- Systementwurf -

Software-Architektur für „Lernsoftware zum Verstehen und Programmieren von Turing-Maschinen“

Version: 0.8

|  |  |  |
| --- | --- | --- |
| Projektbezeichnung | TMSim | |
| Projektleiter | Tobias Lettner | |
| Verantwortlich | Team A | |
| Erstellt am | 26.04.2022 | |
| Zuletzt geändert | 28.05.2022 20:34 | |
| Bearbeitungszustand | X | in Bearbeitung |
|  | Vorgelegt |
|  | fertig gestellt |
| Dokumentablage | In Git-Branch main | |

Änderungsverzeichnis

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Änderung | | | Geänderte Kapitel | Beschreibung der Änderung | Autor | Zustand |
| Nr. | Datum | Version |
| 1 |  | 0.1 | Alle | Initiale Produkterstellung |  |  |

Prüfverzeichnis

Die folgende Tabelle zeigt einen Überblick über alle Prüfungen – sowohl Eigenprüfungen wie auch Prüfungen durch eigenständige Qualitätssicherung – des vorliegenden Dokumentes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Datum | Geprüfte Version | Anmerkungen | Prüfer | Neuer Produktzustand |
|  |  |  |  |  |

Inhalt

[1 Einleitung 4](#_Toc382581489)

[2 Architekturprinzipien und Entwurfsalternativen 4](#_Toc382581490)

[3 Übersicht über die Zerlegung des Systems 6](#_Toc382581491)

[4 Schnittstellenübersicht 7](#_Toc382581492)

[5 Systemkomponenten 8](#_Toc382581493)

[6 Designabsicherung 11](#_Toc382581494)

[7 Abkürzungsverzeichnis 16](#_Toc382581495)

[8 Literaturverzeichnis 16](#_Toc382581496)

[9 Abbildungsverzeichnis 17](#_Toc382581497)

# Einleitung

Dieses Dokument soll ein Grundverständnis der Systemstruktur vermitteln, ohne den Entwurf bis in letzte Einzelheiten darzulegen. Das Grundverständnis soll jedoch ausreichen, um sich ggf. anhand des Quellcodes in weitere Einzelheiten leicht einarbeiten zu können.

Kernthemen in diesem Dokument sind:

* Übersicht über die Zerlegung des Systems: Welche (größeren) Systemkomponenten gibt es? Wofür ist jede einzelne davon zuständig? Wie hängen diese Komponenten voneinander ab?

* Schnittstellenübersicht: Welche Schnittstellen stellt das System und jede Systemkomponente für seine/ihre Umgebung bereit?
* Systemkomponenten: Wie ist jede Systemkomponente aufgebaut?
* Designabsicherung: Zeigt für ausgewählte „architektur-relevante“ Use-Case-Szenarien, dass und wie diese mit dem gewählten Systementwurf realisierbar sind.

Der Systementwurf wird auf Grundlage der funktionalen und nicht-funktionalen Anforderungen sowie des konzeptuellen Datenmodells gewonnen, etwa indem man für ausgewählte „architektur-relevante“ Use-Case-Szenarien untersucht, welche Teile des Systems zur Realisierung in welcher Weise zusammenarbeiten müssen.

Die Gliederung dieses Dokuments orientiert sich grob am Aufbau der V-Modell-XT®[[1]](#footnote-1)-Produkte „System-Architektur“ und „SW-Architektur“, ist jedoch zur Verwendung für die Veranstaltung **„Software-Projekte“** in Informatik-Curricula der **OTH-Amberg-Weiden** angepasst worden (und nicht konform zum V-Modell-XT).

# Architekturprinzipien und Entwurfsalternativen

Das System ist unter Einhaltung des **„model-view-viewmodel“-Prinzips** realisiert, damit die Benutzerschnittstelle klar von der internen Logik abgegrenzt ist. Dadurch können spätere Änderungen am User Interface ohne Abwandlungen des Programmcodes erfolgen. MVVM bietet außerdem die Möglichkeit Unit-Tests für das UI und das „model“ separat durchzuführen. Ein weiter Vorteil ist die parallele Entwicklung von „model“ und“ view“, die durch die Separation dieser beiden Bereiche durch MVVM ermöglicht wird. Da das „model“ unabhängig von der „view“ ist, kann letztere ganz einfach ausgetauscht werden ohne, dass das „model“ verändert werden muss. Ein weiterer Vorteil ist die klare Strukturierung des Projekts, da dieses in drei Teile aufgegliedert ist, welche jeweils eine klar spezifizierte Aufgabe haben.

Eine andere Möglichkeit für die Systemarchitektur war anfangs das **„model-view-controller“-Prinzip**, welches jedoch nicht ausgewählt wurde, da dafür separate Controller-Instanzen notwendig wären, um die Kommunikation zwischen UI und der Programmlogik zu ermöglichen. Diese würde den Implementierungsaufwand erhöhen und es nicht ermöglichen UI und Logik separat zu entwickeln, da die passenden Schnittstellen vorab definiert werden müssten.

