted by the regex."  
 }  
 Return  
 }  
  
 Integer regexIndex = 0  
 Integer wordIndex = 0  
  
 While (regexIndex < regex.length() and wordIndex < word.length()) {  
 Char regexChar = regex.charAt(regexIndex)  
  
 If (regexChar == '\*') {  
 If (regexIndex == 0) {  
 Print "Invalid regex."  
 Return  
 }  
  
 Char loopedChar = regex.charAt(regexIndex - 1)  
  
 While (wordIndex < word.length() and word.charAt(wordIndex) == loopedChar) {  
 wordIndex++  
 }  
  
 regexIndex++  
 } Else {  
 If (wordIndex < word.length() and word.charAt(wordIndex) == regexChar) {  
 regexIndex++  
 wordIndex++  
 } Else {  
 Print "The word \"" + word + "\" is not accepted by the regex."  
 Return  
 }  
 }  
 }  
  
 While (regexIndex < regex.length() and regex.charAt(regexIndex) == '\*') {  
 regexIndex++  
 }  
  
 If (regexIndex == regex.length() and wordIndex == word.length()) {  
 Print "The word \"" + word + "\" is accepted by the regex."  
 } Else {  
 Print "The word \"" + word + "\" is not accepted by the regex."  
 }  
 }  
 }

Класът RecognizeCommand имплементира интерфейса Command и се използва за разпознаване дали дадена дума съответства на регулярен израз от списъка с регулярни изрази, базиран на предоставен идентификатор (ID).

**Метод execute:**

* **Параметри:** Методът execute приема масив от аргументи args.
* **Проверка на аргументите:**
  + Ако дължината на масива args не е 2, се извежда съобщение за грешка: "Usage: recognize <id> <word>" и методът се връща.
* **Преобразуване на аргумента id:**
  + Опитва се да преобразува първия аргумент в цяло число id. Ако преобразуването не успее, се извежда съобщение за грешка: "Invalid ID." и методът се връща.
* **Извличане на списъка с регулярни изрази:**
  + Списъкът с регулярни изрази се извлича от RegCommand.getRegexList().
* **Проверка на валидността на ID:**
  + Ако id е по-малко от 0 или по-голямо или равно на размера на списъка, се извежда съобщение за грешка: "Invalid ID." и методът се връща.
* **Извличане на регулярния израз и думата:**
  + Регулярният израз с дадения id се извлича от списъка.
  + Думата за разпознаване се взима от втория аргумент.
* **Обработка на регулярни изрази, съдържащи символа +:**
  + Ако регулярният израз съдържа символа +, той се разделя на части чрез split("\\+").
  + За всяка част се проверява дали думата съвпада с частта чрез matches. Ако има съвпадение, се извежда съобщение за успешно разпознаване и методът се връща.
  + Ако няма съвпадение, се извежда съобщение за неуспешно разпознаване и методът се връща.
* **Обработка на регулярни изрази без символа +:**
  + Използват се два индекса, regexIndex и wordIndex, за обход на регулярния израз и думата.
  + Ако текущият символ в регулярния израз е \*, се проверява дали предишният символ съвпада с текущия символ от думата. Ако съвпада, индексът за думата се увеличава, докато символите съвпадат.
  + Ако текущият символ в регулярния израз не е \*, се проверява дали текущият символ от думата съвпада с него. Ако съвпада, и двата индекса се увеличават.
  + След като регулярният израз или думата бъдат обходени, се проверява дали са достигнали края си и се извежда съответното съобщение за успешно или неуспешно разпознаване.

**

*- ProjectCommands/RegCommand*

Class RegCommand implements Command {  
 Static List<String> regexList = new ArrayList<>()  
 Static List<Automaton> automatonList = new ArrayList<>()  
  
 Static Method getAutomaton(int id) {  
 If id is valid (0 <= id < automatonList.size()) {  
 Return automatonList.get(id)  
 } Else {  
 Return null  
 }  
 }  
  
 Static Method setRegexList(List<String> uniqueRegexes) {  
 regexList.clear()  
 regexList.addAll(uniqueRegexes)  
 }  
  
 Override Method execute(String[] args) {  
 If (args.length != 1) {  
 Print "Usage: reg <regex>"  
 Return  
 }  
  
