1 Anmerkungen zum Pflichtenheft

1.1 Klarstellungen

1.2 Änderungen

1.2.1 GUI

Graph-Vorschau In der Graph-Preview Ansicht in der GUI werden die einzelnen Graphen, seien sie generiert oder importiert, unter einem neuen Tab angezeigt.

Diese Anzeige war bisher so gestaltet, dass die Graphen in einer Grid-View gesetzt werden. Dies würde in einer tabellenartigen Darstellung resultieren, bei der Beispielsweise 2 Spalten und 3 Reihen für die Graphen gleichzeitig dargestellt werden.

Diese Ansicht hatte den Nachteil, dass der User immer gezeichnete Graphen vor sich sieht. Dies führt zu deutlich geringerer Übersichtlichkeit. Außerdem bestand kein großes Interesse des Kunden daran, dass man die zuvor generierten Graphen sofort betrachten kann. Das graphische Darstellen der Graphen wurde eher an anderer Stelle gewünscht. Darüber hinaus ist diese Art der Ansicht nicht besonders gut skalierbar, wenn der User die Fenstergröße anpassen möchte, besteht die Gefahr, dass die Graphen-Bilder zu klein werden, um anschaulich zu sein.

Aus diesem Grund haben wir die Ansicht zu einer Tab-View geändert. Dies bedeutet, dass man nun eine Liste an ausklappbaren Tabs mit den jeweiligen Graphen-Namen vor sich sieht. Demzufolge kann man bei Interesse die Graphen-Tabs ausklappen. Beim Ausklappen wird dann genau dieser zu betrachtende Graph gezeichnet. Daraus folgt, dass man nicht mehr mit Graphen-Zeichnungen überschüttet wird.

Durch diese Änderung entsteht ein weiterer Vorteil. Die Performance des Programms wird verbessert, da das Programm nicht sofort alle Graphen zeichnen muss, sondern diesen Task erst bei Bedarf starten muss.

Graph-Generierung Möchte man die Heuristiken anwenden, benötigt man selbstverständlich hierfür erst einmal Graphen. Unser Programm stellt zu diesem Zweck mehrere Beschaffungsmöglichkeiten zur Verfügung: Automatische zufällige Generierung mit zuvor getätigten Einstellungen. Import bereits generierter Graphen. Im Graph-Editor von Grund auf neue Graphen von Hand erstellen.

Unter dem Tab "Graphen Generieren"der GUI war es bisher so gehalten, dass man als erstes die möglichen Einstellungsmöglichkeiten zur Generierung hat und sich darunter dann die verschiedenen Knöpfe befinden, welche die Generierung, den Import, oder das Zeichnen von Hand starten.

Diese Anordnung macht nur wenig Sinn, da man im Falle eines Imports oder auch des Editors keine Einstellungsmöglichkeiten benötigt.

Aus diesem Grund befinden sich nun die Buttons, welche die einzelnen möglichen Aktionen (Starten der Generierung, des Zeichnens oder Imports) ausführen, an oberster Stelle. Außerdem werden die Einstellungsmöglichkeiten zur zufälligen Generierung so lange vor dem User verborgen, bis er/sie aktiv auswählt diese Funktionalität wirklich zu benutzen.

Graph-Editor Beim Graph-Editor kann man standardmäßig sowohl einen „Siple-Undirected-Graph“, als auch „Simple-Hyper-Graph“ editieren oder auch erstellen. Dabei gibt es unterschiedliche Funktionen, die dem User geboten werden um dies zu tun.

Bisher wurden diese nicht auf spezielle Graphentypen eingeschränkt.

Allerdings entsteht bei einigen der angebotenen Funktionen die Gefahr, dass der User den Graphentyp durch die gemachten Änderungen verändert, oder gar den gesamten Graphen ungültig für die weitere Bearbeitung macht.

Die daraus von uns getroffene Anpassung war es die Funktionen auf den Graphen-Typ einzuschränken und den Graph-Editor den Typ des editierten Graphen überprüfen zu lassen.

2 Übersicht

Allgemein Das von uns als Grundstein des gesamten Projekts gewählte Entwurfsmuster ist das „Model-View-Controller“ (MVC) Entwurfsmuster. Es gliedert das Programm, wie der Name schon sehr stark vermuten lässt in die folgenden drei grundlegenden Bestandteile

[Modell] Das Model beschäftigt sich mit der gesamten Logik des Programms Hierunter zählen in unserem Fall hauptsächlich die verschiedenen Heuristiken.

