Design Specifications

Washington State University – Tri Cities

CPT_S 322: Software Engineering Principles I

Spring Semester, 2025

Project Tittle:

Interacting Turing Machine Application

Submitted by:

Raul Y. Martinez

Course Instructor:

Dr. Niel B. Corrigan

Date:

March 23rd, 2025

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

List of Figures

Revision History

1st Edition----------3/23/2025

2nd Edition----------4/21/2025

Changes: Minor content accuracy alignments.

1.0 Introduction

This document presents an overview of the detailed object-oriented design for the Turing Machine Application software engineering project, developed as the main coursework for CPT_S 322 Software Engineering Principles I with Dr. Niel B. Corrigan. This design specification is intended for instructors, teaching assistants, and student developers involved in reviewing, implementing, or assessing this application. The design is based on the *Turing Machine Application Requirements Specification*, which defines the functional and non-functional goals of the system. This design specification serves as a blueprint for implementation and provides a complete design ready for development, aligned with the course's instruction on object-oriented design principles (OOD) and Unified Modeling Language (UML).

The remainder of the document is organized as follows:

- The **Title Page**, **Table of Contents**, **List of Figures**, and **Revision History** provide document metadata, navigation, and version tracking.

- **Section 2.0 – Architecture** presents the UML class diagram for the overall design, showing relationships, attributes, and methods.

- **Section 3.0 – Data Dictionary** offers in-depth descriptions of each class in the system, including purpose, associations, attributes, and method operations, with some pseudocode for complex behavior.

- **Section 4.0 – User Interface** illustrates the expected inputs and outputs of the application, covering each supported command with multiple examples.

- **Section 5.0 – Files** describes the external files used by the application, including Turing machine definition and input string file.

- **References** lists all documents and sources cited during the design process.

- **Appendix** includes any supplementary material that supports or extends the design.

This structured design document ensures traceability from requirements to implementation and lays the foundation for building a robust, interactive Turing Machine simulator.

2.0 Architecture

The architecture of the Turing Machine application is designed using object-oriented principles intended to be implemented in the C++ language (compiled with the GNU GCC compiler in a Ubuntu Linux environment) and is represented by the UML class diagram shown below. The diagrams outline the core components of the system, including their attributes, methods, and associations. All attributes include their data types, and all methods include parameter types and return types where applicable. These architectural models provide the foundation for the implementation and ensures traceability to the requirements specification.

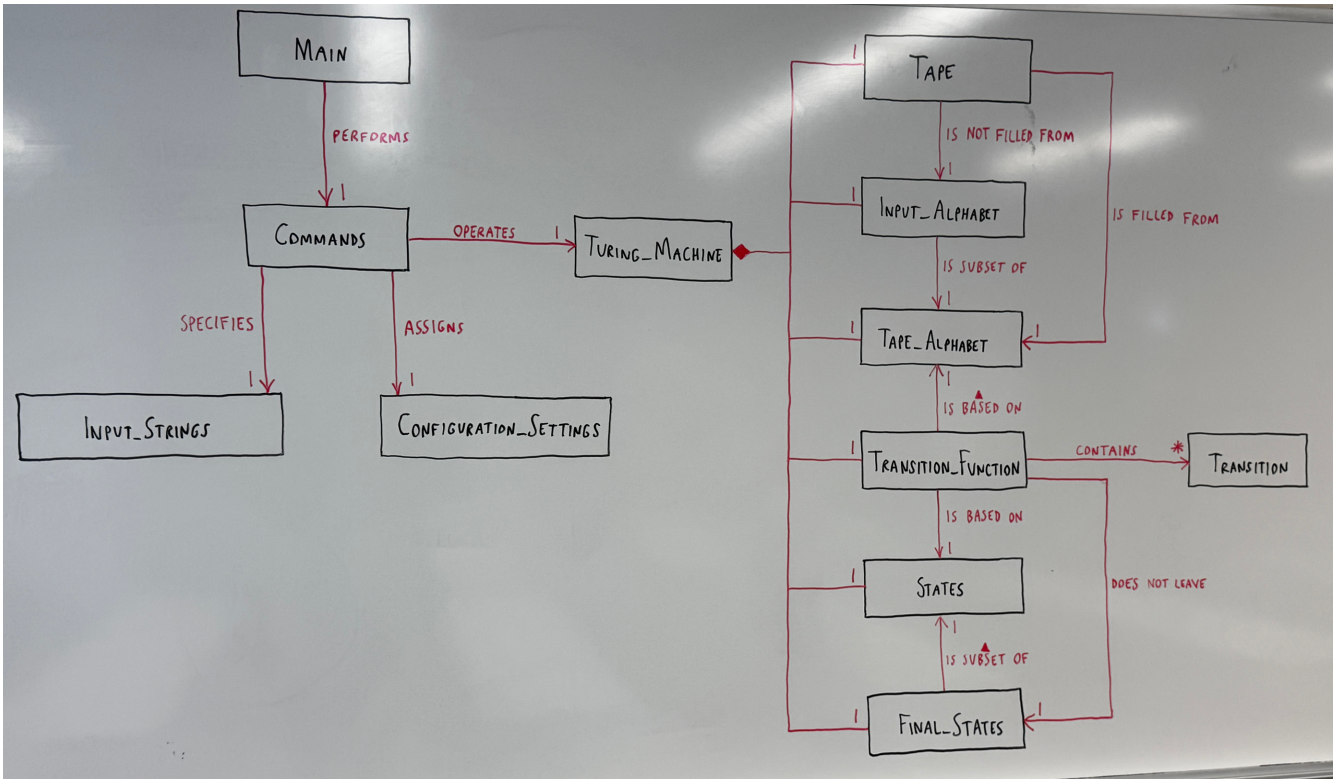


Figure 2.1: Complete TM UML Diagram.

The following image is of the Turing Machine Class's Attributes (green) and Methods (blue). These will be explained in more detail in sections **3.10.3 and 3.10.4.**

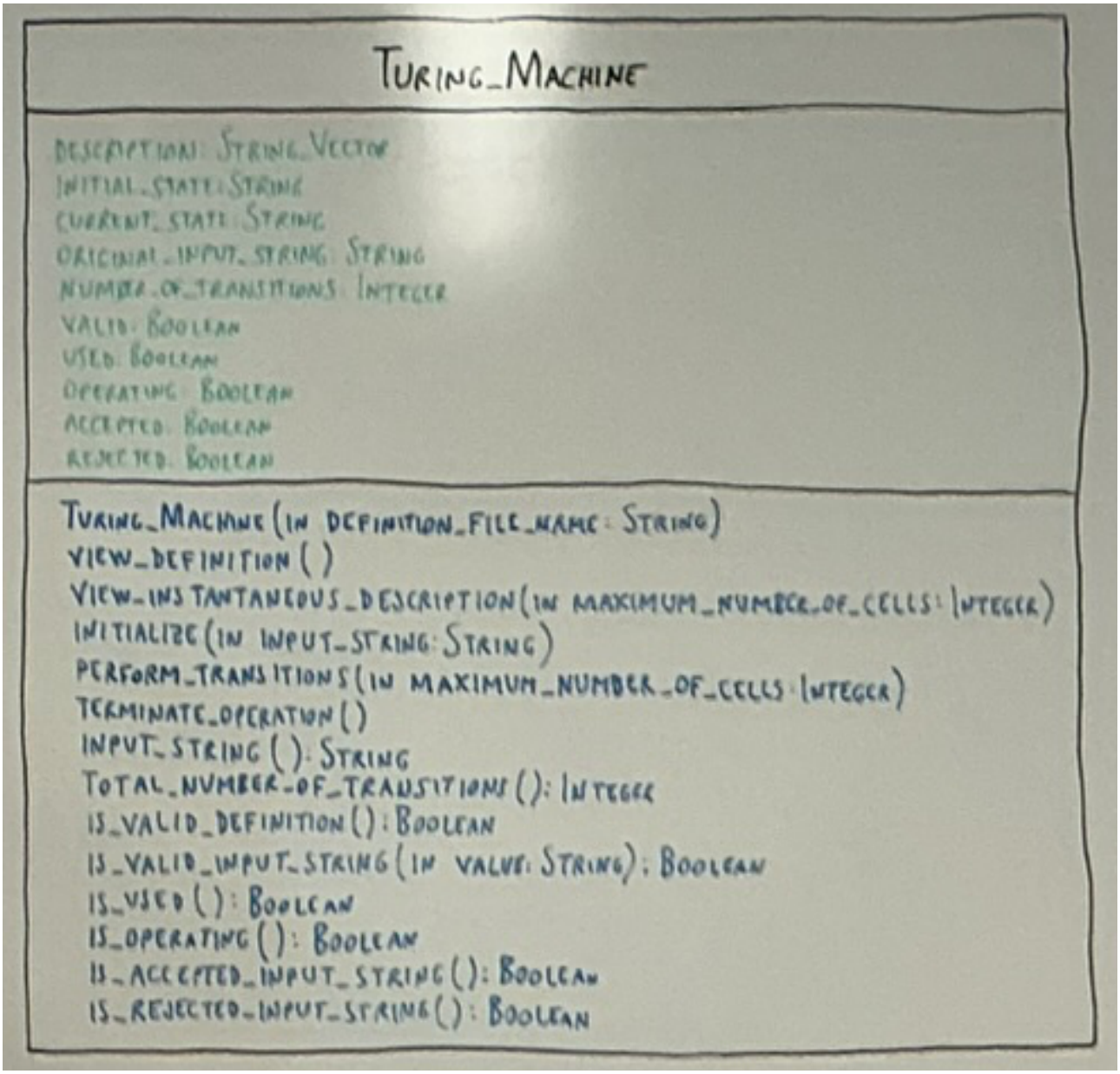


Figure 2.2: TM Table.