 String regex = args[0]  
 If (regex.equals("emptyreg")) {  
 regex = "&"  
 } Else If (!regex.matches("[a-zA-Z\*+]+")) {  
 Print "Invalid regex. It should contain only letters, '\*' or '+'!"  
 Return  
 }  
  
 int state = 0  
 int nextState = 1  
 List<Automaton.Transition> transitions = new ArrayList<>()  
  
 For each character in regex {  
 If (character is '&') {  
 Add transition (state, character, state) to transitions  
 } Else If (next character is '\*') {  
 Add transition (state, character, state) to transitions  
 Skip next character  
 } Else If (next character is '+') {  
 Add transition (state, character, nextState) to transitions  
 state = 0  
 Skip next character  
 } Else {  
 Add transition (state, character, nextState) to transitions  
 state = nextState  
 nextState++  
 }  
 }  
  
 int id = regexList.size()  
 Print "Automaton created for regular expression " + regex + " with ID: " + id  
  
 regexList.add(regex)  
 automatonList.add(new Automaton(transitions))  
 }  
  
 Static Method getRegexList() {  
 Return regexList  
 }  
 }

Класът RegCommand имплементира интерфейса Command и се използва за добавяне на регулярни изрази и създаване на автомати, базирани на тези изрази.

**Променливи:**

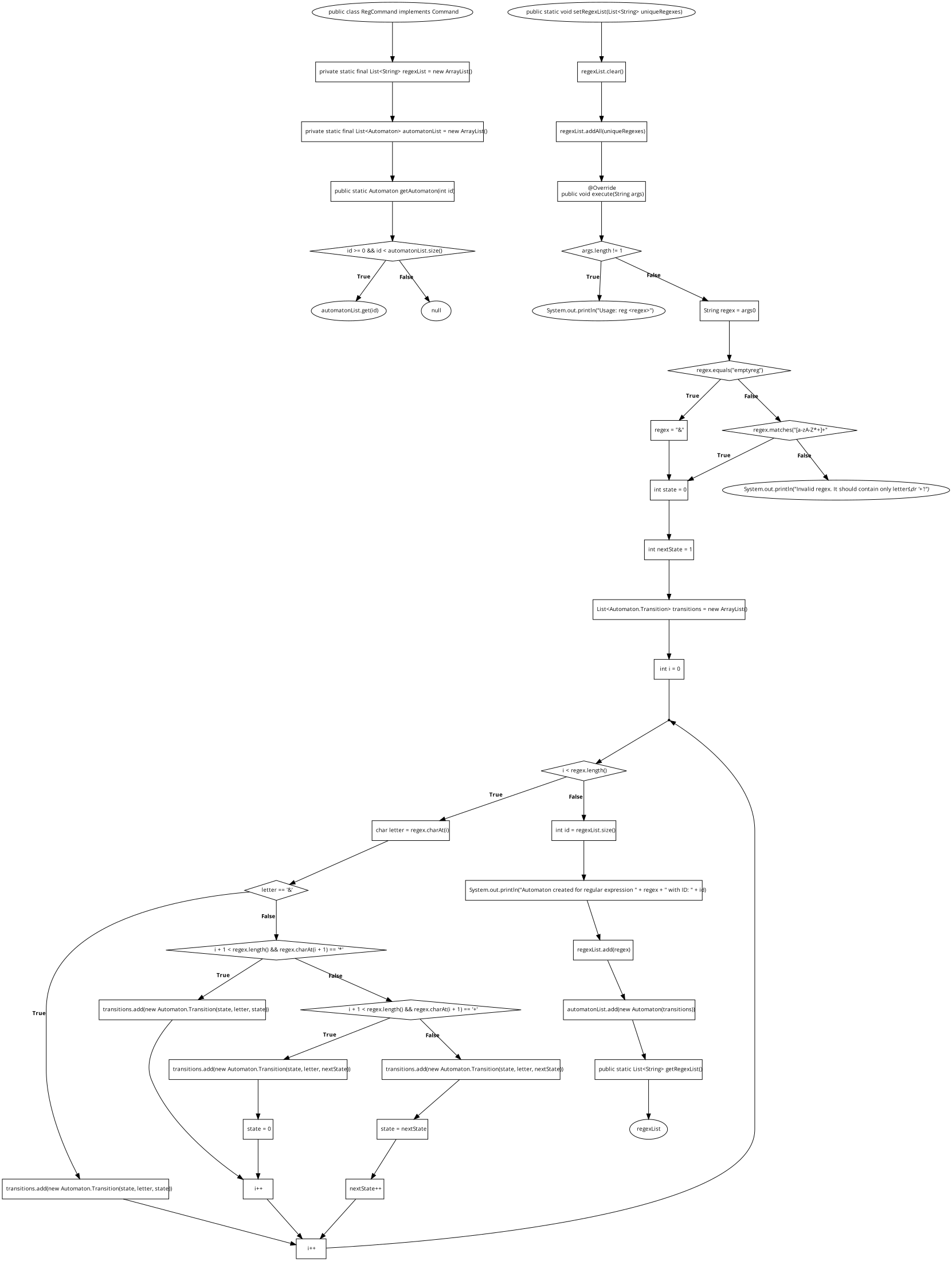
* **regexList**: Статичен списък, който съхранява всички регулярни изрази.
* **automatonList**: Статичен списък, който съхранява автомати, създадени за съответните регулярни изрази.

