Turingmaschine

Ziel dieser Aufgabe ist es, eine Turingmaschine (https://de.wikipedia.org/wiki/Turingmaschine) zu erstellen. Hierzu wird zunächst im ersten Teil ein abstrakter endlicher Automat (https://de.wikipedia.org/wiki/Endlicher_Automat) definiert, der als Grundlage für die Turingmaschine in Teil 2 fungieren wird. In Teil 3 wird dann eine Klasse erstellt, die eine Turingmaschine aus einem Datenstrom (z.B. einer Datei) lesen kann. Mithilfe der mitgelieferten graphischen Oberfläche kann die Implementierung getestet werden.
Die benötigten theoretischen Grundlagen (endlicher Automat, Turingmaschine...) werden jeweils an der entsprechenden Stelle kurz erläutert.

Hinweis: Zeilenumbrüche in `toString` -Methoden sind generell mit "\n" zu erzeugen (bzw. bei Verwendung von Streams mit `println(String)`).

Teil 1: Endlicher Automat (10 Punkte)

In diesem Abschnitt wird zunächst ein abstrakter, endlicher Automat erstellt. Anschließend erfolgt eine beispielhafte Implementierung eines konkreten, endlichen Automaten: einer Mealy-Maschine.

Ein endlicher Automat dient dazu, ein zustandsbasiertes System zu modellieren. Ein solches System befindet sich immer in einem genau identifizierbaren und beschreibbaren Zustand, und kann durch Ereignisse von außen (Eingabe) dazu gebracht werden, den Zustand zu wechseln. Je nach System erfolgt beim Zustandswechsel eine Ausgabe. Im Gegensatz zu anderen Rechenmaschinen besitzt ein endlicher Automat keinen Speicher. Damit sind nicht alle Probleme, die Computer lösen können auch auf endlichen Automaten lösbar.

Ein Automat besteht also formal aus

- einer Menge von genau definierten Zuständen
- einem Eingabealphabet
- einem Ausgabealphabet
- einer Übergangsfunktion, die ausgehend von einem Zustand und einer Eingabe die Ausgabe und den Folgezustand berechnet.

Damit der Automat irgendwo starten kann, wird ein einzelner Zustand als sogenannter Startzustand ausgezeichnet. Wird ein Automat dazu eingesetzt, eine Ja/Nein-Frage zu beantworten (z.B. "ist die Eingabe das Muster 'aba'?"), gibt es eine Menge sogenannter akzeptierter Zustände, in welchen sich der Automat nach vollständiger Eingabe befindet, wenn die Frage mit Ja beantwortet wird.

Wir erstellen nun einen zunächst abstrakten, endlichen Automaten in Java. Hierbei soll das Zustands-Entwurfsmuster (http://de.wikipedia.org/wiki/Zustand_%28Entwurfsmuster%29) verwendet werden, welches vorschreibt, dass das Verhalten eines Automaten in Abhängigkeit seiner Zustände definiert wird. Wir erstellen 3 Klassen:

- Eine abstrakte Klasse `AbstractFiniteStateMachine<I,O>` , die den abstrakten Automaten darstellt, der einen Verweis auf seinen aktuellen Zustand und eine Aufzählung aller möglichen Zustände enthält. Eine konkrete Implementierung dieser Klasse wird später noch die Aufgabe haben, eine Ausgabe, die bei einem Zustandsübergang zurückgegeben wird, zu verarbeiten.
- Eine Klasse `State<I, O>` , die einen konkreten, an einen festen Automaten gebundenen Zustand darstellt. Sie verwaltet die möglichen Zustandsübergänge und setzt im Falle eines Übergangs den aktuellen Zustand im zugehörigen Automaten neu und liefert die zugehörige Ausgabe zurück.
- Eine Klasse `Transition<O>` , welche die Informationen zu einem konkreten Übergang speichert.

Da wir bisher noch nicht wissen, welche Art von Ein- und Ausgaben unser Problem benötigt, werden an dieser Stelle die generische Typen (http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_07_001.htm) I, O für das Eingabe- bzw. Ausgabealphabet des Automaten benutzt.

Erstellen Sie diese Klassen mit mindestens den im Folgenden beschriebenen Methoden.