3.0 Data Dictionary

Each of the following subsections describes a class used in the Turing Machine application. For each class, a brief description is provided along with its associations, attributes, and methods (including pseudocode for complex operations when appropriate).

3.1 Input_Alphabet

3.1.1 Description

The *Input_Alphabet* class represents the set of characters that are valid as input symbols for the Turing Machine. These are the characters that may appear in the input strings processed by the machine, and they form a subset of the tape alphabet. This class is responsible for loading the input alphabet from the Turing machine definition file, validating its correctness, and supporting operations that determine whether a given character is part of the alphabet. The lifetime of an *Input_Alphabet* object spans the duration of the Turing Machine's operation; it is created during initialization and remains constant throughout execution.

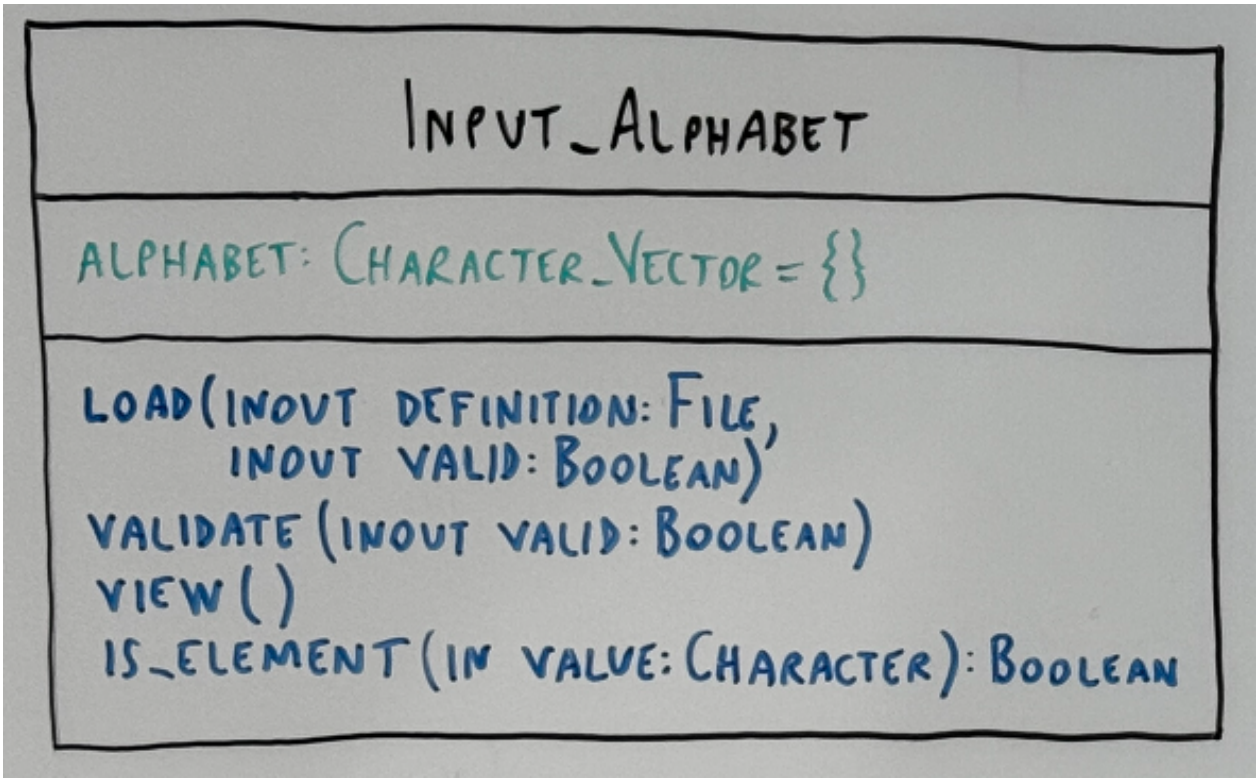


Figure 3.1.1.1: Input Alphabet Table

3.1.2 Associations

The *Input_Alphabet* class is associated with the *Turing_Machine* class, which aggregates and manages it as one of its core components. During initialization, the Turing Machine loads the input alphabet as part of the machine definition and uses it throughout the operation of the system.

The *Input_Alphabet* class also has a direct association with the *Tape_Alphabet* class. Specifically, the input alphabet must be a subset of the tape alphabet, meaning all input symbols must also be valid tape symbols. This relationship is enforced during validation.

Additionally, the *Tape* class references the *Input_Alphabet* indirectly to ensure that the blank character used on the tape does not appear in the input alphabet, preserving the correctness of machine initialization and execution.

3.1.3 Attributes

alphabet: character_vector = {}

A vector or list of characters representing the input alphabet. This set is initialized from the definition file and used throughout the Turing Machine's execution for input validation and runtime checks.

3.1.4 Methods

load(inout definition: File, inout valid: Boolean)

Reads the list of input alphabet characters from the Turing machine definition file. If an invalid or duplicate character is found, or the format is incorrect, an error message is displayed and valid is set to false.

validate(inout valid: Boolean)

Verifies that each character in the input alphabet is a printable symbol and does not include the reserved blank character. If any violation is found, valid is set to false and an error message is displayed.

view()

Displays the characters in the input alphabet to the user, typically as part of a diagnostic or View command.

is_element(in value: Character): Boolean

Returns true if the given character exists in the input alphabet; otherwise, returns false. This function is used in various stages of machine setup and input validation.

3.2 Tape_Alphabet

3.2.1 Description

The *Tape_Alphabet* class represents the set of all characters that may legally appear on the Turing Machine's tape during execution. This includes the input alphabet symbols, special marker characters (such as overwritten symbols), and the blank character used to initialize or extend the tape. The tape alphabet is loaded from the definition file and is used throughout the lifecycle of the Turing Machine to validate transitions, tape content, and runtime operations. An object of this class is created during the initialization phase and remains active throughout execution.

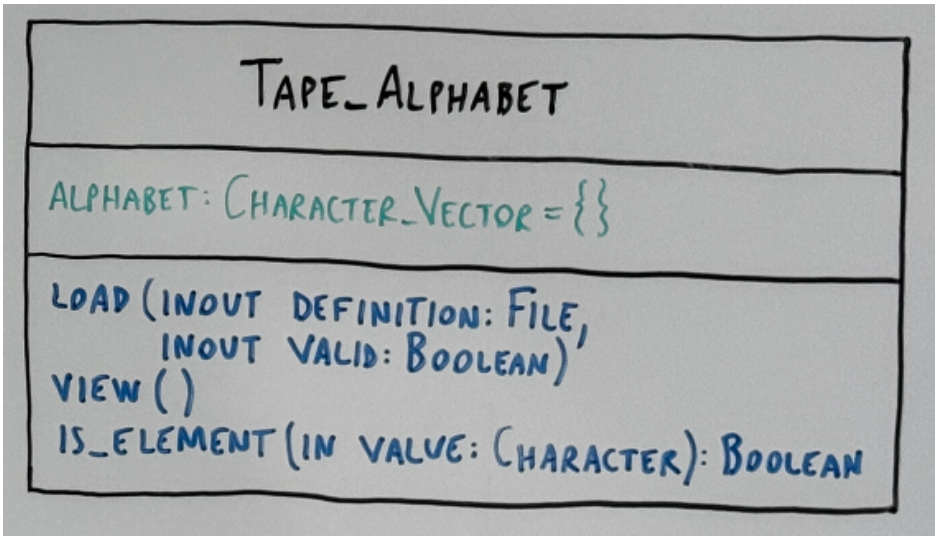


Figure 3.2.1.1: Tape Alphabet Table

3.2.2 Associations

The *Tape_Alphabet* class is aggregated by the *Turing_Machine* class, which instantiates and manages it as part of the machine definition.

- The *Input_Alphabet* class is directly associated with *Tape_Alphabet*, as it must be a strict subset of the tape alphabet. This subset relationship is enforced during validation.

- The *Transition_Function* class depends on the *Tape_Alphabet* to verify that all symbols referenced in transitions (read, write) are valid tape characters.

- The *Tape* class is also directly associated with *Tape_Alphabet*, as the tape is initialized, updated, and extended using characters from the tape alphabet. The blank character used on the tape must be a member of this class's alphabet.

3.2.3 Attributes

alphabet: character_vector = {}

A dynamically constructed collection of all valid characters that can appear on the Turing Machine's tape, including the input characters, blank symbol, and any intermediate symbols used during transitions.

3.2.4 Methods

load(inout definition: File, inout valid: Boolean)

Loads the tape alphabet from the definition file. Ensures that each character is printable and unique. If an invalid or duplicate character is found, or the syntax of the definition is incorrect, an error message is displayed and valid is set to false.

view()

Displays the tape alphabet characters to the user, for debugging or inspection.

is_element(in value: Character): Boolean

Checks whether a specific character exists in the tape alphabet. Returns true if the character is present; otherwise, returns false. This is used for validation during transitions and tape operations.