**MVVM-Architektur bei der grafischen Benutzungsschnittstelle**

Innerhalb der Präsentationsschicht ist die „view“ für die Darstellung der Daten zuständig. Das „viewmodel“ dient als Schnittstelle zwischen der Präsentationsschicht und der Problembereichsschicht. Das „model“ enthält die Logik der Turingmaschine und zusammen bilden diese drei Elemente die MVVM-Architektur.

**Weitere Entwurfsmuster**

* **Observer Pattern:**

Innerhalb der Präsentationsschicht kommt zwischen der „view“ und dem „viewmodel“ das Entwurfsmuster „Observer“ zum Einsatz: Das „viewmodel“ beobachtet die „view“ und wird von dieser über alle Änderungen an der Oberfläche benachrichtigt und gibt diese dann an das “model“ weiter.

* **Facade Pattern:**

An der Schnittstelle zwischen dem „model“ und dem „viewmodel“ in der Präsentationsschicht kommt das Entwurfsmuster „Facade“ zum Einsatz: Das „viewmodel“ kommuniziert ausschließlich über die Klasse „TuringMachine“ mit dem „model“ und hat sonst keine weiteren Kommunikationsschnittstellen mit anderen Klassen des „models“.

* **Memento Pattern:**

Das Entwurfsmuster „Memento“ kommt vor dem Starten der Simulation einer Turingmaschine zum Einsatz, indem dieser Zustand gespeichert wird und im Falle eines Abbruches der Simulation kann zu diesem Zustand zurückgekehrt werden. In zukünftigen Implementierungen könnte man darauf zurückgreifen und damit Rückschritte in der Simulation ermöglichen.

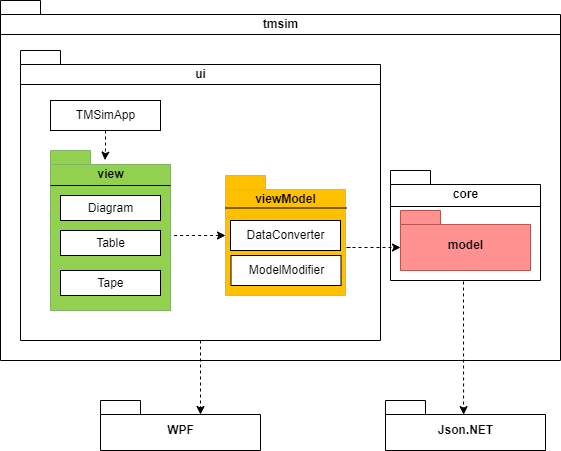
# Übersicht über die Zerlegung des Systems

Gesamtübersicht über die Architektur der „tmsim“-Anwendung: Komponenten/Pakete und deren Abhängigkeiten.

