ТЕХНИЧЕСКИ УНИВЕРСИТЕТ ВАРНА

Катедра: СИТ

**КУРСОВ ПРОЕКТ**

по

**Обектно-ориентирано програмиране 1 част**

на

**Полина Атанасова Кръстева**

**СИТ, 2 курс, 1 група, 22621687**

**Проект 4: Недетерминиран краен автомат**

Съдържание

[Глава 1. Увод 4](#_Toc167044808)

[Условие на проекта 4](#_Toc167044809)

[**Анализ на заданието** 4](#_Toc167044811)

[Изисквания към програмата 4](#_Toc167044812)

[Специфични операции за работа с автоматите 5](#_Toc167044813)

[Мутатори и проверки 5](#_Toc167044814)

[Технически изисквания 5](#_Toc167044815)

[Заключение 6](#_Toc167044816)

[**Глава 2. Преглед на предметната област** 6](#_Toc167044817)

[2.1. Основни дефиниции, концепции и алгоритми, които ще бъдат използвани 6](#_Toc167044818)

[Детерминирани и недетерминирани крайни автомати 6](#_Toc167044819)

[SOLID Принципи 7](#_Toc167044820)

[2.2. Дефиниране на проблеми и сложност на поставената задача 7](#_Toc167044821)

[2.3. Подходи и методи за решаване на поставените проблемите 8](#_Toc167044823)

[Глава 3. Проектиране 9](#_Toc167044824)

[3.1. Обща структура на проекта, пакети който ще се реализират 9](#_Toc167044825)

[3.2. Диаграми/Блок схеми 9](#_Toc167044826)

[Глава 4. Реализация, тестване 9](#_Toc167044827)

[4.1. Реализация на класове 9](#_Toc167044828)

[Клас FileOpener 9](#_Toc167044829)

[Клас AutomatonManager 12](#_Toc167044830)

[Клас FileAutomatonWriter 13](#_Toc167044831)

[Клас Open 16](#_Toc167044832)

[Клас Save 17](#_Toc167044833)

[Клас SaveAs 18](#_Toc167044834)

[Клас Close 18](#_Toc167044835)

[Клас Concat 19](#_Toc167044836)

[Клас Deterministic 21](#_Toc167044837)

[Клас Empty 23](#_Toc167044838)

[Клас Help 25](#_Toc167044839)

[Клас IsLanguageFinite 26](#_Toc167044840)

[Клас ListCommand 29](#_Toc167044841)

[Клас Print 30](#_Toc167044842)

[Клас Recognize 31](#_Toc167044843)

[Клас Un 33](#_Toc167044844)

[Клас Union 35](#_Toc167044845)

[Клас Mutator 38](#_Toc167044846)

[Клас Reg 42](#_Toc167044848)

[Клас Automaton 44](#_Toc167044849)

[Клас AutomatonList 47](#_Toc167044850)

[Клас Edge 49](#_Toc167044851)

[Клас Node 50](#_Toc167044852)

[Клас (енумерация) Operations 51](#_Toc167044853)

[Класове interface Command и abstract class DefaultCommand 52](#_Toc167044854)

[Клас CLI 52](#_Toc167044855)

[Клас Application 54](#_Toc167044856)

[Клас Exit 54](#_Toc167044857)

[Класове за изключения 54](#_Toc167044858)

# Глава 1. Увод

## Условие на проекта

Да се реализира програма, която поддържа операции с недетерминиран краен автомат с Ɛ-преходи,над азбука, състояща се от цифрите и малките латински букви. Автоматите да се сериализират по разработен от Вас формат. Всеки прочетен автомат да получава уникален идентификатор.След като приложението отвори даден файл, то трябва да може да извършва посочените по-долу операции, в допълнение на общите операции (open, close, save, save as, helpи exit):

* *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 автомат 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
* Да се реализира мутатор, който детерминира даден автомат
* Да се реализира операция, която проверяват дали езикът на даден автомат е краен

## **Анализ на заданието**

Целта на заданието е разработването на програма, която поддържа операции върху недетерминиран краен автомат (NFA) с Ɛ-преходи. Азбуката, върху която автоматите работят, се състои от цифрите и малките латински букви. Програмата трябва да поддържа сериализация и десериализация на автоматите във формат, в разработен от мен формат. Всеки автомат, прочетен от файл, трябва да получава уникален идентификатор.

### Изисквания към програмата

Програмата трябва да може да изпълнява следните основни операции:

* ***open <filename>*** - Отваря и зарежда автомат от файл.
* ***close*** - Затваря текущо отворения файл.
* ***save*** - Записва направените промени в текущо отворения файла.
* ***save as <filename>*** - Записва автомат в нов файл.
* ***help*** - Извежда списък с всички команди и тяхното описание.
* ***exit*** - Затваря програмата.

### Специфични операции за работа с автоматите

Програмата трябва да поддържа следните операции върху автоматите:

1. ***List*** - Извежда списък с идентификаторите на всички прочетени автомати.
2. ***print <id>*** - Извежда информация за всички преходи в автомата.
3. ***save <id> <filename>*** - Записва автомат с дадения идентификатор във файл.
4. ***empty <id>*** - Проверява дали езикът на автомата е празен.
5. ***deterministic <id>*** - Проверява дали автоматът е детерминиран.
6. ***recognize <id> <word>*** - Проверява дали дадена дума е в езика на автомата.
7. ***union <id1> <id2>*** - Намира обединението на два автомата и създава нов автомат. Отпечатва идентификатора на новия автомат.
8. ***concat <id1> <id2>*** - Намира конкатенацията на два автомата и създава нов автомат. Отпечатва идентификатора на новия автомат.
9. ***un <id>*** - Намира позитивна обвивка на автомат и създава нов автомат. Отпечатва идентификатора на новия автомат.
10. ***reg <regex>*** - Създава нов автомат по даден регулярен израз (теорема на Клини). Отпечатва идентификатора на новия автомат.

### Мутатори и проверки

Програмата трябва да включва следните мутатори и проверки:

* Мутатор за детерминиране на даден автомат.
* Операция за проверка дали езикът на даден автомат е краен.

### Технически изисквания

1. **Сериализация и десериализация:**
   * Сериализираният формат трябва да поддържа информация за състоянията, преходите (включително Ɛ-преходите), началното състояние и крайните състояния на автомата.
2. **Уникални идентификатори:**
   * Всеки автомат, прочетен от файл, трябва да получава уникален идентификатор, който се използва за идентифициране на автомата при изпълнение на операции.

### Заключение

Заданието изисква разработването на програма, която да поддържа набор от операции върху недетерминирани и детерминирани крайни автомати с Ɛ-преходи. Това включва основни операции за работа с файлове, специфични операции за манипулиране и проверка на автоматите, както и допълнителни мутатори и проверки, както и поддържането на уникални идентификатори за автоматите.

# **Глава 2. Преглед на предметната област**

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

### Детерминирани и недетерминирани крайни автомати

Крайните автомати са математически модели, използвани за представяне на изчислителни процеси. Те са съставени от множество състояния и преходи между тези състояния, определени от символи от дадена азбука. Недетерминираните крайни автомати (NFA) позволяват наличието на ε-преходи, както и множество преходи за един и същ символ от дадено състояние. Детерминираните крайни автомати (DFA) не позволяват такива преходи, като всеки символ води до точно едно следващо състояние.

#### Текстови файлове за съхранение на автомати

В проекта се използват текстови файлове за съхранение и управление на данните. Форматът на съхраннение ще включва информация за преходите, началните и крайните състояния на автомата.

#### Singleton Pattern

За да се осигури единствена инстанция на обект - списък с всички прочетени автомати, ще използвам шаблона Singleton. Той гарантира, че класът има само една инстанция и предоставя глобална точка за достъп до нея.

## 2.2. Дефиниране на проблеми и сложност на поставената задача

Реализирането на програма за работа с недетерминиран краен автомат (NFA) е сложна задача, която включва както теоретични, така и практически знания.

#### 1. Структура на текстовите файлове и сериализация

Основната задача е да се дефинира формат за сериализация и десериализация на автоматите. Основните предизвикателства включват:

* **Разбиране на структурата на текстовия файл**.
* **Избор на подходящ формат** за изобразяване на информацията.
* Избор на метод за записване на автоматите в лесно четим текстов формат.

#### 2. Команден интерфейс (CLI)

Работата с Command Line Interface (CLI) също е предизвикателство поради необходимостта от:

* **Дефиниране на командите**.
  + **Обработка на параметри**. Трябва да се осигури правилна обработка и проверка на параметрите, подадени към командите. Това включва проверка на типа и броя на аргументите.
  + **Валидиране на команди**.
  + **Управление на достъпа**. За определени команди се налага проверка за наличие на текущо отворен файл.

#### 3.Работа с автомати

Този аспект от проекта изисква както владеене на основните принципи на програмиране, така и теоритични познания в сферата на автоматите и дискретната математика.

## 2.3. Подходи и методи за решаване на поставените проблемите

За решаване на поставените проблеми в проекта могат да бъдат използвани различни подходи и методи.

* + Дефиниране на ясни правила за структуриране на информацията във файла.
  + Дефиниране на ясни и ефективни команди със съответните параметри и синтаксис.
  + Изучаване и прилагане на теоретични концепции от областта на автоматите и дискретната математика.
  + Разработване на алгоритми за проверка и операции.
  + Провеждане на тестове за проверка на функционалността на програмата и сигурността на командите.

