ALAN

EIN TURINGMASCHINEN-INTERPRETER

SEP 97

Entwurf
&
Spezifikation

Gerald Grote
Heiner Grote
Björn Heggemann
Stefan Milius

Gruppe 1

9. Juli 1997

Inhaltsverzeichnis

1 Einleitung

Das vorliegende Dokument beschreibt den Entwurf des TM-Interpreters.

Zunächst werden die Klassen des Systems und die Vererbungshierarchie mit der standardisierten OMT-Methode dargestellt. In diesem Diagramm sind auch die Assoziationen zwischen den Klassen dargestellt, die nicht voneinander abgeleitet sind.

Der größte Abschnitt in diesem Dokument behandelt die Vorstellung der einzelnen Klassen. In diesem Abschnitt werden die Bedeutung der Attribute und die Aufgaben der Methoden erläutert.

Das zweite Diagramm zeigt ein dynamisches Modell des Systems, das die Interaktion mit dem Benutzer berücksichtigt. Da es sich um ein System handelt, bei dem durch die Benutzerschnittstelle zu jeder Zeit verschiedene Ereignisse auftreten können, die die Simulation steuern, beschränkt sich das dynamische Modell auf einen typischen Ablauf.

Abschließend wird die Klasse `Parser` spezifiziert. Es werden die wesentlichen privaten Methoden und Attribute näher erklärt. Außerdem wird Pseudocode für sämtliche Methoden angegeben.

2 OMT-Modell

Die Abbildung 1 zeigt den Entwurf des Systems in OMT-Darstellung. Im folgenden wird ein Überblick über die dargestellten Klassen und ihren Assoziationen gegeben.

2.1 Die Klassen

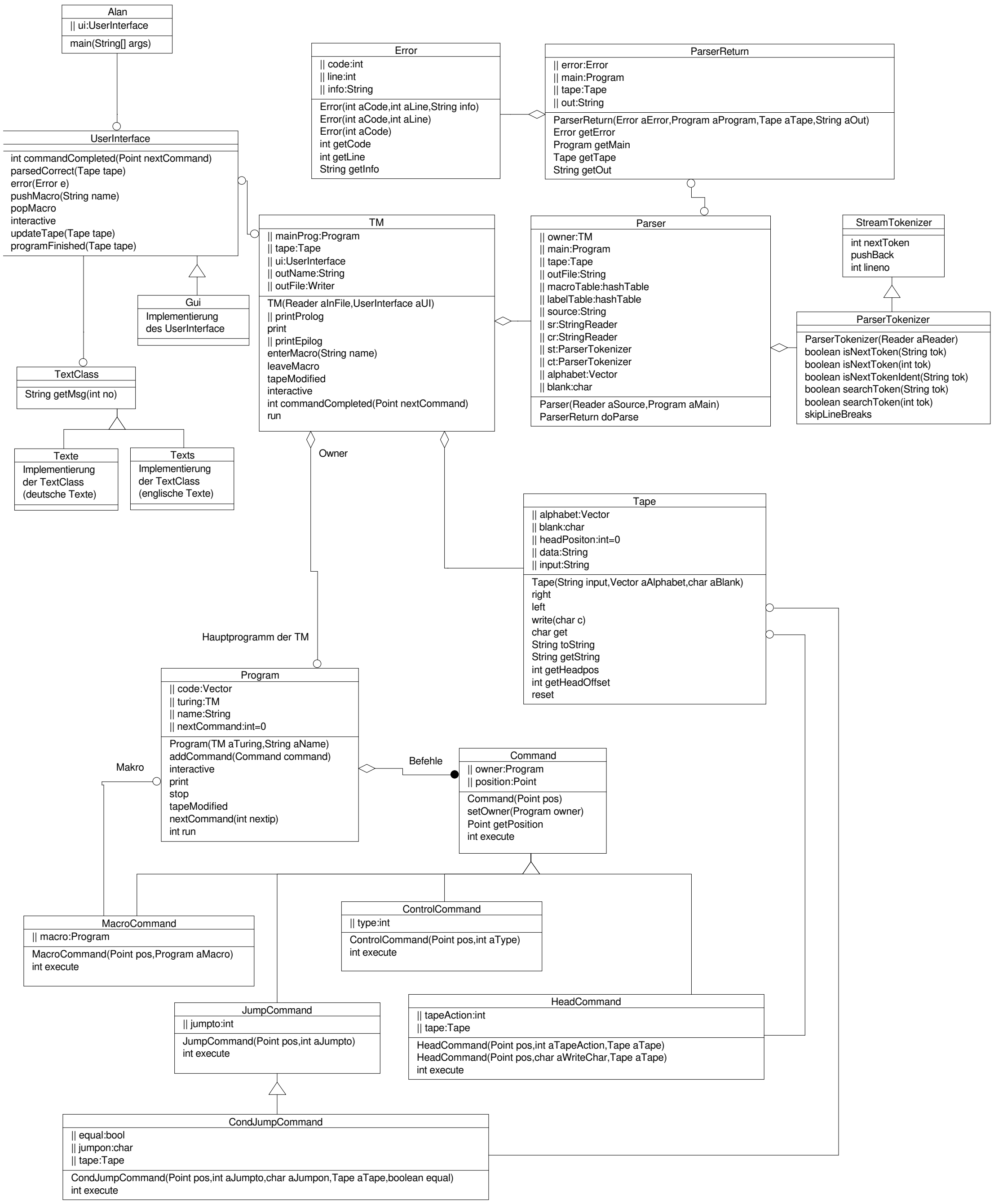
Die Klasse `TM` bildet den Kern des Systems. Sie enthält die Daten der Turingmaschine, also das Band, das Hauptprogramm, die Ausgabedatei und weitere wichtige Referenzen. Die Klasse `Tape` realisiert das Band und stellt alle Methoden zu dessen Bearbeitung zur Verfügung.

Außerdem instantiiert die Klasse `TM` ein Parser-Objekt, das die Übersetzung des Programmtextes in eine interne Datenstruktur übernimmt. Die interne Repräsentation des Turing-Programms wird durch die Klasse `Program` und die von `Command` ausgehende Klassenhierarchie realisiert. Jedes Programm (Hauptprogramm sowie Makros) enthält einen Vector von `Command`-Objekten. Das ist ein Objekt der Klasse Vector, die das Konzept einer verketteten Liste mit dem eines Arrays verbindet. Sie kann also beliebig groß werden. Von der `Command`-Klasse abgeleitet sind die einzelnen Befehlstypen, die in einem Turingprogramm auftreten können. Die Klasse `HeadCommand` ist z.B. für Befehle zuständig, die den Bandinhalt ändern oder den Kopf bewegen.

Um die Simulation der Turingmaschine zu starten, wird die `run`-Methode ihres Hauptprogramms aufrufen. In dieser Methode werden die `execute`-Methoden der `Command`-Objekte aufgerufen.

Von der Datenstruktur der Turingmaschine abgekoppelt ist die Benutzerschnittstelle. Jede Benutzerschnittstelle, die den Ablauf der Turingmaschine darstellen und dem Benutzer Möglichkeit zur Ablaufsteuerung geben will, muß die Schnittstelle `UserInterface` implementieren. Diese Schnittstelle definiert alle Methoden, über die die Turingmaschine der Benutzeroberfläche ihren Status mitteilt.

Das Programm wird gestartet, indem die `main`-Methode der Klasse `Alan` aufgerufen wird. Dort werden die Kommandozeilenparameter ausgewertet und eine Benutzerschnittstelle erzeugt, die die weitere Steuerung übernimmt.



"OBJECT DOMAIN" UNREGISTERED EVALUATION COPY!!!!Alan

Abbildung 1: Das OMT-Modell des Systems

2.2 Die Assoziationen

Im OMT-Modell sind auch die Verbindungen zwischen den Klassen dargestellt. Man sieht z.B., daß Objekte der `TM`-Klasse jeweils ein Objekt der `Program`-Klasse enthalten, nämlich das Hauptprogramm der Turingmaschine. Umgekehrt hat jedes `Program`-Objekt eine Referenz auf ihr `TM`-Objekt. Die `Program`-Objekte enthalten ihrerseits 0 oder mehr Objekte der `Command`-Klasse (die Befehle, die in dem Programm ausgeführt werden sollen).

Da die Befehle vom Typ `CondJumpCommand` und `HeadCommand` Zugriff auf das Band der Turingmaschine benötigen, haben sie eine Assoziation mit diesem. Ebenso ist die `MacroCommand`-Klasse mit der `Program`-Klasse verbunden, damit ein derartiger Befehl die Ausführung des entsprechenden Makros anstoßen kann.

3 Anforderungen an die Programmiersprache

In diesem Entwurf werden einige Datenstrukturen vorausgesetzt, die nicht tiefergehend erläutert und spezifiziert werden (z.B. Hashtabellen oder Tokenizer). Diese Funtionalität muß die Programmiersprache zur Verfügung stellen, in der das System implementiert wird. Ist dies nicht der Fall, müssen die genannten Klassen in der verwendeten Sprache erst noch implementiert werden.

Folgende Datenstrukturen werden vorausgesetzt:

3.1 Vector

Ein Vector ist eine dynamische Datenstruktur, auf die indiziert zugegriffen werden kann. Einerseits ist die Größe eines Vectors veränderlich, d.h. man muß sich nicht im voraus auf eine maximale Feldgröße festlegen, sondern kann beliebig viele Elemente zur Laufzeit hinzufügen. Andererseits ist jedes Element über einen Index erreichbar.

Ein Vector besitzt mindestens folgende Funktionen:

1. Hinzufügen eines Elementes. Hinzugefügte Elemente werden am Ende des Vectors eingefügt und fortlaufend indiziert.
   In diesem Entwurf verwendeter Methodenname: `addElement(data)`

2. Beliebiger Zugriff auf ein Element über dessen Index
   Methodenname: `elementAt(int)`
   Gibt das Element an der gegebenen Position zurück.

3. Entfernen eines beliebigen Elements innerhalb des Vectors. Nachfolgende Elemente rücken dabei automatisch einen Platz nach vorne.
   Methodenname: `remove(int)`

4. Abfragen der Größe des Vektors.
   Methodenname: `size()`
   Gibt die Größe des Vectors zurück.

3.2 Hashtabelle

Eine Hashtabelle ist eine Datenstruktur, in der Elemente sehr effizient gespeichert und wieder gelesen werden. Sie stellt folgende Funktionen bereit:

1. Einfügen eines Elementes an einer Position, die aus einem gegebenen Schlüssel berechnet wird.
   Methodenname: `put(key, data)`

2. Zugriff auf ein Element anhand eines gegebenen Schlüssels.
   Methodenname: `data get(key)`
   Gibt das Element zurück, dessen Schlüssel `key` ist.

3. Abfrage, ob ein Element in der Hashtabelle steht oder nicht.
   Methodenname: `boolean contains(key)`
   Gibt `true` zurück, wenn das Element in der Tabelle steht.

3.3 Reader/Writer

Ein Reader/Writer ist ein Eingabe- bzw. Ausgabestrom, der sequentielles Lesen bzw. Schreiben von Zeichen erlaubt. Die Methode `write(int)` schreibt in einen Writer, mit `read()` liest man Daten aus einem Reader. Ein Aufruf von `close()` schliesst den Eingabe- bzw. Ausgabestrom.

3.4 StreamTokenizer

Der StreamTokenizer setzt auf einen Reader auf. Der Eingabestrom wird in frei definierbare Tokens eingeteilt, die zusammenhängend gelesen werden können. Ein Tokenizer kann beliebige Worte oder Nummern erkennen und Kommentare automatisch ignorieren.

In diesem Entwurf werden folgende Funktionen benötigt:

1. Definition der Tokens und der Zeichen, die ignoriert werden (z.B. Kommentare)

2. Lesen des nächsten Tokens aus dem Eingabestrom.
   Methodenname: `nextToken()`
   Gibt den Typ des Tokens zurück, wenn es sich um ein Wort oder eine Zahl handelt, sonst das Token selbst. Wenn ein Wort oder eine Zahl gelesen wurde, läßt sich der Wert aus dem Attribut `sval` bzw. `nval` des Tokenizers lesen.

3. Zurückstellen von Tokens. Ein zurückgestelltes Token wird bei der nächsten Anfrage an den Tokenizer ein weiteres mal geliefert.
   Methodenname: `pushBack()`

4. Es ist möglich, die Zeilennummer innerhalb des Textes zu ermitteln.
   Methodenname: `int lineno()`

4 Die Klassen des Systems

Soweit nicht anders angegeben, sind alle Attribute der Klassen `privat` deklariert, alle angegebenen Methoden hingegen `public`.

4.1 Die Klasse `Command`

Diese Klasse ist eine abstrakte Klasse, da die Methode `execute` abstrakt ist. Die Klasse wird nicht selbst instantiiert, sondern bildet die Oberklasse aller speziellen `Command`-Klassen, die die einzelnen Befehle der TM-Code-Sprache darstellen. Alle weiteren expliziten `Command`-Klassen werden von dieser Hauptklasse abgeleitet, sie überschreiben lediglich die `execute`-Methode und fügen die Attribute hinzu, die sie zusätzlich benötigen. Ihre Konstruktoren werden nur durch Instanzen der Klasse `Parser` aufgerufen.

4.1.1 Konstruktor

Deklaration `Command(Point pos);`

Dieser Konstruktor weist lediglich dem Attribut `position` die übergebene Postition `pos` zu. Er wird nur durch die Konstruktoren seiner Unterklassen aufgerufen.