3.3 States

3.3.1 Description

The *States* class represents the complete set of states defined in the Turing Machine. Each state is uniquely identified by a name and plays a role in controlling the flow of execution during input processing. States are used in the transition function to define machine behavior. The list of state names is loaded from the machine definition file during initialization. The set of states remains fixed throughout the lifetime of the Turing Machine once loaded.

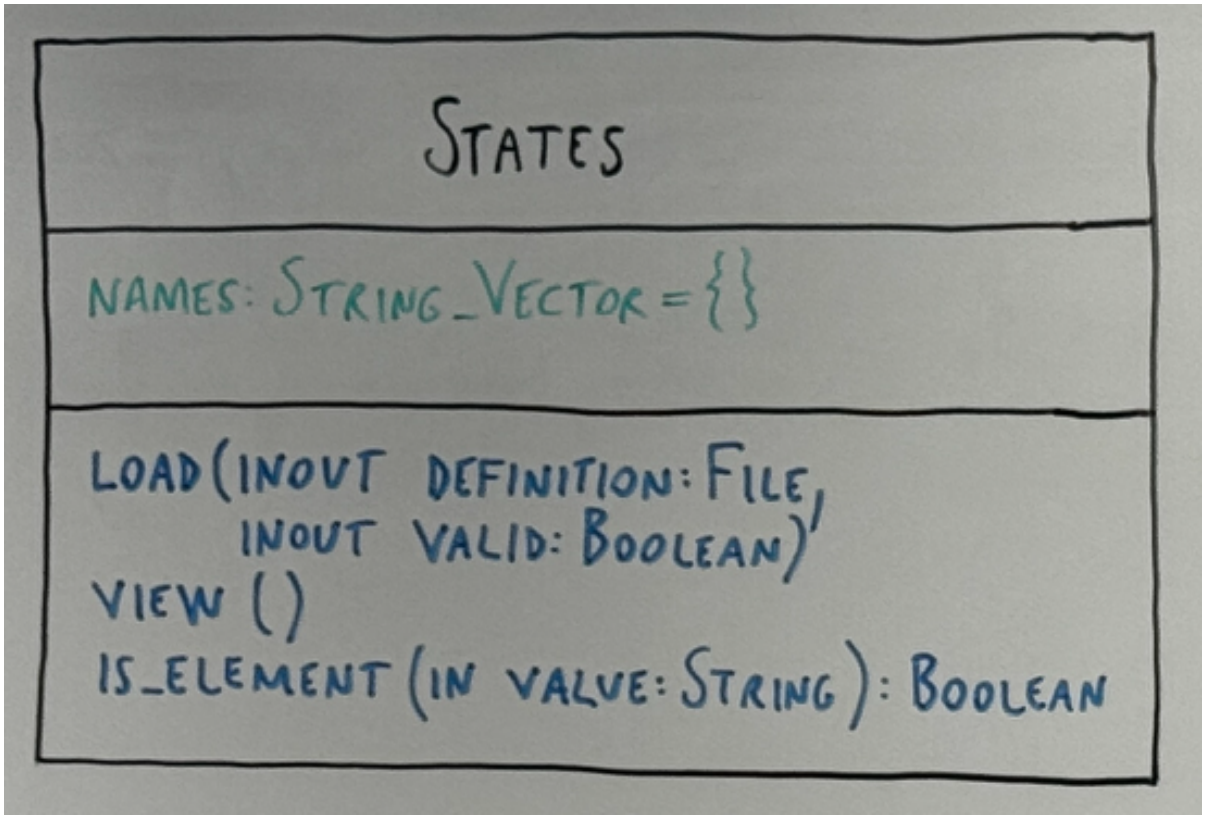


Figure 3.3.1.1: States Table

3.3.2 Associations

The *States* class is aggregated by the *Turing_Machine* class, which manages and initializes it at startup.

- The *Transition_Function* class is directly associated with *States*, as each transition references a current state and a next state, both of which must be valid members of this set.

- The *Final_States* class is a subset of *States*. During validation, each final state is checked to ensure it exists in the full set of states.

3.3.3 Attributes

names: string_vector = {}

A collection of unique state names defined for the Turing Machine. These strings represent the possible states the machine may enter during its operation.

3.3.4 Methods

load(inout definition: File, inout valid: Boolean)

Loads the list of state names from the Turing Machine definition file. Verifies uniqueness and correct formatting. If an error is found, an appropriate message is displayed and valid is set to false.

view()

Displays the list of state names for inspection.

is_element(in value: String): Boolean

Returns true if the specified state name is present in the set of states, false otherwise. This is used during validation of transitions and final states.

3.4 Final_States

3.4.1 Description

The *Final_States* class represents the subset of states that are designated as accepting or halting states in the Turing Machine. If the machine reaches one of these states during execution, it stops processing the input string and may signal acceptance. This class loads the set of final states from the definition file and validates that each belongs to the full set of defined states. Once initialized, the set of final states remains fixed for the duration of the Turing Machine's operation.

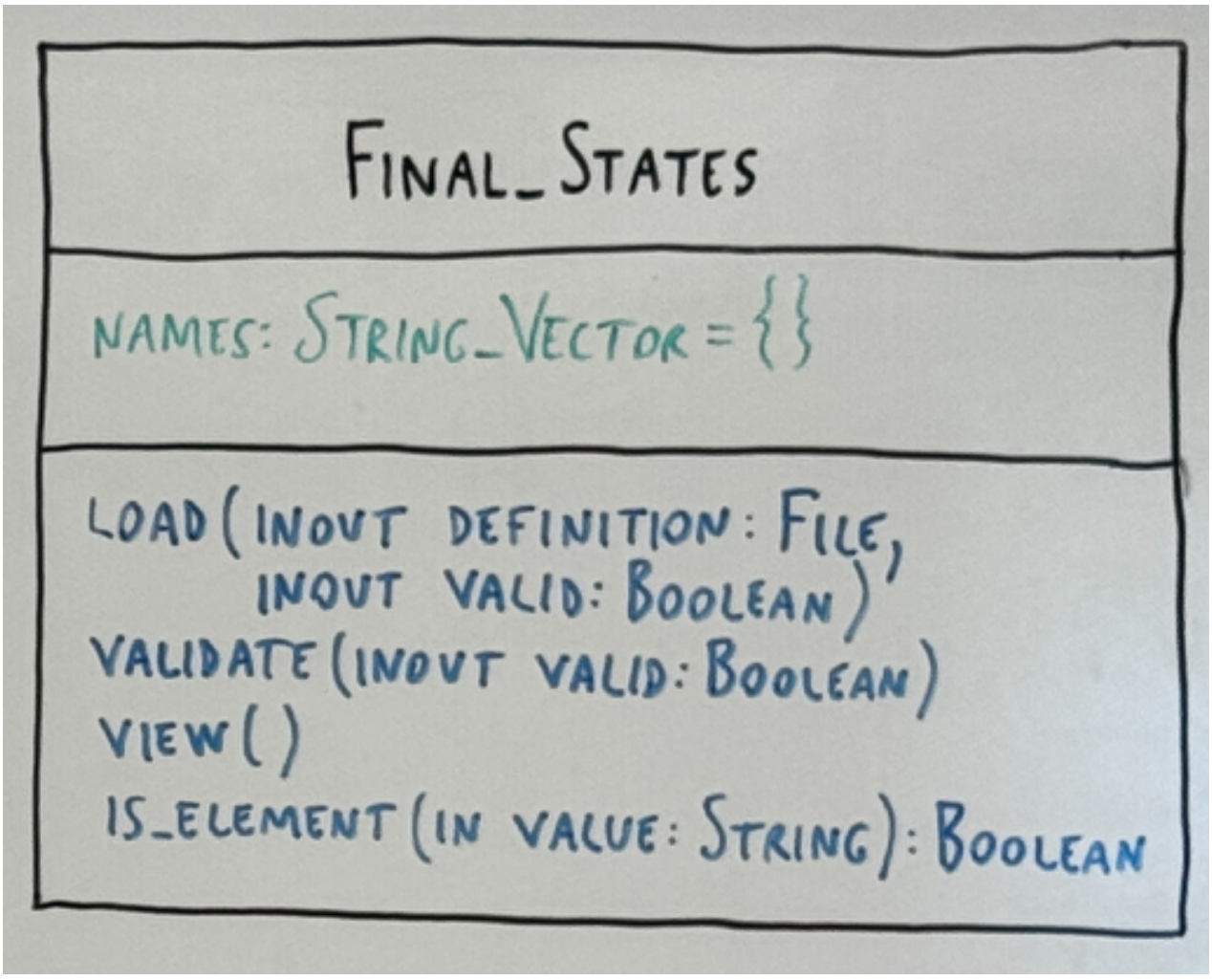


Figure 3.4.1.1: Final States Table.

3.4.2 Associations

The *Final_States* class is aggregated by the *Turing_Machine* class, which instantiates and manages it as part of the machine's definition.

- It is a subset of the *States* class. During validation, each final state must be confirmed as a valid member of the overall state set.