# Глава 3. Проектиране

3.1. Обща структура на проекта, пакети който ще се реализиратA screenshot of a computer program

Description automatically generated Фигура 1: Commands

A screenshot of a computer

Description automatically generated

Фигура 2: Exceptions

A screenshot of a computer program

Description automatically generated

Фигура 3: Realization

# Глава 4. Реализация, тестване

# 4.1. Реализация на класове

# Клас FileOpener

public class FileOpener {  
 private String filePath;  
 private static String *content*;  
 private AutomatonManager automatonManager = AutomatonManager.*getInstance*();  
  
 public FileOpener(String filePath) {  
 this.filePath = filePath;  
 this.*content* = "";  
 }  
  
 public static String getContent() {  
 return *content*;  
 }  
  
 public boolean openFile() throws IOException {  
 File file = new File(filePath);  
 if (!file.exists()) {  
 createNewFile(filePath);  
 }  
 try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {  
 StringBuilder sb = new StringBuilder();  
 String line;  
 String[] lineContent;  
 int id = 0;  
 Node from = null, to = null;  
 String transition = "";  
 Automaton automaton = new Automaton("abcdefghijklmnopqrstuvwxyz123456789");  
  
 while ((line = reader.readLine()) != null) {  
 sb.append(line).append("\n");  
  
 if (line.equals("-")) {  
 AutomatonList.*getInstance*().addAutomaton(id, automaton);  
 automaton = new Automaton("abcdefghijklmnopqrstuvwxyz123456789");  
 continue;  
 }  
  
 lineContent = line.split("=");  
  
 switch (lineContent[0].toLowerCase()) {  
 case "id" -> {  
 id = Integer.*parseInt*(lineContent[1]);  
 }  
 default -> {  
 from = new Node(lineContent[0]);  
 if (lineContent.length == 1) {  
 automaton.addNode(from);  
 automaton.addEndNode(from);  
 continue;  
 }  
 transition = lineContent[1].substring(0, lineContent[1].length() - 1);  
 to = new Node(String.*valueOf*(lineContent[1].charAt(lineContent[1].length() - 1)));  
 automaton.addEdge(from, to, transition);  
 }  
 }  
 }  
 *content* = sb.toString();  
 System.*out*.println("Successfully opened " + filePath);  
 return true;  
 }  
 }  
  
 public void createNewFile(String filePath) throws CreatingFileException, IOException {  
 File file = new File(filePath);  
 if (!file.createNewFile()) {  
 throw new CreatingFileException("Failed to create new file: " + filePath);  
 }  
 }  
}

Класът *FileOpener* отговаря за отварянето и четенето на файлове, които съдържат информация за автомати.

Класът може да отвори указаният файл и да прочете неговото съдържание. Чрез използването на *BufferedReader* и *FileReader* се създава входен поток за четене на файл на указания път *filePath*. Ако процесът завърши успешно, се извежда съобщение за успешно отваряне на файл. Методът *openFile()* връща булева стойност, която показва дали отварянето на файла е преминало успешно.

Класът извлича информация за автоматите, включително техните състояния, преходи и други характеристики:

*StringBuilder* се използва се за конкатенация на редовете от файла.

*String line с*ъхранява текущия ред, който се чете от файла.

*String[] lineContent* е масив за съхранение на отделните части на текущия ред след разделянето му.

За всеки ред се извършва анализ за различни видове информация. Текущият автомат се добавя към списъка с автомати *(AutomatonList*) .

Използвайки метода *split()* на стринговете, текущият ред се разделя на части, като се отделят ключ и стойността му.

Извлечената информация се използва за инициализиране на съответните обекти от класовете *Node* и *Automaton*. Състоянията се добавят към автомата, а преходите се добавят към съответните състояния. Съдържанието на файла се съхранява в променлива за бъдещо използване.

В случай че посоченият файл не съществува, класът може да създаде нов.

# Клас AutomatonManager

public class AutomatonManager {  
 private static AutomatonManager *instance* = null;  
 private File openedFile = null;  
  
 private AutomatonManager() {  
 }  
  
 public static AutomatonManager getInstance() {  
 if (*instance* == null) {  
 *instance* = new AutomatonManager();  
 }  
 return *instance*;  
 }  
  
 public File getOpenedFile() {  
 return openedFile;  
 }  
  
 public void setOpenedFile(File openedFile) {  
 this.openedFile = openedFile;  
 }  
  
 public void save() {  
 if (openedFile == null) {  
 System.*err*.println("Error: No file is currently open.");  
 return;  
 }  
 FileAutomatonWriter writer = new FileAutomatonWriter(openedFile.getPath(), FileOpener.*getContent*());  
 writer.write();  
 }  
  
 public void save(int id, String file) {  
 FileAutomatonWriter writer = new FileAutomatonWriter(file, FileOpener.*getContent*());  
 writer.append(id);  
 }  
  
 public void saveAs(String file) {  
 FileAutomatonWriter writer = new FileAutomatonWriter(file, FileOpener.*getContent*());  
 writer.write();  
 }  
  
 public void close() throws NoOpenFileException {  
 if (openedFile != null) {  
 System.*out*.println("Successfully closed " + openedFile.getName());  
 this.openedFile = null;  
 AutomatonList.*getInstance*().setAutomatons(new HashMap<>());  
 } else {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
 }  
}

Класът *AutomatonManager* осигурява функционалност за файловете и тяхното отваряне, записване, затваряне и управление на автоматите.

*setOpenedFile(File openedFile)* задава отворения файл.

*save() з*аписва съдържанието на текущия отворен файл. Проверява дали има отворен файл и ако няма, извежда съобщение за грешка.

*save(int id, String file)* записва съдържанието на автомат с определен идентификатор в указан файл.

*saveAs(String file) з*аписва съдържанието на текущия отворен файл в нов файл с указаното име.

# Клас FileAutomatonWriter

public class FileAutomatonWriter {  
  
 private String filePath;  
 private String content;  
  
 public FileAutomatonWriter(String filePath, String content) {  
 this.filePath = filePath;  
 this.content = content;  
 }  
  
 public String getFilePath() {  
 return filePath;  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 public void write() {  
 StringBuilder contentInfo = new StringBuilder();  
 Map<Integer, Automaton> automatons = AutomatonList.*getInstance*().getAutomatons();  
 for (Map.Entry<Integer, Automaton> entry : automatons.entrySet()) {  
 contentInfo.append("id=").append(entry.getKey()).append("\n");  
 Map<Node, List<Edge>> currentAutomaton = entry.getValue().getAutomaton();  
 for (Map.Entry<Node, List<Edge>> automatonEntry : currentAutomaton.entrySet()) {  
 if (automatonEntry.getValue() == null || automatonEntry.getValue().isEmpty()) {  
 contentInfo.append(automatonEntry.getKey().getValue()).append("=\n");  
 continue;  
 }  
 for (Edge edge : automatonEntry.getValue()) {  
 contentInfo.append(automatonEntry.getKey().getValue()).append("=")  
 .append(edge.getTransition()).append(edge.getEndNode().getValue()).append("\n");  
 }  
 }  
 contentInfo.append("-\n");  
 }  
 this.content = contentInfo.toString();  
 try (PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter(filePath)))) {  
 writer.println(contentInfo.toString());  
 System.*out*.println("Successfully saved to " + filePath);  
 } catch (WritingToFIleException | IOException ex) {  
 System.*err*.println("Error writing to file: " + ex.getMessage());  
 }  
 }  
  
 public void append(int id) {  
 Automaton automaton = AutomatonList.*getInstance*().getAutomaton(id);  
 StringBuilder contentInfo = new StringBuilder(content);  
 contentInfo.append("id=").append(id).append("\n");  
 Map<Node, List<Edge>> automatonMap = automaton.getAutomaton();  
 for (Map.Entry<Node, List<Edge>> automatonEntry : automatonMap.entrySet()) {  
 if (automatonEntry.getValue().isEmpty()) {  
 contentInfo.append(automatonEntry.getKey().getValue()).append("=\n");  
 continue;  
 }  
 for (Edge edge : automatonEntry.getValue()) {  
 contentInfo.append(automatonEntry.getKey().getValue()).append("=")  
 .append(edge.getTransition()).append(edge.getEndNode().getValue()).append("\n");  
 }  
 }  
 contentInfo.append("-\n");  
 this.content = contentInfo.toString();  
 try (PrintWriter writer = new PrintWriter(new BufferedWriter(new FileWriter(filePath, true)))) {  
 writer.println(contentInfo.toString());  
 System.*out*.println("Successfully appended to " + filePath);  
 } catch (WritingToFIleException | IOException ex) {  
 System.*err*.println("Error appending to file: " + ex.getMessage());  
 }  
 }  
}

Класът *FileAutomatonWriter* е предназначен за записване и добавяне на данни за автомати в текстови файлове.

Методът *write* записва информацията за всички автомати в AutomatonList във файла, посочен от *filePath*. Създава *StringBuilder* за съдържанието и звлича всички автомати от *AutomatonList*. За всеки автомат се извличат неговите състояния и преходи. За всяко състояние се проверява дали има преходи. Ако няма преходи, се записва само състоянието. Ако има преходи, те се записват във формат състояние=преход към състояние. След като цялата информация за автоматите е добавена в *StringBuilder*, тя се записва във файла, посочен от *filePath*. При успех се изписва съобщение на конзолата. При грешка се хваща изключение и се извежда съобщение за грешка.

Методът *append* добавя информация за конкретен автомат с даден идентификатор към съществуващото съдържание във файла, чрез променливата content, в която се записва новото съдържание.