Grün: „View“ – für die Darstellung und Benutzerinteraktion zuständig

Gelb: „ViewModel“ – ermöglicht Kommunikation zwischen „View“ und „Model“

Rot: „Model“ – enthält die gesamte Logik der Turingmaschine



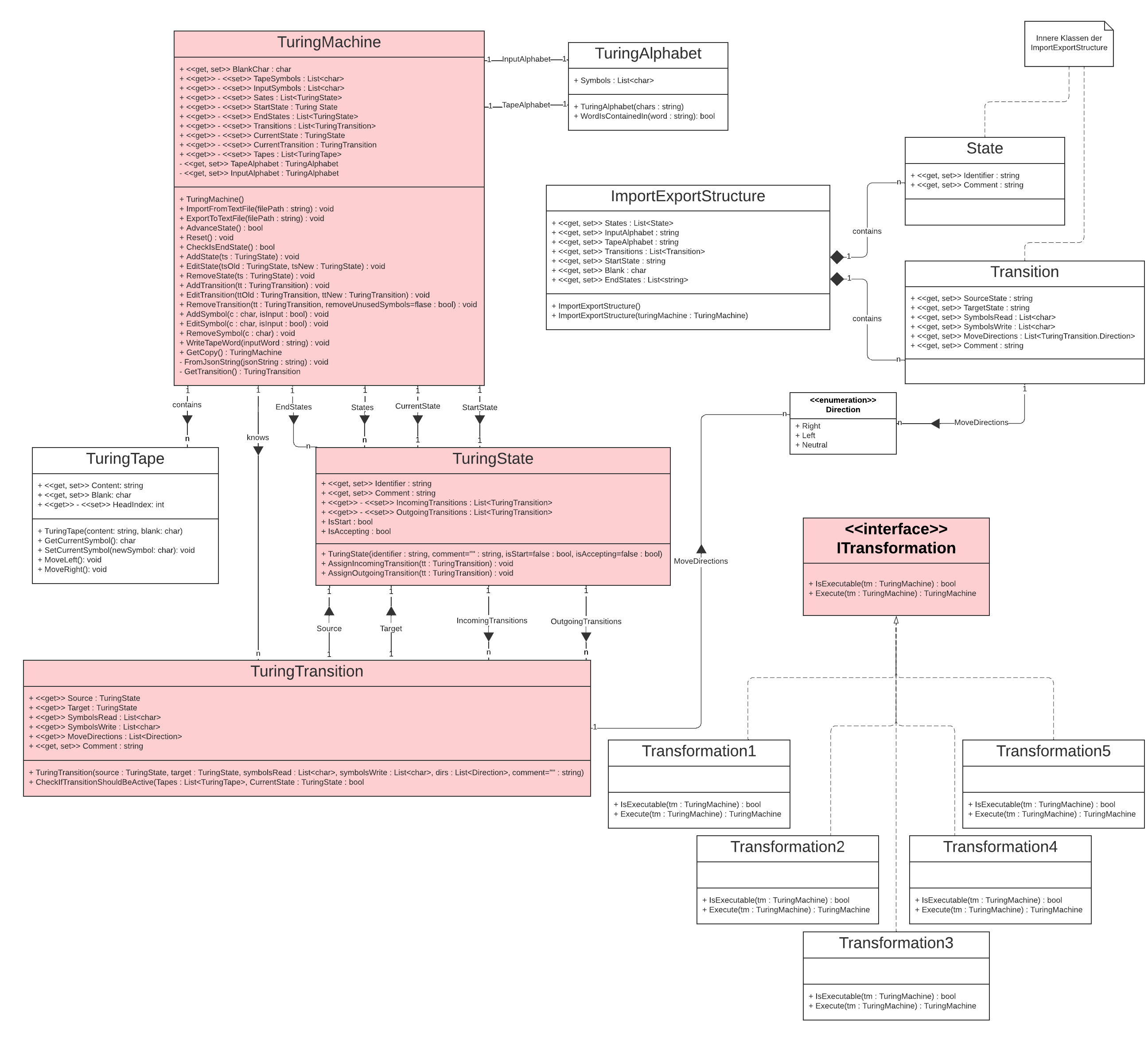
|  |  |
| --- | --- |
| **Paket/Klasse** | **Zuständigkeit** |
| tmsim.ui.TMSimApp | Anwendungsklasse für tmsim als WPF-Anwendung. Enthält das Hauptfenster. |
| tmsim.ui.view | Enthält Klassen für die Darstellung der Turingmaschine als Tabelle oder Zustandsdiagramm und das Band. |
| tmsim.ui.viewModel | Das „ViewModel“ enthält und formatiert alle nötigen Informationen des „Models“ für die „View“, damit diese dargestellt werden können. Außerdem gibt es Änderungen in der „View“ an das „Model“ weiter, damit dieses sich aktualisieren kann. |
| Tmsim.core.model | Repräsentiert den Problembereich. Klassen dieses Pakets bilden beispielsweise das Konzept der Turingmaschine mit dessen Bestandteilen, wie Zuständen und deren Übergängen, das Eingabe- und Bandalphabet oder die Bänder ab, sowie die damit verbundene Logik. |
| **Externe Pakete/Klassen** | |
| WPF | Enthält die nötigen Funktionen, um ein User Interface mit MVVM zu realisieren. Ermöglicht UI-Design durch XAML und die Kommunikation zwischen „View“ und „ViewModel“ mithilfe von „Data Binding“. |
| Json.NET | Erlaubt das Serialisieren und Deserialisieren von Textdateien im JSON-Format, um in diesem Format die Turingmaschine abzuspeichern. Ermöglicht außerdem das direkte zuweisen der Inhalte an eine Klasse, wenn die Struktur dieser mit der JSON-Datei übereinstimmen. |

# Schnittstellenübersicht

Die Klassen des View-Pakets (Tape, Table, MainWindow und Diagram) sind aus Gründen der Übersichtlichkeit hier nur angedeutet. Deren Funktionalität soll den restlichen Paketen nicht bekannt sein und ist somit für die Schnittstellenübersicht nicht relevant