1.1 Die Klasse `AbstractFiniteStateMachine<I, O>`

Erstellen Sie die Klasse im Paket `finiteStateMachine` .

- `public void addState(String state)`
  Legt in der Maschine einen neuen Zustand mit dem Namen *state* an.Die Zustände innerhalb eines Automaten sollen durch ihren Namen eindeutig identifizierbar (und auch ansprechbar) sein. Achten Sie hierbei auch auf eine effiziente Speicherung der Zustände, vor allem soll der Zugriff auf Transitionen effizient sein.
  Ist bereits ein Zustand mit dem gewünschten Namen vorhanden, soll eine `IllegalStateException` mit aussagekräftiger Fehlermeldung geworfen werden.
- `public State<I,O> getState(String state)`
  Liefert den Zustand mit dem Namen *state* zurück. Ist kein Zustand mit dem angegebenen Namen vorhanden, soll eine `IllegalStateException` geworfen werden.
- `public void addTransition(String startState, I input, String targetState, O output)`
  Fügt einen Übergang von *startState* nach *targetState* bei Eingabe input mit der Ausgabe *output* hinzu. Der Übergang selbst soll im Zustand mit dem Namen *startState* gespeichert werden.
  Ist einer der beiden Zustände *startState* oder *targetState* nicht vorhanden, soll eine `IllegalStateException` geworfen werden.
- `public void setCurrentState(String state)`
  Setzt den aktuellen Status des Automaten auf den Zustand mit dem Namen *state*. Ist kein solcher Zustand vorhanden, soll eine `IllegalStateException` geworfen werden.
- `public State<I,O> getCurrentState()`
  Liefert den aktuellen Zustand zurück.
- `public boolean isInAcceptedState()`
  Gibt an, ob der aktuelle Zustand des Automaten akzeptiert ist. Ist kein aktueller Zustand gesetzt, soll `false` zurückgegeben werden.
- `public void transit(I input)`
  Führt einen Übergang vom aktuellen Zustand mit der Eingabe *input* aus. Der Zustandswechsel selbst wird hierbei in der Zustands-Klasse implementiert. Die zurückgelieferte Ausgabe dieses Zustandsübergangs wird dann mit der Methode `processOutput(O)` verarbeitet. Die Verarbeitung der Ausgabe hängt von dem konkreten Automatentyp ab. Dieser ist in der abstrakten Klasse natürlich nicht bekannt. Daher wird hier entsprechend des Schablonen-Entwurfsmusters (http://de.wikipedia.org/wiki/Template_Method) entsprechend lediglich die abstrakte Methode aufgerufen.
  Ist kein Übergang für die Eingabe definiert, soll eine `IllegalArgumentException` mit aussagekräftiger Meldung geworfen werden.
  Ist kein Zustand als aktuell gekennzeichnet, soll eine `IllegalStateException` geworfen werden.
- `public Transition<O> getTransition(I input)`
  Liefert den Übergang zurück, der bei aktuellem Zustand mit der Eingabe *input* ausgeführt werden würde. Ist kein Übergang für die Eingabe definiert, soll `null` zurückgeliefert werden.
  Ist kein aktueller Zustand gesetzt, soll eine `IllegalStateException` geworfen werden.

- `protected abstract void processOutput(O output)`
  Dies ist die einzige abstrakte Methode in dieser Klasse. Bei konkreten Implementierungen eines Automatentyps wird hier festgelegt, wie mit der Ausgabe bei einem Übergang verfahren werden soll.

1.2 Die Klasse `State<I, O>`

Erstellen Sie die Klasse im Paket `finiteStateMachine.state`.