- The *Transition_Function* class is associated with *Final_States* in that transitions do not leave a final state — i.e., the final states are terminal and no further transitions should occur once the machine enters one.

3.4.3 Attributes

names: string_vector = {}

A collection of strings representing the names of the final (accepting) states. These are a proper subset of the total states defined in the Turing Machine.

3.4.4 Methods

load(inout definition: File,

inout valid: Boolean)

Loads the list of final state names from the Turing Machine definition file. Ensures proper formatting and uniqueness. If an error is detected, an error message is displayed and valid is set to false.

validate(inout valid: Boolean)

Confirms that every final state listed is present in the overall set of machine states. If any state is not found in the States class, an error is reported and valid is set to false.

view()

Displays the names of the final states for inspection.

is_element(in value: String): Boolean

Returns true if the specified state name exists in the set of final states; otherwise, returns false.

3.5 Transition

3.5.1 Description

The *Transition* class defines a single transition rule for the Turing Machine. A transition specifies what the machine should do when it is in a certain state and reads a specific character from the tape. It includes the source state, the character to read, the destination state, the character to write, and the direction to move the tape head. Each instance of the *Transition* class represents one such rule, and all transition rules collectively define the behavior of the machine during execution.

A *Transition* object is typically created when parsing the transition function from the definition file and remains unchanged throughout the operation of the machine.

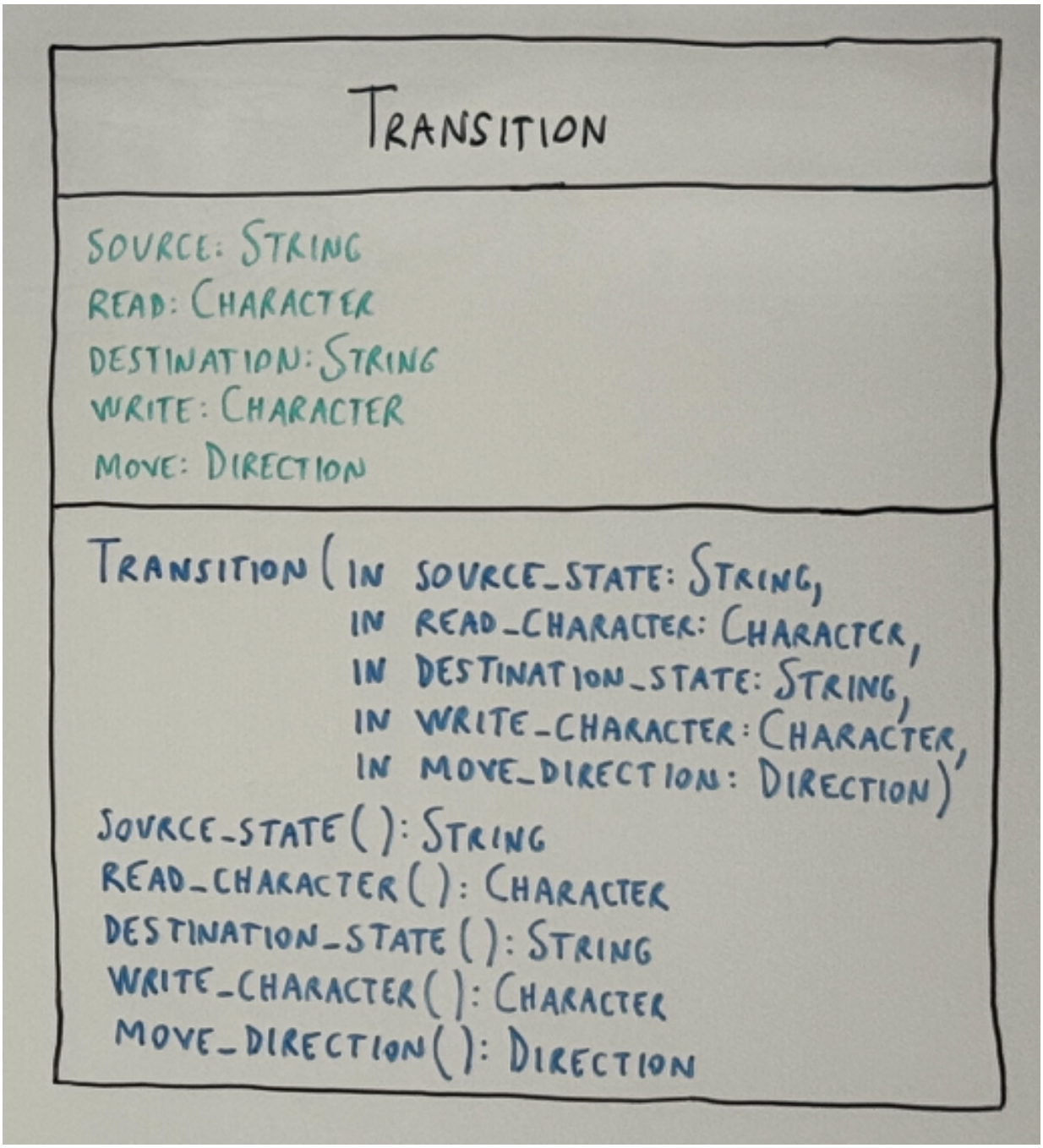


Figure 3.5.1.1: Transition Table.

3.5.2 Associations

The *Transition* class is contained by the *Transition_Function* class, which manages a collection of transitions and handles queries related to machine behavior.

- The *Turing_Machine* class does not directly associate with *Transition*, but relies on *Transition_Function* to retrieve applicable transitions.

- Each transition depends on valid references to the states and tape alphabet characters that it uses for execution.

3.5.3 Attributes

source: String

The current state from which the transition originates.

read: Character

The character that must be read from the tape in order for this transition to apply.

destination: String

The state to which the machine will transition after performing the action.

write: Character

The character that will be written to the tape at the current position.

move: Direction

The direction (left or right) in which the tape head will move after writing. Typically represented as an enumerated type with values L and R.

3.5.4 Methods

transition(in source_state: String, in read_character: Character, in destination_state: String, in write_character: Character, in move_direction: Direction)

Constructor method to initialize the transition with all required values.

source_state(): String

Returns the name of the source state for this transition.

read_character(): Character

Returns the character that must be read for this transition to trigger.

destination_state(): String

Returns the name of the destination state for this transition.

write_character(): Character

Returns the character that will be written to the tape by this transition.

move_direction(): Direction

Returns the direction in which the tape head will move when this transition is executed.

3.6 Transition_Function

3.6.1 Description

The *Transition_Function* class represents the full set of transition rules that define the behavior of the Turing Machine. Each transition is an instance of the *Transition* class, and collectively they determine how the machine changes state, modifies the tape, and moves the tape head in response to the symbols it reads.

The *Transition_Function* class is responsible for loading these rules from the Turing Machine definition file, validating their correctness, displaying them for user reference, and providing lookup functionality to locate applicable transitions during machine execution. It does not store any data beyond the set of *Transition* objects, making it purely functional in nature.