**Методи:**

* **getAutomaton(int id)**: Връща автомата с дадения id, ако той съществува. Ако id е извън границите на списъка, връща null.
* **setRegexList(List<String> uniqueRegexes)**: Задава нов списък с регулярни изрази, като първо изчиства текущия списък и след това добавя новите изрази.
* **execute(String[] args)**: Основният метод, който се изпълнява при командата reg.

**Метод execute:**

* **Проверка на аргументите:**
  + Ако дължината на аргументите не е 1, се извежда съобщение за грешка: "Usage: reg <regex>" и методът се връща.
* **Валидиране на регулярния израз:**
  + Ако регулярният израз е emptyreg, той се заменя с &.
  + Ако регулярният израз съдържа невалидни символи, се извежда съобщение за грешка и методът се връща.
* **Създаване на автомата:**
  + Инициализират се променливите state и nextState.
  + Обхожда се всеки символ от регулярния израз и се създават преходи според правилата:
    - Символът & създава преход към същото състояние.
    - Символът \* създава преход към същото състояние за предишния символ.
    - Символът + създава преход към ново състояние и връща обратно в началното състояние.
    - Всички други символи създават преход към ново състояние.
* **Добавяне на регулярния израз и автомата в списъците:**
  + Регулярният израз и съответният автомат се добавят в regexList и automatonList.
  + Извежда се съобщение за успешно създаване на автомата с идентификатор.

**

*- ProjectCommands/SaveaCommand*

Class SaveaCommand implements Command {  
 Override Method execute(String[] args) {  
 If (args.length != 2) {  
 Print "Usage: savea <id> <filename>"  
 Return  
 }  
  
 String id = args[0]  
 String filename = args[1]  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 Try to parse id to int (index)  
 If (index is valid) {  
 String regex = regexList.get(index)  
 Try (PrintWriter writer = new PrintWriter(new FileWriter(filename))) {  
 writer.println(regex)  
 Print "Automaton saved successfully to " + filename  
 } Catch (IOException e) {  
 Print "Error saving automaton to file: " + e.getMessage()  
 }  
 } Else {  
 Print "Invalid ID: " + id  
 }  
 }  
 }