- `public void setAccepted(boolean accepted)`
  `public boolean isAccepted()`
  Diese beiden Methoden kennzeichnen einen Zustand als akzeptiert, bzw. zeigen an, ob ein Zustand akzeptiert ist.
- `public void addTransition(I input, String targetState, O output)`
  Fügt eine Transition zum Zustand hinzu, die bei der Eingabe *input* ausgeführt werden soll. Ist bereits ein Übergang für die Eingabe festgelegt, wird dieser überschrieben.
- `public Transition<O> getTransition(I input)`
  Liefert die Transition, die bei der Eingabe *input* ausgeführt wird. Ist keine Transition für die Eingabe definiert, soll `null` zurückgeliefert werden.
- `public O transit(I input)`
  Diese Methode führt den Zustandsübergang bei Eingabe von *input* aus. Hierzu wird in der zum Zustand gehörigen Maschine der aktuelle Zustand auf den entsprechenden Nachfolgezustand gesetzt und die entsprechende Ausgabe zurückgeliefert. Ist kein Übergang für die übergebene Eingabe definiert, soll eine `IllegalArgumentException` mit aussagekräftiger Meldung geworfen werden.
- `public AbstractFiniteStateMachine<I, O> getMachine()`
  `public String getName()`
  Diese Getter liefern die zum Zustand gehörige Maschine und den Namen des Zustands.
- `public String toString()`
  Die Methode soll den Namen des Zustandes liefern.

Ein Zustand soll durch den Konstruktor `public State(AbstractFiniteStateMachine<I, O> machine, String name)` erstellt werden.

1.3 Die Klasse `Transition<O>`

Erstellen Sie die Klasse im Paket `finiteStateMachine.state`.

- `public String getNextState()`
  `public O getOutput()`
  Diese Getter liefern den Nachfolgezustand und die zugehörige Ausgabe.
- `public String toString()`
  Liefert einen String folgender Form zurück:

  ```
  <Name des Nachfolgezustandes>,<toString() der Ausgabe>
  ```

Eine Transition soll mit dem Konstruktor `public Transition(String targetState, O output)` erstellt werden können.

1.4 Die Klasse `MealyMachine<I, O>`

Erstellen Sie die Klasse im Paket `finiteStateMachine.mealy`.

Nachdem wir nun einen abstrakten Automaten definiert haben, erstellen wir auf dieser Basis einen konkreten Automaten, den Mealy-Automaten (https://de.wikipedia.org/wiki/Mealy-Automat). Dieser liefert zu jedem Zustandsübergang eine zugehörige Ausgabe. Unser Mealy-Automat soll diese Ausgabe einfach in die Konsole schreiben.

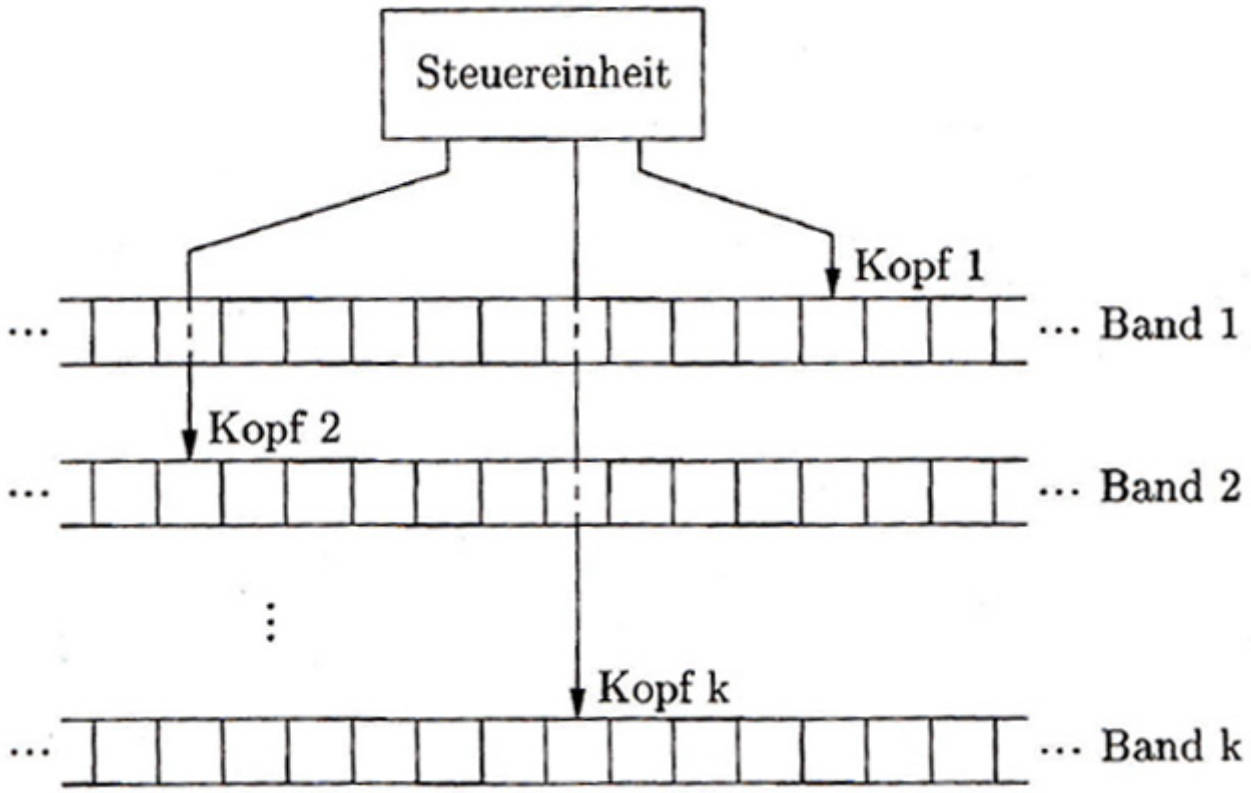
Erstellen Sie eine Klasse `MealyMachine<I, O>`, welche die Klasse `AbstractFiniteStateMachine<I, O>` erweitert und die Ausgaben ohne Zwischenzeichen wie Leerzeichen oder Zeilenumbruch auf die Standardausgabe schreibt.