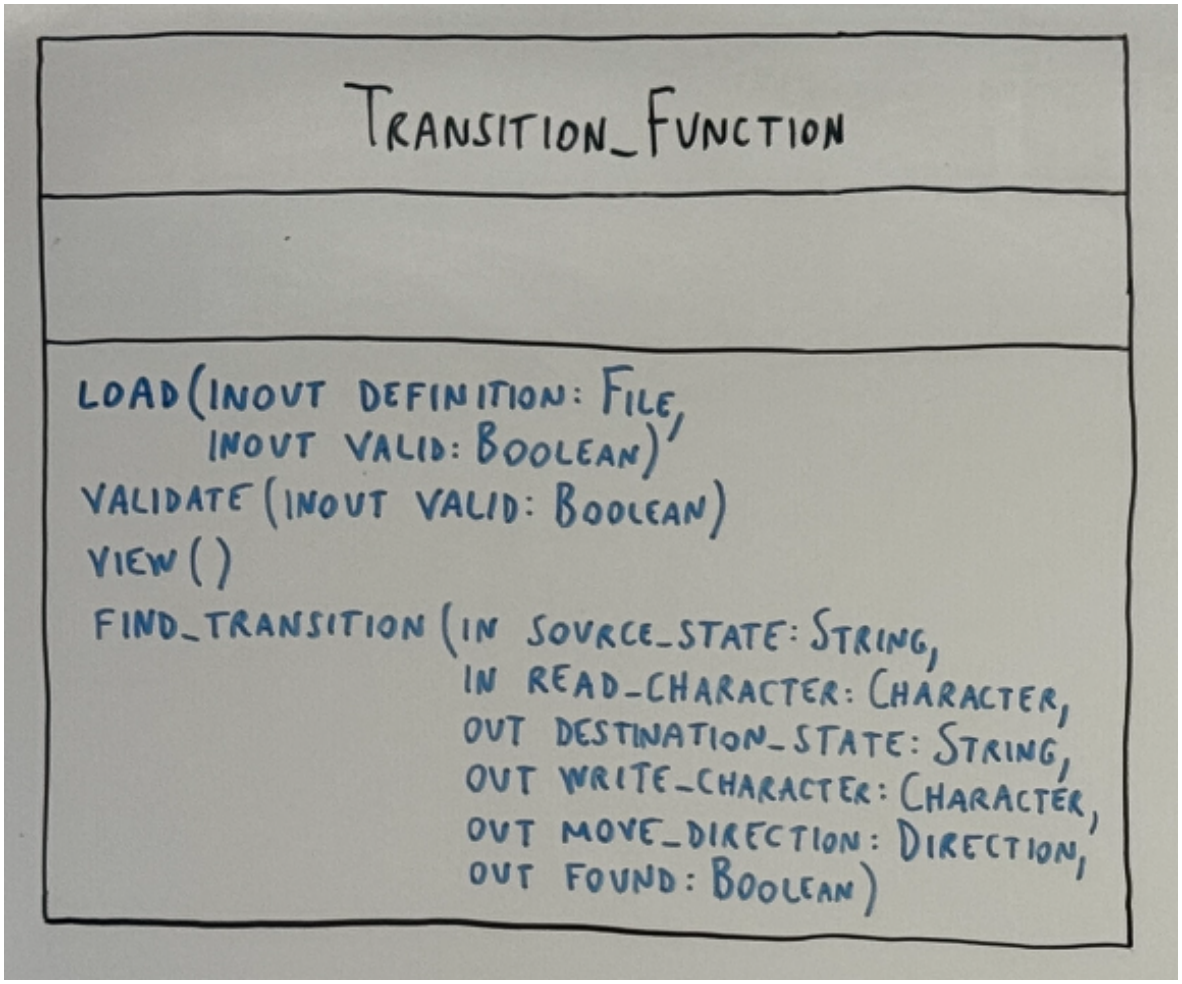


Figure 3.6.1.1: Transition Function Table.

3.6.2 Associations

The *Transition_Function* class is aggregated by the *Turing_Machine* class, which invokes its methods during initialization and simulation.

- It contains a collection of *Transition* objects that it manages and queries.
- It depends on the *States* class to validate that all source and destination states in its transitions are valid.
- It also checks that transitions do not leave the *Final_States*, as entering a final state should terminate machine execution.

3.6.3 Attributes

This class has no standalone data attributes beyond its internal collection of *Transition* objects. All operational data is encapsulated within those objects.

3.6.4 Methods

load(inout definition: File, inout valid: Boolean)

Reads transition rules from the Turing Machine definition file and constructs corresponding Transition objects. If any rule is improperly formatted or references invalid states or characters, an error message is displayed and valid is set to false.

validates(inout valid: Boolean)

Checks all transitions to ensure they conform to the rules of the machine. Verifies that all states and symbols used are valid, and that no transitions originate from final states. If any inconsistency is found, valid is set to false.

view()

Displays the entire list of transition rules for inspection, formatted to reflect source state, read symbol, destination state, write symbol, and direction of movement.

find_transition(in source_state: String,

in read_character: Character,
out destination_state: String,
out write_character: Character,
out move_direction: Direction,
out found: Boolean)

Searches for a transition matching the given source state and read character. If a matching transition is found, the output parameters are populated accordingly and found is set to true. If no match is found, found is set to false.

3.7 Tape

3.7.1 Description

The tape of a Turing machine consists of an ordered sequence of cells, indexed starting at 0, which may grow to any size needed up to the limit of storage during operation of the machine on an input string. Each cell contains a character in the tape alphabet. An input string is stored in the lowest numbered tape cells at the beginning of operation, and all other tape cells initially contain the blank character. The current cell starts at the first cell on the tape. In performing a transition of the Turing machine, the character contained in the current cell may be read and written, and the current cell may be moved one cell to the left or right. The tape exists only as part of a Turing machine.

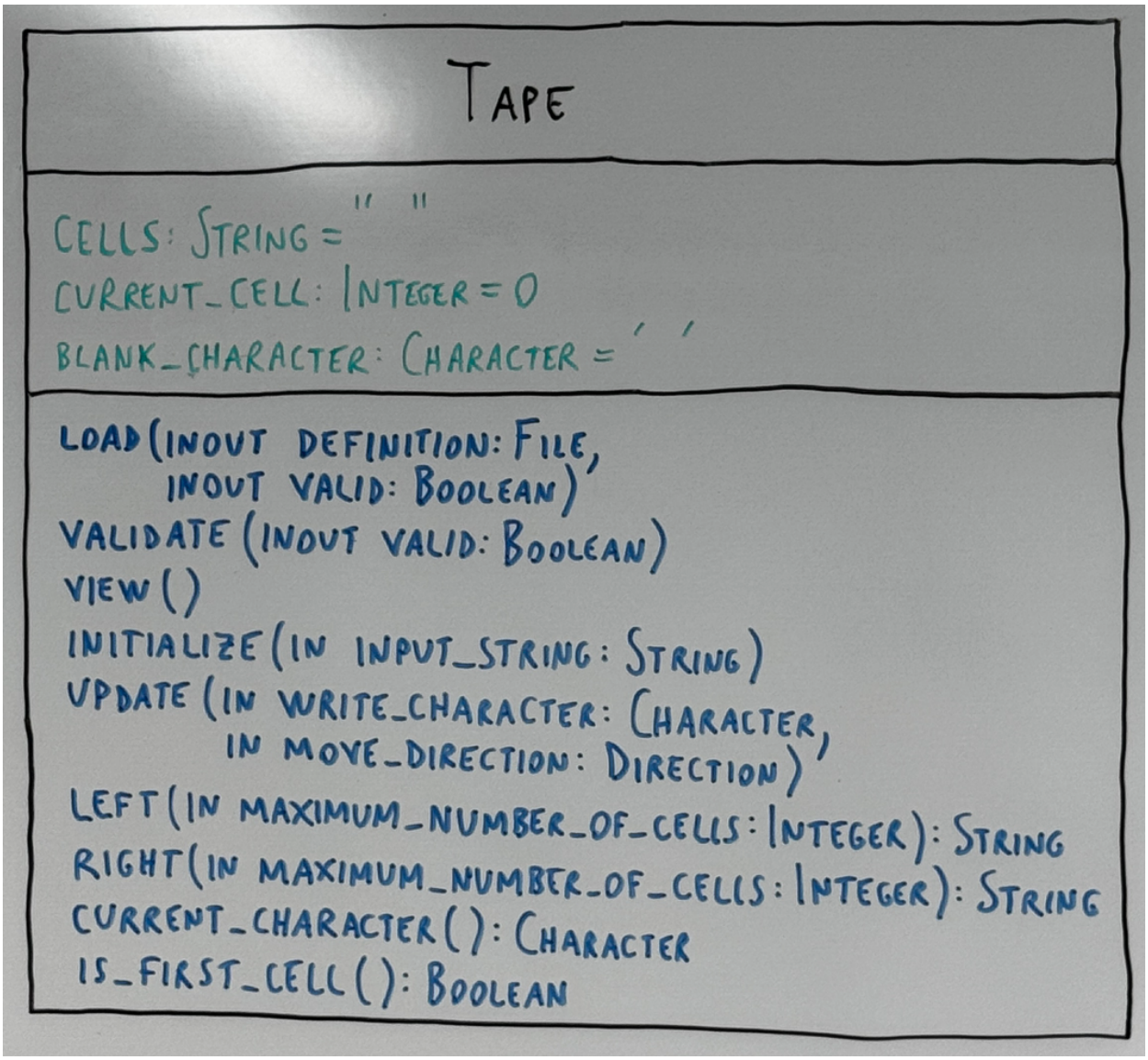


Figure 3.7.1.1: Tape Class Table.

3.7.2 Associations

The class Tape is a component of the class Turing_Machine, receiving messages delegated to it by the Turing machine. The association is not filled from with the class Input_Alphabet is used to validate that the blank character for initialization and extension of the Turing machine tape is not in the input alphabet. The association is filled from with the class Tape_Alphabet is used to validate that the blank character for initialization and extension of the Turing machine tape is in the tape alphabet.

3.7.3 Attributes

Cells:String= “”

The attribute cells is a dynamically growing character string containing the Turing machine tape. In performing an update, the tape may be extended by appending a blank character.

Current_cell:Integer = 0

The index of the current cell on the Turing machine tape is stored in the attribute current_cell

Blank_character:Character= ‘’

The blank character used to initialize and extend the Turing machine tape is contained in the attribute blank_character.

3.7.4 Methods

load(inout definition:File, inout valid:Boolean)

The method load reads the blank character from the Turing machine definition file. If the blank character is reserved or not printable, or the next keyword does not follow it in the file, an error message is displayed and valid is set to false.

validate(inout valid:Boolean)

The method validate determines if the blank character of the Turing machine is in the tape alphabet but not the input alphabet. If the blank character is in the input alphabet or is not in the tape alphabet, an error message is displayed and valid is set to false.

view()

The method view displays the blank character of the Turing machine.

initialize(in input_string:String)

The method initialize sets the Turing machine tape to the input string followed by a blank character, replacing the previous contents of the tape. The current cell is set to the first cell on the tape, indicated by the index 0.

update(in write_character:Character, in move_direction:Direction)

The method update first determines if the update of the Turing machine tape is possible. The method returns if a left move is specified from the first cell. If a right move is specified from the last cell, a blank character is appended to the tape. If no storage is available for this character, an out of storage error will be thrown. Assuming that the update may be performed, the character to write on the tape is stored in the current cell, replacing the previous character in that cell. To move the current cell one cell to the left, the index is decremented, or to move the current cell one cell to the right, the index is incremented.