# Клас Open

public class Open extends DefaultCommand {  
  
 public Open() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) throws ErrorOpeningException, IOException {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() != null) {  
 throw new AlreadyOpenedFileException("A file is already opened!");  
 }  
 if (arguments.size() != 1) {  
 throw new InvalidArguments("Invalid number of arguments. Usage: open <file>");  
 }  
 String filePath = arguments.get(0);  
 FileOpener fileOpener = new FileOpener(filePath);  
 if (fileOpener.openFile()) {  
 AutomatonManager.*getInstance*().setOpenedFile(new File(filePath));  
 }  
 } catch (AlreadyOpenedFileException | InvalidArguments e) {  
 System.*err*.println(e.getMessage());  
 } catch (ErrorOpeningException e) {  
 System.*err*.println("An error occurred while opening the file: " + e.getMessage());  
 }  
 }  
}

Класът Open е предназначен за отварянето на файл в система, която работи с недетерминирани крайни автомати. Той проверява дали вече има отворен файл, проверява аргументите, опитва да отвори нов файл и обработва възможни грешки. Основният метод на класа е *execute*, който извършва отварянето на файл. Проверява дали вече има отворен файл в *AutomatonManager* и дали броят на аргументите е точно един. Присвоява на променлива пътя до файла от аргументите. Създава се обект *FileOpener* с пътя до файла и се опитва да отвори файла с метода *openFile()*.

# 

# Клас Save

public class Save extends DefaultCommand {  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.isEmpty()) {  
 AutomatonManager.*getInstance*().save();  
 } else if (arguments.size() >= 2) {  
 int id = Integer.*parseInt*(arguments.get(0));  
 String filename = arguments.get(1);  
   
 if (AutomatonList.*getInstance*().getAutomatons().containsKey(id)) {  
 AutomatonManager.*getInstance*().save(id, filename);  
 System.*out*.println("Successfully saved '" + filename + "'.");  
 } else {  
 throw new AutomatonNotFoundException("Automaton with ID " + id + " not found.");  
 }  
 } else {  
 System.*err*.println("Error: 'save' command requires an ID and a filename.");  
 }  
 } catch (NoOpenFileException | AutomatonNotFoundException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
}

Класът Save е предназначен за записване на направени промени във файл. Той наследява DefaultCommand и реализира метода execute, който изпълнява командата за записване.

Методът execute приема списък от аргументи *List<String>,* които определят начина на запис. Ако няма подадени аргументи, се извиква методът *save* на *AutomatonManager*, който запазва всички отворени автомати. Ако има подадени аргументи, се очаква те да бъдат два: идентификатор на автомата и път до файла.

# 

# Клас SaveAs

public class SaveAs extends DefaultCommand {  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 1) {  
 throw new InvalidArguments("Usage: saveas <filepath>");  
 }  
  
 String filePath = arguments.get(0);  
 AutomatonManager.*getInstance*().saveAs(filePath);  
 } catch (NoOpenFileException | InvalidArguments e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
}

Класът *SaveAs* представлява команда за запазване на всички отворени автомати в посочен файл. Извлича пътя до файла от аргументите. Извиква метода *saveAs* на *AutomatonManager* с пътя до файла, за да запази информацията в указания файл.

# Клас Close

public class Close extends DefaultCommand {  
  
 public Close() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
  
 if (!arguments.isEmpty()) {  
 throw new InvalidArguments("Usage: close");  
 }  
  
 AutomatonManager.*getInstance*().close();  
 } catch (InvalidArguments e) {  
 System.*err*.println(e.getMessage());  
 } catch (NoOpenFileException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
}

Класът *Close* представлява команда за затваряне на текущо отворен файл, като се обръща към мерода *сlose*() в мениджъра.

# Клас Concat

public class Concat extends DefaultCommand {  
  
 public Concat() {  
 }  
  
 private final AutomatonList automatonList = AutomatonList.*getInstance*();  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 2) {  
 throw new IllegalArgumentException("Error: Invalid number of arguments. Usage: concat <id1> <id2>");  
 }  
  
 int id1, id2;  
 try {  
 id1 = Integer.*parseInt*(arguments.get(0));  
 id2 = Integer.*parseInt*(arguments.get(1));  
 } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Error: Invalid argument(s). IDs must be integers.");  
 }  
  
 if (!automatonList.getAutomatons().containsKey(id1) || !automatonList.getAutomatons().containsKey(id2)) {  
 throw new AutomatonNotFoundException("Error: Automaton with ID " + id1 + " or " + id2 + " not found.");  
 }  
  
 Automaton automaton1 = automatonList.getAutomaton(id1);  
 Automaton automaton2 = automatonList.getAutomaton(id2);  
  
 Automaton concatAutomaton = findConcatenation(automaton1, automaton2);  
  
 int newAutomatonId = automatonList.addAutomaton(concatAutomaton);  
  
 System.*out*.println("New automaton with ID: " + newAutomatonId + " is created!");  
 } catch (NoOpenFileException | IllegalArgumentException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private Automaton findConcatenation(Automaton automaton1, Automaton automaton2) {  
 Automaton concatAutomaton = new Automaton(automaton1.getAlphabet());  
  
  
 for (Node node : automaton1.getAutomaton().keySet()) {  
 concatAutomaton.addNode(node);  
 if (automaton1.isEndNode(node)) {  
 concatAutomaton.addEndNode(node);  
 }  
  
 List<Edge> edges = automaton1.getAutomaton().get(node);  
 for (Edge edge : edges) {  
 concatAutomaton.addEdge(node, edge.getEndNode(), edge.getTransition());  
 }  
 }  
  
  
 for (Node endNode : automaton1.getEndNodes()) {  
 for (Node startNodeAutomaton2 : automaton2.getAutomaton().keySet()) {  
 concatAutomaton.addEdge(endNode, startNodeAutomaton2, "");  
 }  
 }  
  
  
 for (Node node : automaton2.getAutomaton().keySet()) {  
 concatAutomaton.addNode(node);  
 if (automaton2.isEndNode(node)) {  
 concatAutomaton.addEndNode(node);  
 }  
  
 List<Edge> edges = automaton2.getAutomaton().get(node);  
 for (Edge edge : edges) {  
 concatAutomaton.addEdge(node, edge.getEndNode(), edge.getTransition());  
 }  
 }  
  
 return concatAutomaton;  
 }  
}

**Конкатенацията** е операция, при която два автомата се комбинират така, че новополученият да разпознава всички думи, които могат да бъдат формирани чрез “слепване” на дума от езика на първия автомат с дума от езика на втория автомат.

Класът *Concat* представлява команда за конкатенация на два автомата и създаване на нов автомат. Методът execute изпълнява командата и приема списък от аргументи *List<String>* и проверява дали има отворен файл и дали броят на аргументите е точно 2. Проверява дали автоматите с дадените идентификатори съществуват. Извличат се автоматите от *AutomatonList* и се извършва конкатенация чрез метода *findConcatenation*. Новият автомат се добавя в *AutomatonList* и принтира новия идентификатор. Методът *findConcatenation* се извиква от *execute* и създава нов автомат, резултат от конкатенацията на двата подадени автомата. За всеки възел от първия автомат се добавя съответния възел в новия автомат. Ако възелът от първия автомат е краен, също се маркира като краен и в новия автомат. Преходите от първия автомат се добавят към новия със съответните начален и краен възел. За всеки възел от втория автомат се добавя съответния възел в новия автомат. Ако възелът от втория автомат е краен, се маркира като краен и в новия автомат. Преходите от втория автомат се добавят към новия със съответните начален и краен възел. Новият автомат се връща като резултат от метода.

# Клас Deterministic

public class Deterministic extends DefaultCommand {  
  
 public Deterministic() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 1) {  
 throw new IllegalArgumentException("Error: Invalid number of arguments. Usage: deterministic <id>");  
 }  
  
 String idString = arguments.get(0);  
 try {  
 int automatonId = Integer.*parseInt*(idString);  
 Automaton automaton = AutomatonList.*getInstance*().getAutomatons().get(automatonId);  
 if (automaton == null) {  
 throw new AutomatonNotFoundException("Error: Automaton with ID " + automatonId + " not found.");  
 }  
  
 boolean isDeterministic = isAutomatonDeterministic(automaton);  
 if (isDeterministic) {  
 System.*out*.println("Automaton with ID " + automatonId + " is deterministic.");  
 } else {  
 System.*out*.println("Automaton with ID " + automatonId + " is not deterministic.");  
 }  
 } catch (NumberFormatException e) {  
 System.*err*.println("Error: Invalid automaton ID. Please provide a valid integer ID.");  
 }  
 } catch (NoOpenFileException | IllegalArgumentException | AutomatonNotFoundException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 public boolean isAutomatonDeterministic(Automaton automaton) {  
 Map<Node, List<Edge>> automatonMap = automaton.getAutomaton();  
  
 for (Map.Entry<Node, List<Edge>> entry : automatonMap.entrySet()) {  
 List<Edge> edges = entry.getValue();  
 List<String> transitions = new ArrayList<>();  
 for (Edge edge : edges) {  
 transitions.add(edge.getTransition());  
 }  
 long duplicateCount = transitions.stream().distinct().count();  
 if (duplicateCount < transitions.size()) {  
 return false;  
 }  
 }  
 return true;  
 }  
}

Крайните автомати са детерминирани или недетерминирани. При детерминираните може да има най-много по един преход от дадено състояние за всяка буква от азбуката на автомата, докато при недетерминираните са възможни няколко различни прехода за една и съща буква.