4.1.2 Attribute

Deklaration `Point position;`

Im Attribut `position` wird die genaue Position eines jeden Befehls im TM-Quelltext gespeichert. Dieses beinhaltet die Zeilennummer und die Position des Befehls in seiner Zeile (also um den wievielten Befehl in dieser Zeile es sich handelt). Beide Werte sind positive ganze Zahlen.

Deklaration `Program owner;`

Hier wird eine Referenz auf das `Program`-Objekt, zu dem der Befehl gehört, festgehalten. Es kann sich dabei um das Hauptprogramm oder aber auch um ein Makro handeln.

4.1.3 Methoden

Deklaration `int execute();`

Diese Methode ist abstrakt. Sie wird von den abgeleiteten `Command`-Klassen überschrieben.

Deklaration `void setOwner(Program aOwner);`

Mit dieser Methode wird das `Program`-Objekt festgelegt, zu dem der Befehl gehört.

Deklaration `Point getPosition();`

Diese Methode gibt die Position des Befehls im TM-Code, den das `Command`-Objekt darstellt, zurück.

4.2 Die Klasse `JumpCommand`

Die `JumpCommand`-Klasse ist von der `Command`-Klasse abgeleitet und realisiert einfache Sprünge, also Sprünge, die in jedem Fall ausgeführt werden, unabhängig vom aktuellen Zeichen bei der Kopfposition des Turingbandes.

4.2.1 Konstruktor

Deklaration `JumpCommand(Point pos, int aJumpto);`

Der Konstruktor erstellt ein `JumpCommand`-Objekt. Die Attribute werden mit den als Parametern gegebenen Werten initialisiert.

4.2.2 Attribute

Deklaration `int jumpto;`

`jumpto` enthält die genaue Position, zu der gesprungen werden soll. Diese ist ein Index in den `code`-Vektor des `Program`-Objektes, zu dem der Befehl gehört.

4.2.3 Methoden

Deklaration `int execute();`

Diese Methode führt den Sprung aus, indem sie `owner.nextCommand(jumpto)` der `Program`-Klasse aufruft. Die Position der Stelle, zu der gesprungen werden soll, wird übergeben. Der Rückgabewert der Methode selbst ist `prog_OK` (vgl. `Program`-Klasse in 4.7).

4.3 Die Klasse `CondJumpCommand`

Die `CondJumpCommand`-Klasse ist von der `JumpCommand`-Klasse abgeleitet und erweitert diese um bedingte Sprünge.

4.3.1 Konstruktor

Deklaration `CondJumpCommand(Point pos, int aJumpto, char aJumpon, Tape aTape, boolean aEqual);`

Der Konstruktor erstellt ein `CondJumpCommand`-Objekt. Die Attribute, also die Befehlsposition, Sprungposition, Sprungzeichen sowie das aktuelle Band, werden mit den als Parameter übergebenen Werten initialisiert.

4.3.2 Attribute

Deklaration `boolean equal;`

Dieses Flag gibt an, ob auf Gleichheit oder Ungleichheit getestet wird. Wenn sein Wert `true` ist, wird auf Gleichheit getestet, ansonsten auf Ungleichheit.

Deklaration `char jumpon;`

`jumpon` enthält das Zeichen, mit dem das Bandzeichen verglichen wird.

Deklaration `Tape tape;`

`tape` ist eine Referenz auf das Band, von dem eingelesen werden soll.

4.3.3 Methoden

Deklaration `int execute();`

Diese Methode überprüft, ob das Kriterium für einen Sprung erfüllt ist und führt ihn gegebenenfalls aus, indem sie `owner.nextCommand(jumpto)` der `Program`-Klasse aufruft. Die Position der Stelle, zu der gesprungen werden soll, wird dabei übergeben. Ist das Sprungkriterium nicht erfüllt, führt die Methode keine Aktion aus. Der Rückgabewert der Methode selbst ist `prog_OK` (vgl. `Program`-Klasse in 4.7).

4.4 Die Klasse `HeadCommand`

Die `HeadCommand`-Klasse ist von der `Command`-Klasse abgeleitet. Sie realisiert die möglichen Bandoperationen, also das Bewegen des Kopfes nach rechts bzw. links oder das Schreiben eines neuen Symbols. Das Lesen eines Zeichens ist keine explizite Bandoperation und gehört deshalb nicht zu `HeadCommand`.

4.4.1 Konstruktoren

Deklaration `HeadCommand(Point pos, char aWriteChar, tape aTape);`

Der Konstruktor erstellt ein `HeadCommand`-Objekt, das ein gegebenes Zeichen auf das Band schreibt. Die Attribute, also die aktuelle Befehlsposition, das zu schreibende Bandzeichen sowie das Band, auf das sich die Operation bezieht, werden auf die übergebenen Werte gesetzt. Das Attribut `tapeAction` wird so gesetzt, daß das Objekt als schreibendes Kommando identifiziert werden kann.

Deklaration `HeadCommand(Point pos, int aTapeAction, tape aTape);`

Der Konstruktor erstellt ein `HeadCommand`-Objekt, das eine Bewegung des Schreib-/Lesekopfes durchführt. Die Attribute, also die aktuelle Befehlsposition, die Art der Operation sowie das Band, auf das sich die Operation bezieht, werden auf die übergebenen Werte gesetzt. Das Attribut `writeChar` ist bei einem über diesen Konstruktor instantiierten Objekt undefiniert.

4.4.2 Attribute

Deklaration `int tapeAction;`

Anhand dieses Attributes wird entschieden, welche Bandoperationen durchgeführt wird. Die möglichen Operationen sind das Bewegen des Kopfes nach rechts bzw. links oder das Schreiben eines neuen Symbols.

Deklaration `char writeChar;`

`writeChar` ist das zu schreibende Zeichen.

Deklaration `Tape tape;`

`tape` ist eine Referenz auf das Band, das von dem Befehl verändert werden soll.

4.4.3 Methoden

Deklaration `int execute();`

Diese Methode überprüft anhand von `tapeAction`, welche Bandoperation ausgeführt werden soll und handelt dementsprechend. Der Rückgabewert der Methode selbst ist `prog_OK` (vgl. `Program`-Klasse in 4.7).

4.5 Die Klasse `MacroCommand`

Die `HeadCommand`-Klasse ist von der `Command`-Klasse abgeleitet und realisiert den Aufruf eines Makros. Makros sind Objekte der Klasse `Program` und können dementsprechend behandelt werden.

4.5.1 Konstruktor

Deklaration `MacroCommand(Point pos, Program aMacro);`

Der Konstruktor erstellt ein neues `MakroCommand`-Objekt. Die Attribute (Position des Befehls und Referenz auf das aufzurufende Makro) werden mit den als Parameter übergebenen Werten initialisiert.

4.5.2 Attribute

Deklaration `Program macro;`

`macro` ist die Referenz auf das `Program`-Objekt, dessen Ausführung von dem Makroaufruf angestoßen wird.

4.5.3 Methoden

Deklaration `int execute();`

Diese Methode ist für das Ausführen eines Makros zuständig. Dies geschieht, indem die Methode `Program.run()` aufgerufen wird. Bei der Rückkehr aus dem Makro wird der Rückgabewert von `Program.run()` von der `execute()` Methode zurückgegeben.

4.6 Die Klasse `ControlCommand`

Die `ControlCommand`-Klasse ist von der `Command`-Klasse abgeleitet. Die Klasse `ControlCommand` faßt die Direktiven `.STOP`, `.PRINT` und `.INTERACTIVE` zusammen. Führt man für jeden dieser Direktiven eine eigene Unterklasse von Command ein, so erhält man drei Klassen, die fast identisch aufgebaut und sehr klein sind. Deshalb ist es sinnvoll, diese Direktiven durch eine Klasse darzustellen.

4.6.1 Konstruktor

Deklaration `ControlCommand(Point pos, int aCommandType);`

Der Konstruktor erhält als Parameter die Position des Befehls im Quelltext als Point und den Typ des Befehls, d.h., um welche Art von Steuerbefehl es sich handelt.

4.6.2 Attribute

Deklaration `int commandType;`

Dieses Attribut gibt den Typ des Kommandos an. Im aktuellen Entwurf sind `.STOP`, `.PRINT` und `.INTERACTIVE` vorgesehen. Für jeden dieser Typen ist ein bestimmter Wert vorgesehen. Die Werte für die Typen werden als klassenbezogene, konstante Werte in der `ControlCommand`-Klasse definiert. Wenn man eine Instanz dieser Klasse erzeugt, sollte man nur diese Konstanten verwenden (z.B. `ControlCommand.STOP`), damit beim Ändern oder Hinzufügen von Typen nur diese Klasse geändert werden muß.

4.6.3 Methoden

Deklaration `int execute();`

Ruft `owner.print()`, `owner.interactive()` oder `owner.stop()` auf, je nachdem, von welchem Typ das `ControlCommand`-Objekt ist. Der Rückgabewert der Methode selbst ist `prog_OK` (vgl. `Program`-Klasse in 4.7).

4.7 Die Klasse `Program`

Diese Klasse stellt die Funktionalität für ein Makro bzw. das Hauptprogramm der Turingmaschine bereit. Instanzen dieser Klasse werden von der Klasse `Parser` erzeugt.

4.7.1 Konstruktor

Deklaration `Program(TM aTuring, String aName);`

Der Konstruktor erzeugt ein `Program`-Objekt und initialisiert die Attribute `turing` und `name` mit den als Parameter angegebenen Werte. Das Attribut `code` wird als leerer `Vector` initialisiert.

4.7.2 Attribute

Deklaration `Vector code;`

Diese Datenstruktur speichert die einzelnen Befehle des Programms. Diese sind Instanzen von Unterklassen der abstrakten Klasse `Command`.

Deklaration `TM turing;`

Das Attribut `turing` ist eine Referenz auf die Instanz der Klasse `TM` (d.h. die Turingmaschine), zu der das `Program`-Objekt gehört.

Deklaration `String name;`

`name` ist der Name des `Program`-Objektes.

Deklaration `int nextCommand = prog_OK;`

Normalerweise arbeitet die Methode `run()` die Befehle des Programms nacheinander ab. Es gibt jedoch Fälle, in denen dieses nicht gewünscht wird, z.B. Sprunganweisungen oder wenn das Programm beendet werden soll. Über die Methode `nextCommand(int nextip)` kann in einem solchen Fall dieses Attribut auf einen Wert gesetzt werden, der vor dem Ablauf des nächsten Programmbefehls durch die Methode `run()` ausgewertet wird. Die Werte haben dabei folgende Bedeutung:

≥ 0 und <`code.size`	Es wird mit dem Befehl mit der Nummer, die in `nextCommand` steht, fortgefahren.
=`code.size`	Falls der letzte Statement des Makros ein Label ist, wird der Wert des Feldes genau `code.size` sein, also gibt es keinen Befehl, mit dem fortgefahren wird. In diesem Fall wird die Ausführung des Makros beendet.
`prog_Stopped`	Das Programm wird angehalten, d.h. `run()` wird beendet und liefert den Wert `prog_Stopped` zurück.
`prog_OK`	Die Befehlsausführung wird beim nächsten Befehl fortgesetzt.

Nach der Auswertung des Attributes durch `run()` wird dieses durch ebendiese Methode auf `prog_OK` zurückgesetzt. `code.size` ist die Größe des Vektors `code` (siehe 3.1).

4.7.3 Methoden

Deklaration `void addCommand(Command command);`

Diese Methode fügt den Befehl `command` zu dem `Program`-Objekt hinzu. Es handelt sich dabei um eine Instanz der Klasse `Command` oder einer davon abgeleiteten Klasse. Das neue `Command`-Objekt wird immer am Ende des Programms angefügt. Die Methode wird nur von Instanzen der Klasse `Parser` aufgerufen.

Deklaration `void nextCommand(int nextip);`

`nextCommand(int nextip)` setzt das Attribut `nextCommand` (siehe 4.7.2) auf den Wert von `nextip`. Damit kann der Befehl festgelegt werden, der als nächstes ausgeführt werden soll. Dies ist beispielsweise bei Sprunganweisungen vonnöten. Im aktuellen Entwurf wird die Methode lediglich von den `execute()`-Methoden der Klassen `JumpCommand` und `CondJumpCommand` aufgerufen.

Die normale sequentielle Abarbeitung einzelner Befehle wird durch die Methode `run()` gesteuert. `nextCommand()` wird also nur dann aufgerufen, wenn von dieser Abarbeitungweise abgewichen werden soll.

Deklaration `void print();`

Die Methode `print()` wird durch bestimmte Instanzen der Klasse `ControlCommand` aufgerufen. Durch diese Methode werden `Program`-Objekte davon benachrichtigt, daß eine `.PRINT`-Direktive ausgeführt wird. Diese Benachrichtigung wird über einen Aufruf von `turing.print()` an die zuständige Turingmaschine (d.h. eine Instanz der Klasse `TM`) weitergeleitet.

Deklaration `void interactive();`

Diese Methode wird ebenfalls durch Instanzen der Klasse `ControlCommand` aufgerufen. Durch ihren Aufruf werden `Program`-Objekte davon benachrichtigt, daß eine `.INTERACTIVE`-Direktive aufgetreten ist. Diese Benachrichtigung wird wiederum durch den Aufruf von `turing.interactive()` an die zuständige Turingmaschine weitergeleitet.