Demonstrieren Sie die Funktionstüchtigkeit ihrer Klasse anhand eines Beispiels in der `main()`-Methode der Mealy-Klasse. Als Ein- und Ausgabetyp könnte z.B. `Character` gewählt werden.

Teil 2: Turingmaschine (59 Punkte)

Nachdem in Teil 1 ein abstrakter endlicher Automat definiert wurde, widmet sich der zweite Teil der Erstellung einer Turingmaschine.

Eine Turingmaschine ist ein endlicher Automat, der seine Eingaben von sogenannten Bändern liest und seine Ausgaben auch wieder auf selbige schreibt. Diese Bänder sind (theoretisch) unendlich lange Aneinanderreihungen von Speicherzellen und es gibt eine fest gelegte Anzahl von ihnen. Jedes dieser Bänder besitzt genau einen Lese-/Schreibkopf, der pro Zustandsübergang im Automaten maximal um eine Stelle nach links oder rechts bewegt werden kann und genau von einer Speicherzelle liest bzw. auf sie schreibt. Zustandsübergänge hängen vom aktuellen Zustand der Turingmaschine und von allen Daten in den Speicherzellen unter den Lese-/Schreibköpfen der Bänder ab. Die gelesenen Daten entsprechen also der Eingabe in einem Automaten. Ein Übergang erfolgt natürlich zu einem Nachfolgezustand und die Ausgabe setzt sich zusammen aus den auf die Bänder zu schreibenden Daten und den Richtungen, in die die einzelnen Köpfe bewegt werden sollen.



Erstellen Sie folgende Klassen mit mindestens den angegebenen Methoden.

2.1 Die Klasse `TuringMachine<T>`

Erstellen Sie die Klasse im Paket `turingMachine`.

Die Klasse `TuringMachine<T>` erweitert die Klasse `AbstractFiniteStateMachine<MultiTapeReadWriteData<T>, TuringTransitionOutput<T>>`.
Sie hat nur einen generischen Parameter, denn Eingabe- und Ausgabealphabet müssen gleich sein, da von den selben Bändern gelesen wird, auf die geschrieben wird.

- `public void run()`
  Diese Methode führt so lange Zustandsübergänge aus, bis die Maschine einen akzeptierten Zustand erreicht hat oder der Fall eintritt, dass kein Zustandsübergang definiert ist.
- `public void transit()`
  Führt basierend auf den aktuell von den Bändern gelesenen Daten (und natürlich dem aktuellen Zustand) einen Zustandsübergang aus. Befindet sich die Maschine bereits in einem akzeptierten Zustand, soll eine `IllegalStateException` mit aussagekräftigem Text geworfen werden.
- `public int getTapeCount()`
  Liefert die Anzahl der Bänder der Maschine.
- `public MultiTape<T> getTapes()`
  Liefert die Bänder der Maschine.
- `public void processOutput(TuringTransitionOutput<T> output)`
  In dieser überschriebenen Methode werden zunächst die neuen Daten auf die Bänder geschrieben und anschließend die Lese-/Schreibköpfe in die entsprechenden Richtungen bewegt.