Pseudocode (update):

```
method update(in write_character:Character,
                in move_direction:Direction) is
begin
 if move_direction = L and current_cell = 0 then
        return;
 end if;
 if move_direction = R and current_cell = cells.length() - 1 then
        cells.append(blank_character);
 end if;
 cells[current_cell] := write_character;
 if move_direction = L then
         current_cell := current_cell - 1;
 else
        current_cell := current_cell + 1;
```

```
 end if;

end update;
```

left(in maximum_number_of_cells:Integer):String

The method left returns a character string of up to the maximum number of cells from the Turingmachine tape to the left of the current cell, excluding that cell. The length of the string will be less than the maximum if there are fewer cells to the left of the current cell. If the string is truncated from the tape, the reserved character '<' will be added to the beginning of the string.

right(in maximum_number_of_cells:Integer):String

The method right returns a character string of up to the maximum number of cells from the Turing machine tape to the right of the current cell, including that cell. The length of the string will be less than the maximum if there are fewer cells to the right of the current cell up to the rightmost nonblank character. If the string is truncated from the tape, the reserved character '>' will be added to the end of the string.

current_character():Character

The method current_character returns the character contained in the current cell on the Turing machine tape.

is_first_cell():Boolean

The method is_first_cell returns a value of true if the current cell on the Turing machine tape is the first cell, indicated by the index 0. Otherwise, it returns a value of false.

3.8 Input_Strings

3.8.1 Description

The *Input_Strings* class manages the list of input strings that may be processed by the Turing Machine. These strings represent candidate inputs for acceptance or rejection based on the current machine definition. The list can be populated from a file or directly by the user during runtime. It supports insertion, deletion, lookup, and saving of the input strings.

This class maintains state throughout the Turing Machine's lifecycle, tracking both the list of strings and whether any modifications have occurred. Changes to the list set a flag used to determine if the data should be written back to the file.

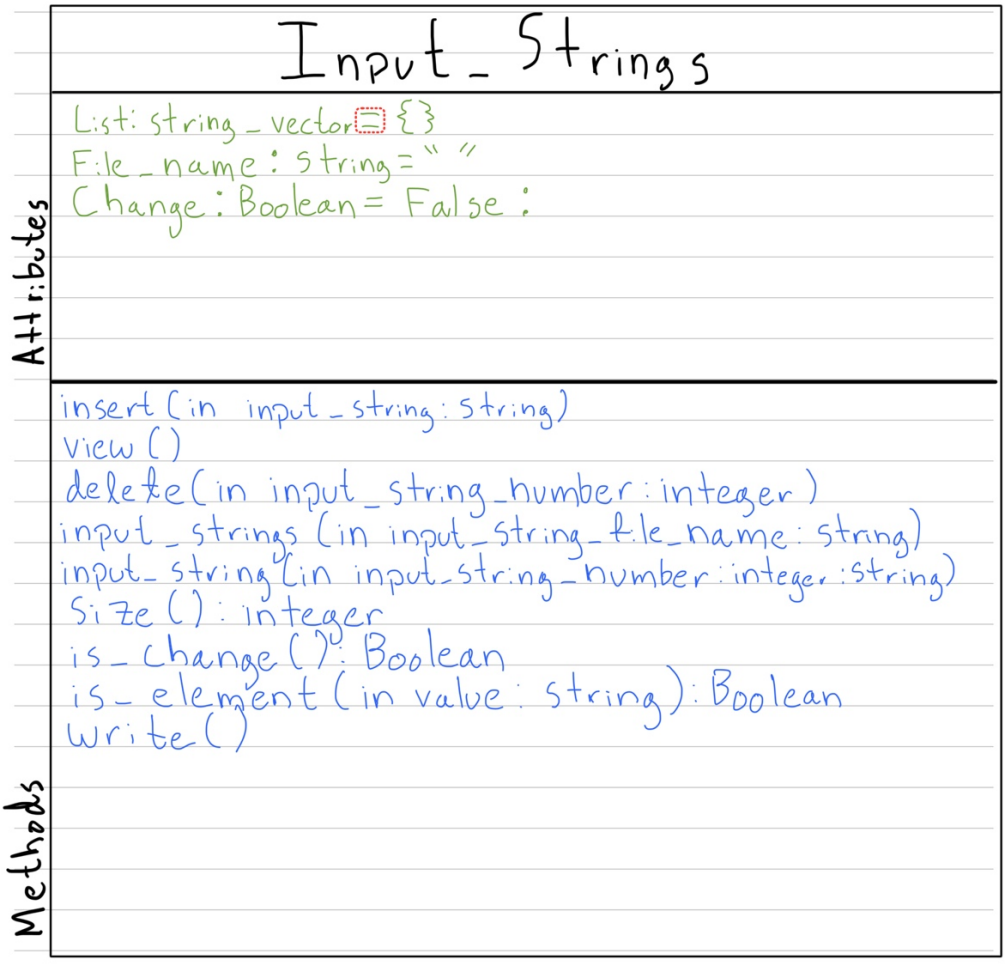


Figure 3.8.1.1: Input String Table.

3.8.2 Associations

The *Input_Strings* class is used by the *Commands* class, which references it to perform operations such as insert, delete, view, and run. In the UML diagram, the relationship is labeled as specifies, indicating a behavioral dependency rather than ownership.

The *Turing_Machine* class does not directly interact with *Input_Strings*; it is accessed through the Commands interface during user interaction.

3.8.3 Attributes

list: string_vector = {}

A list of input strings currently stored in memory, representing all candidate strings available for execution.

file_name: string = ""

The name of the file from which input strings are loaded or to which they are saved.

change: boolean = false

A flag indicating whether the list of input strings has been modified during execution. This is used to determine if the strings need to be rewritten to the file upon program termination

3.8.4 Methods

insert(in input_string: String)

Adds a new input string to the list and sets the change flag to true.

view()

Displays all stored input strings, each with its index number for easy reference during other operations.

delete(in input_string_number: Integer)

Removes an input string from the list based on its index number. If successful, sets the change flag to true.

input_strings(in input_string_file_name: String)

Loads input strings from the specified file into the internal list. This replaces any previously stored strings.

input_string(in input_string_number: Integer): String

Returns the string located at the given index in the list. This is used during Run and Quit commands to reference specific strings.

size(): Integer

Returns the number of input strings currently stored in the list.

is_change(): Boolean

Returns the value of the change flag, indicating whether the list has been modified since it was loaded.

is_element(in value: String): Boolean

Returns true if the specified string exists in the current list of input strings, false otherwise.

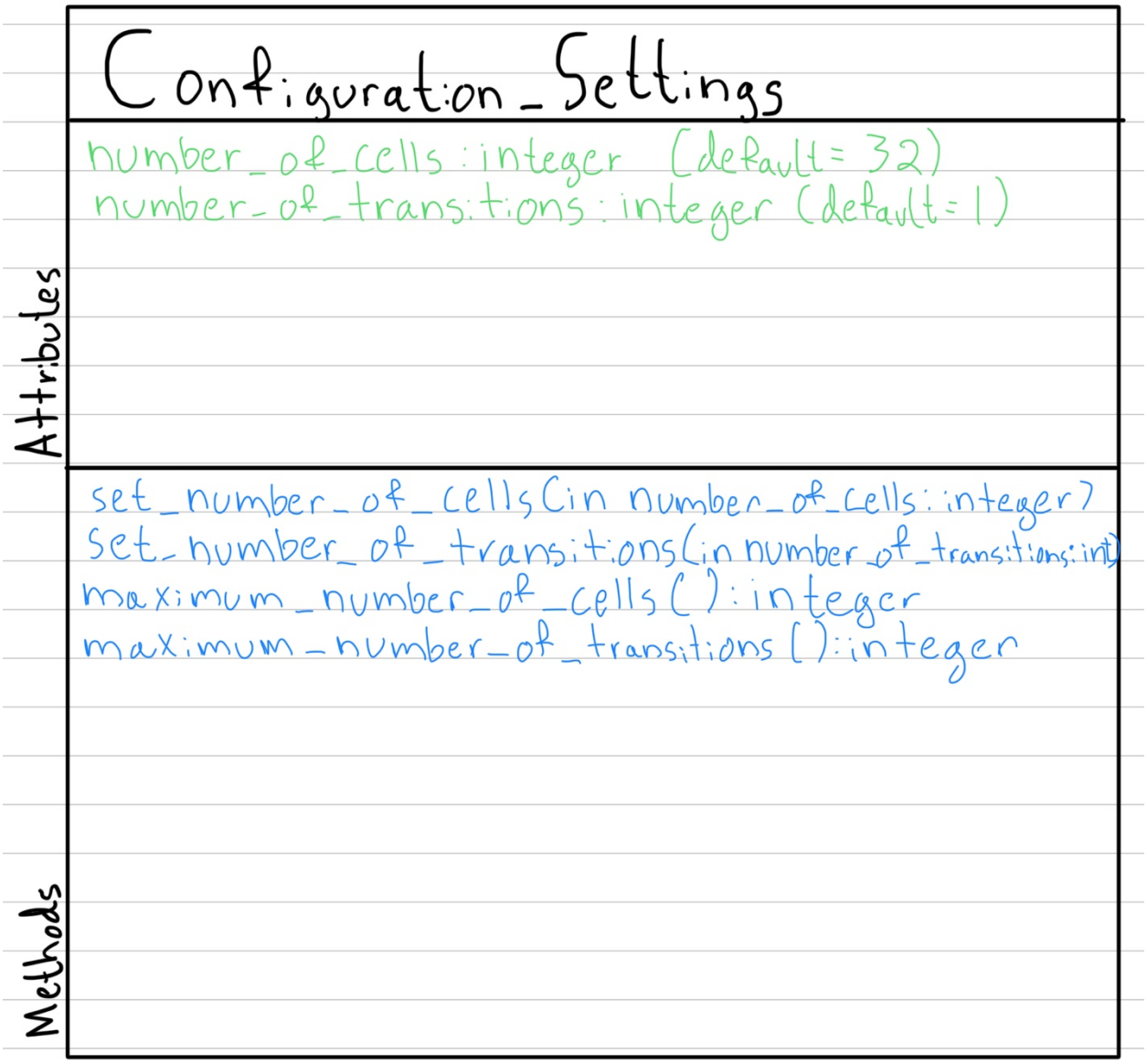
write()

Writes the current list of input strings to the file specified by file_name, if any changes have been made.