Deklaration `void stop();`

Auch diese Methode wird durch Instanzen der Klasse `ControlCommand` aufgerufen. Durch ihren Aufruf wird ein `Program`-Objekt von einer aufgetretenen `.STOP`-Direktive benachrichtigt. Die Methode ruft `nextCommand(prog_Stopped)` auf, damit die Methode `run()` die Befehlsabarbeitung abbricht.

Deklaration `void tapeModified();`

Diese Methode wird durch Instanzen der Klasse `HeadCommand` aufgerufen, um das `Program`-Objekt davon zu benachrichtigen, daß sich das Band der Turingmaschine geändert hat. Diese Nachricht wird durch den Aufruf von `turing.tapeChanged()` an die übergeordnete Turingmaschine weitergeleitet.

Deklaration `int run();`

Das ist die zentrale Methode der `Program`-Klasse. Handelt es sich bei der betrachteten Instanz um das Hauptprogramm der Turingmaschine, so wird diese Methode direkt durch die `TM`-Instanz bzw. deren Methode `run()` aufgerufen, ansonsten erfolgt der Aufruf durch eine Instanz der Klasse `MacroCommand`.

Die Methode arbeitet die Befehle des `Program`-Objekts, die in `code` stehen, nacheinander ab. Da es sich bei den Befehlen um Instanzen von `Command` oder davon abgeleiteten Klassen handelt, geschieht die Abarbeitung durch den Aufruf der Methode `execute()` dieser Objekte. Spunganweisungen und Makroaufrufe werden bei der Programmausführung berücksichtigt. Ersteres geschieht über die Auswertung des Attributes `nextCommand` nach jeder Befehlsabarbeitung, letzteres automatisch durch Ausführung der Methode `execute()` eines `MacroCommand`-Objektes.

Zusätzlich wird vor der Abarbeitung jedes Befehls dessen Position im Programmquelltext durch Aufruf von `Command.getPosition()` ermittelt und dann die Turingmaschine, zu dem das `Program`-Objekt gehört, durch Aufrufen der Methode `turing.commandCompleted` davon benachrichtigt, welcher Befehl des Quelltextes als nächstes zur Ausführung kommt.

Ebenfalls vor der Befehlsabarbeitung wird die übergeordnete Turingmaschine durch Aufruf von `turing.enterMacro(name)` davon benachrichtigt, daß das Programm „betreten“ wurde. Nach der Abarbeitung wird diese Turingmaschine durch Aufruf von `turing.leaveMacro()` vom Ende der Befehlsabarbeitung benachrichtigt.

Je nach Status bei Beendigung der Befehlsausführung liefert `run()` verschiedene Rückgabewerte. Diese sind:

`prog_OK`	Abarbeitung erfolgreich; keine Fehler aufgetreten
> 0	Der Laufzeitfehler mit der angegebenen Nummer ist aufgetreten.
`prog_Stopped`	Das Programm wurde durch ein „User-Break“ oder durch die Direktive .STOP beendet.

4.8 Die Klasse `Tape`

Die Klasse `Tape` ist ein abstrakter Datentyp, mit dem ein Turingband modelliert wird.

4.8.1 Konstruktor

Deklaration `Tape(char aBlank, String input);`

Der Konstruktor erzeugt ein `Tape`-Objekt. Die Attribute `blank` und `inputString` werden auf die als Parameter angegebenen Werte gesetzt. Außerdem wird das Band mit dem in `input` angebenen Wort beschrieben und der Schreib-/Lesekopf auf dem ersten Zeichen dieses Wortes positioniert. Enthält das Eingabewort nur Blank-Symbole ist seine Position irgendwo auf dem leeren Band.

4.8.2 Attribute

Deklaration `String inputString;`

Dieses Attribut speichert das Eingabewort des Bandes. Dieses wird durch den Konstruktor bei der Instantiierung auf das Band geschrieben sowie bei jedem Aufruf der Methode `reset()`.

Deklaration `char blank;`

Dieses Attribut speichert das Blank-Symbol des Bandalphabets.

Deklaration `Vector data;`

Dieses Attribut enthält die aktuelle Beschriftung des Bandes.

Deklaration `int headPosition;`

Dieses Attribut speichert die aktuelle Position des Schreib-/Lesekopfes auf dem Band.

4.8.3 Methoden

Deklaration `void right();`

Diese Methode bewegt den Schreib-/Lesekopf um eine Position nach rechts auf dem Turingband.

Deklaration `void left();`

Diese Methode bewegt den Schreib-/Lesekopf um eine Position nach links auf dem Turingband.

Deklaration `void write(char c);`

Diese Methode schreibt das Zeichen, das in `c` gespeichert ist, an der aktuellen Kopfposition auf das Band.

Deklaration `char get();`

Diese Methode liefert das Zeichen, das sich an der aktuellen Position des Schreib-/Lesekopfes befindet, als Rückgabewert.

Deklaration `int getHeadPos();`

Diese Methode liefert die aktuelle Kopfposition zurück. Die Methode wird immer dann aufgerufen, wenn der Bandinhalt abgerufen werden soll. Dies geschieht immer dann, wenn die Benutzerschnittstelle das Band darstellt. Der Rückgabewert dieser Methode ist eine Position in dem String, in dem die Banddaten stehen. Dieser String kann durch Aufruf der Methode `toString` ermittelt werden.

Deklaration `int getHeadOffset();`

Diese Methode liefert die Kopfposition relativ zu einem Fixpunkt zurück. Dieser Fixpunkt wird immer beim Aufruf von `reset()` festgelegt, d.h. er bezeichnet die Position auf dem Turingband, bei der die Kopfposition initialisiert wird. Dies ist für eine bessere Darstellung der Kopfbewegung innerhalb einer Nutzerschnittstelle nötig, z.B. wenn der Kopf unter dem feststehenden Band bewegt dargestellt werden soll, anstatt das Band über einem feststehenden Kopf bewegt darzustellen.

Deklaration `String toString();`

Diese Methode formatiert den Bandinhalt in einen String und gibt diesen an die aufrufende Methode zurück. Dies ist ebenfalls immer dann vonnöten, wenn der Bandinhalt abgerufen werden soll. Es wird nur der tatsächlich beschriebene Teil des Bandes und davor und dahinter jeweils genau ein Blank-Symbol zurückgegeben. Dabei ist jedoch darauf zu achten, daß die aktuelle Kopfposition unbedingt innerhalb dieses Strings liegen muß. Liegt der Kopf z.B. rechts vom beschriebenen Teil des Bandes, werden die Blank-Symbole bis zur Kopfposition ebenfalls mit in den Rückgabestring geschrieben.

Deklaration `String getString();`

Auch diese Methode formatiert den Bandinhalt in einen String und gibt diesen an die aufrufende Methode zurück. Der Unterschied zu `toString` besteht einzig und allein darin, daß die Kopfposition innerhalb des zurückgegebenen Strings durch die Zeichen „`<`“ und „`>`“ gekennzeichnet ist und am Beginn und am Ende des Strings jeweils drei Punkte stehen, z.B.:

```
...b11b11<1>11b...
```

Deklaration `void reset();`

Diese Methode dient dem Zurücksetzen des Bandes auf den Ursprungszustand. Das Attribut `data` wird zurückgesetzt, das Eingabewort `inputString` auf das Band geschrieben und der Schreib-/Lesekopf auf das erste Zeichen dieses Eingabewortes gesetzt.

4.9 Die Klasse `Parser`

Die Klasse Parser dient dazu, den Quelltext eines Turingprogramms zu lesen und die interne Datenstruktur aufzubauen. Der Entwurf sieht vor, daß jede Instanz von `TM` genau eine Instanz der Parser-Klasse enthält. Um die interne Datenstruktur aufzubauen, erzeugt eine bereits existierende (aber noch „leere“) TM eine neue Parser-Instanz und ruft deren Methode `doParse()` auf. Daraufhin wird die Datenstruktur entsprechend dem Quelltext aufgebaut bzw. ein Fehler im Quelltext signalisiert.

Die Parserklasse wird an keiner anderen Stelle des Systems genutzt und dient im wesentlichen dazu, die zum Parsen notwendigen Methoden und temporären Datenstrukturen zu kapseln.

4.9.1 Konstruktor

Deklaration `Parser(Reader aSource, TM aOwner);`

Der Konstruktor bekommt als Parameter den Reader, aus dem der Quelltext gelesen wird und eine Referenz auf das `TM`-Objekt, das den Parser benutzt.

4.9.2 Methoden

Deklaration `ParserReturn doParse();`

Diese Methode dient dazu, das Parsen eines Programmquelltextes zu starten. Es wird ein Container vom Typ `ParserReturn` zurückgegeben. Dieser enthält vier Attribute, eine `Error`-Instanz, falls ein Fehler während des Parsens aufgetreten ist und ansonsten je eine Instanz von `Program` (das geparste Programm als Objektstruktur), `Tape` (das initialisierte Turingband) und den Ausgabedateinamen als String.

Der weitere Aufbau der Parserklasse und der Ablauf des Parserdurchlaufs orientiert sich an der Definition der Sprache in EBNF-Form (siehe Pflichtenheft). Die Turingmaschine ist dort als Folge von einer Definition (Alphabet, Blank, usw.), Null oder mehr Makrodefinitionen und einem Hauptprogramm definiert. Dementsprechend besitzt die Parserklasse weitere private Methoden (siehe Kapitel 6).

4.10 Die Klasse `ParserReturn`

Diese Klasse dient als Sammelklasse, um die Datenstrukturen, die der Parser erzeugt, an die aufrufende Methode zurückgeben zu können.

4.10.1 Attribute

Deklaration `Error error;`

Dieses Attribut ist das vom Parser erzeugte Fehlerobjekt. War das Parsen fehlerfrei, wird ein Fehler mit dem Code 0 zurückgegeben, ansonsten ein Fehler, dessen Fehlercode die Art des Fehlers beschreibt.

Deklaration `Program main;`

Das Hauptprogramm der Turingmaschine.

Deklaration `Tape tape;`

Dies ist das mit dem Eingabewort initialisierte Band, auf dem die Turingmaschine arbeitet, wenn die Simulation gestartet wird.

Deklaration `String outFile;`

Dies ist der Name der Ausgabedatei, in die durch die Direktive `.PRINT` während des Simulationsablaufs Informationen geschrieben werden.

4.10.2 Konstruktor

Deklaration `ParserReturn(Error aError, Program aProgram, Tape aTape, String aOutFile);`

Der Konstruktor erhält die Referenzen auf die erzeugten Daten, und weist sie den entsprechenden Attributen zu.

4.10.3 Methoden

Alle Methoden der Klasse sind Zugriffsmethoden auf die Attribute:

Deklaration `Error getError();`

Deklaration `Program getMain();`

Deklaration `Tape getTape();`

Deklaration `String getOutFile();`

4.11 Die Klasse `Error`

Ein Objekt der Klasse `Error` wird erzeugt, wenn ein Fehler auftritt.

4.11.1 Attribute

Deklaration `int code;`

Jeder Fehlertyp hat eine Nummer, damit ein UserInterface für jeden Fehler einen passenden Text ermitteln und ausgeben kann. Der Fehlercode ist in diesem Attribut gespeichert.

Deklaration `int line;`

Bei Fehlern, die der Parser entdeckt, enthält dieses Attribut die Zeilennummer, in der der Fehler aufgetreten ist. Kann einem Fehler keine Zeilennummer zugeordnet werden, wird `line` auf -1 gesetzt.

Deklaration `String info;`

Über diesen String kann man dem Fehlerobjekt noch zusätzliche Informationen über den Fehler mitgeben. Beispiel: Findet der Parser ein Label, das bereits definiert wurde, kann dieser String den Namen des Labels enthalten. Wird keine zusätzliche Information über einen Fehler gegeben, ist `info` eine `null`-Referenz.

4.11.2 Konstruktoren

Die Konstruktoren der Klasse haben nur die Aufgabe, die Attribute mit den übergebenen Werten zu initialiseren.

Deklaration `Error(int aCode, int aLine, String aInfo);`

Initialisiert alle Attribute mit den gegebenen Werten.

Deklaration `Error(int aCode, int aLine);`

Der `info`-String bleibt eine null-Referenz.

Deklaration `Error(int aCode, String info);`

`line` wird auf -1 gesetzt.

Deklaration `Error(int aCode);`

Initialisiert nur den Fehlercode.

4.11.3 Methoden

Alle Methoden dieser Klassse sind Zugriffsmethoden, die den Wert des entsprechenden Attributs zurückgeben:

Deklaration `int getCode();`

Deklaration `int getLine();`

Deklaration `String getInfo();`

4.12 Die Klasse `TM`

Diese Klasse ist der Kern des Systemteils, sie ist die eigentliche Darstellung der Turingmaschine. Ihre Instanzen werden von der `GUI` erzeugt. Diese Klasse besitzt Referenzen auf alle wichtigen Komponenten des Programms und führt den gesamten Ablauf aus, der von der `GUI` gesteuert wird.

4.12.1 Konstruktor

Deklaration `TM(Reader aInfile, UserInterface aUI);`

Der Konstruktor erhält als Parameter den Eingabestrom, aus dem der Parser den Programmtext lesen kann. Er erzeugt ein Parser-Objekt und startet den Übersetzungsvorgang. Der Parser führt eine Syntaxüberprüfung durch und wandelt den Text in die interne Datenstruktur um. Er gibt dem `TM`-Objekt ein initialisiertes Band (ein `Tape`-Objekt), das Hauptprogramm (ein `Program`-Objekt) und den Namen der Ausgabedatei zurück bzw. ein `Error`-Objekt, falls während des Parsens ein Fehler aufgetreten ist. Außerdem benötigt die Turingmaschine eine Referenz auf das UserInterface, das den Ablauf der Simulation steuert. Nachdem der Konstruktor die Turingmaschine initialisiert hat, kann die Simulation durch Aufruf von `run()` gestartet werden.