Eine Turingmaschine soll durch den Konstruktor `public TuringMachine(int tapeCount)` mit einer vorgegebenen Anzahl von Bändern erstellt werden können.

2.2 Die Enumeration `Direction`

Die im Paket `turingMachine.tape` zu erstellende Enumeration stellt drei Konstanten zur Verfügung, die die Möglichkeiten zur Bewegungsrichtung des Schreib-/Lesekopfes darstellen:
`LEFT`, `RIGHT`, `NON`

2.3 Das Interface `TapeChangeListener<T>`

Erstellen Sie das Interface im Paket `turingMachine.tape`.

Dieses Interface stellt das Gerüst für einen Listener dar, der die Änderung von `Tape`-Objekten verfolgt. Für die Aufgabe genügt es, nur das Interface zu erstellen, die Implementierung ist im mitgelieferten Teil der graphischen Oberfläche enthalten.

- `public void onMove(Direction direction)`
  Wird aufgerufen, nachdem das Band bewegt wurde (auch bei `Direction.NON`)
- `public void onExpand(Direction direction)`
  Wird aufgerufen, nachdem eine Zelle zum ersten Mal vom Lesekopf besucht wurde. Tritt dies in Kombination mit einer Bandbewegung auf, ist `onMove(Direction)` nach dieser Methode aufzurufen.
- `public void onWrite(T value)`
  Wird aufgerufen, wann immer auf das Band geschrieben wurde.

2.4 Die Klasse `Tape<T>`

Erstellen Sie die Klasse im Paket `turingMachine.tape`.

Diese Klasse stellt ein einzelnes Band dar. Es ist zu beachten, dass, je nachdem wie der Lese-/Schreibkopf bewegt wird, vorne oder hinten zusätzliche Speicherzellen (ggf. mit leerem Inhalt) eingefügt werden müssen. Auch bei der Erstellung sollen, bei einer initialen Kopfposition außerhalb der ursprünglichen Werte, neue Zellen

hinzugefügt werden.

- `public void move(Direction direction)`
  Bewegt den Kopf in die angegebene Richtung.
- `public T read()`
  Gibt das aktuelle Element unter dem Kopf zurück. Steht der Kopf über einer leeren Stelle, soll `null` zurückgegeben werden.
- `public List<T> getContents()`
  Gibt den Inhalt des Bandes inklusive leerer Zellen an Anfang und Ende (alle Zellen auf denen der Kopf schon einmal war) zurück. Die Liste soll nicht mehr modifizierbar sein.
- `public int getPosition()`
  Gibt die Position des Kopfes auf dem Band zurück (entspricht der Position in der von `getContents()` zurückgegebenen Liste).
- `public void write(T content)`
  Schreibt den Inhalt von *content* an die aktuelle Stelle des Kopfes.
- `public void addListener(TapeChangeListener<T> listener)`
  Fügt einen neuen `TapeChangeListener` zu den zu informierenden Listenern hinzu. Auf diese Art hinzugefügte Listener müssen über Bandoperationen entsprechend der Interfacevereinbarung informiert werden.
- `public String getCurrent()`
  Gibt das aktuelle Element unter dem Kopf als String zurück (`toString()` -Methode)). Befindet sich unter dem Kopf eine leere Zelle, soll ein Unterstrich "_" zurückgegeben werden.
- `public String getLeft()`
  Gibt den Bandinhalt links von der aktuellen Position als String (verknüpfte `toString()` -Methoden der Speicherzelleninhalte) zurück. Führende, leere Zellen werden abgeschnitten. Ist der linke Teil des Bandes leer, soll also ein leerer String zurückgegeben werden.
- `public String getRight()`
  Gibt den Bandinhalt rechts von der aktuellen Position als String (verknüpfte `toString()` -Methoden der Speicherzelleninhalte) zurück. Abschließende leere Zellen werden abgeschnitten. Ist der rechte Teil des Bandes leer, soll also ein leerer String zurückgegeben werden.
- `public String toString()`
  Gibt den Inhalt des kompletten Bandes als zweizeiligen String an. Die erste Zeile ist der Inhalt des Bandes (verknüfte `toString()` -Methoden der Speicherzelleninhalte) ohne führende und abschließende Leerzellen (maximal wird bis zu der Zelle abgeschnitten, auf der gerade der Kopf steht). In der zweiten Zeile soll sich das Zeichen „^“ unter der Zelle befinden, auf der sich der Kopf befindet. Vor dem „^“ soll mit Leerzeichen aufgefüllt werden.
  Beispiel:

  ```
  1111_1_11111
       ^
  ```