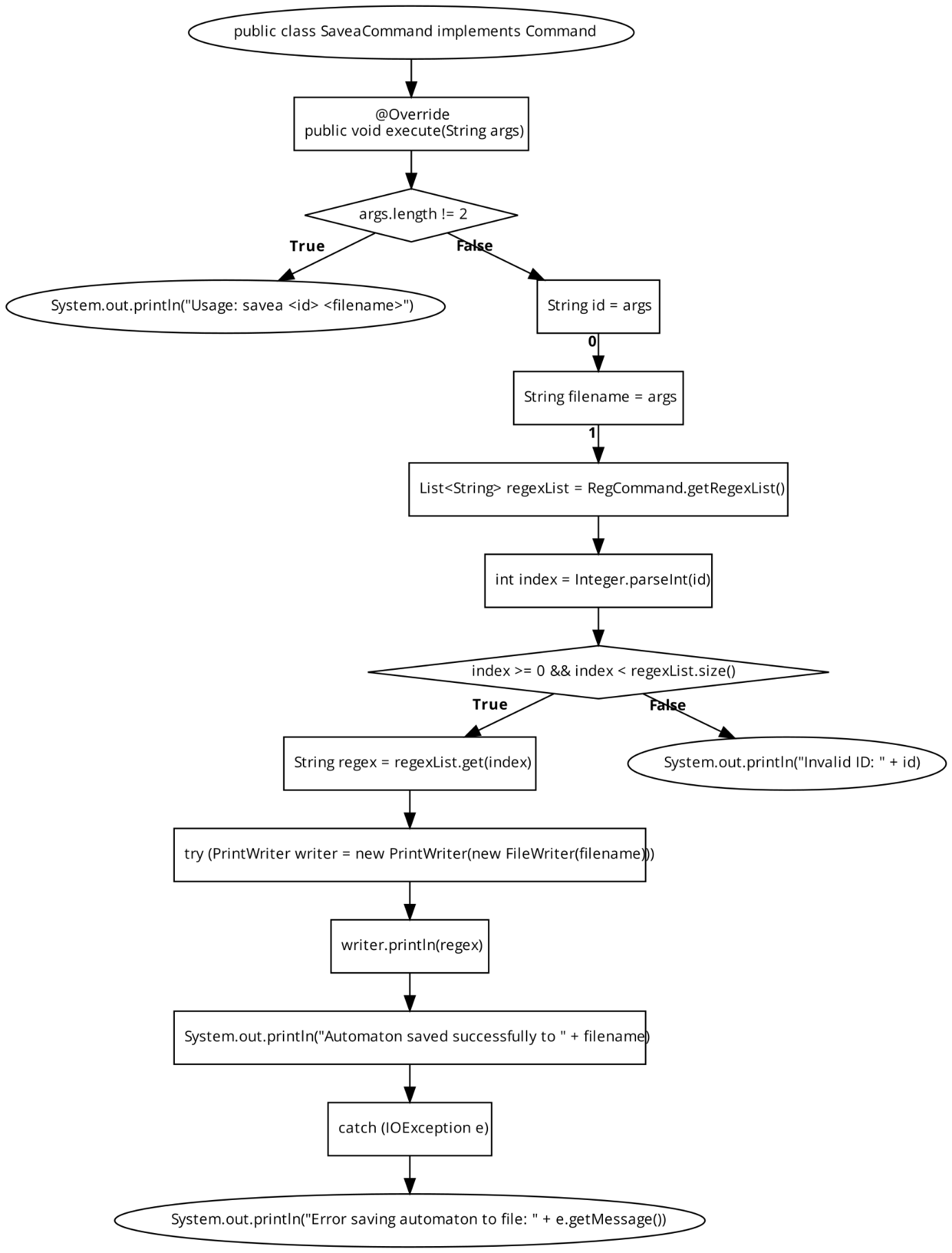
Класът SaveaCommand имплементира интерфейса Command и се използва за запазване на регулярни изрази в даден файл.

**Методи:**

* **execute(String[] args)**: Основният метод, който се изпълнява при командата savea.

**Метод execute:**

* **Проверка на аргументите:**
  + Ако дължината на аргументите не е 2, се извежда съобщение за грешка: "Usage: savea <id> <filename>" и методът се връща.
* **Извличане на аргументите:**
  + id и filename се извличат от аргументите.
* **Получаване на списъка с регулярни изрази:**
  + Извиква се методът getRegexList от RegCommand, за да се получи списъкът с регулярни изрази.
* **Проверка на валидността на id:**
  + Преобразува се id в цяло число и се проверява дали индексът е валиден.
  + Ако индексът е валиден, се извлича съответният регулярен израз.
  + Ако индексът не е валиден, се извежда съобщение за грешка: "Invalid ID: " + id.
* **Запазване на регулярния израз във файл:**
  + Използва се PrintWriter и FileWriter за записване на регулярния израз във файла с име filename.
  + Ако записът е успешен, се извежда съобщение: "Automaton saved successfully to " + filename.
  + Ако възникне грешка при запис, се извежда съобщение за грешка: "Error saving automaton to file: " + e.getMessage().

