# Datenmodell

Als Datenmodell für den Graphen verwenden wir diverse Klassen aus JGraphT.

# Parser

Wir wollten in unserem Parser die Möglichkeit haben, mittels Leerzeilen (optische) Trennungen innerhalb der Dateien vornehmen und mittels „//“ am Zeilenanfang Zeilen auskommentieren zu können. Deswegen haben wir ihn um diese Möglichkeiten im Gegensatz zum vorgegebenen Format erweitert.

Der Parser arbeitet zeilenweise auf der Eingabedatei und versucht mittels diverser Regex die Zeilen zu matchen. Dabei wird zuerst die Headerzeile gesucht. Ist diese gefunden worden, wird daraus die Konfiguration gelesen, ob der Graph gerichtet, gewichtet oder attributiert sein soll. Stößt der Parser bei der Suche nach der Headerzeile auf eine Definitionszeile, so wird die Suche nach der Headerzeile abgebrochen und eine Standardkonfiguration für einen weder gerichteten oder gewichteten noch attributierten Graphen verwendet. Nachdem die Konfiguration ermittelt wurde, wird der entsprechende Graph aus der Konfiguration erzeugt. Wurde statt der Headerzeile eine Definitionszeile gefunden, so muss diese nach der Erzeugung des Graphen noch verarbeitet werden. Anschließend werden auch die übrigen Zeilen unter Berücksichtigung der Konfiguration verarbeitet.

# Benutzeroberfläche

Die Benutzeroberfläche verfolgt ein Model-View-Controller-Konzept das mit einem eigens entwickelten (kleinen) Framework umgesetzt wird. Bei diesem Ansatz werden die Daten / das Modell von der Logik der Anwendung und der Darstellung getrennt, sodass besser wartbarer Code entsteht und die Komponenten eine geringere Koppelung erfahren. Als positiven Nebeneffekt erhalten wir damit auch die Austauschbarkeit der Darstellung.

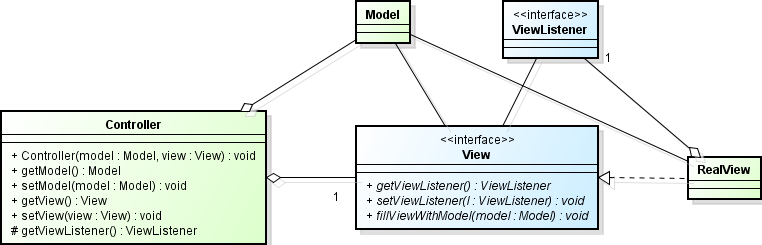


Abbildung - Generelles Muster für MVC

Der Controller setzt die Logik der Anwendung um und verwaltet das Model und die View (Darstellung). Das Model kann eine beliebige Klasse sein und ist in unserem Fall ein Graph aus JGraphT. Die View ist etwas komplexer und besteht aus den übrigen drei Komponenten. Das Interface "View" ist die Schnittstelle für den Controller zum Zugriff auf die View. Sie bietet eine Methode zum übergeben des Modells an, damit das Modell angezeigt wird und eine Methode um einen Listener auf die View zu registrieren. Der an der View registrierte Listener (ViewListener) stellt eine Schnittstelle dar, um Benachrichtigungen von der View an den Controller zu übermitteln (zum Beispiel ein Button-Click). Die RealView ist dann die Implementation des Interfaces View.

Soweit der generische Ansatz. Für unser Projekt haben wir eine Benutzeroberfläche zur Anzeige (und zukünftig vielleicht dem Bearbeiten) von Graphen angelegt. Diese wurde mit dem oben beschriebenen Muster entwickelt und soll exemplarisch darstellen, wie das Framework in der Praxis in diesem Projekt eingesetzt wird:

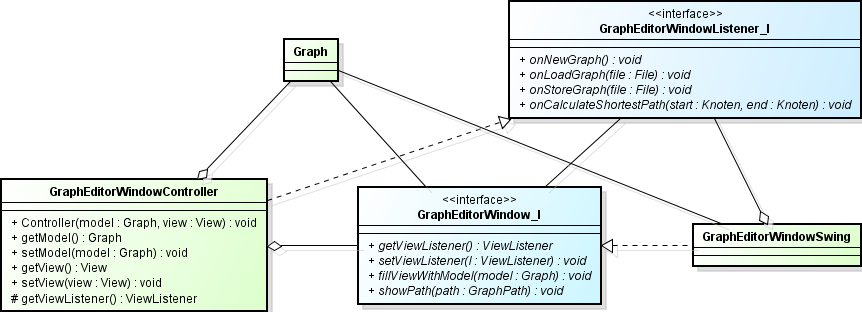


Abbildung - MVC-Aufbau der GraphEditor-Oberfläche

Hier kann man am GraphEditorWindowListener\_I sehen, dass dieser nun Methoden bereitstellt, die auf gewisse Interaktionen auf der Benutzeroberfläche reagieren. Der Controller implementiert das Listener-Interface und registriert sich so selbst auf der Oberfläche (GraphEditorWindow\_I) und wird somit über die Interaktion benachrichtigt. Die View (GraphEditorWindow\_I) bietet nun eine Methode zum Anzeigen eines Pfades an. Und zuletzt kann man sehen, dass die Graph-Klasse aus JGraphT als Modell für die Benutzeroberfläche verwendet wird.

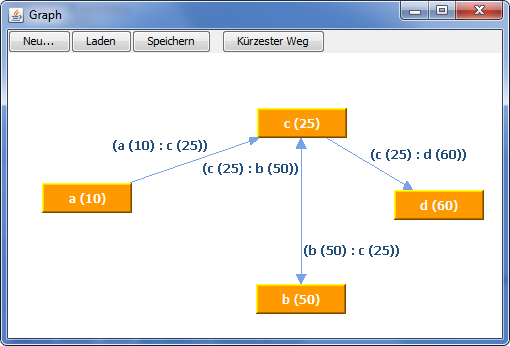


Abbildung - Die Benutzeroberfläche

Die Buttons (Neu, Laden, Speichern, Kürzester Weg) in der Benutzeroberfläche entsprechen den Aktionen, über die der Listener benachrichtigt werden kann. So wird gewährleistet, dass der Controller die Logik zum z.B. Laden oder Ermitteln des kürzesten Weges umsetzt, während die GUI dem Controller sozusagen nur den Wunsch äußert, dies zu tun.