4.12.2 Attribute

Deklaration `Tape tape;`

Das ist eine Referenz auf eine Instanz der Klasse `Tape`. Sie stellt das Band dar, auf dem die Turingmaschine arbeitet. Das Band wird vom Parser erzeugt und an das `TM`-Objekt zurückgegeben.

Deklaration `Program mainProg;`

Das Hauptprogramm, das hier in der Klasse `TM` benutzt wird, erfordert eine Referenz auf eine Instanz der Klasse `Program`.

Deklaration `Writer outFile=System.out;`

Das ist Ausgabestrom, in den die `print`-Methoden die Ausgaben der Turingmaschine schreiben. Das Attribut ist mit `System.out` initialisiert, damit die Ausgabe automatisch auf die Standardausgabekanal des Betriebssystems geleitet wird, wenn die Ausgabedatei nicht angegeben ist.

Deklaration `UserInterface ui;`

Das Attribut `ui` ist eine Referenz auf eine Instanz der Klasse `UserInterface`.

Deklaration `String outName;`

In diesem Attribut steht der Dateiname der Ausgabedatei.

4.12.3 Methoden

Deklaration `private void printProlog();`

Hier wird die Ausgabedatei geöffnet. Sie wird zu Beginn der Methode `run()` aufgerufen und benutzt den Dateinamen `outName`. Bevor der eigentliche Ablauf der TM beginnt, werden noch einige allgemeine Informationen über die aktuelle Turingmaschine in die Ausgabedatei geschrieben. Diese sind:

- Name und Autoren des Systems
- der Bandinhalt vor dem Start der Simulation

Deklaration `void print();`

Diese Methode schreibt den Bandinhalt, die Anzahl der bis zu diesem Zeitpunkt simulierten Schritte sowie den Makrostack in die Ausgabedatei.

Deklaration `private void printEpilog();`

Diese Methode wird nach Beendigung der Methode `run()` aufgerufen, um den endgültigen Status der Turingmaschine (Bandinhalt) und einige statistische Informationen (Anzahl der ausgeführten Schritte und Anzahl der Makroaufrufe) in die Ausgabedatei zu schreiben. Anschließend wird der Ausgabestrom wieder geschlossen.

Deklaration `void enterMacro(String name);`

Diese Methode ruft selbst nur `ui.pushMacro(name)` auf. Damit wird dem Benutzerinterface der Name des Makros mitgeteilt.

Deklaration `void leaveMacro();`

Und diese Methode wird von `Program.run()` nach der Ausführung aufgerufen. Sie ruft selbst nur `ui.popMacro()` auf.

Deklaration `void tapeModified();`

Die Methode meldet dem UserInterface mit der `ui.updateTape()`-Methode, daß das Band aktualisiert werden muß, denn nicht alle Befehle bewirken eine Veränderung des Bandinhaltes.

Deklaration `void interactiv();`

Die Methode meldet dem UserInterface mit der `ui.interactive()`-Methode, daß der interaktive Modus durch die Programmdirektive `.INTERACTIVE` erzwungen wird.

Deklaration `int commandCompleted(Point nextCommand);`

Diese Methode meldet der Benutzerschnittstelle, daß die Abarbeitung eines Befehls abgeschlossen ist. Soll die Simulation normal weiterlaufen, liefert `commandCompleted()` eine Null zurück, wenn gestoppt werden soll, einen Wert ungleich Null.

Deklaration `void run();`

Diese Methode startet den Programmablauf. Sie darf erst aufgerufen werden, wenn die Turingmaschine korrekt geparst und initialisiert wurde. Zunächst ruft sie die Methode `tape.reset()` auf, die das Band auf den Anfangszustand setzt, indem sie den Eingabestring auf das Band setzt. Anschließend öffnet sie die Ausgabedatei mit `void printProlog()`. Der Aufruf von `mainProg.run()` startet die Simulation der Turingmaschine. Mit `printEpilog()` wird die Ausgabedatei wieder geschlossen. Der Abschluß der Simulation wird dem Userinterface mit der `ui.programFinished`-Methode mitgeteilt.

4.13 Die Klasse `Alan`

Dieses ist die Hauptklasse, aus der das Programm gestartet wird (sie enthält die `main`-Methode). Sie erzeugt das Userinterface, von dem aus der weitere Ablauf gesteuert wird.

4.13.1 Attribute

Deklaration `UserInterface ui;`

Das einzige Attribut der Klasse Alan ist eine Referenz zum UserInterface. Dieses ist, wie dort beschrieben, das abstrakte Bindeglied vom Systemteil zur Oberfläche, in unserem Fall der GUI. Sie wird in der Methode `static void main(Strings[] args)` instanziiert.

4.13.2 Methoden

Deklaration `static void main(String[] args);`

Die Methode wertet die Kommandozeile aus und erkennt die Parameter. Als Parameter kann der Name einer Textdatei mit einer Turingmaschinenbeschreibung sowie eine Option zur Angabe der in der GUI verwendeten Sprache angegeben werden. Dann erzeugt sie über das `UserInterface` eine Instanz der `GUI`, die dann alles weitere steuert.

4.14 Die Klasse `UserInterface`

Die Klasse `UserInterface` ist eine Interface-Klasse, d.h. sie enthält nur abstrakte Methoden-Deklarationen, jedoch keine Attribute oder Implementierungen. In dieser Klasse werden alle Methoden deklariert, die das Turingmaschinen-System zur Ansteuerung der Benutzeroberfläche benötigt.

Durch die Realisierung der Benutzeroberfläche durch ein solches Interface wird eine geringe Kopplung des Turingmaschinen-Systems an seine Oberfläche erreicht. Einerseits muß die konkrete Realisierung der Benutzerschnittstelle den restlichen Klassen des Systems nicht bekannt sein, da nur die in `UserInterface` deklarierten Methoden zur Ansteuerung verwendet werden. Andererseits muß die Benutzerschnittstelle nicht direkt auf Datenstrukturen in den `TM`- und `Program`- Klassen zugreifen.

Im folgenden wird der Zweck der einzelnen abstrakten Methoden und deren Parameter beschrieben. Das bedeutet jedoch nicht, das sich jede Implementierung von `UserInterface` gemäß dieser Beschreibung verhält. So kann es z.B. sinnvoll sein, die Methoden des Interface so zu implementieren, daß keine Ausgaben gemacht werden.

4.14.1 Methoden

Deklaration `void updateTape(Tape tape);`

Dieser Methodenaufruf aktualisiert die Darstellung des Turingmaschinen-Bandes. `tape` ist die Referenz auf das `tape`-Objekt, das dargestellt werden soll. Um auf die Daten des Bandes zuzugreifen, müssen die Methoden `tape.getHeadPos()` und `tape.toString()` bzw. `tape.getString()` verwendet werden (siehe 4.8.3).

Deklaration `int commandCompleted(Point nextCommand);`

Die Methode `commandCompleted()` wird nach der Ausführung eines jeden Turingmaschinen-Befehls aufgerufen. Dadurch hat die Benutzeroberfläche die Möglichkeit, den Ablauf der Simulation zu steuern. Hier kann z.B. der Zeitraum zwischen der Ausführung zweier Befehle festgelegt oder etwas ausgegeben werden. In dem Parameter `nextCommand` werden zwei `int`-Werte übergeben. Der erste Wert gibt die Zeile innerhalb des Programmtextes an, in der der Befehl steht, der als nächstes ausgeführt wird. Der zweite Wert gibt die entsprechende Befehlsnummer innerhalb dieser Zeile an. Mit diesen beiden Werten kann die Benutzeroberfläche den jeweils nächsten Befehl anzeigen.

Um der Turingmaschine mitzuteilen, ob der Benutzer den Ablauf der Simulation unterbrechen will, wird ein int-Wert zurückgegeben. Soll die Simulation normal weiterlaufen, liefert `commandCompleted()` eine Null zurück, wenn gestoppt werden soll, einen Wert ungleich Null.

Deklaration `void pushMacro(String name);`

`pushMacro()` teilt der Benutzeroberfläche mit, daß die Turingmaschine das Makro `name` ausführt. Die Oberfläche kann dieses Makro dem Makrostack hinzufügen und die Anzeige des Makrostack aktualisieren.

Das Userinterface verwaltet einen eigenen Makrostack, da das Turingmaschinen-System selbst keinen Makrostack verwalten muß. Der von der Benutzeroberfläche verwaltete Makrostack dient also nur der Anzeige der Makroaufrufe.

Deklaration `void popMacro();`

`popMacro()` teilt der Benutzeroberfläche mit, das die Turingmaschine die Ausführung des aktuellen Makros beendet hat. Das Makro kann dann von dem Makrostack entfernt werden.

Deklaration `void error(Error e);`

Diese Methode dient zur Ausgabe von Fehlermeldungen durch das Userinterface. Es wird ein `Error`-Objekt übergeben. Das Userinterface kann den entsprechenden Text aus der `TextClass` holen und anzeigen.

Deklaration `void parsedCorrect(Tape tape);`

Wenn der Parser erfolgreich durchgelaufen ist, wird die Methode `parsedCorrect()` aufgerufen, damit die Benutzeroberfläche entsprechend darauf reagieren kann. Das Starten einer TM darf z.B. erst nach einem Aufruf von `parsedCorrect()` möglich sein. Nach dem erfolgreichen Parsen eines Turingprogramms soll das Eingabeband und die Anfangsposition des Kopfes angezeigt werden. Dazu wird auch hier eine Referenz auf das Band übergeben.

Deklaration `void interactive();`

Durch einen Aufruf der Methode `interactiv()` wird der Benutzerschnittstelle mitgeteilt, daß Sie ab sofort im interaktiven Modus laufen soll, d.h. der Benutzer selbst kann den Ablauf der Simulation steuern. Wie der interaktive Modus konkret realisiert wird, bleibt der jeweiligen Realisierung der Benutzerschnittstelle vorbehalten.

Deklaration `void programFinished(Tape tape);`

`programFinished()` wird aufgerufen, wenn das Turingprogramm beendet worden ist. Nach Ablauf eines Turingprogramms soll der letzte Zustand des Bandes angezeigt werden. Dazu wird auch hier eine Referenz auf das Band übergeben.

4.15 Die Klasse `GUI`

Die Klasse `GUI` ist eine Realisierung der Interface-Klasse `UserInterface` als graphische Benutzeroberfläche.

Diese Klasse wird in diesem Entwurf jedoch nicht näher spezifiziert, da sie einerseits keine interessanten Entwurfsaspekte enthält und anderseits ihre Realisierung sehr stark von der verwendeten Programmiersprache bzw. eventuell verwendeten Oberflächenbibliotheken abhängt.

Die im folgenden beschriebenen Grundkonzepte der `GUI`-Klasse beziehen sich auf die funktionalen Aspekte der Benutzerschnittstelle.

Interessant an der `GUI`-Klasse ist die Ablaufsteuerung der Simulation im interaktiven Modus. Die Turingmaschine selbst (also die Klassen `TM`, `Program`, `Command`, etc.) kennt den Simulationsmodus nicht, der gesamte Ablauf wird von der GUI gesteuert. Im nichtinteraktiven Modus wartet die `commandCompleted()`-Methode der GUI einen einstellbaren Zeitraum, bevor mit der Simulation fortgefahren wird. Im interaktiven Modus wird dieser Zeitraum vom Benutzer bestimmt, d.h. die `commandCompleted()`-Methode wartet auf die Interaktion des Benutzers.

Etwas komplizierter wird die Ablaufsteuerung durch die Funktion `Trace` im interaktiven Modus. Das entsprechende Makro muß ohne Darstellung abgearbeitet werden, danach soll aber wieder in den interaktiven Modus zurückgekehrt werden. In der GUI wird das folgendermaßen realisiert:

Der Ablaufmodus der Turingmaschine wird durch einen Zähler bestimmt. Ist dieser Zähler 0, ist der interaktive Modus aktiviert, ansonsten läuft die TM automatisch ab.

Wird der Tracebutton gedrückt, wird ein Flag gesetzt und die Simulation wird durch Beenden der `commandCompleted()`-Methode fortgesetzt. Ist der nächste Befehl ein Makro, wird vor dem erneuten Aufruf von `commandCompleted()` die Methode `pushMacro()` aufgerufen. Hier wird nun, falls das Trace-Flag gesetzt ist, der Interaktiv-Zähler um eins erhöht. Danach läuft die TM also automatisch durch das Makro. Sind die Befehle des Makros abgearbeitet, wird die Methode `popMacro()` aufgerufen, die den Interaktiv-Zähler um eins verringert, wenn das Trace-Flag gesetzt ist. Ist der Zähler dann wieder gleich null, läuft die TM anschließend wieder im interaktiven Modus weiter.

Der Interaktiv-Zähler ist nötig, da ein Makro selbst Makroaufrufe enthalten kann. Wird ein solches Makro „getraced“, dürfen alle rekursiven Makroaufrufe ebenfalls nicht angezeigt werden. Durch das Hoch- und Herunterzählen des Interaktiv-Zählers wird gewährleistet, daß ein Makro mitsamt seiner Makroaufrufe ausgeführt wird, bevor wieder in den Interaktiv-Modus zurückgekehrt wird.