**

*- ProjectCommands/UnCommand*

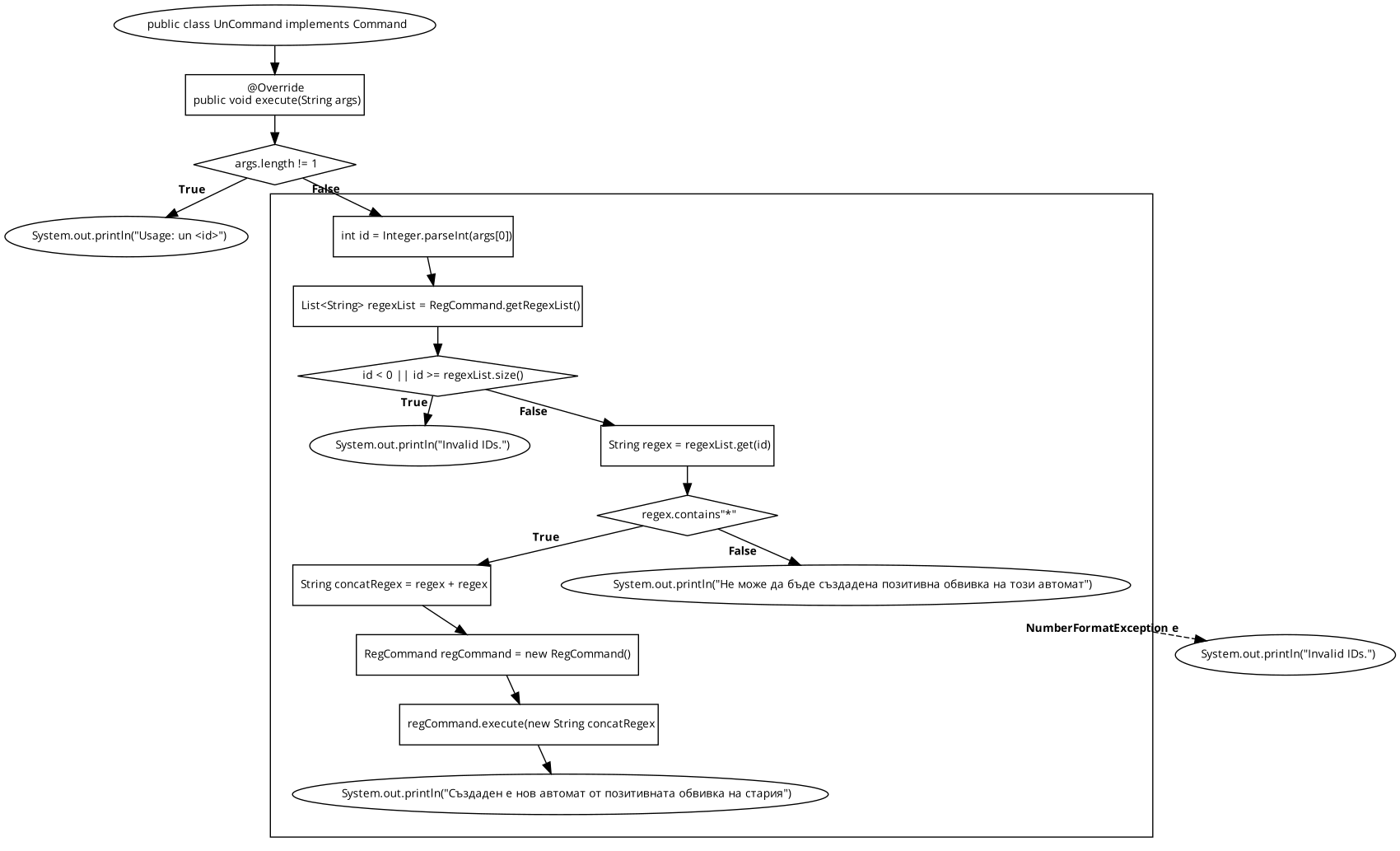
Class UnCommand implements Command {  
 Override Method execute(String[] args) {  
 If (args.length != 1) {  
 Print "Usage: un <id>"  
 Return  
 }  
  
 Try {  
 Parse args[0] to int (id)  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 If (id < 0 || id >= regexList.size()) {  
 Print "Invalid IDs."  
 Return  
 }  
  
 String regex = regexList.get(id)  
  
 If(regex contains "\*"){  
 String concatRegex = regex + regex  
 RegCommand regCommand = new RegCommand()  
 regCommand.execute(new String[] {concatRegex})  
 Print "Създаден е нов автомат от позитивната обвивка на стария"  
 }  
 Else {  
 Print "Не може да бъде създадена позитивна обвивка на този автомат"  
 }  
  
 } Catch (NumberFormatException e) {  
 Print "Invalid IDs."  
 }  
 }  
 }

Класът UnCommand имплементира интерфейса Command и представлява команда за обработка на регулярни изрази.