[View / Präsentation] Die View stellt die Graphische Schnittstelle zum User dar.

[Controller / Steuerung] Der Controller bildet das Zwischenstück zu den bisherigen Teilen und verteilt die Aufgaben.

3 Model

4 View

Allgemein Kommen wir nun zum nächsten Großen Abschnitt des Programm-Entwurfs. Nachdem wir im letzten Abschnitt über das Model gesprochen haben folgt nun der View-Teil des Model-View-ControllerEntwurfsmusters. Die View beschäftigt sich, wie der Name andeutet mit dem Aussehen des Programms und somit mit der graphischen Repräsentation.

Wie im Pflichtenheft beschreiben haben wir uns für die Entwicklung mit Java entschieden. Unter Java gibt es mehrere Möglichkeiten eine GUI zu erstellen.

(a) Standart Widget Toolkit (SWT)

(b) Abstract Widget Toolkit (AWT)

(c) Swing

(d) JavaFx

Uns war allerdings relativ schnell klar, dass die Wahl auf JavaFx fallen wird. Dies lag nicht zuletzt an FXML und der bisherigen Entwicklungs-Erfahrung. Dazu gleich mehr.

[JavaFx] JavaFx ist eine Abkürzung für Java Graphics. JavaFx ist eine Möglichkeit unter Java eine graphische Oberfläche zu erstellen. JavaFx ist eine komplette Neuentwicklung von Oracle. Es ist unabhängig von den bisherigen Methoden AWT und Swing. JavaFx wurde 2014 veröffentlicht. Es ist seit Version 7.6 in x86 Java Standard Edition (JavaSE) Runtime Installation enthalten. Da wir mit Java 8 arbeiten werden ist dies somit kein Problem.

JavaFX arbeitet mit einem Szenengraphen (engl. scene graph), der die einzelnen Bestandteile einer GUI verwaltet. Auf diesen werden dann alle weiteren Bestandteile gesetzt.

[FXML] Wie auch bei den alternativen kann man natürlich auch mit JavaFx über zu schreibenden Code GUI-Objekte erstellen und diese auf den Scenen-Graphen aufbringen. Allerdings besteht mit JavaFx erstmals die Möglichkeit eine neue Form der GUI Entwicklung zu beschreiten. Diese erfolgt in Form von FXML.

FXML ist eine deklarative Beschreibung der grafischen Oberflächen auf XML-Basis. Dies bietet einige Vorteile gegenüber der konventionellen GUI-Entwicklung. Zum einen ist durch diese Technologie die

Trennung des Designs der GUI und deren Funktionalität strikt getrennt. Zum anderen ist das Einfügen von GUI-Bestandteilen, die an mehreren Stellen der Benutzeroberfläche zum Einsatz kommen sehr einfach möglich. Dies Ermöglicht, dass der mehrfachverwendbare Code nur einmal in einem Separatem FXML-Dokument abgespeichert werden muss und dann über den „include-Tag“ an allen Stellen verwendet werden kann. Darüber hinaus können für die Gestaltung auch Web-Technologien wie CSS eingesetzt werden. Dies sorgt zusätzlich für eine Trennung von Layout auf der einen und Style und Design auf der anderen Seite, da separate CSS-Dateien erstellt werden können. Diese können dann in den FXML-Code eingebettet werden, sodass die GUI das Design übernehmen kann.

Die Entwicklung der FXML-Dateien erfolgt zuerst über den SceneBuilder. Dieser ist ein grafisches Tool, das die Erstellung von FXML-Dateien vereinfacht. Der daraus generierte Code wird bei Bedarf dann nochmals per Hand nachbearbeitet. Zur Nachbereitung zählen unter anderem auch das Einfügen der „include-Tags“ (wie oben beschrieben).

Entwurf Der Entwurf der View gliedert sich prinzipiell in folgende Pakete auf:

(a) Graphic

(b) Drawer

(c) Sound

Diese sind Sup-Pakete des "View-Packagesünd werden im folgenden genauestens unter die Lupe genommen.

Paket Graphic

The Graphic-Package is a Package for some adaptations and expansions with the JavaFx Stuff.

Paket Graphic.UIElements

The UI-Elements-Package contains new created UI-Elements that expand the JavaFx-UI.

Klasse ZoomableScrollPane

Beschreibung This is an expansion to the JavaFx-ScrollPane. This adds the ScrollPane that it can be zoomed.