Класът *Deterministic* реализира командата за проверка. Извлича се идентификаторът на автомата от списъка с подадените аргументи и се търси съответният автомат по този идентификатор. Извиква се методът *isAutomatonDeterministic*, който приема като аргумент автомат и проверява дали той е детерминиран. За всяко състояние от автомата се проверява дали има единствен преход за всеки символ от азбуката. Събират се всички преходи от съответното състояние и се проверява броят на уникалните преходи. Ако за някое състояние има повече от един преход за един и същи символ, то автоматът не е детерминиран и методът връща *false*.

# Клас Empty

public class Empty extends DefaultCommand {  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 1) {  
 throw new IllegalArgumentException("Error: Invalid number of arguments. Usage: empty <id>");  
 }  
  
 int id;  
 try {  
 id = Integer.*parseInt*(arguments.get(0));  
 } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Error: Invalid automaton ID. Please provide a valid integer ID.");  
 }  
  
 Automaton automaton = AutomatonList.*getInstance*().getAutomaton(id);  
  
 if (automaton == null) {  
 throw new AutomatonNotFoundException("Error: Automaton with id " + id + " not found.");  
 }  
  
 boolean isEmpty = isEmptyLanguage(automaton);  
 if (isEmpty) {  
 System.*out*.println("The language of automaton with id " + id + " is empty.");  
 } else {  
 System.*out*.println("The language of automaton with id " + id + " is not empty.");  
 }  
 } catch (NoOpenFileException | IllegalArgumentException | AutomatonNotFoundException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 public boolean isEmptyLanguage(Automaton automaton) {  
 Set<Node> reachableStates = new HashSet<>();  
 for (Node startNode : automaton.getAutomaton().keySet()) {  
 exploreReachableStates(startNode, reachableStates, automaton);  
 }  
 for (Node endNode : automaton.getEndNodes()) {  
 if (reachableStates.contains(endNode)) {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 private void exploreReachableStates(Node currentNode, Set<Node> reachableStates, Automaton automaton) {  
 if (reachableStates.contains(currentNode)) {  
 return;  
 }  
 reachableStates.add(currentNode);  
  
 List<Edge> edges = automaton.getAutomaton().get(currentNode);  
 if (edges != null) {  
 for (Edge edge : edges) {  
 exploreReachableStates(edge.getEndNode(), reachableStates, automaton);  
 }  
 }  
 }  
}

Един автомат има празен език, ако няма път от началното състояние на автомата до някое от крайните състояния.

Класът *Empty* представлява командата за проверка дали езикът на даден автомат е празен.

Методът *isEmptyLanguage* приема като аргумент автомат и проверява дали неговият език е празен. Извиква методът *exploreReachableStates*, който обхожда всички достижими състояния от началните състояния на автомата. Проверява се дали състоянията, които са крайни, са достижими. Ако някое крайно състояние е достижимо, то езикът на автомата не е празен и методът връща *false*. Ако нито едно крайно състояние не е достижимо, методът връща true. Ако текущото състояние е вече посетено, методът приключва, а ако не е посетено, то се добавя към множеството с достижими състояния и за всеки преход от текущото състояние се извиква рекурсивно същият метод за следващото състояние. Това продължава, докато не се обходят всички достижими състояния от началните.

# Клас Help

1. public class Help extends DefaultCommand {  
     
    public Help() {  
    }  
     
    @Override  
    public void execute(List<String> arguments) {  
    for (Operations value : Operations.*values*()) {  
     
    System.*out*.printf("%-20s%s\n",value.getCommand().toUpperCase(), value.getDescription());  
     
    }  
     
    }  
   }

Класът *Help* представлява командата за извеждане на помощна информация за всички операции, достъпни в програмата. Тази команда обхожда всички операции, дефинирани в енумерацията *Operations* и извежда името на командите с кратко описание.

# Клас IsLanguageFinite

public class IsLanguageFinite extends DefaultCommand {  
  
 private final AutomatonList automatonList = AutomatonList.*getInstance*();  
  
 public IsLanguageFinite() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 1) {  
 throw new IllegalArgumentException("Error: Invalid number of arguments. Usage: finite <id>");  
 }  
  
 int id;  
 try {  
 id = Integer.*parseInt*(arguments.get(0));  
 } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Error: Invalid automaton ID. Please provide a valid integer ID.");  
 }  
  
 Automaton automaton = automatonList.getAutomaton(id);  
  
 if (automaton == null) {  
 throw new AutomatonNotFoundException("Error: Automaton with ID " + id + " not found.");  
 }  
  
 boolean isFinite = isLanguageFinite(automaton);  
  
 if (isFinite) {  
 System.*out*.println("The language of automaton with ID " + id + " is finite.");  
 } else {  
 System.*out*.println("The language of automaton with ID " + id + " is infinite.");  
 }  
 } catch (NoOpenFileException | IllegalArgumentException | AutomatonNotFoundException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 public boolean isLanguageFinite(Automaton automaton) {  
 Set<Node> reachableStates = findReachableStates(automaton);  
  
 return !hasCycle(automaton, reachableStates);  
 }  
  
 private Set<Node> findReachableStates(Automaton automaton) {  
  
 Set<Node> reachableStates = new HashSet<>();  
 Queue<Node> queue = new LinkedList<>();  
 Node startState = new Node("S");  
  
 queue.add(startState);  
 reachableStates.add(startState);  
  
 while (!queue.isEmpty()) {  
 Node currentState = queue.poll();  
 List<Edge> edges = automaton.getAutomaton().get(currentState);  
 if (edges != null) {  
 for (Edge edge : edges) {  
 Node nextState = edge.getEndNode();  
 if (!reachableStates.contains(nextState)) {  
 reachableStates.add(nextState);  
 queue.add(nextState);  
 }  
 }  
 }  
 }  
  
 return reachableStates;  
 }  
  
 private boolean hasCycle(Automaton automaton, Set<Node> reachableStates) {  
 Set<Node> visited = new HashSet<>();  
 Set<Node> recStack = new HashSet<>();  
  
 for (Node node : reachableStates) {  
 if (detectCycle(automaton, node, visited, recStack)) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 private boolean detectCycle(Automaton automaton, Node node, Set<Node> visited, Set<Node> recStack) {  
 if (recStack.contains(node)) {  
 return true;  
 }  
 if (visited.contains(node)) {  
 return false;  
 }  
 visited.add(node);  
 recStack.add(node);  
  
 List<Edge> edges = automaton.getAutomaton().get(node);  
 if (edges != null) {  
 for (Edge edge : edges) {  
 if (detectCycle(automaton, edge.getEndNode(), visited, recStack)) {  
 return true;  
 }  
 }  
 }  
  
 recStack.remove(node);  
 return false;  
 }  
}

Езикът на автомата се нарича краен, ако може да бъде описан от крайно множество от думи.

Класът *IsLanguageFinite* представлява командата за проверка дали езикът на даден автомат е краен или безкраен. Той анализира достижимите състояния в автомата и проверява дали има цикли, които биха превърнали езика в безкраен.

Методът *execute* се извиква, когато се изпълни командата "finite" в програмата. Извиква се методът *isLanguageFinite* , който приема като аргумент автомат и проверява дали неговият език е краен. Първо намират достижимите състояния в автомата чрез метода *findReachableStates*.След това се проверява дали в достижимите състояния има цикли, които биха превърнали езика в безкраен, чрез метода *hasCycle*.

Методът *findReachableStates* намира всички достижими състояния в автомата, използвайки алгоритъма на BFS (обхождане в ширина).Започва с началното състояние на автомата и добавя всички достижими състояния в опашката.При обхождане на състоянията, достигнатите състояния се добавят в множество *reachableStates.*

Методът *hasCycle* проверява дали автоматът има цикъл в достижимите състояния. Използва се алгоритъм на DFS (обхождане в дълбочина), за да се провери за наличие на цикли.Използва се вътрешният метод *detectCycle*, за да се извърши рекурсивна проверка за цикли.

Методът *detectCycle*  извършва рекурсивна проверка за наличие на цикъл в автомата.Използва се DFS за обхождане на състоянията на автомата.Проверява се дали има цикъл чрез проверка на състоянията, които са били посетени, както и чрез проверка на за състояния, които се срещат повторно.

# Клас ListCommand

public class ListCommand extends DefaultCommand {  
 public ListCommand() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 AutomatonManager manager = AutomatonManager.*getInstance*();  
 if(manager.getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
 AutomatonList automatonList = AutomatonList.*getInstance*();  
 List<Integer> allAutomatonIds = automatonList.getAllAutomatonIds();  
  
 if (allAutomatonIds.isEmpty()) {  
 System.*out*.println("There are no automatons.");  
 } else {  
 System.*out*.println("List of automatons IDs:");  
 for (Integer id : allAutomatonIds) {  
 System.*out*.print(id);  
 if (allAutomatonIds.indexOf(id) < allAutomatonIds.size() - 1) {  
 System.*out*.print(", ");  
 }  
 }  
 }  
 System.*out*.print("\n");  
 } catch (NoOpenFileException | AutomatonNotFoundException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
}

Класът *ListCommand* представлява команда за извеждане на списък с идентификаторите на всички налични автомати.

Методът *execute(List<String> arguments)* се извиква, когато се изпълнява командата. След това получава списъка с всички идентификатори на автоматите от AutomatonList. Ако списъкът е празен, се извежда съобщение, че няма никакви автомати. В противен случай се извеждат идентификаторите на автоматите, разделени със запетайки.