3.9 Configuration_Settings

3.9.1 Description

The *Configuration_Settings* class defines runtime parameters that affect how the Turing Machine operates and how its execution is displayed to the user. These settings include the maximum number of tape cells to show when displaying the machine's state and the maximum number of transitions the machine should perform before halting. These values can be set by the user via commands and are used during execution and output formatting. The settings persist during a single session and are reset to default values at startup.



3.9.1.1: Configuration Settings Table.

3.9.2 Associations

The *Configuration_Settings* class is used by the *Commands* class, which assigns its values based on user input. This is a behavioral association rather than structural aggregation.

The *Turing_Machine* class does not directly associate with *Configuration_Settings*, but instead receives runtime configuration indirectly through *Commands*.

3.9.3 Attributes

number_of_cells: Integer = 32

Specifies the maximum number of tape cells to display on either side of the tape head during a transition. Used in truncating long instantaneous descriptions.

number_of_transitions: Integer = 1

Defines the maximum number of transitions the Turing Machine will perform when executing a single input string. Execution halts if this limit is reached.

3.9.4 Methods

set_number_of_cells(in number_of_cells: Integer)

Sets the value for number_of_cells. Typically invoked through the Set command.

set_number_of_transitions(in number_of_transitions: Integer)

Sets the maximum number of transitions allowed during machine execution. Also set via the Set command.

maximum_number_of_cells(): Integer

Returns the current value of number_of_cells.

maximum_number_of_transitions(): Integer

Returns the current value of number_of_transitions.

3.10 Turing_Machine

3.10.1 Description

The *Turing_Machine* class encapsulates the core behavior and state of the Turing Machine simulator. It brings together all components—alphabet definitions, transition rules, tape structure, and current execution state—to simulate the operation of a formal Turing Machine. This class is responsible for loading the machine definition, initializing the tape and input string, performing transitions, displaying state, and tracking acceptance or rejection of input.

A *Turing_Machine* object is initialized with a definition file and maintains all data necessary to execute multiple input strings during a program session. It is central to the machine's lifecycle, responding to configuration settings and input provided through the *Commands* class.

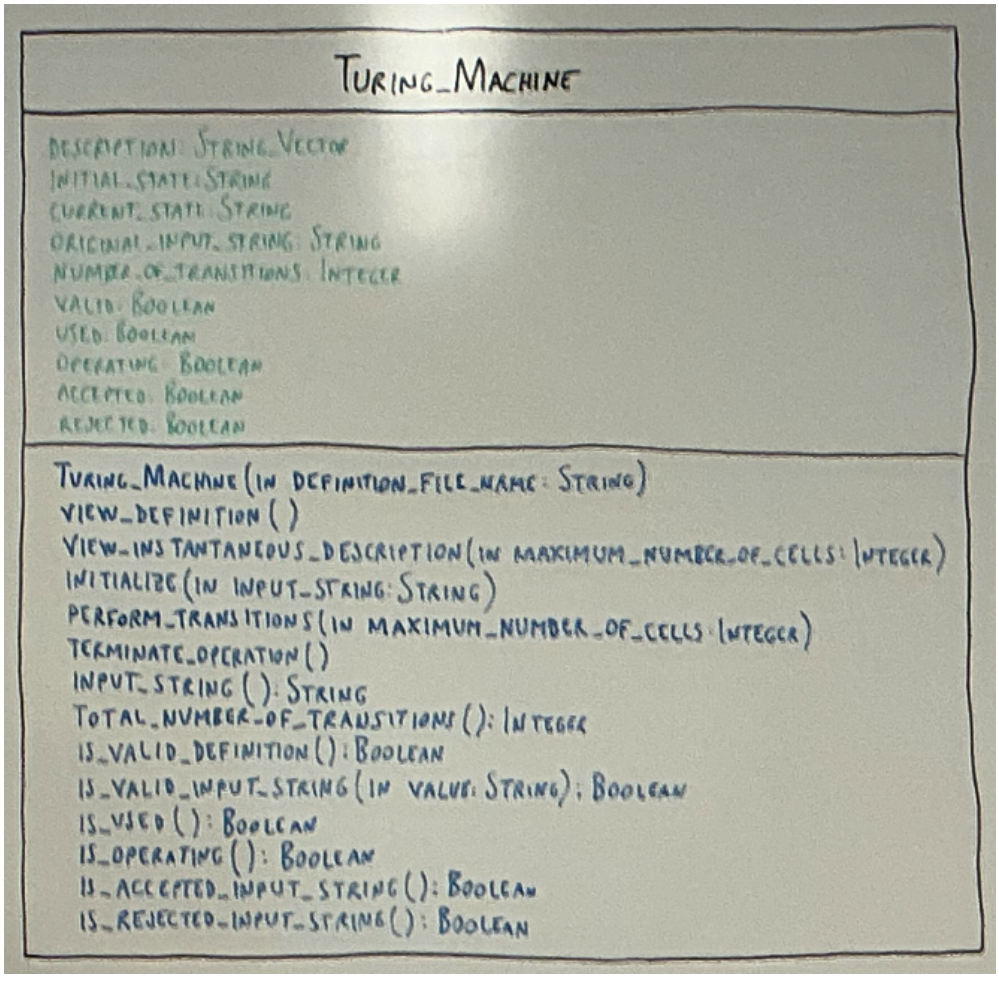


Figure 3.10.1.1: Turing Machine Table.

3.10.2 Associations

The *Turing_Machine* class is operated by the *Commands* class, which interacts with it to perform all user-requested operations such as Run, Set, View, and Quit.

Internally, the *Turing_Machine* class aggregates the following classes, each representing a fundamental component of the machine:

- *Tape*: holds and updates the current tape configuration.
- *Input_Alphabet*: defines the valid characters for input strings.
- *Tape_Alphabet*: defines all symbols that can appear on the tape.
- *Transition_Function*: contains the set of transition rules and logic.
- *States*: defines all possible machine states.
- *Final_States*: subset of States that indicate halting/acceptance conditions.

3.10.3 Attributes

description: string_vector

A list of descriptive lines (comments) from the machine definition file, typically used to explain the machine's purpose or behavior.

initial_state: string

The name of the state where the machine begins execution.

current_state: string

The machine's current state during execution, updated as transitions occur.

original_input_string: string

Stores the original input string given to the machine before any modifications during simulation.

number_of_transitions: integer

Tracks the number of transitions executed for the current input string.

valid: boolean

Indicates whether the machine definition is valid and complete. This flag is used to prevent execution if the machine is not properly configured.

used: boolean

Set to true once the machine has been initialized with an input string and has been operated on at least once.

operating: boolean

Indicates whether the machine is currently in an operational state (actively processing a string).

accepted: boolean

True if the current input string has been accepted by the machine (i.e., reached a final state.

rejected: boolean

True if the machine encountered an undefined transition.

3.10.4 Methods

turing_machine(in definition_file_name: string)

Constructor method. Initializes all components by reading from the specified definition file. Performs validation of each component.

view_definition()

Displays the description and definition of the Turing Machine, including its alphabets, states, transitions, and blank character.

view_instantaneous_description(in maximum_number_of_cells: integer)

Displays the current instantaneous configuration of the machine, including the tape contents and current state, truncated according to the configuration setting.

initialize(in input_string: string)

Initializes sets up all of the required parameters parameters needed for execution of the turing machine. Like setting up tape and machine state based on the given input string file (anbn.str). As well as reads from the definition and string files (anbn.def, and anbn.str) to configure the transition function, accepted states, rejected states etc. This method sets up the parameters based on the contents of the definiton files.

perform_transitions(in maximum_number_of_cells: integer)

Simulates the Turing Machine's operation by executing transitions up to the configured limit. Updates the current state, tape, and flags accordingly. Consider this the "brain" of the operating part of this application. Here is where the performing turing machine-like logic lies. In-case there is an error with an undefined transition an error will be thrown. When accepting and rejecting states are reached based on the input string, a message will be printed to the console with that information.

terminate_operation()

Ends the current machine run with the Quit [Q] command. Updates the operating, accepted, and rejected flags based on the machine's operating state at the time of "quit".

input_string(): string

Returns the original input string for reference or display.

total_number_of_transitions(): integer

Returns the total number of transitions performed during the most recent run.

is_valid_definition(): boolean

Returns whether the machine definition is valid.

is_valid_input_string(in value: string): boolean

Returns true if the input string contains only characters in the input alphabet.

is_used(): boolean

Returns whether the machine has been initialized and run at least once.

is_accepted_input_string(): boolean

Returns true if the current input string was accepted.

is_rejected_input_string(): boolean

Returns true if the current input string was rejected.