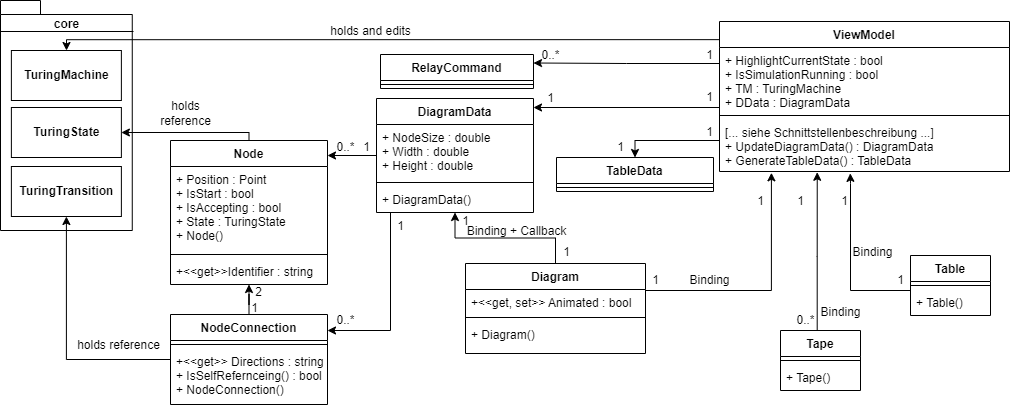
# Systemkomponenten

**Paket tmsim.core**



|  |  |
| --- | --- |
| **Klasse** | **Zuständigkeit** |
| TuringMachine | Beschreibt die Turingmaschine als Ganzes. Enthält die komplette Definition der Turingmaschine, erlaubt es eine zu importieren oder zu exportieren. Des Weiteren bietet sie die Funktionalität die Turingmaschine zu bearbeiten. Die Turingmaschine kann außerdem kopiert, in den nächsten Zustand versetzt oder in den Ausgangszustand zurückgesetzt werden. |
| TuringTape | Enthält das Eingabewort des Benutzers, die Position des LSK, kann das aktuelle Symbol unter dem LSK lesen und ändern. Kann außerdem den LSK nach dem aktuellen Zustandsübergang bewegen. |
| TuringState | Repräsentiert einen Zustand der Turingmaschine. |
| TuringTransition | Beinhaltet alle Informationen eines Zustandsübergangs, wie Bezeichnung und Ein- und Ausgangszustände. |
| TuringAlphabet | Generalisierung von Input- und Bandalphabet. Ein Alphabet besteht aus einer Liste einzelner Zeichen. |
| ITransformation | Generalisierung der einzelnen Transformationen. Das Interface stellt eine Methode zur Verfügung, um zu überprüfen, ob die Transformation ausgeführt werden kann. Eine weitere Methode dient zur Ausführung der Transformation. |
| Transformation1 | Die Turingmaschine stellt sicher, dass man nicht in den Anfangszustand zurückkehren kann. |
| Transformation2 | Der LSK bleibt bei keinem Zustandsübergang stehen. |
| Transformation3 | Ersetzt das Leerzeichen des Bandalphabets durch ein anderes Zeichen. |
| Transformation4 | Zuerst werden alles Übergänge entfernt, bei denen der LSK stehen bleibt. Dann wird die Menge aller Zustände in zwei Hälften geteilt. Alle Zustandsübergänge, bei denen der LSK nach links bewegt wird, stellen eine Hälfte dar und die restlichen Übergänge bilden die andere. |
| Transformation5 | Die Turingmaschine besitzt maximal einen akzeptierenden Zustand. |