# Клас Print

public class Print extends DefaultCommand {  
  
 public Print() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 1) {  
 throw new IllegalArgumentException("Usage: print <id>");  
 }  
  
 int id;  
 try {  
 id = Integer.*parseInt*(arguments.get(0));  
 } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Error: Invalid automaton ID. Please provide a valid integer ID.");  
 }  
 Automaton automaton = getAutomatonById(id);  
  
 if (automaton == null) {  
 throw new AutomatonNotFoundException("Automaton with id " + id + " not found.");  
 }  
  
 System.*out*.println(automaton.toString());  
 } catch (NoOpenFileException | IllegalArgumentException | AutomatonNotFoundException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private Automaton getAutomatonById(int id) {  
 AutomatonList automatonList = AutomatonList.*getInstance*();  
 return automatonList.getAutomatons().get(id);  
 }  
}

Класът *Print* е команда, която извежда информация за автомат с определен идентификатор.

Ако всички проверки са успешни, се извежда информацията за автомата, като се използва метода *toString()* на класа *Automaton*, който връща текстово представяне на състоянието на автомата. Методът *getAutomatonById(int id)* получава автомат по даден идентификатор от *AutomatonList.*

# Клас Recognize

public class Recognize extends DefaultCommand {  
  
 public Recognize() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 2) {  
 throw new IllegalArgumentException("Usage: recognize <id> <word>");  
 }  
  
 int id = parseId(arguments.get(0));  
 String word = arguments.get(1);  
  
 Automaton automaton = getAutomatonById(id);  
  
 if (automaton == null) {  
 System.*out*.println("Automaton with id " + id + " not found.");  
 return;  
 }  
  
 boolean result = containsWord(word, automaton);  
  
 if (result) {  
 System.*out*.println("The word '" + word + "' is in the language of the automaton.");  
 } else {  
 System.*out*.println("The word '" + word + "' is not in the language of the automaton.");  
 }  
 } catch (NoOpenFileException | IllegalArgumentException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private int parseId(String idString) {  
 try {  
 return Integer.*parseInt*(idString);  
 } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Error: Invalid automaton ID. Please provide a valid integer ID.");  
 }  
 }  
  
 private boolean containsWord(String word, Automaton automaton) {  
 String generatedWord = automaton.generateWord();  
 return generatedWord.contains(word);  
 }  
  
 private Automaton getAutomatonById(int id) {  
 AutomatonList automatonList = AutomatonList.*getInstance*();  
 return automatonList.getAutomatons().get(id);  
 }  
}

Класът *Recognize* представлява команда за проверка дали дадена дума принадлежи на езика на даден автомат.

Методът *getAutomatonById(int id)* връща автомата със зададения идентификатор. Извиква се метода *containsWord(String word, Automaton automaton)* за да провери дали думата принадлежи на езика на автомата.

Методът *containsWord(String word, Automaton automaton*) приема дума и автомат като аргументи. Генерира дума от езика на автомата, като извиква метода *generateWord()* на автомата и проверява дали генерираната дума съдържа подадената дума.

# Клас Un

public class Un extends DefaultCommand {  
 private final AutomatonList automatonList = AutomatonList.*getInstance*();  
  
 public Un() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 1) {  
 throw new IllegalArgumentException("Error: Invalid number of arguments. Usage: un <id>");  
 }  
  
 int id;  
 try {  
 id = Integer.*parseInt*(arguments.get(0));  
 } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Error: Invalid argument. ID must be an integer.");  
 }  
  
 if (!automatonList.getAutomatons().containsKey(id)) {  
 throw new AutomatonNotFoundException("Error: Automaton with ID " + id + " not found.");  
 }  
  
 Automaton automaton = automatonList.getAutomaton(id);  
 Automaton unAutomaton = findUnreachableStates(automaton);  
  
 int newAutomatonId = automatonList.addAutomaton(unAutomaton);  
  
 System.*out*.println("New automaton with ID: " + newAutomatonId + " is created!");  
 } catch (NoOpenFileException | IllegalArgumentException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private Automaton findUnreachableStates(Automaton automaton) {  
 Set<Node> reachableStates = findReachableStates(automaton);  
 Automaton unAutomaton = new Automaton(automaton.getAlphabet());  
  
 for (Node node : reachableStates) {  
 unAutomaton.addNode(node);  
 if (automaton.isEndNode(node)) {  
 unAutomaton.addEndNode(node);  
 }  
  
 List<Edge> edges = automaton.getAutomaton().get(node);  
 for (Edge edge : edges) {  
 if (reachableStates.contains(edge.getEndNode())) {  
 unAutomaton.addEdge(node, edge.getEndNode(), edge.getTransition());  
 }  
 }  
 }  
  
 return unAutomaton;  
 }  
  
 private Set<Node> findReachableStates(Automaton automaton) {  
 Set<Node> reachableStates = new HashSet<>();  
 Set<Node> visited = new HashSet<>();  
 Set<Node> queue = new HashSet<>();  
  
 Node startNode = new Node("S");  
  
 queue.add(startNode);  
  
 while (!queue.isEmpty()) {  
 Node node = queue.iterator().next();  
 queue.remove(node);  
  
 if (!visited.contains(node)) {  
 visited.add(node);  
 reachableStates.add(node);  
  
 List<Edge> edges = automaton.getAutomaton().get(node);  
 if (edges != null) {  
 for (Edge edge : edges) {  
 queue.add(edge.getEndNode());  
 }  
 }  
 }  
 }  
  
 return reachableStates;  
 }  
}

Позитивната обвивка на автомат е множеството от всички състояния, до които може да се достигне от началното състояние, като се следват всички възможни преходи в автомата.

Класът *Un* реализира командата *un*, която намира позитивна обвивка на даден автомат и създава нов автомат, който съдържа само достижимите състояния от началния и отпечатва идентификатора на новия.

*Методът findUnreachableStates(Automaton automaton)* приема автомат като аргумент и връща нов автомат, от който са премахнати недостижимите състояния. Използва метода *findReachableStates(Automaton automaton)* за намиране на достижимите състояния от дадения автомат. Създава нов празен автомат, базиран на азбуката на подадения.

Методът *findReachableStates(Automaton automaton)* приема автомат като аргумент и връща множество от достижимите състояния от дадения автомат. Създава празно множество за достижимите състояния, множество за посетени състояния и опашка за обхождане. Започва от стартовото състояние на автомата и добавя го към опашката. Докато опашката не е празна, взима следващото състояние и добавя всички достижими състояния към множеството на достижимите състояния. Връща множеството на достижими състояния.

# Клас Union

public class Union extends DefaultCommand {  
 private final AutomatonList automatonList = AutomatonList.*getInstance*();  
  
 public Union() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 2) {  
 throw new IllegalArgumentException("Error: Invalid number of arguments. Usage: union <id1> <id2>");  
 }  
  
 int id1, id2;  
 try {  
 id1 = Integer.*parseInt*(arguments.get(0));  
 id2 = Integer.*parseInt*(arguments.get(1));  
 } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Error: Invalid argument. IDs must be integers.");  
 }  
  
 if (!automatonList.getAutomatons().containsKey(id1) || !automatonList.getAutomatons().containsKey(id2)) {  
 throw new IllegalArgumentException("Error: Automaton with ID " + id1 + " or " + id2 + " not found.");  
 }  
  
 Automaton automaton1 = automatonList.getAutomaton(id1);  
 Automaton automaton2 = automatonList.getAutomaton(id2);  
  
 Automaton unionAutomaton = findUnion(automaton1, automaton2);  
  
 int newAutomatonId = automatonList.addAutomaton(unionAutomaton);  
  
 System.*out*.println("New automaton with ID: " + newAutomatonId + " is created!");  
 } catch (NoOpenFileException | IllegalArgumentException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private Automaton findUnion(Automaton automaton1, Automaton automaton2) {  
  
 Automaton unionAutomaton = new Automaton(automaton1.getAlphabet());  
  
 for (Node node : automaton1.getAutomaton().keySet()) {  
 unionAutomaton.addNode(node);  
 if (automaton1.isEndNode(node)) {  
 unionAutomaton.addEndNode(node);  
 }  
 }  
  
 for (Node node : automaton2.getAutomaton().keySet()) {  
 unionAutomaton.addNode(node);  
 if (automaton2.isEndNode(node)) {  
 unionAutomaton.addEndNode(node);  
 }  
 }  
  
 for (Node from : automaton1.getAutomaton().keySet()) {  
 List<Edge> edges = automaton1.getAutomaton().get(from);  
 for (Edge edge : edges) {  
 Node to = edge.getEndNode();  
 String transition = edge.getTransition();  
 unionAutomaton.addEdge(from, to, transition);  
 }  
 }  
  
 for (Node from : automaton2.getAutomaton().keySet()) {  
 List<Edge> edges = automaton2.getAutomaton().get(from);  
 for (Edge edge : edges) {  
 Node to = edge.getEndNode();  
 String transition = edge.getTransition();  
 unionAutomaton.addEdge(from, to, transition);  
 }  
 }  
  
 return unionAutomaton;  
 }  
}

Обединението на два автомата е нов автомат, който включва всички състояния и преходи от двете начални автомата. Обединеният автомат представлява комбинация от двата автомата, като е в състояние да разпознава думите, които са приемани от поне един от тях.

Класът *Union* реализира командата за обединение на два автомата. Взимат се двата автомата от списъка с автомати. Създава нов автомат, който ще бъде обединението на двете входни автомата. Добавя състоянията на първия и тези на втория автомат в новия. По същия начин добавя преходите на първия и втория автомат в новия автомат. Новият автомат се записва в списъка с автомати.

Методът *findUnion* извършва стъпките за създаване на обединението на двата автомата. Той добавя състоянията и преходите от двата автомата в новия.