Tritt während des Abarbeiten eines Makros eine `.INTERACTIVE`-Direktive auf, wird einfach der Interaktiv-Zähler auf Null gesetzt.

4.16 Die Klasse `TextClass`

Das Interface `TextClass` dient als Kapselung sämtlicher Texte, die eine Benutzerschnittstelle benutzt oder ausgibt. Jede Klasse, die dieses Interface implementiert, besitzt die Methode `getMsg()`, die einen Text abhängig von einem Integerwert zurückgibt. Soll ein Fehler ausgegeben werden, reicht es, `getMsg()` mit der Fehlernummer als Parameter aufzurufen, und man erhält den gewünschten Text. Abhängig davon, welche Implemetierung der Schnittstelle gerade genutzt wird, kann dies z.B. ein deutscher Text sein oder ein englischer.

4.16.1 Methoden

Deklaration `String getMsg(int no);`

Die `getMsg()`-Methode hat als Parameter einen Integerwert. Sie gibt den String zurück, der diesem Wert zugeordnet ist.

4.17 Die Klasse `Texte`

Diese Klasse ist eine Implementierung der `TextClass`-Schnittstelle und enthält alle Texte der Benutzerschnittstelle in deutscher Sprache.

4.17.1 Methoden

Deklaration `String getMsg(int no);`

Die Methode gibt den gewünschten Text in deutsch zurück.

4.18 Die Klasse `Texts`

Diese Klasse implementiert wie `Texte` die TextClass-Schnittstelle, enthält jedoch alle Texte in englischer Sprache.

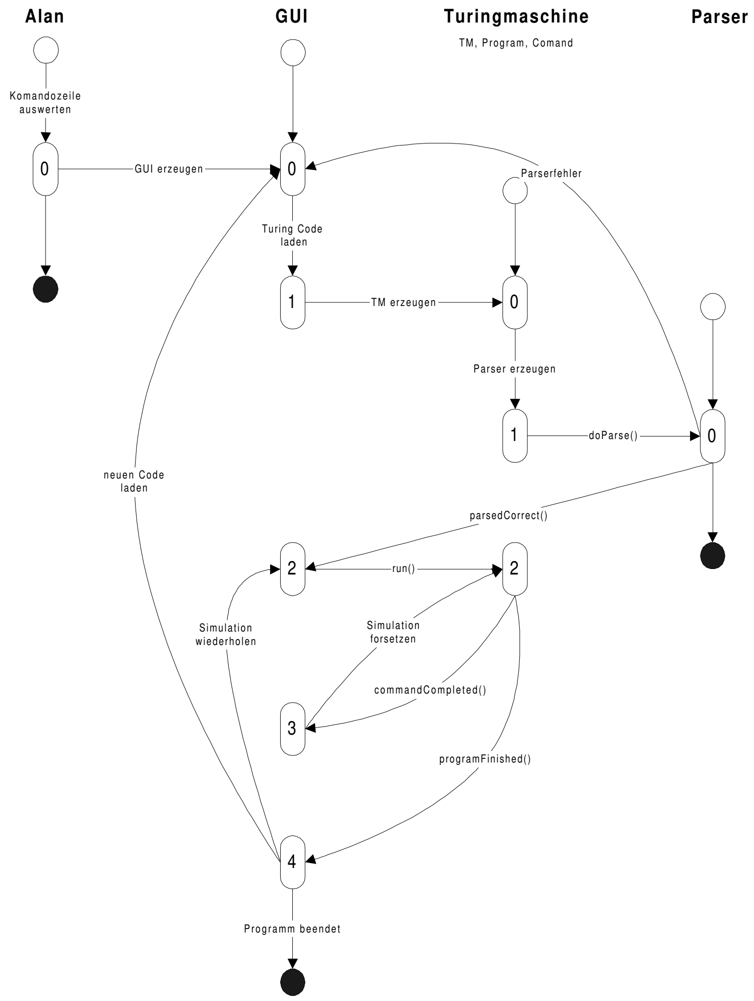


Abbildung 2: Das dynamische Modell des Systems

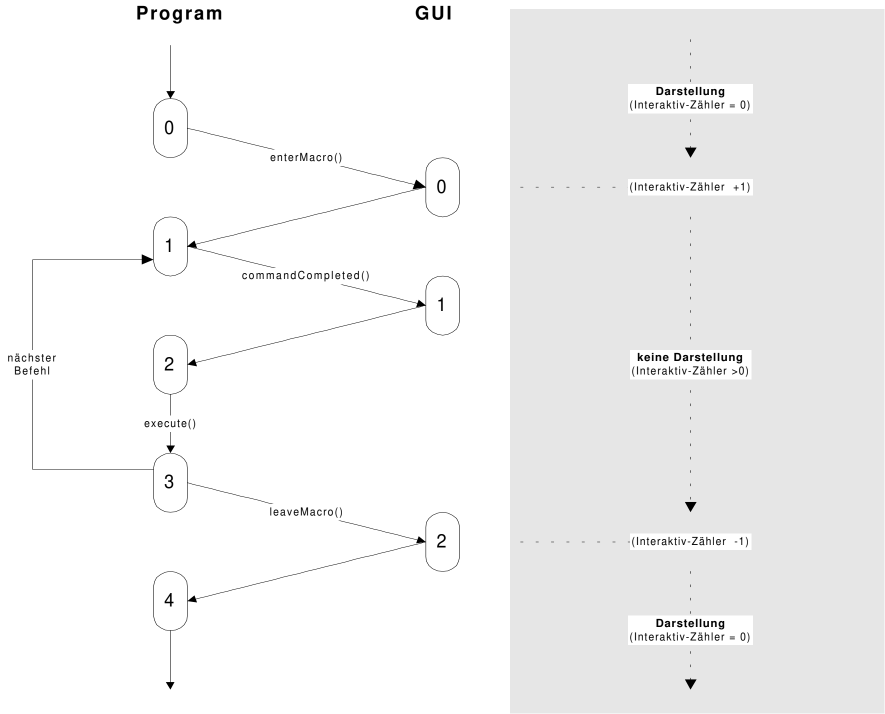


Abbildung 3: Das dynamische Modell des Systems (Ablauf der `Program`-Klasse)

5 Das dynamische Modell

In den Abbildungen 2 und 3 wird der dynamische Ablauf des Systems gezeigt. Die Spalten stellen die Klassen dar, die numerierten Symbole sind die Zustände der Klassen und die Pfeile zwischen den Symbolen Zustandsübergänge. Um die Übersichtlichkeit des Diagramms zu erhalten, werden nur die wichtigen Zustände aufgezeigt. Außerdem fehlen die Befehlsklassen. Sie wurden in der Spalte Turingmaschine zusammengefaßt.

Anmerkung: Das System läßt sich mit einem solchen dynamischen Diagramm nur zum Teil beschreiben, da es über eine graphische Benutzerschnittstelle gesteuert wird, die dem Benutzer zu jeder Zeit viele Funktionen bereitstellt. Beispielsweise läßt sich jederzeit während der Simulation die Ablaufgeschwindigkeit ändern, die Simulation abbrechen oder die Ausgabedatei anzeigen. Diese Details werden hier nicht dargestellt, sondern nur die wichtigen bzw. kritischen Abläufe.

5.1 Der globale Ablauf

Die Abbildung 2 stellt den Programmablauf des Gesamtsystems dar.

Die Hauptklasse `Alan` hat nur einen Zustand (Alan 0). Nach dem Auswerten der Kommandozeilenparameter wird eine Instanz der `GUI` erzeugt, womit Alan seine Aufgabe erfüllt hat. Die GUI lädt den Programmtext der Turingmaschine ein und erzeugt nun ihrerseits eine neue Turingmaschine (TM). Diese wiederum erzeugt einen Parser und startet diesen mit dem Methodenaufruf `doParse()`. Nach erfolgreichem Parsen des Quelltextes benachrichtigt der Parser die GUI durch die Methode `parsedCorrect()`. Erst in diesem Zustand (GUI 2) ist es möglich, die Simulation zu starten. Im Zustand (TM 2) findet die eigentliche Simulation der Turingmaschine statt. Nach jeder Befehlsausführung wird jedoch `commandCompleted()` aufgerufen, damit die GUI die Möglichkeit hat, den Ablauf der Simulation zu steuern (GUI 3). Nach Ablauf des gesamten Programms wechselt das System mit dem Methodenaufruf `programFinished()` in den Zustand (GUI 4), in dem entweder ein neues Programm geladen, die existierende TM neu gestartet oder das Programm verlassen werden kann.

5.2 Die `Program`-Klasse

In der Abbildung 3 ist der dynamische Ablauf der `Program`-Klasse und deren Zusammenarbeit mit der Benutzeroberfläche dargestellt.

Bevor die Befehle des Makros abgearbeitet werden, wird die GUI mit `enterMacro()` davon benachrichtigt, daß ein neues Makro ausgeführt wird. In der nun folgenden Schleife werden nacheinander alle Befehle des `Program`-Objektes abgearbeitet. Vor jedem Befehl wird der GUI durch die Methode `commandCompleted()` die Position des Befehls im Programmtext mitgeteilt. Außerdem hat die Benutzeroberfläche innerhalb dieser Methode die Möglichkeit, den Ablauf der Simulation zu steuern, indem z. B. auf eine Benutzerinteraktion gewartet wird. Sind alle Befehle ausgeführt worden, wird der GUI das mit `leaveMacro()` mittgeteilt.

Rechts vom dynamischen Modell der `Program`-Klasse ist ein kritischer Sonderfall des Interpreterablaufs dargestellt: das Verhalten der GUI im interaktiven Modus, wenn ein Makro „getraced“ werden soll, also ohne Darstellung ausgeführt wird. Das Problem in einem solchen Fall ist, daß der interaktive Modus während der Ausführung dieses Makros (inklusive aller seinerseits aufgerufenen Makros) verlassen werden muß.

In diesem Entwurf wird das Problem mit Hilfe eines Interaktiv-Zählers gelöst. Im interaktiven Modus ist der Interaktiv-Zähler gleich Null. Wenn ein Makro „getraced“ werden soll, wird innerhalb der Methode `enterMacro()` der Interaktiv-Zähler um

Eins erhöht und damit der interaktive Modus verlassen. Alle folgenden Befehle werden ohne Darstellung abgearbeitet. In der Methode `leaveMacro()` wird der Zähler wieder um Eins verringert, die Darstellung also wieder angeschaltet.

Auch rekursive Makroaufrufe werden durch diese Technik korrekt wiedergegeben. Jeder rekursive Makroaufruf erhöht den Zähler, beim Verlassen eines Makros wird der Zähler verringert. So ist sichergestellt, daß die Darstellung erst bei Verlassen des ursprünglich „getraceten" Makros wieder eingeschaltet wird, da erst in diesem Moment der Interaktiv-Zähler wieder gleich Null ist.

6 Spezifikation der Klasse `Parser`

6.1 Allgemeines

Das Parsen der Turingmaschinenbeschreibung läuft in zwei Abschnitten ab:

1. Zunächst durchläuft der Parser einmal den Text und sucht nach Makrodefinitionen. Für jedes gefundene Makro wird ein `Program`-Objekt erstellt und unter dem Makronamen in die Hash-Tabelle `macroTable` eingefügt. Außerdem werden bei diesem Durchlauf die vier Definitionen am Anfang des Textes (`$ALPHABET{..}`, `$BLANK{b}`, `$TAPE{...}` und `$OUT{...}`) ausgewertet.

2. Danach durchläuft der Parser noch einmal den Programmtext. Wenn er eine Makrodefinition findet, sucht er das dazugehörende `Program`-Objekt aus `macroTable` und fügt die Befehle des Makros ein. Wenn beispielsweise ein Makroaufruf erkannt wird, sucht der Parser ebenfalls in `macroTable` nach der zugehörigen Referenz auf das `Program`-Objekt und erzeugt damit eine Instanz der `MacroCommand`-Klasse.

   Zum Schluß des zweiten Durchgangs wird das Hauptprogramm geparst.

Durch dieses Vorgehen wird ermöglicht, daß aus einem Makro ein anderes Makro aufgerufen werden kann, welches erst später im Quelltext definiert wird. Auf die gleiche Weise wird beim Parsen der einzelnen Makros bzw. des Hauptprogramms vorgegangen: Im ersten Durchgang wird der Programmtext nach Labeldefinitionen durchsucht, die in der HashTable `labelTable` gespeichert werden. Im zweiten Durchlauf werden die Informationen in `labelTable` genutzt, um festzustellen, ob ein Label, das angesprungen wird, existiert, und wenn ja, auf welchen Befehl innerhalb des `Program`-Objektes es verweist.

Im angegebenen Pseudocode wird der Quelltext im Konstruktor zunächst in einen String eingelesen, mit dem die Methoden des Parsers dann arbeiten. Dies schien uns aus folgenden Gründen notwendig:

- Da der Parser mit mehreren Durchläufen durch den Quelltext arbeitet, ist es effizenter, den Quelltext einmal zu laden und in einem String zu speichern, als ihn mehrmals von der Festplatte zu laden.

- Der Parser erhält den Quelltext in Form eines `Reader`-Objekts, aus dem nur sequentiell gelesen werden kann. Deshalb ist es unwahrscheinlich daß man mehrere Tokenizer auf den Reader aufsetzen kann, die parallel lesen ohne sich gegenseitig zu beeinflussen.