**Методи:**

* **execute(String[] args)**: Основният метод, който се изпълнява при командата un.

**Метод execute:**

* **Проверка на аргументите:**
  + Проверява дали дължината на подадените аргументи е точно 1. Ако не е, извежда съобщение за грешка и прекратява изпълнението на метода.
* **Извличане и валидация на id:**
  + Аргументът id се преобразува от стринг в цяло число (int).
  + Проверява дали полученото id е валидно, т.е., дали е в интервала [0, размер на списъка с регулярни изрази).
  + Ако id не е валиден, извежда съобщение за грешка и прекратява изпълнението на метода.
* **Обработка на регулярния израз:**
  + Извлича се регулярният израз от списъка на базата на получения id.
  + Проверява дали регулярният израз съдържа символа \*.
  + Ако регулярният израз съдържа \*, се създава нов регулярен израз concatRegex, който представлява конкатенацията на текущия регулярен израз със себе си.
  + Извиква се команда от тип RegCommand с новия регулярен израз като аргумент, което води до актуализиране на списъка с регулярни изрази с новия.
  + Извежда се съобщение за успешно създаден автомат от позитивната обвивка на стария регулярен израз.
* **Обработка на случай без \*:**
  + **Ако регулярният израз не съдържа \*, извежда се съобщение, че не може да бъде създадена позитивна обвивка на този автомат.

*- ProjectCommands/UnionCommand*

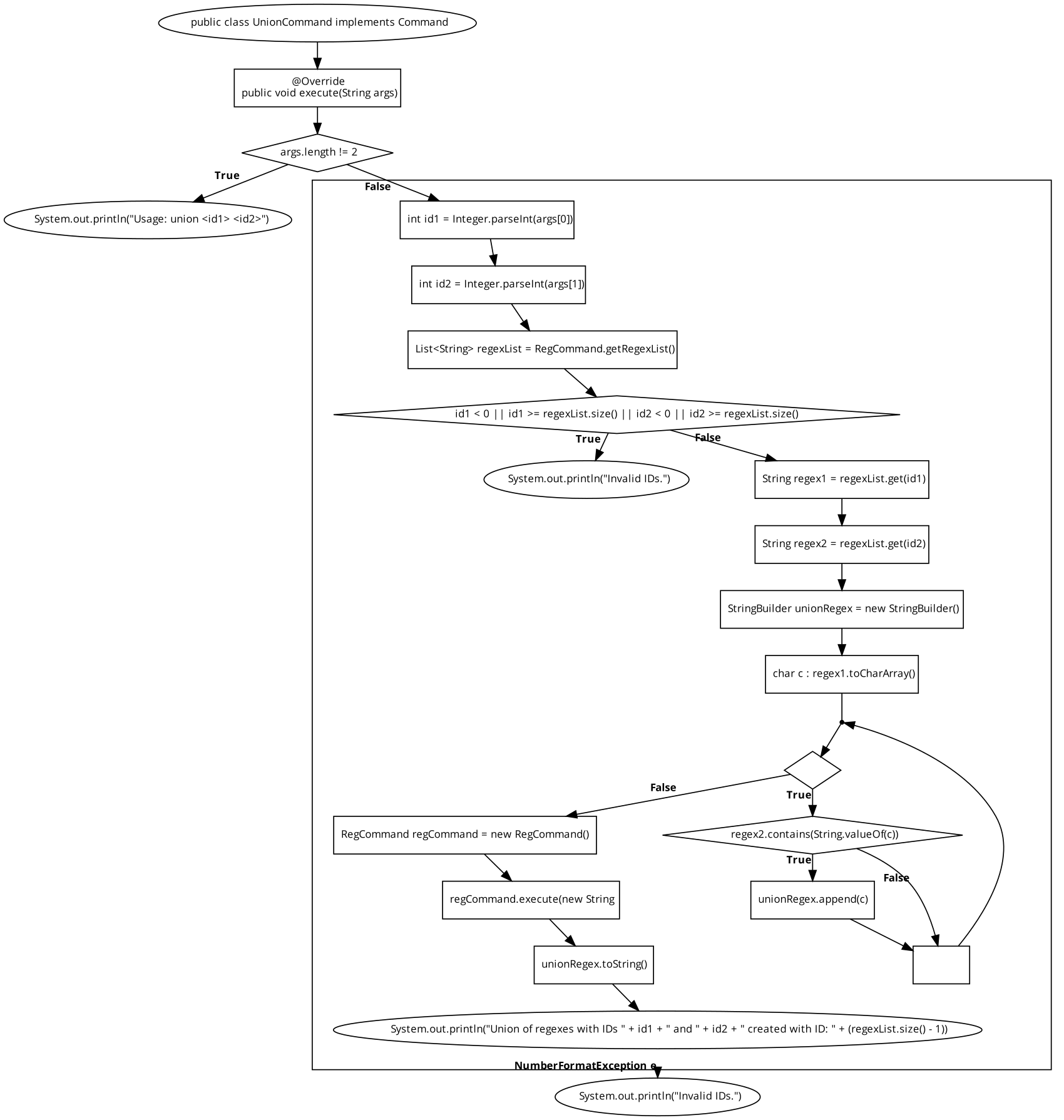
Class UnionCommand implements Command {  
 Override Method execute(String[] args) {  
 If (args.length != 2) {  
 Print "Usage: union <id1> <id2>"  
 Return  
 }  
  