**Paket tmsim.ui**

****

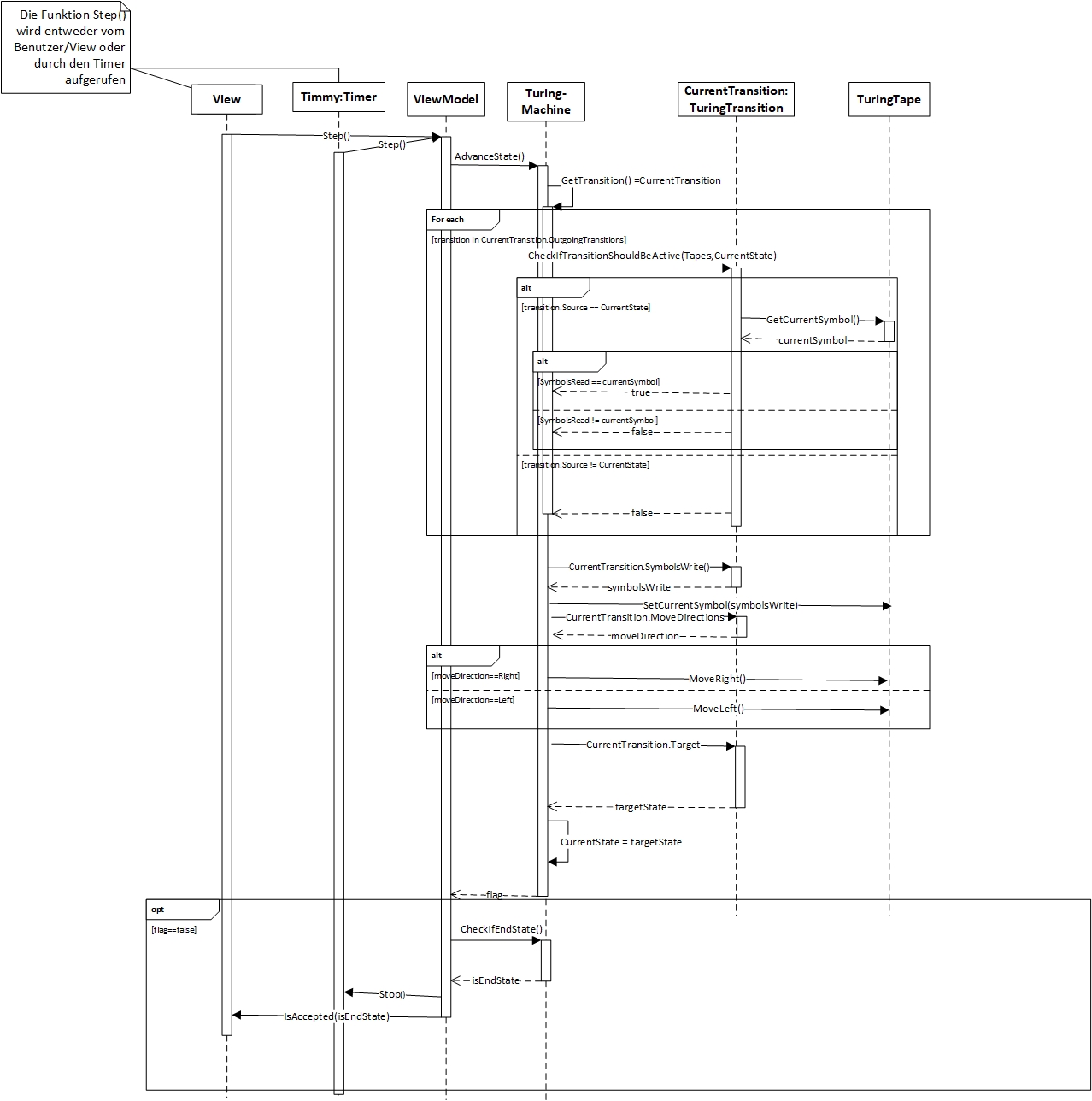
|  |  |
| --- | --- |
| **Klasse** | **Zuständigkeit** |
| Node | Stellt einen Knoten im Zustandsdiagramm dar und enthält Informationen über die Darstellung des Zustands, den es repräsentiert. |
| NodeConnection | Stellt die Verbindung zwischen zwei Knoten (Nodes) dar und enthält Informationen über die Darstellung des Übergangs, den es repräsentiert. |
| DiagramData | Enthält alle Knoten mit deren Bezeichnung und eine Liste aller NodeConnections, die gezeichnet werden müssen. Bestimmt außerdem die Größe der Nodes in der Benutzeroberfläche. |
| Tape | Stellt das Band der Turingmaschine in der Oberfläche dar. |
| Table | Stellt die Tabelle in der Oberfläche dar, in der die Turingmaschine definiert werden kann. |
| Diagram | Stellt das Zustandsdiagramm in der Oberfläche dar, in dem die Turingmaschine definiert werden kann. |
| ViewModel | Dient als Verbindung zwischen der „View“ und dem „Model“. Gibt Veränderungen des Benutzers von der „View“ an das „Model“ weiter und aktualisiert die Oberfläche beim Durchlaufen der Simulation entsprechend dem aktuellen Stand der Turingmaschine im „Model“. |
| RelayCommand | Bindet Funktionen des „ViewModels“ an Elemente der Benutzeroberfläche und ermöglicht dadurch Interaktion zwischen Benutzer und Programm. |
| TableData | Enthält die Daten der Tabelle. |
| TuringMachine | Siehe Paket „tmsim.core“ |
| TuringState | Siehe Paket „tmsim.core“ |
| TuringTransition | Siehe Paket „tmsim.core“ |

Die Verbindung des Core-Pakets mit Klassen des View-Pakets durch Referenzen wurde hier aus zweierlei Gründen gewählt. Zum einen hätte sich die Performance bei größeren Turing Maschinen potenziell problematisch gestaltet. Beim Verändern (oder lediglich neu rendern, je nach Implementation) des Diagramms müssten beispielsweise für jede NodeConnection die richtige TuringTransition durch Abgleichen der einzelnen State-Identifier gefunden werden. Eine Aufgabe, die beindruckend schlecht auf größere Anwendungsfälle skaliert. Zum anderen gestaltet sich somit der Quellcode deutlich übersichtlicher und somit weniger fehleranfällig, da beim Editieren und Entfernen von Zuständen und Übergängen bereits eine Referenz auf das relevante Objekt im Backend existiert.  
Im Sinne von MVVM Kann die View das Backend trotzdem nicht direkt modifizieren. Es hält lediglich Referenzen auf bestehende Objekte. In einigen Kastenschaubildern des MVVM-Prinzips wird diese Kenntnis auch als Pfeil von der View auf das Backend gekennzeichnet.