Das gewählte Verfahren hat den Nachteil, daß es zu Speicherplatzproblemen führen kann, da der gesamte Quelltext in einem String gespeichert werden muß. Dies ist in unserem Fall jedoch in Kauf genommen werden, da nicht zu erwarten ist, daß Turingprogramme eine Größe erreichen, die zu Platzproblemen führt. Sollte sich diese Einschränkung als zu groß erweisen, ist der Entwurf in diesem Punkt noch zu verändern. Die nötigen Veränderungen beziehen sich jedoch nur auf den Konstruktor der `Parser`-Klasse, da alle anderen Methoden aus `ParserTokenizer`-Objekten lesen, deren Realisierung unabhängig von der Datenquelle ist.

6.2 Die Klasse `ParserTokenizer`

Die Klasse `ParserTokenizer` erweitert den StreamTokenizer um einige nützliche Methoden, die beim Parsen häufiger eingesetzt werden.

6.2.1 Konstruktor

Deklaration `ParserTokenizer(Reader aReader, boolean isEOLSignificant);`

Der Konstruktor dieser Klasse hat als Parameter den Reader, aus dem der Tokenizer seine Daten liest. Er ruft den Konstruktor der Oberklasse mit diesem Parameter auf. Als zweiter Parameter wird ein boolean gegeben, der angibt, ob der Tokenizer die End-Of-Line Zeichen als eigenes Token zurückgeben (`true`) oder überlesen (`false`) soll. Beide Fälle werden in der `Parser`-Klasse verwendet.

6.2.2 Methoden

Deklaration `boolean isNextToken(int tok);`

Deklaration `boolean isNextToken(String tok);`

`isNextToken` liefert `true`, wenn das nächste Token im Eingabestrom dem als Parameter gegebenen Integer bzw. String entspricht. Die Methode liest genau ein Token, ohne es wieder in den Strom zurückzustellen.

Deklaration `boolean isNextTokenIdent();`

`isNextTokenIdent` liefert `true`, wenn das nächste Token im Eingabestrom ein Ident im Sinne der Sprachdefinition ist (siehe Pflichtenheft). Das heißt, daß das Token vom Typ Wort sein muß, der erste Buchstabe jedoch keine Ziffer sein darf. Die Methode liest genau ein Token aus dem Eingabestrom, ohne es wieder zurückzustellen.

Deklaration `boolean searchToken(int tok);`

Deklaration `boolean searchToken(String tok);`

Diese Methode liest solange aus dem StreamTokenizer, bis das gegebene Token gefunden wird. Sie gibt `false` zurück, wenn das Token im Stream nicht mehr vorkommt und `true`, wenn das Token gefunden wurde. Das nächste Lesen eines Token aus diesem Strom gibt dann das erste Token nach dem gefundenen zurück.

Deklaration `void skipLineBreaks();`

Diese Methode überliest solange alle Tokens, bis eines gefunden wird, welches kein Zeilenumbruch ist. Dieses wird in den Strom zurückgestellt.

6.3 Private Attribute und Methoden

6.3.1 Attribute

Deklaration `TM owner;`

Diese Referenz auf das `TM`-Objekt benötigt der Parser, um sie den `Program`-Objekten, die er erzeugt, zu übergeben.

Deklaration `Program main;`

Dies ist die Referenz auf das Hauptprogramm der Turingmaschine, das der Parser mit Befehlen füllen soll. Dieses `Program`-Objekt wird von der Methode `doParse()` zusammen mit dem Band `Tape`, einem Ausgabedateinamen und evtl. einem `Error`-Objekt zurückgegeben.

Deklaration `Tape tape;`

Dies ist eine Referenz auf das Band, welches von dem `Parser`-Objekt initialisiert wird. Das Band wird von `doParse()` zusammen mit einem `Program`-Objekt, einem Ausgabedateinamen und evtl. einem `Error`-Objekt an das aufrufende `TM`-Objekt zurückgegeben.

Deklaration `String outFile;`

In diesem String wird der Name der Ausgabedatei, der im Definitionsteil des Turing-Programms durch die Direktive `$OUT` definiert wird, gespeichert. Er wird von `doParse()` zurückgegeben.

Deklaration `HashTable macroTable;`

In diese Hashtabelle werden die Makros eingefügt, die im Turing-Programm definiert werden. Die Makrotabelle enthält also `Program`-Objekte, wobei der Name der Makros als Key dient, unter dem die Objekte eingefügt werden.

Deklaration `HashTable labelTable;`

Während der Übersetzung eines `Program`-Objekts (Hauptprogramm oder Makros) dient diese Tabelle zum Speichern der Label und ihrer Einsprungpositionen. Als Key für die Hashtabelle dient der Labelname, als Datum die Position die Befehlsnummer, auf die das Label weist.

Deklaration `String source;`

In `source` wird der Quelltext der Turingmaschinenbeschreibung gespeichert. Dieses Attribut wird vom Konstruktor der Klasse mit diesem Text gefüllt.

Deklaration `StringReader sReader;`

Dieser Reader liest aus dem String `source` den Quelltext einer Turingmaschinenbeschreibung. Er wird zunächst beim ersten Durchlauf durch den Quelltext benutzt.

Bei der zweiten Bearbeitung des gesamten Quelltextes wird `sReader` für den Durchlauf durch ein Makro oder das Hauptprogramm verwendet, bei dem die Labeltabelle aufgebaut wird.

Deklaration `StringReader cReader;`

Auch dieser Reader liest aus dem String `source`. Er wird lediglich beim zweiten Durchlauf benutzt. Er kommt zum Einsatz, wenn die einzelnen Kommandos aus dem Quelltext extrahiert werden.

Deklaration `ParserTokenizer sTok;`

Dieser Tokenizer setzt auf dem Reader `sReader` auf und dient der Einteilung dieses Stromes in Token. Durch Aufrufe von `sTok.nextToken()` werden Daten aus `source` gelesen, d.h. alle Zugriffe auf `sReader` geschehen durch dieses Attribut.

Deklaration `ParserTokenizer cTok;`

Dieser Tokenizer setzt auf dem Reader `cReader` auf und teilt diesen Strom somit in Token ein. Zugriffe auf `cReader` geschehen nur durch Methoden von `cTok`.

Deklaration `Vector alphabet;`

Dieses Attribut dient der Speicherung des Alphabets der Turingmaschine.

Deklaration `char blank;`

Dieses Attribut repräsentiert das Blanksymbol.

6.3.2 Methoden

Deklaration `ParserReturn doParse();`

Dies ist die einzige öffentliche Methode der `Parser`-Klasse. Sie ruft `passOne()` und `passTwo()` auf. Das sind die Methoden, die die beiden Parserdurchläufe realisieren.

Zum Schluß wird ein `ParserReturn`-Objekt erzeugt, das als Container für alle vom Parser aufgebauten Datenstrukturen dient: das Hauptprogram, das Tape, der Name der Ausgabedatei und ein Fehlerobjekt, über das die aufrufende Methode feststellen kann, ob das Parsen erfolgreich war.

Deklaration `Error passOne();`

Dies ist der erste Durchlauf durch den Quelltext des Turingprogramms, in dem der Definitionsteil geparst und die Makrotabelle initialisiert wird. Dazu wird zunächst ein `StringReader`-Objekt erzeugt, der aus dem String `source` liest. Auf diesem Reader wird ein ParserTokenizer aufgesetzt, durch den der Quelltext sequentiell in Tokens aufgeteilt gelesen werden kann. Zum Parsen des Definitionsteils wird dann die Methode `parseDefinitions()` aufgerufen, gefolgt von der Methode `parseMakroNames()`, die die Makrotabelle initialisiert.

Deklaration `Error parseDefinitions();`

Diese Methode versucht, den Definitionsteil des Programms zu parsen und baut die entsprechenden Datenstrukturen auf (Alphabet, Blank-Symbol, Eingabeband, Dateiname der Ausgabedatei). Dazu wird die Methode `getDefinition` für jede Definition (ALPHABET, BLANK, TAPE und OUT) einmal aufgerufen. Wenn die Definitionen fehlerfrei sind, kann mit dem zurückgegebenen String die entsprechende Datenstruktur aufgebaut werden. Dabei muß überprüft werden, ob die Angaben im Definitionteil zulässig sind:

- Das Alphabet darf nur aus Buchstaben oder Ziffern bestehen.
- Das Blank-Symbol muß ein Zeichen sein, das nicht im Alphabet enthalten ist.
- Das Eingabe-Band darf nur aus Zeichen des Alphabets bestehen.

Die Methode gibt ein `Error`-Objekt zurück.

Deklaration `String getDefinition(String def, boolean emptyAllowed);`

Dies ist eine Hilfsmethode zum Parsen einer Zeile des Definitionsteils. Der übergebene String ist der Name der Definition, die gelesen werden soll (z.B. ALPHABET oder TAPE). Sie erwartet, daß als nächstes Token ein $ gefunden wird, gefolgt von dem gegebenen String. Dann prüft sie, ob hinter dem Definitionsstring ein Wort steht, das von geschweiften Klammern eingeschlossen wird. Dieses Wort wird als String zurückgegeben. Wenn `emptyAllowed true` ist, dann darf dieses gesuchte Wort auch leer sein, sonst nicht. Tritt bei einer der obigen Aktionen ein Fehler auf, wird das `Error`-Objekt entsprechend gesetzt und ein Leerstring zurückgegeben.

Deklaration `Error parseMakroNames();`

Aufgabe dieser Methode ist es, alle im Quelltext definierten Makros zu finden, und für jedes Makro ein `Program`-Objekt zu erzeugen, das unter dem Makronamen in die Makrotabelle eingefügt wird. Voraussetzung für die Methode ist, daß als nächstes Token die öffnende Klammer des ersten Makros gefunden wird. Dann wird geprüft, ob das nächste Token (der Name des Makros) ein IDENT im Sinne der Sprachdefinition ist und ungleich „STOP“, „PRINT“ bzw. „INTERACTIVE“. Ist dies der Fall, wird das Makro in die Makrotabelle eingefügt. Schließlich wird die schließende Klammer des Makros gesucht. Dies wird solange wiederholt, bis keine öffnende Klammer eines Makros mehr gefunden wird.

Wenn ein Makroname ungültig ist oder ein Makro doppelt definiert wird, gibt die Methode ein entsprechendes `Error`-Objekt zurück.

Deklaration `Error passTwo();`

Diese Methode realisiert den zweiten Durchlauf durch den gesamten Text der Turingmaschinenbeschreibung. Dabei werden zuerst durch `skipDefinitions` die vier Definitionen am Anfang des Textes überlesen. Danach werden alle vorkommenden Makros und zum Schluß das Hauptprogramm geparst. Dies geschieht durch Aufrufe von `parseProgram`, wobei `passTwo` jeweils die passenden `Program`-Objekte aus `macroTable` heraussucht. Falls diese Methoden ein `Error`-Objekt zurückgeben, so wird die Bearbeitung abgebrochen und die Methode liefert dieses `Error`-Objekt zurück.

Deklaration `void skipDefinitions(ParserTokenizer pt);`

Diese Methode dient zum Überlesen der vier Definitionen am Anfang des Quelltextes. Da diese bereits beim ersten Durchlauf umgesetzt werden, erübrigt sich eine Fehlerbehandlung. Die Informationen werden aus dem Tokenizer `pt` gelesen.

Deklaration `Error parseProgram(Program prg, boolean isMacro);`

Diese Methode parst ein Makro bzw. das Hauptprogramm der Turingmaschinenbeschreibung. Die Methode erwartet, daß die Tokenizer `sTok` und `cTok` als nächstes Token den ersten Befehl liefern. `isMacro` gibt an, ob es sich um ein Makro oder das Hauptprogramm handelt. Die instantiierten `Command`-Objekte werden zu dem `Program`-Objekt `prg` hinzugefügt.

Das Parsen geschieht wiederum in zwei Durchläufen. Dabei wird im ersten Durchlauf die Labeltabelle erzeugt. Die Einträge dieser Tabelle sind die jeweilige Position des auf ein Label folgenden Befehls. Um diese Position korrekt zu ermitteln, wird ein Zähler mitgeführt. In diesem Durchlauf werden mittels des Tokenizers `sTok` Daten gelesen.

Im zweiten Durchlauf werden die einzelnen Befehle geparst. Dabei kann jeweils am ersten Zeichen eines Befehls erkannt werden, um welchen Befehl es sich handelt, z.B. eine Sprunganweisung oder ein Makroaufruf, und dann entsprechend `Command`-Objekte instantiiert werden. Außerdem wird durch das Mitführen eines Spaltenzählers (Spalte = der wievielte Befehl innerhalb der Zeile der betrachtete ist) die genaue Position des Befehls im Quelltext festgestellt. Daten werden in diesem Durchgang aus dem Reader `cTok` gelesen.

Sollte während der Bearbeitung ein Fehler auftreten oder durch aufgerufene Methoden zurückgegeben werden, so wird die Bearbeitung abgebrochen und der Fehler an die aufrufende Methode zurückgegeben.

Deklaration `Error parseCommand(Program prg, Point p);`

Diese Methode parst einen Aufruf. Es kann sich dabei um einen Makroaufruf oder um einen der Befehle `.PRINT`, `.STOP`, `.INTERACTIVE`, `.l` oder `.r` handeln. Die Methode erwartet, daß der Reader `ct` als nächstes Token den Bezeichner des Aufrufs liest. Das instantiierte `Command`-Objekt wird dann zu `prg` hinzugefügt. Die Position des Befehls innerhalb des Quelltextes wird bereits durch die aufrufende Methode in `p` gegeben.