  Befindet sich bei den String-Ausgaben eine leere Zelle zwischen zwei beschriebenen Zellen (also nicht abgeschnittene leere Zellen), so soll diese leere Zelle durch einen Unterstrich „_“ dargestellt werden (zu sehen im Beispiel).

Ein leeres Band wird durch den Konstruktor `public Tape()` erstellt.

2.5 Die Klasse `MultiTape<T>`

Erstellen Sie die Klasse im Paket `turingMachine.tape` .

Diese Klasse stellt eine Menge aus mehreren einzelnen Bändern dar.

- `public MultiTapeReadWriteData<T> read()`
  Liest die Werte unter allen Köpfen der einzelnen Bänder und gibt diese zurück.
- `public void write(MultiTapeReadWriteData<T> values)`
  Schreibt die Werte in *values* auf die einzelnen Bänder. Befinden sich in *values* zu wenige oder zu viele Werte, soll eine `IllegalArgumentException` geworfen werden.
- `public void move(Direction[] directions)`
  Bewegt die Lese-/Schreibköpfe der einzelnen Bänder. Befinden sich in *directions* zu wenige oder zu viele Richtungen, soll eine `IllegalArgumentException` geworfen werden.
- `public List<Tape<T>> getTapes()`
  Liefert eine Liste der Bänder.
- `public int getTapeCount()`
  Gibt die Anzahl der Bänder zurück.
- `public String toString()`
  Gibt eine durch Zeilenumbruch getrennte `toString()` -Darstellung der einzelnen Bänder zurück. Endet mit einem Zeilenumbruch.

Ein Multiband wird durch den Konstruktor `public MultiTape(int tapeCount)` mit *tapeCount* leeren Bändern erstellt.

2.6 Die Klasse `MultiTapeReadWriteData<T>`

Erstellen Sie die Klasse im Paket `turingMachine.tape` .

`MultiTapeReadWriteData<T>` repräsentiert die Daten, die von einem `MultiTape<T>` (genauer: den Köpfen der einzelnen Bänder) gelesen wurden, oder auf dieses geschrieben werden sollen.

- `public T get(int i)`
  Liefert den Wert, der für den Inhalt der Zelle unter dem Lese-/Schreibkopf des *i*-ten Bandes steht.
- `public void set(int i, T value)`
  Setzt den Wert, der für den Inhalt der Zelle unter dem Lese-/Schreibkopf des *i*-ten Bandes steht.
- `public int getLength()`
  Anzahl der Werte (entspricht der Anzahl der repräsentierten Bänder-Köpfe).
- `public String toString()`
  Gibt die einzelnen Werte als String (verknüfte `toString()` -Methoden) zurück. `null` -Werte sollen hierbei als Unterstrich ("_") dargestellt werden.

Die Indizierung erfolgt 0-basiert. Wird irgendwo versucht auf eine Zelle mit einem Index außerhalb des zulässigen Bereiches zuzugreifen, soll eine `IndexOutOfBoundsException` geworfen werden.

Der Konstruktor `public MultiTapeReadWriteData(int length)` erstellt ein Objekt mit *length* Werten.
Da zwei Objekte vom Typ `MultiTapeReadWriteData<T>` vergleichbar sein sollen, ist es notwendig die `equals()`– und die `hashCode()`–Methode dieser Klasse zu überschreiben.
Zwei Objekte dieses Typs sollen genau dann gleich sein, wenn die Werte auf den einzelnen Bändern paarweise gleich sind.