# Designabsicherung

**Sequenzdiagramm zum Befehl „Einzelschritt durchführen“:**

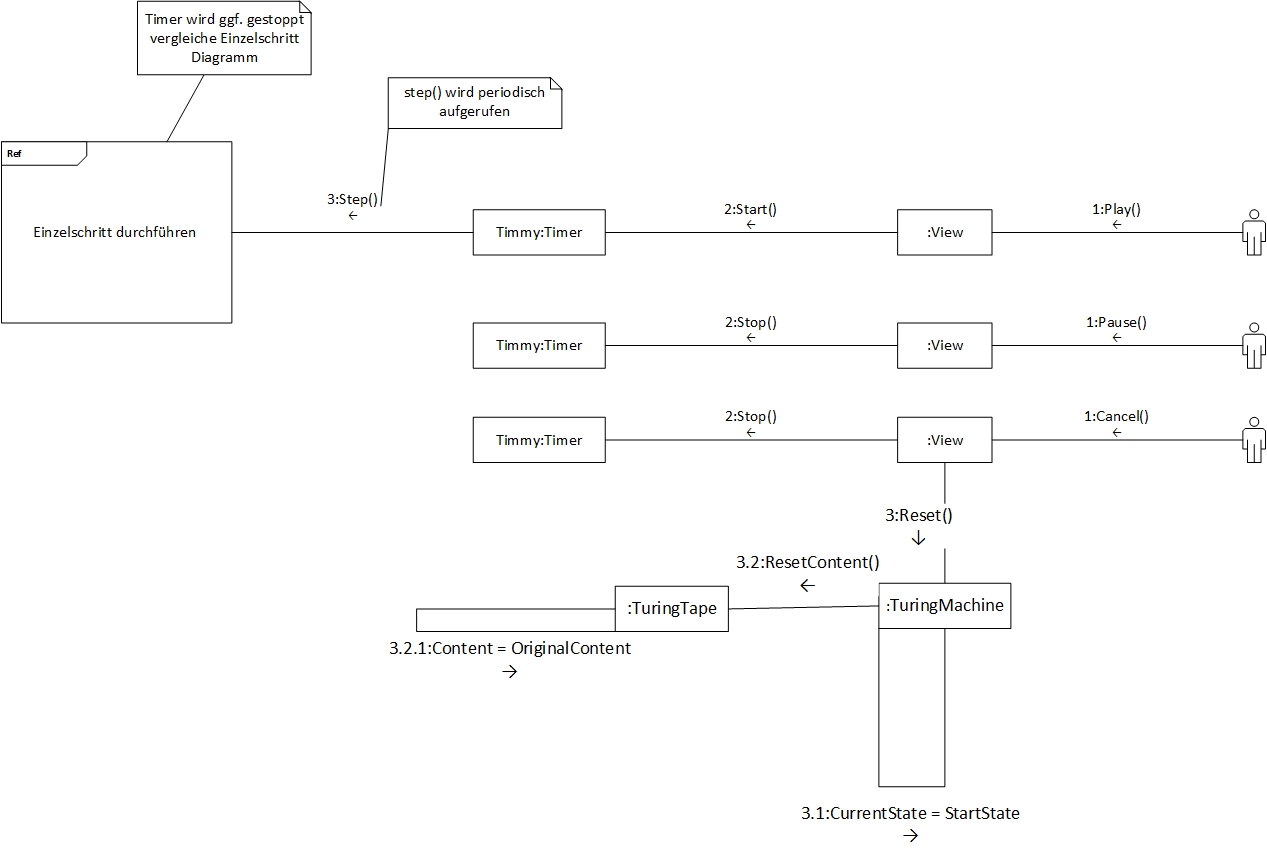
**Der Befehl führt einen einzelnen Zustandsübergang aus. Dabei ermittelt er den zugehörigen Übergang mit Hilfe des gelesenen Zeichens. Wurde ein passender Übergang zum Zeichen gefunden, so wird die Bewegungsrichtung des LSK und das zu schreibende Zeichen gemäß dem Übergang gesetzt. Danach wird der aktuelle Zustand auf den Zielzustand des Übergangs gesetzt. Gibt es keinen Übergang zum gelesenen Symbol, überprüft die Turingmaschine, ob der aktuelle Zustand ein akzeptierender Zustand ist. Der Benutzer wird darüber informiert, ob das Eingabewort von der Turingmaschine akzeptiert wurde oder nicht.**

****

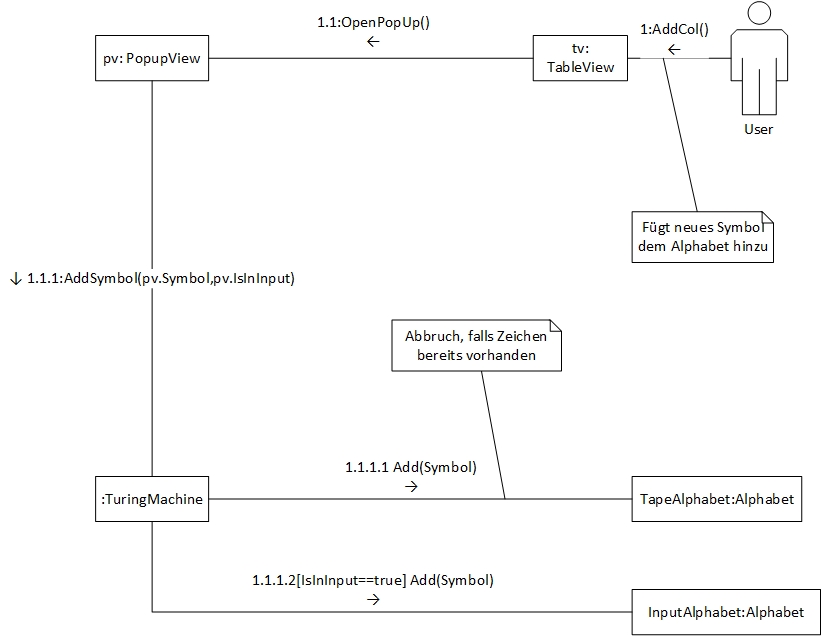
**Kommunikationsdiagramme zu den Befehlen „Simulation starten/stoppen/pausieren“:**