# Клас Mutator

public class Mutator extends DefaultCommand {  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() < 1) {  
 throw new IllegalArgumentException("Please provide the ID of the automaton to determinize.");  
 }  
  
 int automatonId;  
 try {  
 automatonId = Integer.*parseInt*(arguments.get(0));  
 } catch (NumberFormatException e) {  
 throw new IllegalArgumentException("Invalid automaton ID. Please provide a valid integer ID.");  
 }  
  
 AutomatonList automatonList = AutomatonList.*getInstance*();  
 Automaton nfa = automatonList.getAutomaton(automatonId);  
  
 if (nfa == null) {  
 throw new IllegalArgumentException("Automaton with ID " + automatonId + " does not exist.");  
 }  
  
  
 Deterministic deterministicCommand = new Deterministic();  
 boolean isDeterministic = deterministicCommand.isAutomatonDeterministic(nfa);  
  
 if (isDeterministic) {  
 System.*out*.println("Automaton with ID " + automatonId + " is already deterministic.");  
 return;  
 }  
  
 Automaton dfa = determinize(nfa);  
 if (dfa == null) {  
 System.*out*.println("Failed to determinize automaton with ID " + automatonId + ".");  
 return;  
 }  
  
 int newAutomatonId = automatonList.addAutomaton(dfa);  
  
 System.*out*.println("Automaton " + automatonId + " has been determinized.");  
 System.*out*.println("New DFA has been added with ID: " + newAutomatonId);  
 } catch (NoOpenFileException | IllegalArgumentException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private Automaton determinize(Automaton nfa) {  
 String alphabet = nfa.getAlphabet();  
 Automaton dfa = new Automaton(alphabet);  
  
 Map<Set<Node>, Node> dfaStates = new HashMap<>();  
 Queue<Set<Node>> queue = new LinkedList<>();  
  
 Set<Node> startState = new HashSet<>();  
 startState.add(new Node("S"));  
 queue.add(startState);  
 Node dfaStartState = new Node(stateSetToString(startState));  
 dfa.addNode(dfaStartState);  
 dfaStates.put(startState, dfaStartState);  
  
 while (!queue.isEmpty()) {  
 Set<Node> currentStateSet = queue.poll();  
 Node currentDfaState = dfaStates.get(currentStateSet);  
  
 for (char c : alphabet.toCharArray()) {  
 Set<Node> nextStateSet = new HashSet<>();  
 for (Node node : currentStateSet) {  
 List<Edge> edges = nfa.getAutomaton().get(node);  
 if (edges != null) {  
 for (Edge edge : edges) {  
 if (edge.getTransition().equals(String.*valueOf*(c))) {  
 nextStateSet.add(edge.getEndNode());  
 }  
 }  
 }  
 }  
  
 if (!nextStateSet.isEmpty()) {  
 Node dfaNextState;  
 if (!dfaStates.containsKey(nextStateSet)) {  
 dfaNextState = new Node(stateSetToString(nextStateSet));  
 dfa.addNode(dfaNextState);  
 dfaStates.put(nextStateSet, dfaNextState);  
 queue.add(nextStateSet);  
 } else {  
 dfaNextState = dfaStates.get(nextStateSet);  
 }  
  
 dfa.addEdge(currentDfaState, dfaNextState, String.*valueOf*(c));  
  
 if (containsEndNode(nextStateSet, nfa.getEndNodes())) {  
 dfa.addEndNode(dfaNextState);  
 }  
 }  
 }  
 }  
  
 return dfa;  
 }  
  
 private String stateSetToString(Set<Node> stateSet) {  
 List<String> sortedStateNames = new ArrayList<>();  
 for (Node node : stateSet) {  
 sortedStateNames.add(node.getValue());  
 }  
 Collections.*sort*(sortedStateNames);  
 return String.*join*(",", sortedStateNames);  
 }  
  
 private boolean containsEndNode(Set<Node> stateSet, List<Node> endNodes) {  
 for (Node endNode : endNodes) {  
 if (stateSet.contains(endNode)) {  
 return true;  
 }  
 }  
 return false;  
 }  
}

Класът *Mutator* реализира командата за детерминизация на автомат.

Инициализират се две структури *dfaStates*:

*Map* -карта, която отразява връзката между множества от състояния на недетерминирания автомат и съответните състояния в детерминирания.

*Queue* - Опашка, която съдържа множествата от състояния, които още не са обработени.

Методът *determinize* осъществява процеса на детерминация на автомат. В него се създава началното множество от състояния на детерминирания автомат (*startState*). Това множество съдържа само едно ново състояние, което се създава с идентификатор "S". Това състояние също се добавя в опашката и в *dfaStates*. Започва процеса на детерминизация чрез цикъл, който продължава докато опашката не е празна. Взима се текущото множество от състояния за обработка от опашката, създава се съответното състояние в детерминирания автомат и се добавя в *dfaStates*. Търси се множеството от състояния, до които може да се стигне от текущото множество състояния с тази буква. Проверява се дали новото множество от състояния е празно. Ако не е, се създава съответното състояние в детерминирания автомат, ако то не съществува вече. Добавя се съответната връзка между текущото и следващото състояние с тази буква в новия автомат. Ако новото множество от състояния съдържа поне едно крайно състояние от автомата, то следващото състояние става крайно състояние в новия автомат.

Методът *stateSetToString*, преобразува множество от състояния в низ. Създава се празен списък от низове *sortedStateNames*, който ще съдържа имената на състоянията от множеството *stateSet*. За всяко състояние *node* от *stateSet*, методът взима името на състоянието, извлечено чрез метода *getValue()*, и го добавя към списъка sortedStateNames.

### Методът *containsEndNode(Set<Node> stateSet, List<Node> endNodes)* проверява дали даден набор от състояния (*stateSet*) съдържа поне едно крайно състояние от списъка с крайни състояния на автомата (*endNodes*). Методът преминава през всяко крайно състояние в списъка endNodes и проверява дали този елемент се съдържа в множеството *stateSet*. Във всяка итерация се проверява дали текущото крайно състояние (*endNode*) се съдържа в множеството *stateSet* чрез метода *contains*.

# Клас Reg

public class Reg extends DefaultCommand {  
  
 private final AutomatonList automatonList = AutomatonList.*getInstance*();  
 private final String ALPHABET = "abcdefghijklmnopqrstuvwxyz123456789";  
  
 @Override  
 public void execute(List<String> arguments) {  
 try {  
 if (AutomatonManager.*getInstance*().getOpenedFile() == null) {  
 throw new NoOpenFileException("Error: No file is currently open.");  
 }  
  
 if (arguments.size() != 1) {  
 throw new IllegalArgumentException("Error: Invalid number of arguments. Usage: reg <regex>");  
 }  
  
 String regex = arguments.get(0);  
  
 if (!isValidRegex(regex)) {  
 throw new IllegalArgumentException("Error: Invalid regular expression format. Only characters from the alphabet are allowed.");  
 }  
  
 Automaton regAutomaton = createAutomatonFromRegex(regex);  
 if (regAutomaton == null) {  
 System.*out*.println("Error: Failed to create automaton from regex.");  
 return;  
 }  
  
 int newAutomatonId = automatonList.addAutomaton(regAutomaton);  
  
 System.*out*.println("New automaton with ID: " + newAutomatonId + " is created!");  
 } catch (NoOpenFileException | IllegalArgumentException e) {  
 System.*err*.println(e.getMessage());  
 }  
 }  
  
 private boolean isValidRegex(String regex) {  
 for (char c : regex.toCharArray()) {  
 if (ALPHABET.indexOf(c) == -1 && c != '+' && c != '\*') {  
 return false;  
 }  
 }  
 return true;  
 }  
  
 private Automaton createAutomatonFromRegex(String regex) {  
 Automaton automaton = new Automaton(ALPHABET);  
  
 int m = regex.length();  
 Stack<Node> stack = new Stack<>();  
 Node start = new Node("S");  
 automaton.addNode(start);  
 stack.push(start);  
  
 String capitals = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
 int idIndex = 0;  
  
 List<Node> endNodes = new ArrayList<>();  
  
 for (int i = 0; i < m; i++) {  
 char c = regex.charAt(i);  
 if (c == '+') {  
 stack.pop();  
 }else if (c == '\*') {  
 if (!stack.isEmpty()) {  
 Node prevNode = stack.pop();  
 if (!stack.isEmpty()) {  
 Node previousNode = stack.peek();  
 automaton.addEdge(previousNode, prevNode, "\*"); //\*=Ɛ  
 }  
 automaton.addEdge(prevNode, prevNode, "\*");  
 stack.push(prevNode);  
 } else {  
 System.*out*.println("Error: Invalid regular expression.");  
 return null;  
 }  
 }else {  
 Node newNode = new Node(Character.*toString*(capitals.charAt(idIndex)));  
 idIndex++;  
 automaton.addNode(newNode);  
 automaton.addEdge(stack.peek(), newNode, Character.*toString*(c));  
 stack.push(newNode);  
 }  
 }  
  
 Node lastNode = stack.peek();  
 endNodes.add(lastNode);  
 automaton.addEndNode(lastNode);  
  
 return automaton;  
 }  
}

Теоремата на Клини установява еквивалентност между регулярните изрази и краен автомат. Теоремата казва, че за всеки регулярен израз съществува краен автомат, който разпознава точно тези низове, които са описани от регулярния израз, и обратно - за всеки краен автомат съществува регулярен израз, който описва езика, разпознаван от автомата.

Класът *Reg* предоставя функционалност за създаване на автомати от регулярни изрази и проверява валидността на входните данни.

Методът *isValidRegex* проверява дали всички символи от регулярния израз са част от азбуката или от операторите.