Sollte während der Abarbeitung ein Fehler auftreten, so wird die Bearbeitung abgebrochen, ein passendes `Error`-Objekt generiert und zurückgegeben.

Deklaration `Error parseBranch(Program prg, Point p, int type);`

Diese Methode parst eine Sprunganweisung. Sollte dies fehlerfrei möglich sein, wird die entsprechende `JumpCommand`- oder `CondJumpCommand`-Instanz zu dem `Program`-Objekt `prg` hinzugefügt. Die Position des Befehls wird durch die aufrufende

Methode in `p` gegeben. Außerdem teilt diese der Methode in `type` mit, um welche Sprunganweisung (unbedingt, = - Sprung, # - Sprung) es sich handelt.

Falls während der Abarbeitung ein Fehler auftritt, wird die Methode abgebrochen und ein passendes `Error`-Objekt generiert und zurückgegeben.

6.4 Pseudocode

Für den Pseudocode benutzen wir mangels einer einheitlichen Definition im wesentlichen die C++ Syntax. Das bedeutet z. B., daß

- als Operatoren die bekannten C-Operatoren benutzt werden. Zuweisungen werden durch ein einfaches Gleichheitszeichen dargestellt, der Gleichheitsoperator durch ein doppeltes Gleichheitszeichen.
- bei `if`-Ausdrücken der boolesche Ausdruck in Klammern hinter dem `if` folgt. Die restliche Notation (`if ... then ... else ... fi`) ist allerdings nicht an die C++ Syntax angelehnt.
- Anweisungsblöcke und Methodenrümpfe durch geschweifte Klammern begrenzt werden.

Kommentare werden durch ein doppelten Schrägstrich eingeleitet.

```
//-------------------------------------------
// Konstruktor
// Der Konstruktor erhaelt einen Reader, aus dem der vollstaendige
// Quelltext gelesen werden kann. Der Quelltext wird zur weiteren
// Verarbeitung eingelesen und in einem String abgelegt, um bei
// den Parserdurchlaeufen nicht immer wieder auf eine Datei
// zugreifen zu muessen.

Parser(Reader aSource, TM aOwner) {

  owner = aOwner;
  // Einen StringWriter erzeugen, mit dem der Sourcecode
  // in einen String geschrieben wird

  StringWriter sw = new StringWriter();
  int c;  //gelesenes Zeichen

  while (Dateiende noch nicht erreicht) do
    c=aSource.read();
    sw.write(c);
  od

  // gelesenen Text in source speichern;
  source = sw.toString();

  // Hauptprogramm initialisieren
  main = new Program(owner, "$Main" );

}


//--------------------------------
```

```
// doParse wird aufgerufen, um das Parsen zu starten

public ParserReturn doParse() {

  // erster Durchlauf: Definitionen lesen und makroTable initialisieren
  passOne();
  if (ein Fehler ist aufgetreten) gib den Fehler zurueck fi

  // zweiter Durchlauf: Macros und Hauptprogramm parsen
  if (kein Fehler aufgetreten ist) then
    passTwo();
    if (ein Fehler ist aufgetreten) gib den Fehler zurueck fi
  fi

  // Rueckgabe zusammenbasteln
  gib main, tape, outFile zurueck;
}


//-------------------------------
// Erster Durchlauf durch den gesamten Quellcode
// Definitions parsen und Makrotabelle aufbauen

private Error passOne() {

  Error e = null;
  sReader = new StringReader(source);        // Reader erzeugen
  sTok = new ParserTokenizer(sReader, false);  // Tokenizer erzeugen
                                             // EOL's werden ueberlesen

  // zunaechst alle Definitionen einlesen
  e = parseDefinitions();

  // dann Makros identifizieren und in makroTable eintragen
  if (kein Fehler ist aufgetreten) then
    e = parseMakroNames();
  fi

  sReader.close();  // Reader schliessen
  return e;    // Fehler zurueckgeben
}


//-------------------------------
// Diese Methode versucht, den Definitionsteil des Programms
// zu parsen und baut die entsprechenden Datenstrukturen auf
// (Alphabet, Blank, Band, Dateiname der Ausgabedatei).


private Error parseDefinitions() {

  int i; // Laufvariable

  // Alphabet lesen
```

```
String alph = getDefinition("ALPHABET");

if (Fehler ist aufgetreten) then
  Fehler zurueckgeben;
else
  // Aufbauen des alphabet-Vektors
  for i = 0 to alph.length() do

    //Zeichen an Position c merken
    Character c = new Character(alph.charAt(i));

    // pruefen, ob c schon vorhanden ist
    if (c ist in alphabet vorhanden) then
      entsprechenden Fehler zurueckgeben;
    else
      c in den Vector alphabet einfuegen;
    fi
  od
fi

// Blank lesen. Das Blanksymbol wird zunaechst in einen
// String gelesen, dessen Laenge ueberprueft wird
String blk = getDefinition("BLANK");

if (Fehler ist aufgetreten) then
  Fehler zurueckgeben;
else

  // Blank setzen
  if (blk ist laenger als 1) then
    Fehler zurueckgeben;
  else
    if (blk ist im Vector alphabet enthalten) then
      Fehler zurueckgeben;
    else
      blank = blk;  //Blanksymbol merken
    fi
  fi
fi

// Tapedefinition lesen
String t = getDefinition("TAPE");
if (Fehler ist aufgetreten) then
  Fehler zurueckgeben;
else
  // Testen, ob alle Zeichen gueltig sind
  for i=0 to t.length() do
    if (Zeichen an Position i nicht im Alphabet und
        ungleich dem BLANK-Zeichen) then
      Fehler zurueckgeben;
    fi
  od
  tape = new Tape( ... )
fi
```

```
  // Ausgabedateinamen lesen
  out = getDefinition("OUT");
  if (Fehler ist aufgetreten) then
    Fehler zurueckgeben;
  fi

  return null;  // Kein Fehler ist aufgetreten

}

//-------------------------------
// Hilfsmethode, um den Wert einer bestimmten Definition
// zurueckzugeben. Es wird erwartet, das der StreamTokenizer
// ein $ gefolgt vom gegebenen Namen def der Definition
// als die naechsten beiden Tokens findet. Wenn die Definition
// nicht gelesen werden kann, wird das error-Objekt entsprechend
// gesetzt.
// Rueckgabe: Den Wert der Definition als String, wenn sie gefunden
//            wurde, sonst null


private String getDefinition(String def, boolean emptyAllowed ) {

 String defVal;

 error = null;   // Fehler zuruecksetzen

 // '$' suchen

  if (!sTok.isNextToken('$')) then
    error neuen Fehler zuweisen;
    return null;
  fi

  // richtige Definition ?
  if (!sTok.isNextToken(def)) then
    error neuen Fehler zuweisen;
    return null;
  fi

  // '{' suchen
  if (!sTok.isNextToken('{')) then
    error neuen Fehler zuweisen;
    return null;
  fi

  // Wert der Definition lesen
  if (sTok.nextToken() ist ein Word-Token) then

    // wenn erfolgreich merken;
    defVal = sTok.sval;

    // Test, ob '}' folgt
```

```
    if (!sTok.isNextToken('}')) then
      error neuen Fehler zuweisen;
      return null;
    fi
  else // Darf die Definition leer sein?
       if ((sTok.ttype == '}') && emptyAllowed) then
         defVal = "";
       else
         error neuen Fehler zuweisen;
         return null;
       fi
  fi

  // Wert zurueckgeben
  return defVal;
}



//--------------------------------
// Diese Methode sucht im Programmtext nach Makros und
// traegt Program-Objekte unter dem Namen der Makros in die
// macroTable ein.

private Error parseMacroNames() {

  // Schleife: fuer jede gefundene { Makro parsen
  while (sTok.isNextToken('{')) do

    // Test, ob Makroname vorhanden ist und IDENT ist
    if (!sTok.isNextTokenIdent()) then
      Fehler zurueckgeben;
    fi

    // Programmobjekt in macroTable einfuegen
    if (makro ist noch nicht in macroTable) then
      if (Makroname ist ungleich STOP, PRINT, INTERACTIVE) then
        neues Program-Objekt erzeugen;
        Program-Objekt unter seinem Makronamen in macroTable einfuegen;
      else
        Fehler zurueckgeben: ungueltiger Makroname
      fi
    else
      Fehler zurueckgeben: doppelter Makroname;
    fi

    // Schliessende Klammer suchen

    if (!sTok.searchToken('}')) then
      Fehler zurueckgeben;
    fi

  od
```

```
  // kein Fehler aufgetreten
  return null;
}



//-------------------------------
// zweiter Durchlauf durch dem gesamten Quelltext
// Makros und Hauptprogramm parsen

private Error passTwo() {

  sReader = new StringReader(source);           // Reader fuer Label erzeugen
  cReader = new StringReader(source);           // Reader fuer Kommados erzeugen
  sTok = new ParserTokenizer(sReader, true);      // Tokenizer fuer Label erzeugen
  cTok = new ParserTokenizer(cReader, true);      // Tokenizer fuer Kommandos erzeugen

  // Bei beiden Parsertokenizer-Objekten sind EOL's signifikant,
  // weil sie fuer die Befehlszaehlung benoetigt werden.


  // Definitions Teil ueberspringen
  skipDefinitions(sTok);
  skipDefinitions(cTok);

  // Leerzeilen ueberspringen
  sTok.skipLinebreaks();
  cTok.skipLinebreaks();

  // solange noch Makros uebrig sind
  while (sTok.isNextToken('{')) do

    // cTok synchronisieren
    ueberlies ein Token in cTok;

    // Makronamen lesen
    sTok.nextToken();
    cTok.nextToken();

    // Makro parsen

    parseProgram(macroTable.get(gefundener Name), true);
    if (ein Fehler ist aufgetreten) then gib den Fehler zurueck;
    else

      // '}' ueberlesen
      sTok.searchToken('}');
      cTok.searchToken('}');

      // Leerzeilen ueberspringen
      sTok.skipLinebreaks();
      cTok.skipLinebreaks();
    fi
  od
```

```
  // letztes Token zurueckschreiben
  sTok.pushBack();
  cTok.pushBack();

  // Hauptprogramm parsen
  parseProgram(main, false);
  if ( Fehler aufgetreten ) gib den Fehler zurueck fi;

  // Streams schliessen und beenden
  sReader.close();
  cReader.close();
}

//--------------------------------
// Diese Methode ueberspringt den Definitionsteil eines Programms
// Sie muss im zweiten Durchlauf aufgerufen werden, weil sie erwartet,
// dass der Definitionsteil korrekt ist und der Stream auf den
// Anfang des Quelltextes zeigt.

private void skipDefinitions(ParserTokenizer pt) {

  pt.searchToken('$');                // erstes '$' suchen
  while (!pt.isNextToken("OUT"))
    pt.searchToken('$');              // naechstes '$' suchen
  od

  // bis zur Klammer zu der OUT-Definition ueberlesen
  pt.searchToken('}');
}

//--------------------------------
// parseProgram uebersetzt das naechste Makro
// Vorraussetzung: beide Tokenizer muessen als erstes Token den
// ersten Befehl des Makros finden. Das Programm p wird mit den
// gefundenen Befehlen aufgefuellt.

private Error parseProgram(Program prg, boolean isMacro) {

  neue labelTable anlegen;
  int commandCount;          // zaehlt die Kommandos beim ersten Durchlauf
  int columnCount;           // zaehlt die Spalten beim zweiten Durchlauf

  // Erster Durchlauf: Tokenizer sTok;
  // Labeltabelle aufbauen
  commandCount = 0;
  do
    sTok.nextToken();

    case (gelesenes Token) of

      // Aufruf oder unbedingten Sprung gefunden
      ('.' oder '>' gefunden):  ueberlies ein Token;
```

```
    // bedingten Sprung gefunden
    ('=' oder '#' gefunden):  ueberlies drei Token;

    // Label gefunden
    (':' gefunden):

      // kein Ident nach dem ':'?
      if(!sTok.isNextTokenIdent()) then Fehler generieren;
      else if(gefundener Ident ist nicht in labelTable) then
             labelTable.put(gefundener Ident, commandCount);
           else
             Fehler (Label doppelt)
           fi
      fi
      commandCount--;      // Label nicht mitzaehlen

    (Zeilenende gefunden): commandCount--;    // EOL nicht mitzaehlen

    (Dateiende gefunden):
    ('}' gefunden):
    (Wort gefunden): ;      // diese drei nicht als Fehler behandeln

    default:                 // unerlaubtes Zeichen gefunden
      Fehler (Anweisung beginnt mit ungueltigem Zeichen)
  esac

    commandCount++;         // naechstes Kommando
od while((!isMacro && Streamende nicht erreicht) ||
      (isMacro && noch kein '}' gefunden)));

// Klammer oder EOF zurueckschreiben
sTok.pushBack();

// Zweiter Durchlauf: Tokenizer cTok
// Syntaxpruefung und Command-Hierarchie aufbauen
columnCount = 1;
do
  cTok.nextToken();

  case (gelesenes Zeichen) of
    '.':                     // Aufruf gefunden
      parseCommand(prg, new Point(columnCount, cTok.lineno()));

    '>':                     // Unbedingten Sprung gefunden
      parseBranch(prg, new Point(columnCount, cTok.lineno()), unbedingter Sprung);

    '#':                     // bedingten Sprung gefunden
       parseBranch(prg, new Point(columnCount, cTok.lineno()), # - Sprung);

    '=':
      parseBranch(prg, new Point(columnCount, cTok.lineno()), = - Sprung);

    ':':                     // Label gefunden
      ueberlies ein Token;   // Identifier ueberlesen
```

```
      columnCount--;           // Spaltenzaehler nicht hochzaehlen

      (ein Wort gefunden):     // Zeichen, das auf das Band geschrieben wird
        if(gef. Wort w hat Laenge 1) &&
          (alphabet.contains(w) || (w == blank))) then
          prg.addCommand(new HeadCommand(Point(columnCount, cTok.lineno()),
              Zeichen schreiben, w, tape));
        else
          Fehler (eine ungueltige Anweisung)
        fi

      (Zeilenende gefunden):
        columnCount = 0;   // zuruecksetzen und gleich implizit 1 abziehen

      '}':                // bei '}' nur im Hauptprogramm Fehler
        if (!isMacro) then Fehler (ungueltige Anweisung '}') fi

      (Streamende gefunden):  // bei EOF nur in Makros Fehler
      if (isMacro) then Fehler (unerwartetes Ende der Eingabe)

    default:                 // unerlaubtes Zeichen gefunden
      Fehler (Anweisung beginnt mit ungueltigem Zeichen)
    esac

    // Spaltenzaehler erhoehen
    columnCount++;

  od while((!isMacro && (Streamende nicht erreicht)) ||
       (isMacro && noch kein '}' gefunden)));

  // Klammer oder EOF zurueckschreiben
  cTok.pushBack();
}

//-------------------------------
// parst einen Aufruf. Die Position des Befehls im Quelltext
// steht in p. Falls keine Syntaxfehler auftreten, wird der
// Befehl an das Programm prg angehaengt.

private Error parseCommand(Program prg, Point p) {

  if (!cTok.isNextTokenIdent()) then Fehler (kein Identifier nach dem '.')
  else    // die verschiedenen Commandarten testen
    if("PRINT" gefunden) then
      prg.addCommand(new ControlCommand(p, PRINT-Kommando));
    else if("STOP" gefunden) then
      prg.addCommand(new ControlCommand(p, STOP-Kommando));
    else if("INTERACTIVE" gefunden) then
      prg.addCommand(new ControlCommand(p, INTERACTIVE-Kommando));
    else if("r" gefunden) then
      prg.addCommand(new HeadCommand(p, nach_rechts, tape));
    else if("l" gefunden)) then
      prg.addCommand(new HeadCommand(p, nach_links, tape));
    else if( gefundener Name in macroTable enthalten )
```

```
      prg.addCommand(new MacroCommand(p, entsprechende Program-Objekt));
    else Fehler (Aufruf eines undefinierten Makros) fi fi fi fi fi fi
 fi
}

//------------------------------
// parst eine Sprungkommando. Die Position des Befehls im Quelltext
// steht in p. Der Typ des Sprunges steht in type.  Falls keine
// Syntaxfehler auftreten, wird der Befehl an das Programm prg angehaengt.

private Error parseBranch(Program prg, Point p, int type) {
  char c;          // Zeichen, auf dessen Auftreten gesprungen werden soll

  // bei Bedingten Spruengen Zeichen ueberpruefen und '>' suchen
  if(type kein unbedingter Sprung) then
    if (naechstes Token ist ein Wort der Laenge 1, und
        dieses Zeichen ist im alphabet enthalten oder blank) then
      c = dieses Zeichen
    else
      Fehler (ungueltige(s) Zeichen in Sprunganweisung)
    fi

    // '>' suchen
    if(!cTok.isNextToken('>')) then
      Fehler ('>' fehlt in bedingtem Sprung)
    fi
  fi

  // bei allen Spruengen Identifier pruefen
  if(!cTok.isNextTokenIdent())  then
    Fehler (erwarte Identifier nach einem '>')

  // Sprung zu undefiniertem Label?
  else if( labelTable enthaelt den gefundenen Identifier nicht ) then
    Fehler (Sprung zu undefiniertem Label)
  fi fi

  // Command - Objekt instantiieren

   if(type == unbedingter Sprung) then
      prg.addCommand(new JumpCommand(p, labelTable.get( gef. Identifier ));
   else
     prg.addCommand(new CondJumpCommand(p, labelTable.get( gef. Indentifier ),
        c, tape, (type == =-Sprung));
}
```