***Zu Start*: Der Benutzer startet den automatischen Ablauf der Turingmaschine. Dafür wird ein Timer gestartet, welcher periodisch den Einzelschrittbefehl (s.o.) durchführt. Die Periodendauer wird durch einen Slider in der UI durch den Benutzer eingestellt (Geschwindigkeit der Simulation).**

***Zu Pause*: Der Benutzer pausiert den automatischen Ablauf. Dabei wird der Timer gestoppt.**

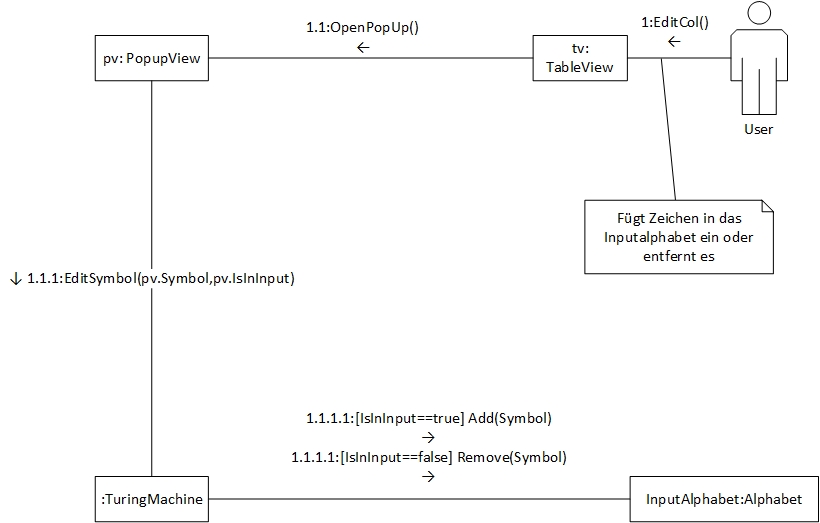
***Zu Abbruch*: Der Benutzer bricht den automatischen Ablauf ab. Dabei wird der Timer gestoppt, der aktuelle Zustand auf den Startzustand gesetzt und das Wort auf dem Band zurückgesetzt.**

**Kommunikationsdiagramm zum Einfügen eines Zeichens/einer Spalte in der Tabelle:**

**Der Benutzer will eine neue Spalte hinzufügen. Dabei wird ein Popup-Window geöffnet, welches den Benutzer auffordert ein neues Zeichen einzugeben. Das neue Zeichen wird dem Bandalphabet hinzugefügt, falls es dort noch nicht vorhanden ist. Zusätzlich kann der Benutzer auswählen, ob das Zeichen in das Eingabealphabet aufgenommen werden soll.**

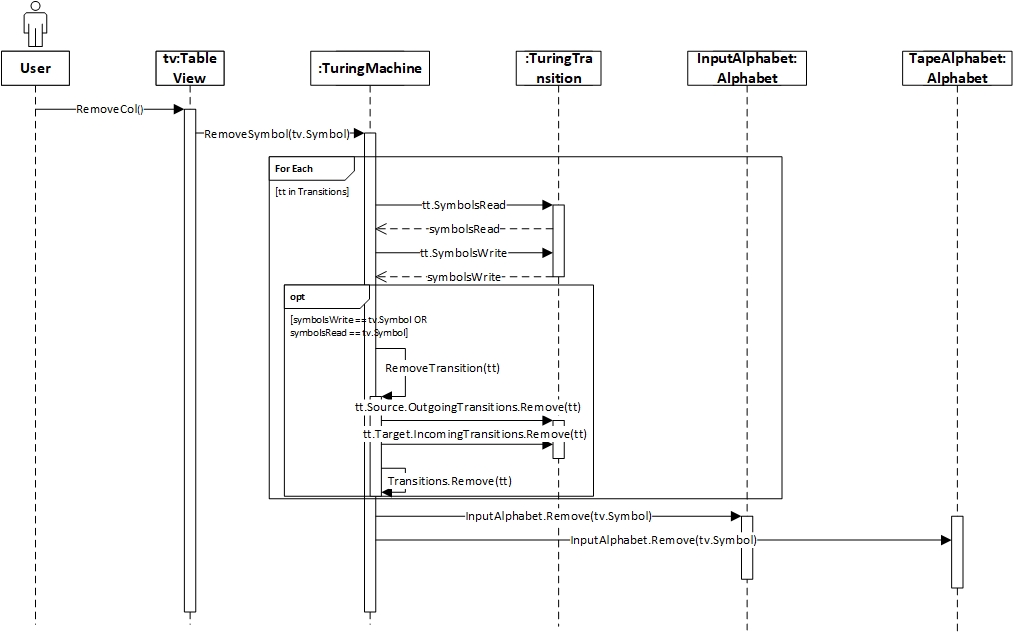
**Kommunikationsdiagramm zum Bearbeiten eines Zeichens/einer Spalte in der Tabelle:**