Методът *createAutomatonFromRegex* създава автомат от регулярния израз. Използва стек за следене на текущото състояние на автомата. Обхожда всеки символ от регулярния израз. Ако символът е оператор "+", премахва горния елемент от стека. Ако символът е оператор "\*", изтегля последното добавено състояние от стека и го маркира като финално. Създава преход от предишното състояние към последното състояние с празен символ (ε), но в програмата се ползва (\*). Ако стекът не е празен, създава преход от предишното състояние към последното състояние с празен символ (\*). Ако символът е буква от азбуката, създава ново състояние и добавя преход към него от предишното състояние. Последното състояние в стека се маркира като финално.

# Клас Automaton

public class Automaton {  
 private Map<Node, List<Edge>> automaton = new HashMap<>();  
 private String alphabet;  
 private List<Node> endNodes = new ArrayList<>();  
  
 public List<Node> getEndNodes() {  
 return endNodes;  
 }  
  
  
 public Automaton(String alphabet) {  
 this.alphabet = alphabet;  
 }  
  
 public Map<Node, List<Edge>> getAutomaton() {  
 return automaton;  
 }  
  
 public String getAlphabet() {  
 return alphabet;  
 }  
  
 public String generateWord() {  
 Set<Node> visited = new LinkedHashSet<>();  
 Queue<Node> queue = new LinkedList<>();  
  
 Node start = new Node("S");  
  
 queue.add(start);  
 visited.add(start);  
  
 StringBuilder result = new StringBuilder();  
  
 while (!queue.isEmpty()) {  
 Node node = queue.poll();  
 List<Edge> edges = automaton.get(node);  
 if (edges != null) {  
 for (Edge edge : edges) {  
 if (!visited.contains(edge.getEndNode())) {  
 visited.add(edge.getEndNode());  
 queue.add(edge.getEndNode());  
 result.append(edge.getTransition());  
 if (isEndNode(edge.getEndNode())) {  
 return result.toString();  
 }  
 }  
 }  
 }  
 }  
  
 return result.toString();  
 }  
  
 public void addNode(Node node) {  
 automaton.put(node, new ArrayList<>());  
 }  
  
 public void addEndNode(Node node) {  
 if (!endNodes.contains(node)) {  
 endNodes.add(node);  
 }  
 }  
  
  
  
 public boolean isEndNode(Node node) {  
 return endNodes.contains(node);  
 }  
  
 public void addEdge(Node from, Node to, String transition) {  
 if (transition == null) {  
  
 transition = "";  
 } else if (transition.isEmpty()) {  
  
 transition = "";  
 }  
  
 if (!automaton.containsKey(from)) {  
 automaton.put(from, new ArrayList<>());  
 }  
  
 if (!automaton.containsKey(to)) {  
 automaton.put(to, new ArrayList<>());  
 }  
  
 automaton.get(from).add(new Edge(transition, to));  
 }  
  
 @Override  
 public String toString() {  
 StringBuilder automatonInfo = new StringBuilder();  
  
 int index = 1;  
 for (Node node : automaton.keySet()) {  
 for (Edge edge : automaton.get(node)) {  
 automatonInfo  
 .append(index++)  
 .append(". ")  
 .append(node.getValue())  
 .append(" -> ")  
 .append(edge.getTransition())  
 .append(edge.getEndNode().getValue())  
 .append("\n");  
 }  
 }  
  
 for (Node node : endNodes) {  
 automatonInfo  
 .append(index++)  
 .append(". ")  
 .append(node.getValue())  
 .append(" -> ")  
 .append("\n");  
 }  
  
 return automatonInfo.toString();  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (!(o instanceof Automaton)) return false;  
 Automaton automaton1 = (Automaton) o;  
 return Objects.*equals*(getAutomaton(), automaton1.getAutomaton()) && Objects.*equals*(getAlphabet(), automaton1.getAlphabet()) && Objects.*equals*(endNodes, automaton1.endNodes);  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(getAutomaton(), getAlphabet(), endNodes);  
 }  
}

Класът Automaton представлява имплементация на автомат.

*automaton* - карта, която съдържа върховете на автомата като ключове и списък от ребрата към съседните върхове като стойности.

*alphabet* - азбуката, която автоматът приема.

*endNodes* - списък от крайните върхове на автомата.

*generateWord()* генерира низ(дума).

Първоначално се инициализират две колекции - *visited* (посетени върхове) и *queue* (опашка). Началния връх е добавен към опашката и посетените върхове. След това се извършва обхождане на автомата чрез *BFS*, като се добавят всички върхове и ребра, които се срещат по пътя, в низа *result*. Когато се стигне до финален връх, низът се връща.

*addNode(Node node)* добавя нов връх към автомата.

*addEndNode(Node node)* добавя връх към списъка на крайните върхове.

*isEndNode(Node node)* проверява дали даден връх е краен.

*addEdge(Node from, Node to, String transition)* добавя ребро между два върха.

*toString()* генерира текстово представяне на автомата, като използва списъците с върхове и ребра.

# Клас AutomatonList

public class AutomatonList {  
  
 private static AutomatonList *singletonInstance*;  
 private Map<Integer, Automaton> automatons = new HashMap<>();  
 private int nextAutomatonId = 1;  
  
 private AutomatonList() {}  
  
 public Map<Integer, Automaton> getAutomatons() {  
 return new HashMap<>(automatons);  
 }  
  
 public void addAutomaton(int id, Automaton automaton) throws InvalidAutomatonException {  
 if (!automaton.getAutomaton().containsKey(new Node("S"))) {  
 throw new InvalidAutomatonException("Automaton must contain the start node - S.");  
 }  
 if (automatons.containsKey(id)) {  
 throw new IllegalArgumentException("Automaton with ID " + id + " already exists.");  
 }  
 automatons.put(id, automaton);  
 }  
  
 public void setAutomatons(Map<Integer, Automaton> automatons) {  
 this.automatons = automatons;  
 }  
  
 public static AutomatonList getInstance(){  
 if(*singletonInstance* == null){  
 *singletonInstance* = new AutomatonList();  
 }  
 return *singletonInstance*;  
 }  
  
 public Automaton getAutomaton(int id) throws AutomatonNotFoundException {  
 Automaton automaton = automatons.get(id);  
 if (automaton == null) {  
 throw new AutomatonNotFoundException("Automaton with ID " + id + " not found.");  
 }  
 return automaton;  
 }  
  
 public List<Integer> getAllAutomatonIds(){  
 return new ArrayList<>(automatons.keySet());  
 }  
  
 public int addAutomaton(Automaton automaton) {  
 int newAutomatonId = generateUniqueAutomatonId();  
 automatons.put(newAutomatonId, automaton);  
 return newAutomatonId;  
 }  
  
 private int generateUniqueAutomatonId() {  
 while (automatons.containsKey(nextAutomatonId)) {  
 nextAutomatonId++;  
 }  
 return nextAutomatonId;  
 }  
}

Класът *AutomatonList* представлява списък от автомати и осигурява основни операции за управление.

*singletonInstance* е статично поле, което съхранява единствената инстанция на класа *AutomatonList.*

*automatons* е карта, където ключовете са уникални идентификатори на автомати, а стойностите са съответните автомати.

*nextAutomatonId* е следващият уникален идентификатор, който ще се използва при добавяне на нов автомат.

*getAutomatons()* връща копие на картата с автоматите.

*addAutomaton(int id, Automaton automaton)* добавя нов автомат към списъка с автомати. Приема идентификатор и автомат. Преди добавянето проверява дали автоматът съдържа стартовия връх "S".

*setAutomatons(Map<Integer, Automaton> automatons)* задава целия списък с автомати.

*getInstance()* връща единствената инстанция на класа *AutomatonList*. Ако тя все още не е създадена я създава.

*getAutomaton(int id)* връща автомат по даден уникален идентификатор.

*getAllAutomatonIds()* връща списък с всички уникални идентификатори на автоматите в списъка.

*addAutomaton(Automaton automaton)* добавя нов автомат към списъка с автомати. Генерира уникален идентификатор за автомата и го добавя към картата с автомати.

# Клас Edge

public class Edge {  
 private String transition;  
 private Node endNode;  
  
 public Edge(String transition, Node endNode) {  
 this.transition = transition;  
 this.endNode = endNode;  
 }  
  
 public String getTransition() {  
 return transition;  
 }  
  
 public Node getEndNode() {  
 return endNode;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (!(o instanceof Edge)) return false;  
 Edge edge = (Edge) o;  
 return Objects.*equals*(getTransition(), edge.getTransition()) && Objects.*equals*(getEndNode(), edge.getEndNode());  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(getTransition(), getEndNode());  
 }  
}

Класът *Edge* представлява връзка между два върха в автомат. Той съдържа информация за прехода и краен върх на тази връзка.

*Transition* е връзката между двата върха.

*endNode* е връхът, към който води връзката.

*Edge(String transition, Node endNode)* е конструктор, който създава нова връзка между върхове. Приема символ за преход и краен връх.

*getTransition()* връща символа за преход на връзката.

*getEndNode()* връща краен връх на връзката.

*equals(Object o)* проверява дали текущият обект е равен на друг обект.

*hashCode()* генерира хеш код за връзката.

# Клас Node

public class Node {  
 private String value;  
  
 public Node(String value) {  
 this.value = value;  
 }  
  
 public String getValue() {  
 return value;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (!(o instanceof Node)) return false;  
 Node node = (Node) o;  
 return Objects.*equals*(getValue(), node.getValue());  
 }  
  
 @Override  
 public int hashCode() {  
 return Objects.*hash*(getValue());  
 }  
}

Класът *Node* представлява възел в автомат.

*Value* е Стринг, който съхранява стойността на възела.

*Node(String value)* е конструктор, който създава нов възел със зададена стойност.