3.11 Commands

3.11.1 Description

The *Commands* class serves as the primary interface between the user and the Turing Machine application. It processes command-line input, displays available commands, and maps each user input to a method call on the appropriate object. This class interacts with *Turing_Machine*, *Input_Strings*, and *Configuration_Settings* to perform user-driven operations such as loading input strings, running the machine, adjusting parameters, or displaying information.

3.11.2 Associations

The *Commands* class operates the *Turing_Machine* class by invoking its key methods based on user input.

It also specifies the *Input_Strings* and *Configuration_Settings* classes, modifying their internal state based on commands like Insert, Delete, Set, and Truncate.

3.11.3 Attributes

Although not explicitly modeled with data attributes in the UML, *Commands* would logically maintain references to:

- A *Turing_Machine* object
- An *Input_Strings* object
- A *Configuration_Settings* object

3.11.4 Methods

No specific methods were provided, but at a high level, *Commands* would likely include:

- process_command(in command: string)

- display_help()
- dispatch_to_<command>()

For each of the command types supported (Run, Quit, View, Help, etc.)

3.12 Main

3.12.1 Description

The *Main* class serves as the entry point of the application. It contains the main() function and is responsible for launching the command-line interface. It initializes core objects such as *Turing_Machine*, *Commands*, and the necessary configuration classes, then enters a loop where it waits for user input.

3.12.2 Associations

Main interacts with the *Commands* class to begin the command-processing loop and pass control to the application logic. It indirectly initiates all associations via object construction and passing references between components.

3.12.3 Attributes

No attributes are modeled for *Main*. It is typically procedural and serves only to bootstrap the application.

3.12.4 Methods

main()

The application's entry function. Initializes the system, displays a startup banner or help menu, and begins reading commands from the user.

4.0 User Interface

This section presents the command-line user interface (CLI) for the Turing Machine application. It includes example interactions between the user and the system for each supported command. These examples demonstrate how the application is invoked, how commands are entered, and what kind of output is produced in response. The goal is to provide a clear and consistent user experience that aligns with the application's functional requirements.

4.1 Command Line Invocation

The Turing Machine application is launched from the terminal using the following syntax:

Example:

```
./tm anbn

Loading definition file...

Turing Machine Loaded Successfully!

Command:
```

4.2 Help Command

The *Help* command *(H or h)* displays a list of all available commands along with their descriptions. This helps users understand the functionality of the application.

```
Command: h

Input Chars  Commands    Description

(D)----------Delete------Delete Input String From List
(X)----------Exit--------Exit Application
(H)----------Help--------Help User
(I)----------Insert------Insert Input String into List
(L)----------List--------List Input Strings
(Q)----------Quit--------Quit Operation of Turing Machine on Input String
(R)----------Run---------Run Turing Machine on Input String
(S)----------Set---------Set Maximum Number of Transitions to Perform
(W)----------Show--------Show Status of Application
(T)----------Truncate----Truncate Instantaneous Descriptions
(V)----------View--------View Turing Machine
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Command:
```

Figure 4.2.1: Help Command Table.

4.3 Show Command

The *Show* command *(W or w)* displays the current status of the application, such as loaded files or internal states. In the prototype, this displays a placeholder message.

```
Command: W
There is currently nothing to show
Command:
```

Figure 4.3.1: Show Command Table.

4.4 View Command

The *View* command *(V or v)* provides a visual or textual representation of the Turing Machine definition.

```
Command: V
View command executed
Command:
```

Figure 4.4.1: View Command Table.

4.5 List Command

The *List* command *(L or l)* lists the currently loaded input strings available for simulation.

```
Command: L
1. AA
2.  \
3. AABB
Command:
```

Figure 4.5.1: List Command Table.

4.6 Insert Command

The *Insert* command *(I or i)* allows the user to insert a new input string into the list for later processing.

```
Command: I
Input String: AAAB
Input String AAAB inserted into list.

Command:
```

Figure 4.6.1: Insert Command Table.

4.7 Delete Command

The *Delete* command *(D or d)* removes a specific input string by number from the list.

```
Command: D
Input String Number: 4
Input String AABBB deleted from list.

Command:
```

Figure 4.7.1: Delete Command Table.

4.8 Set Command

The *Set* command *(E or e)* allows the user to set the maximum number of transitions the Turing Machine should perform when running an input string.

```
Command: E
MAXIMUM NUMBER OF TRANSITIONS [1]: 2
MAXIMUM NUMBER OF TRANSITIONS SET TO 2

Command:
```

Figure 4.8.1: Set Command Table.

4.9 Truncate Command

The *Truncate* command *(T or t)* enables the truncation of instantaneous descriptions, typically to limit verbose output during machine execution.

```
Command: T
Truncate command executed

Command:
```

Figure 4.9.1: Truncate Command Table.

4.10 Run Command

The *Run* command *(R or r)* simulates the Turing Machine's execution on a selected input string by number. The output includes a sequence of configurations.

```
Command: R
Input String Number: 3
0. [s0]AABB
4. AB[s1]xy

Command:
```

Figure 4.10.1: Run Command Table.

4.11 Quit Command

The *Quit* command *(Q or q)* stops the Turing Machine's operation on the current input string before acceptance or rejection.

```
Command: Q
intput string AABB not accepted or rejected in 8 transitions

Command:
```

Figure 4.11.1: Quit Command Table.

4.12 Exit Command

The *Exit* command *(X or x)* terminates the application.

```
Command: X
...Program Terminated by User...
Program ended with exit code: 0
```

Figure 4.11.1: Exit Command Table.

5.0 Files

The Turing Machine application uses two types of external text files: a machine definition file and an input string file. These files are loaded upon application startup and drive the operation of the simulator.

5.1 Turing Machine Definition File

The Turing machine definition file contains the formal specification of the machine to be simulated. This includes the set of states, input and tape alphabets, transition rules, the initial state, the blank symbol, and the set of final (accepting) states.

Example:

anbn.def

```
This Turing machine accepts the language of one or more a's followed by the
same number of b's.

STATES: s0 s1 s2 s3 s4

INPUT_ALPHABET: a b

TAPE_ALPHABET: a b X Y -

TRANSITION_FUNCTION:
  s0 a   s1 X R
  s0 Y   s3 Y R
  s1 a   s1 a R
  s1 b   s2 Y L
  s1 Y   s1 Y R
  s2 a   s2 a L
  s2 X   s0 X R
  s2 Y   s2 Y L
  s3 Y   s3 Y R
  s3 -   s4 - R

INITIAL_STATE: s0

BLANK_CHARACTER: -

FINAL_STATES: s4
```

Figure 5.1.1: TM Definition File.

5.2 Input String file

The input string file contains a list of candidate strings to be tested by the Turing machine. Each string is placed on a separate line and may represent a valid or invalid string for the language defined by the machine.

anbn.str

```
a
ab
\
aaabb
aaaaaaaaaabbbbbbbbbb
aabb
aaaaaabbbbbb
ba
aba
bb
```

Figure 5.2.1: String Definition File.

6.0 References

Corrigan, N. B. *CPT_S 322 – Software Engineering Principles I*. Lecture materials and in-class instruction, Washington State University Tri-Cities, Spring 2025.

Martinez, R. *Turing Machine Application Prototype Source Code*. Unpublished student project, CPT_S 322, Washington State University Tri-Cities, Spring 2025.

7.0 Appendix

This appendix provides brief definitions and explanations of key terms and concepts referenced throughout the design document. These entries are intended to support understanding of technical terminology and the structural foundations of the Turing Machine application.

Command Line Interface (CLI)

A **Command Line Interface (CLI)** is a text-based user interface used to interact with software by typing commands into a terminal or console. The Turing Machine application uses a CLI to allow users to execute commands such as Run, Insert, View, and Quit. This interface facilitates interaction without the need for graphical elements.

Object-Oriented Design (OOD)

Object-Oriented Design (OOD) is a software engineering methodology that structures a system as a collection of interacting objects, each encapsulating data and behavior. The Turing Machine application is designed using OOD principles, where each component (e.g., tape, states, transition function) is modeled as a class with its own attributes and methods. This promotes modularity, reusability, and maintainability of the code.

Unified Modeling Language (UML)

The **Unified Modeling Language (UML)** is a standardized visual language used to model the structure and behavior of software systems. UML diagrams, such as class diagrams, are used in this design document to represent the architecture of the Turing Machine application. These diagrams illustrate relationships among classes, including associations, aggregations, and method interactions, providing a high-level overview of the system's object-oriented structure.

End.