 Try {  
 Parse args[0] to int (id1)  
 Parse args[1] to int (id2)  
  
 List<String> regexList = RegCommand.getRegexList()  
  
 If (id1 < 0 || id1 >= regexList.size() || id2 < 0 || id2 >= regexList.size()) {  
 Print "Invalid IDs."  
 Return  
 }  
  
 String regex1 = regexList.get(id1)  
 String regex2 = regexList.get(id2)  
  
  
 StringBuilder unionRegex = new StringBuilder()  
 For each character c in regex1.toCharArray() {  
 If (regex2.contains(String.valueOf(c))) {  
 unionRegex.append(c)  
 }  
 }  
  
 RegCommand regCommand = new RegCommand()  
 regCommand.execute(new String[] {unionRegex.toString()})  
  
 Print "Union of regexes with IDs " + id1 + " and " + id2 + " created with ID: " + (regexList.size() - 1)  
 } Catch (NumberFormatException e) {  
 Print "Invalid IDs."  
 }  
 }  
 }

Класът UnionCommand имплементира интерфейса Command и представлява команда за обединение на два регулярни израза от списъка с регулярни изрази.

**Методи:**

* **execute(String[] args)**: Основният метод, който се изпълнява при командата union.

**Метод execute:**

* **Проверка на аргументите:**
  + Проверява дали дължината на подадените аргументи е точно 2. Ако не е, извежда съобщение за грешка и прекратява изпълнението на метода.
* **Извличане и валидация на id1 и id2:**
  + Аргументите id1 и id2 се преобразуват от стрингове в цели числа (int).
  + Проверява дали получените id1 и id2 са валидни, т.е., дали са в интервала [0, размер на списъка с регулярни изрази).
  + Ако някой от id1 или id2 не е валиден, извежда съобщение за грешка и прекратява изпълнението на метода.
* **Обработка на регулярните изрази:**
  + Извлича се първият и вторият регулярен израз от списъка базирано на получените id1 и id2.
  + Създава се нов стринг (unionRegex), в който се добавят символите от regex1, които съществуват и в regex2.
  + Извиква се команда от тип RegCommand с новия регулярен израз като аргумент, което води до актуализиране на списъка с регулярни изрази с новия.
* **Извеждане на резултат:**
  + Извежда се съобщение за успешно създадено обединение на регулярните изрази с указани id1 и id2, като се упоменава идентификаторът на новия регулярен израз.

Глава 4. Тестване

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