*getValue()* връща стойността на възела.

*equals(Object o)* проверява дали текущият възел е равен на друг възел.

*hashCode()* генерира хеш код за възела.

# Клас (енумерация) Operations

public enum Operations {  
 *OPEN*("open", "Opens a file and reads its contents."),  
 *CLOSE*("close", "Closes the current file."),  
 *SAVE*( "save", "Saves the contents in the current file."),  
 *SAVEAS*("save as", "Saves the contents in a directory, chosen by the user."),  
 *HELP*("help", "Shows all possible commands."),  
 *EXIT*("exit", "Exits the application."),  
 *LIST*("list", "Outputs a List of IDs of all automatons read."),  
 *PRINT*("print", "Displays information about all transitions in the automaton."),  
 *SAVE\_ID\_FILENAME*("save automaton", "Saves an automaton to a file."),  
 *EMPTY*("empty", "Checks if the language is empty."),  
 *DETERMINISTIC*("deterministic", "Checks whether an automaton is deterministic."),  
 *RECOGNIZE*("recognize", "Checks if a word is in the language."),  
 *UNION*("union", "Finds the union of two automatons and creates a new automaton. Prints the ID of the new automaton."),  
 *CONCAT*("concat", "Finds the concatenation of two automatons and creates a new automaton. Prints the ID of the new automaton."),  
 *UN*("un", "Finds the positive envelope of an automaton and creates a new automaton. Prints the ID of the new automaton."),  
 *REG*("reg", "Creates a new automaton based on a specified regular expression (Cliny's theorem). Prints the ID of the new automaton."),  
 *MUTATOR*("mutator", "Determines a given automaton"),  
 *FINITE*("finite","Check whether the language of a given automaton is finite");  
  
  
 private String command;  
 private String description;  
  
 Operations(String command, String description) {  
 this.command = command;  
 this.description = description;  
  
 }  
  
 public String getCommand() {  
 return command;  
 }  
  
 public String getDescription() {  
 return description;  
 }  
}

Класа *Operations* дефинира възможни операции в програмата.

*Command* е стринг, който представлява името на операцията или командата.

*Description* е стринг, който съдържа описание на операцията.

*Operations(String command, String description)* е конструктор, който инициализира нова операция или команда с подаденото име и описание.

*getCommand()* е метод, който връща името на операцията или командата.

*getDescription()* е метод, който връща описанието на операцията или командата.

# Класове interface Command и abstract class DefaultCommand

public interface Command {  
 void execute(List<String> arguments) throws IOException;  
}

public abstract class DefaultCommand implements Command {  
  
}

Тези два класа са част от дизайна на командната система.

Интерфейсът *Command* дефинира метода *execute*, който приема списък от аргументи и не връща резултат. Този метод трябва да бъде реализиран от всички класове, които искат да бъдат команди в приложението.

*DefaultCommand* е абстрактен клас, който имплементира *Command* и е наследяван от класовете за командите.

Този дизайн позволява лесно добавяне на нови команди към системата, като се придържа към принципите на обектно-ориентираното програмиране, като ползва абстракция и интерфейси.

# Клас CLI

public class CLI {  
 private static Map<Operations, Command> *commands* = new HashMap<>();  
  
 static {  
 *commands*.put(Operations.*OPEN*, new Open());  
 *commands*.put(Operations.*CLOSE*, new Close());  
 *commands*.put(Operations.*SAVE*, new Save());  
 *commands*.put(Operations.*SAVEAS*, new SaveAs());  
 *commands*.put(Operations.*HELP*, new Help());  
 *commands*.put(Operations.*EXIT*, new Exit());  
  
 *commands*.put(Operations.*LIST*, new ListCommand());  
 *commands*.put(Operations.*PRINT*, new Print());  
 *commands*.put(Operations.*EMPTY*, new Empty());  
 *commands*.put(Operations.*DETERMINISTIC*, new Deterministic());  
 *commands*.put(Operations.*RECOGNIZE*, new Recognize());  
 *commands*.put(Operations.*UNION*, new Union());  
 *commands*.put(Operations.*CONCAT*, new Concat());  
 *commands*.put(Operations.*UN*, new Un());  
 *commands*.put(Operations.*REG*, new Reg());  
 *commands*.put(Operations.*MUTATOR*, new Mutator());  
 *commands*.put(Operations.*FINITE*, new IsLanguageFinite());  
 }  
  
 public static void start() {  
 Scanner scanner = new Scanner(System.*in*);  
 while (true) {  
 try {  
 System.*out*.print(">");  
 String input = scanner.nextLine();  
 String[] arguments = input.split(" ");  
 Operations operation = Operations.*valueOf*(arguments[0].toUpperCase());  
 Command command = *commands*.get(operation);  
 if (command != null) {  
 command.execute(Arrays.*asList*(arguments).subList(1, arguments.length));  
 } else {  
 System.*out*.println("Error: Unknown command. Type 'help' for a list of available commands.");  
 }  
 } catch (IllegalArgumentException | IOException e) {  
 System.*err*.println("Error: Unknown command. Type 'help' for a list of available commands.");  
 }  
 }  
 }  
}

Класът *CLI* представлява интерфейса на командния ред. В него се дефинират всички възможни команди и се изпълняват по заявка на потребителя.

Статичното поле *commands* създава карта, която съдържа всяка операция като ключ и съответната й команда като стойност.

Методът *start()* е главният метод на класа, който се използва за стартиране на интерфейса на командния ред. Той използва безкраен цикъл, който чете вход от потребителя и изпълнява съответната команда. Ако операцията не е разпозната, се извежда съобщение за грешка.

# Клас Application

public class Application {  
 public static void main(String[] args) {  
 CLI.*start*();  
 }  
}

Класът *Application* представлява входната точка на приложението. При стартиране, се изпълнява методът *main.* В него се извиква статичният метод *start()* от класа *CLI*, който стартира интерфейса на командния ред. Така програмата започва своето изпълнение и чака входни команди от потребителя.

# Клас Exit

public class Exit extends DefaultCommand {  
  
 public Exit() {  
 }  
  
 @Override  
 public void execute(List<String> arguments) {  
 System.*out*.println("Exiting...");  
 System.*exit*(1);  
 }  
}

Класът *Exit* представлява командата за излизане от програмата.

При изпълнението на метода execute, той извежда съобщение за излизане и след това извиква метода *System.exit(1)*, който прекратява изпълнението на Java виртуалната машина с код за изход 1.

Кодът за изход 0 се използва, за да покаже успешно завършване на програмата.

# Класове за изключения

public class AutomatonException extends RuntimeException{  
 public AutomatonException(String message) {  
 super(message);  
 }  
}

public class AutomatonNotFoundException extends AutomatonException {  
 public AutomatonNotFoundException(String message) {  
 super(message);  
 }  
}

public class InvalidArguments extends AutomatonException {  
 public InvalidArguments(String message) {  
 super(message);  
 }  
}

public class InvalidAutomatonException extends AutomatonException {  
 public InvalidAutomatonException(String message) {  
 super(message);  
 }  
}

public class AlreadyOpenedFileException extends RuntimeException{  
 public AlreadyOpenedFileException(String message) {  
 super(message);  
 }  
}

public class CreatingFileException extends AutomatonException {  
 public CreatingFileException(String message) {  
 super(message);  
 }  
}

public class NoOpenFileException extends AutomatonException {  
 public NoOpenFileException(String message) {  
 super(message);  
 }  
}

public class ErrorOpeningException extends AutomatonException {  
 public ErrorOpeningException(String message) {  
 super(message);  
 }  
}

public class WritingToFIleException extends AutomatonException {  
 public WritingToFIleException(String message) {  
 super(message);  
 }  
}

*AutomatonException* е базов клас за изключенията, свързани с автоматите. Наследява се от всички останали.

*AutomatonNotFoundException* се хвърля когато не е намерен търсеният автомат.

*InvalidArguments* се хвърля когато подадените аргументи за изпълнение на команда са невалидни или липсват.

*InvalidAutomatonException* се хвърля когато автоматът е невалиден или не отговаря на изискванията.

*AlreadyOpenedFileException* се хвърля когато вече е отворен файл и се опитваме да отворим нов.

*CreatingFileException* - грешка при създаването на файл.

*ErrorOpeningException -* грешка при отварянето на файл.

*NoOpenFileException* се хвърля когато не е отворен файл, но се опитваме да извършим операции с него.

# 4.1. Тестване

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

# Глава 5. Заключение

## 5.1. Обобщение на изпълнението на началните цели

Изпълнени са изискванията за функционалностие на автоматите. Предвидени са конкретни грешки, които да се обработват по подходящ начин и да не прекъсват изпълнението на програмата. Възможно е да има неточности при изпълнението на някои команди, поради сложността на проекта и реализацията на автомати. Главните проблеми идват от класа Reg, където преобразуването от израз се случва сложно. Останалите функционалности са разработени според направено проучване и знания от дисциплината „ Дискретни структури“.

## 5.2. Насоки за бъдещо развитие и усъвършенстване

Би се наложило по-подробно проучване поради обширността на темата. Гъвкавостта при използването на автомати е голяма и беше трудно да се реализира всичко в проекта. Трябва да се разучат и анализират пропуските и евентуално да се оптимизират решенията на дадените задачи.

Използвана литература:

<https://bg.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D0%B5%D0%BD_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82>

<https://stackoverflow.com/questions/15297863/combining-multiple-regular-expressions-into-one-automaton>

<https://www.baeldung.com/java-finite-automata>

Връзка към GitHub: https://github.com/polinakryst3va/Project\_OOP1