This is used so that the drawn Graph could be zoomed in/out so that the user can easily look for some Edges.

This Class is not made by ourself. @author https://www.pixelduke.com/2012/09/16/zooming-inside-a-scrollpane/

Dokumentation Because this Class is already fully implemented by the creator, there will be no Documentation from our side.

Paket Drawer

The Drawer-Package. This Package contains everything that belongs to the Drawing of the Graphs. It is a upper-Package, therefore no further Documentation for this.

Paket Drawer.GraphDrawer

The GraphDrawer-Package contains like the Name suggested the GraphDrawer that visualizes the Graph and "drawsït to a JavaFx-Node for the User.

Klasse GraphDrawer

Beschreibung The Drawer that draws the given Graph to the given JavaFx-Node. **Dokumentation**

- **graph : VisualGraph**
  The Graph that should be drawn.
- **fxNode : javafx.scene.Node**
  The JavaFx-Node where the Graph should be drawn on.
- **colourManager : ColourManager**
  The ColourManager of this Drawer to Map the ColourID's to the actual Colours of the to drawn Objects.

  This Object is created at the Constructor as new ColourManager and before the Drawing the ColourID's are added.
- **selectedVertices : List<Integer>**
  The List of Vertices-ID's that the user selected the Vertices at the GUI.

+ **GraphDrawer(graph : VisualGraph, fxNode : javafx.scene.Node)**
  The Constructor of this Class.

  Sets the given Graph and fxNode. Also initializes the ColourManager.
  @param graph The Graph that should be set as the Graph of this Drawer.
  @param fxNode The JavaFx-Node that should be set as the fxNode of this Class.

+ **printGraphTextual()**
  This Method prints the textual Representation onto the given JavaFx-Node.

+ **drawGraph(layout : GraphLayoutEnum)**
  This Method draws the Graph to the given JavaFx-Node by using the given Layout to position it's Vertices.
  @param layout The Enum that indicates which Layout the Drawer should use. If it is null the Drawer will use the Circle Layout.

- **drawVertex(vertexID : Integer)**
  Draw the given Vertex.

  Get the Vertex by searching for the given VertexID at the vertices-List of the given Graph. Use the GraphicLayout to get the correct Position of this Vertex.
  @param vertexID The ID of the to drawn Vertex.

- **drawVertexSelected(vertexID : Integer)**
  Draw the given Vertex as a selected Vertex.

  This Method is called if the to drawn Vertex of the drawVertex-Method is in the selectedVertices-List.

  The Vertex is drawn as selected by adding the corresponding Picture into the Vertex-JavaFx-Shape. Then the standard draw-Method is used to do the rest.

@param vertexID The ID of the Vertex that should be drawn as a selected Vertex.

- **drawEdge(edgePosition : Integer)**
  Draw the Edge that is on the given Position at the Edge-List of the Graph of this Drawer.

  This Method only draws one Edge so that the Editor can show specific Edges. This Method is also called multiple times to draw all Edges.
  @param edgePosition The Position of the to drawn Edge at the List of Edges of the Graph.

+ **addToSelectedVertices(vertexID : Integer)**
  Add the given VertexID to the List of selected ones.
  @param vertexID The Vertex-ID that should be added to the List of selected Vertices.

+ **clearSelectedVerticesList()**
  Clear the List of selected Vertices-ID's.

+ **getGraph() : VisualGraph**
  Get the VisualGraph of this Drawer.
  @return returns The VisualGraph of this Drawer.

+ **setGraph(graph : VisualGraph)**
  Set the VisualGraph of this Drawer.
  @param graph The VisualGraph that should be set.

+ **setFxNode(fxNode : javafx.scene.Node)**
  Set the JavaFx-Node where the Graph should be drawn on.
  @param fxNode The Node that should be set as the JavaFx-Node to draw on.

Paket Drawer.ColourManager

The ColourManager-Package. This Package only contains the ColourManager which maps the abstract ColourID's that are given by the calculation into a real Colour-Value that could be drawn. This Class is separately because it provides a relatively general task, that easily can be (re)used elsewhere.

Klasse ColourManager

Beschreibung The ColourManager manages the different Colours by Mapping the ColourID's to an actual Colour-Value, so that the Drawer can draw the coloured Graph by these ColourID's. **Dokumentation**

- **colourMap : Hashmap<IntegerString>**
  The HashMap of every ColourID to the actual Colour-Value that is represented as a String.

+ **ColourManager()**
  The Empty-Constructor of this Class. The Colours are added step by step at a later point.