**Will der Benutzer ein Zeichen entfernen, dann erscheint dafür ein Popup-Window, über welches er das ausgewählte Zeichen dem Eingabealphabet hinzufügen oder von dort aus entfernen kann.**

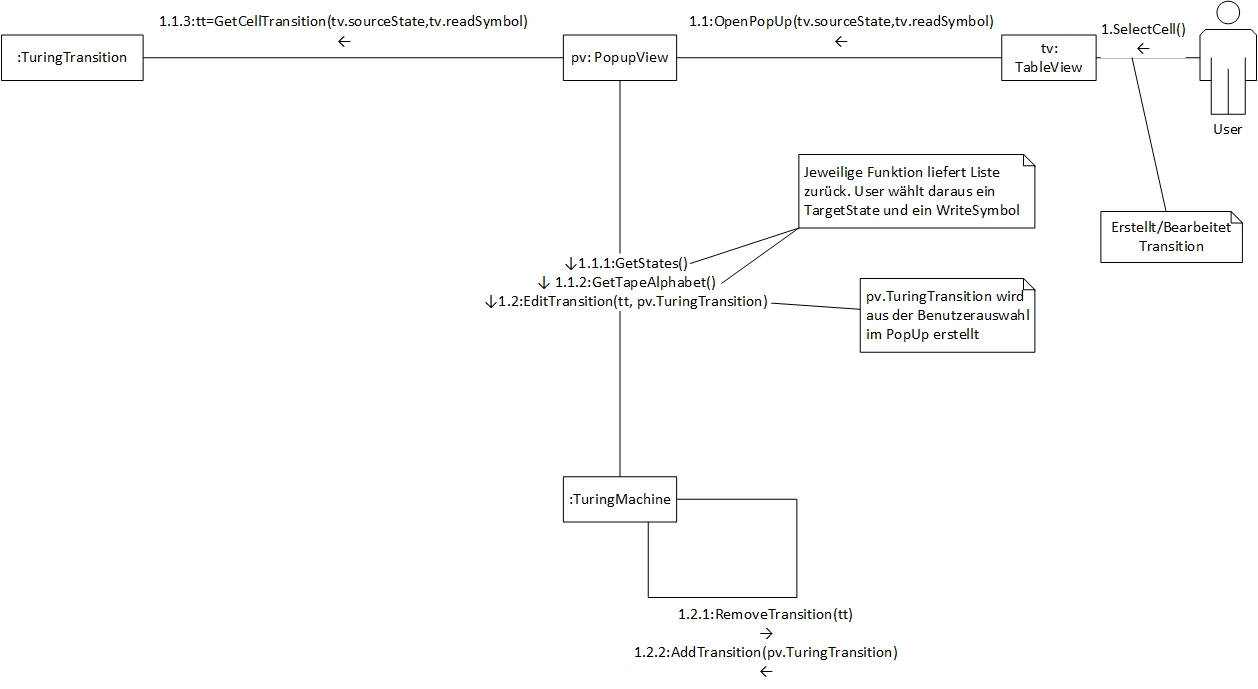
****

**Sequenzdiagramm zum Entfernen eines Zeichens/einer Spalte aus der Tabelle:**

**Der Benutzer löscht eine Spalte bzw. ein Zeichen aus der Tabelle. Dabei werden auch alle Zustandsübergänge, die dieses Zeichen lesen oder schreiben, gelöscht. Abschließend wird das Zeichen aus dem Eingabe- und Bandalphabet entfernt.**

****

**Kommunikationsdiagramm zum Bearbeiten/Einfügen eines Zustandsübergangs/Zelle in der Tabelle:**

Der Benutzer wählt eine Zelle in der Tabelle aus. Dabei spielt es keine Rolle, ob in der Zelle bereits ein Eintrag existiert oder nicht. Es erscheint ein Popup-Window, über welches der Benutzer den ausgewählten Zustandsübergang bearbeiten kann. Das Popup-Window ermittelt anhand der Spalte (Zeichen) und Zeile (Zustand) ob ein Zustandsübergang in dieser Zelle bereits existiert und holt sich diesen. Nachdem der Benutzer mit dem Bearbeiten fertig ist, wird aus den Daten im Popup-Window ein neuer Zustandsübergang erstellt. Der alte Zustandsübergang wird danach aus der Liste der Übergänge gelöscht und der neue hinzugefügt. Falls vorher in der Zelle kein Übergang definiert war, wird nur der neue hinzugefügt.

# Abkürzungsverzeichnis

|  |  |
| --- | --- |
| Abkürzung | Erklärung |
| UI | User Interface |
| LSK | Lese-/Schreibkopf |

# Literaturverzeichnis

# Abbildungsverzeichnis

1. V-Modell® ist eine geschützte Marke der Bundesrepublik Deutschland. [↑](#footnote-ref-1)