2.7 Die Klasse `TuringTransitionOutput<T>`

Erstellen Sie die Klasse im Paket `turingMachine` .

Die Ausgabe einer Turingmaschine setzt sich zusammen aus den Werten, die auf die einzelnen Bänder geschrieben werden (`MultiTapeReadWriteData<T>`) und den Richtungen, in die die einzelnen Lese-/Schreibköpfe der Bänder bewegt werden sollen (`Direction[]`).

- `public MultiTapeReadWriteData<T> getToWrite()`
  Liefert die zu schreibenden Daten.
- `public Direction[] getDirections()`
  Gibt die Bewegungsrichtungen für die einzelnen Köpfe an.

Eine Transitionsausgabe soll durch den Konstruktor `public TuringTransitionOutput(MultiTapeReadWriteData<T> toWrite, Direction... directions)` erstellt werden können. Stimmt die Anzahl der Werte nicht mit der der Richtungen überein, so soll eine `IllegalArgumentException` geworfen werden.

2.8 Demonstration:

Erstellen Sie in der Klasse `TuringMachine<T>` eine `main()` -Methode, die ein Beispiel enthält, das die Funktionstüchtigkeit ihrer Klassen zeigt.

Teil 3: In-/Output (31 Punkte)

In Teil 3 soll die Definition einer Turingmaschine mit einem `Character` -Alphabet von einem Eingabestrom gelesen und daraus ein `TuringMachine` -Objekt erstellt werden.

Die Definition einer Maschine sieht folgendermaßen aus:

```
<Anzahl der Bänder>
<Vorinitialisierung Band 1>
<Vorinitialisierung Band 2>
…
<Komma-separierte Zustandsliste>
<Startzustand>
<Komma-separierte Liste der akzeptierten Zustände>
<Transition 1>
<Transition 2>
…
```

Hierbei haben die Vorinitialisierungen für ein Band das Format

```
<Position des Lesekopfes (startend bei 1)>:<Bandinhalt ab Position 1>
```

und eine Transition hat das Format

```
<Name des Anfangszustands>,<Inhalt der einzelnen Bänder> -> <Name des Zielzustands>,<Werte, die
auf die einzelnen Bänder zu schreiben sind>,<Richtungen für die einzelnen Köpfe: R für Rechts;
L für Links, N für keine Bewegung>
```

Zeilen, die mit einer Raute beginnen, und vollständig leere Zeilen sollen ignoriert werden.
Unterstriche ("_") stehen für leere Speicherzellen.

Beispiel:

```
# Turingmaschinendefinition fuer Character-Baender.
# Leerzeilen und Zeilen mit # beginnend werden ignoriert

# Diese Machine verdoppelt die Einsen auf dem Band
```

```
# Anzahl der Baender
1
# Vorinitalsierung der Baender
1:11111
# Zustaende der Maschine
s1,s2,s3,s4,s5,s6
# Startzustand
s1
# akzeptierte Zustaende
s6
# Transitionen
s1,1 -> s2,_,R
s1,_ -> s6,_,N
s2,1 -> s2,1,R
s2,_ -> s3,_,R
s3,_ -> s4,1,L
s3,1 -> s3,1,R
s4,1 -> s4,1,L
s4,_ -> s5,_,L
s5,1 -> s5,1,L
s5,_ -> s1,1,R
s6,_ -> s6,_,N
```

Erstellen Sie folgende Klasse:

3.1 Die Klasse `TuringMachineReader`

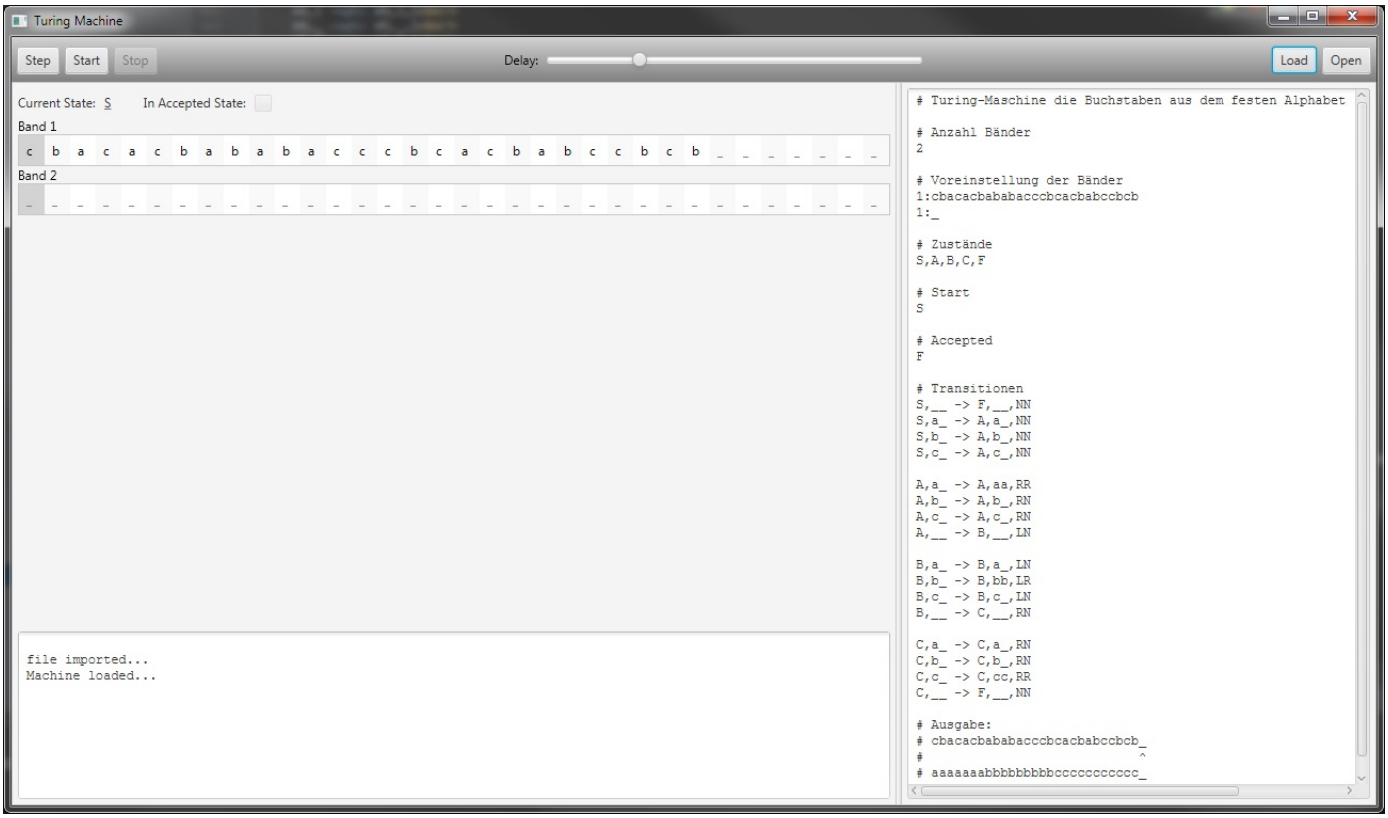
Erstellen Sie die Klasse im Paket `turingMachine.io` .

- `public static TuringMachine<Character> read(InputStream input)`
  Diese Methode liest eine Turingmaschine aus einem `InputStream` und erstellt dazu ein `TuringMachine<Character>` -Objekt.
  Erfolgt eine ungültige Eingabe, soll eine `IllegalArgumentException` mit sinnvoller Fehlermeldung geworfen werden.
- `public static TuringMachine<Character> read(Reader input)`
  Diese Methode liest eine Turingmaschine aus einem `Reader` und erstellt dazu ein `TuringMachine<Character>` -Objekt.
  Erfolgt eine ungültige Eingabe, soll eine `IllegalArgumentException` mit sinnvoller Fehlermeldung geworfen werden.

Zeigen Sie die Funktionstüchtigkeit beider Methoden dieser Klasse an einem Beispiel in der `main()` -Methode.

Graphische Oberfläche

In das Projekt ist eine graphische Oberfläche eingebunden, mit der die Funktionsfähigkeit des Codes zusätzlich geprüft werden kann. Sie kann über die `main()` -Methode der Klasse `turingMachine.Main` gestartet werden.



Viel Erfolg!