7 Änderungen am Entwurf

7.1 Die Klasse `UserInterface`

Die Methoden `parsedCorrect()`, `updateTape()`, und `programFinished()` haben im Gegensatz zum ersten Entwurf nur einen Parameter vom Typ `Tape`. Das Userinterface ruft die Methode `toString()` dieser Klasse nur dann auf, wenn es wirklich benötigt wird. Das führt zu *erheblichen* Beschleunigung der Simulation.

7.2 Die Klasse `Error`

Diese Klasse hat einen weiteren Konstruktor erhalten, bei dem nur der Fehlercode und ein Infostring angegeben werden. Der Konstruktor wird bei Laufzeitfehlern benutzt, denen sich keine Zeilennumer zuordnen läßt.

7.3 Die Klasse `Tape`

Die Klasse `Tape` hat folgende neue Methoden erhalten:

- Es wurde eine neue Methode `getString()` hinzugefügt, die eine Stringrepräsentation des Bandes zurückgibt, wie sie für die Ausgabe in der Ausgabedatei benötigt wird.

- Um eine bessere Darstellung des Bandes in der Benutzerschnittstelle zu ermöglichen, wurde eine neue Methode `getHeadOffset()` eingeführt. Diese Methode gibt den Abstand des Kopfes von der Startposition zurück. Dadurch wird es der GUI ermöglicht, den Kopf unter dem Band hin und her zu bewegen, was die Übersichtlichkeit der Darstellung erhöht.

7.4 Die Klasse `Parser`

Bei der Implementation der Klasse `Parser` hat sich gezeigt, daß noch einige Details geändert werden mußten.

- In der Spezifikation war vorgesehen, daß im Definitionteil der Turingmaschinenbeschreibung in jeder Definition ein String angegeben werden muß. Dies wurde dahingehend geändert, daß sowohl beim Eingabeband als auch bei der Ausgabedatei leere Definition möglich sind. Dazu haben wir der privaten Methode `getDefinition()` einen weiteren Parameter `emptyAllowed` gegeben. Dieser boolesche Parameter gibt an, ob bei der Definition ein Leerstring erlaubt sein soll oder nicht.

- In der Spezifikation des Parsers haben wir übersehen, daß Makros nicht mit den Schlüsselwörtern STOP, PRINT, oder INTERACTIVE bezeichnet werden dürfen. Die Methode `parseMakroNames()` wurde um eine entsprechende Abfrage erweitert.

- Der Parser hat das Fehlen der geschlossenen Klammer bei einer Makrodefinition nicht erkannt, was zu unklaren Fehlermeldungen führen kann. In der Methode `parseProgram` wurden deshalb in dem switch-Block, der die einzelnen Befehlstypen unterscheidet, einige weitere `case`-Fälle hinzugefügt.

- Die im Pseudocode verwendeten Bezeichnungen `sr`, `cr`, `st` und `ct` für die Reader bzw. Tokenizer wurden in `sReader`, `cReader`, sTok und `cTok` umgeändert.

7.5 Die Klasse `Command`

Die Methode `execute()` der Klasse `Command` und aller Unterklassen gibt jetzt einen Wert vom Typ `int` zurück. Über diesen Rückgabewert wird dem Programmobjekt mitgeteilt, ob während der Ausführung des Kommandos die Simulation abgebrochen wurde oder ein Laufzeitfehler aufgetreten ist.

7.6 Die Klasse `HeadCommand`

Der Konstruktor dieser Klasse wurde durch zwei andere mit folgender Signatur ersetzt:

- `Headcommand(Point pos, int aTapeAction, Tape aTape)`
- `Headcommand(Point pos, char aWriteChar, Tape aTape)`

Der erste Konstruktor dient zum Erzeugen von Befehlen, die den Schreib-/Lesekopf nach links oder rechts bewegen, der zweite zum Erzeugen von Befehlen, die das angegebene Zeichen auf das Band schreiben.

7.7 Die Klasse `Program`

Für die Werte des Feldes `nextCommand` und der Parameters der gleichnamigen Methode dieser Klasse wurden symbolische Konstanten eingeführt. Dies sind:

- `prog_OK` zeigt an, daß normal fortgefahren wird (war -2 im Entwurf)
- `prog_Stopped` zeigt an, daß das Programm angehalten wurde (war -1 im Entwurf)

8 Glossar

Container Mit Container wird in diesem Entwurf eine Klasse bezeichnet, die im wesentlichen nur Attribute enthählt. Ein Container dient zur Bündelung und Kapselung von Daten verschieden Types, um diese Daten z.B. als Rückgabe einer Methode übergeben zu können.

Direktive siehe ↑ Programmdirektive

GUI Eine Abkürzung für Graphical User Interface, also eine grafische Benutzerschnittstelle, die der Benutzer meistens mit Hilfe von Menüs und Buttons bedienen kann.

Interpreter Ein Interpreter liest jeden Befehl eines Programms der Reihe nach ein und führt ihn dann aus. Der Programmcode wird also nicht im voraus in eine für einen speziellen Rechner ausführbare Maschinensprache übersetzt, sondern erst bei der Ausführung des Programms. Interpreterprogramme sind daher langsamer als vorübersetzte, aber für den Programmierer leichter zu handhaben.

Java Java ist eine von Sun Microsystems entwickelte Programmiersprache. Sie wurde mit dem Ziel der maximalen Portabilität entworfen. Javaprogramme können auf jedem System laufen, sofern eine entsprechende Laufzeitumgebung, die ↑ Java Virtual Machine, für dieses System existiert. Aufgrund dieser Plattformunabhängigkeit und der in die ↑ Java Virtual Machine integrierten Möglichkeiten des Netzwerkzugriffs sind Java-Programme besonders für verteilte Netzwerkumgebungen wie z.B. das World-Wide-Web geeignet (sog. Applets), aber auch lokal operierende Anwendungen lassen sich mit Java elegant programmieren.

Java Virtual Machine Die virtuelle Java-Maschine ist die Laufzeitumgebung, in der die Java-Anwendung abläuft. Ein Java-Quelltext wird beim Kompilieren nicht wie bei anderen Programmiersprachen üblich in die Maschinen-Sprache einer speziellen CPU übersetzt, sondern in den sogenannten Java-Bytecode.

Der Bytecode wird von der virtuellen Java-Maschine zur Laufzeit interpretiert. Dieses Konzept macht es möglich, das Java-Programme auf jedem System laufen, für das eine Java Virtual Machine existiert.

Makro Mit Makro bezeichnen wir einen Befehl der sich aus mehreren Einzelbefehlen zusammensetzt. Makros bilden also quasi kleine Unterprogramme, die jeder Zeit aufgerufen werden können. Dadurch ist es möglich, Programmtext mehrfach zu verwenden, ohne ihn jedesmal neu einfügen zu müssen. Rekursion ist hierbei möglich.

Makrostack Der Makrostack gibt an, welche Makros aufgerufen und gerade abgearbeitet werden. Wird ein Makro aufgerufen, wird dessen Name auf den Makrostack gelegt und nach seiner Beendigung wieder entfernt. So ist es möglich, die Verschachtelungstiefe der Makroaufrufe mitzuverfolgen.

OMT OMT ist ein Entwurfsverfahren für objektorientierte Systeme. Mit Hilfe eines OMT-Klassendiagramms kann die Systemarchitektur grafisch dargestellt werden. OMT-Werkzeuge erlauben es, eine grafisch erstellte Systemarchitektur in Programmtext umzuwandeln.

Objektorientierte Programmierung (OOP) stellt eine Methode dar, bereits bei der Analyse anders an Probleme heranzugehen als bei konventioneller modularer Programmierung. Die Aufgabenstellung wird in diesem Fall nicht als abstrakte Funktion betrachtet, die schrittweise verfeinert werden muß, sondern es wird von Gegenständen ausgegangen, die in der Realität bearbeitet werden. Diese Gegenstände, die Objekte, haben gewisse Eigenschaften, die man bei der Analyse möglichst gut erfassen und mit einer Programmiersprache abbilden möchte.

Parser Ein Parser überprüft einen Text auf syntaktische Korrektheit. Es werden z.B. alle Schreibfehler oder Klammersetzungsfehler erkannt. Außerdem übersetzt er den Programmtext in die interne Datenstruktur.

Programmdirektive Eine Programmdirektive ist ein spezieller Befehl im Programmtext. Er beeinflußt den Programmablauf nicht direkt, sondern indirekt, indem er auf die Ablaufumgebung wirkt. Beim einem Interpreter geschieht das zur Laufzeit des Programms. Ein Beispiel für eine Programmdirektive ist der Befehl `.INTERACTIVE`, der das System in den interaktiven Modus versetzt, aber den Ablauf und das Ergebnis der Turingmaschine nicht verändert.

Token Ein Token ist ein Element der Syntax einer Programmiersprache. Dazu gehören z.B. die reservierten Wörter der Sprache, Nummern, Operatoren, ↑ Programmdirektiven, aber auch Klammern und Kommentarzeichen. Der gesamte Programmtext läßt sich in Token einteilen und dann vom Parser verarbeiten.