+ **ColourManager(colourIDs : List<Integer>)**
  The Constructor of this Class. It adds the given ColourID's of the List and puts them into the Hashmap. Then the initColours-Method is called so that the mapping is completed for the given ColourId's.

@param colourIDs The List of colourID's that should be mapped to real Colour-Values.

+ **initColours()**
  This Method has to be called when every ColourID is put into the HashMap. Then this Method calculates a Assignment of real Colours to the ColourID's and writes them into the HashMap, where it can be read out at a later Time.

+ **addColourID(colourIDs : Integer)**
  Add a new ColourID to the HashMap, where later the real Colour is mapped to.

  It is checked if the given ColourID is already at the HashMap.
  @param colourIDs The ColourID that should be added.

+ **getColourFromID(colourID : Integer) : javafx.scene.paint.Color**
  Get the real Colour-Object from the given ColourID. This Colour is then used to draw the Vertex/Edge to the screen to represent the Colouring-Solution.

  The initColour-Method has to be called first so that the ColourManager has already mapped the Colour-Values at the HashMap.
  @param colourID The colourID from what the colour should be.
  @return returns The actual Colour of the Object.

Paket Drawer.Layouts

This Package contains the implemented Layouts for the GraphDrawer and the Enum that Lists all of them.

Klasse GraphLayoutEnum

Beschreibung This Enum Contains all implemented GraphLayout's that can be used by the graphDrawer to position the VisualVertices.

This Enum is needed because the Drawer needs to know whitch Layout to use for the drawing of the Graph and this is done via this Enumeration.

In our case there is only one Layout, because we well always draw the Graphs in a Circle.

If someone wants to Expand this Drawer by adding a new Layout he/she/it has to update this Enum as well. This is not against ObjectOrienting Programming because the Programmer that would add this new Layout already needs to recompile the Program and therefore can expand the Enum as well.
Dokumentation

+ **circleLayout**
  The Enum for the possible Layouts.

  There will be only one Value in it because we will only use the Circle-Layout. But this is needed for possible extensions by other Programmers.

Klasse GraphLayout

Beschreibung This is the Layout of the Drawing of the Graph.

It is an abstract class so that there could be multiple Layouts for the Representation that implements this. **Dokumentation**

+ **GraphLayout()**
  The Constructor of this abstract Class. This is used at the Childs if they do not have an separate Constructor because they do not need parameters to set as well.

+ **executeLayout(graph : VisualGraph) : VisualGraph**
  This is an abstract Method and has to be implemented at the Sub-Classes.

  This Method set's the given Graph to the implemented Layout of the particular Child-Class. Therefore it sets the Positions of the Vertices of the given Graph.
  @param graph The Graph that gets the layout set on it. Therfore all Elements of this given Graph will be relocated to the calculated Position this Method calculates.
  @return returns The given Graph with the calculated Layout.

Klasse GraphLayoutCircle

Beschreibung This is the Circle Layout of the Graph. Therefore this Layout orders the Graph-Nodes into a Circle.

It is an Child-Class of the abstract GraphLayout-Class.

- **radius : Double**
  The Radius of the Circle where the Elements should be positioned at.

+ **GraphLayoutCircle(radius : Double)**
  The Constructor of this Class.

  Sets the given Radius as radius of this Layout.
  @param NAME The Radius to set.

+ **executeLayout()**
  This is the overwritten Method from the abstract-Parent-Class.

+ **setRadius(radius : Double)**
  The Setter for the Radius.
  @param radius The Radius to set.

Paket Drawer.Visualization.VisualizationGraph

Klasse VisualVertex

Beschreibung The Vertex of an Visual-Graph. It is the Child of the JavaFx-Circle Object so this Vertex can be drawn. **Dokumentation**

- **ID : Integer**
  The Identification-Number (ID) of this Node. This Variable is Final.

+ **VisualVertex(id : Integer)**
  The Constructor of this Class.

  It contains only the final-ID as Parameter to set. The Parameters of the JavaFx-Node will be set by the Layout if it calculates the Position of this Vertex.

@param id The ID that will be set to this Vertex.

+ **getID() : Integer**
  Get the ID of this Vertex.
  @return returns The Integer-Value of the ID of this Vertex.

+ **toString() : String**
  This Method overwrites the standard toString-Method.
  @return returns It returns a String-Representation of this VisualVertex. «ID>"

Klasse VisualVertexColoured

Beschreibung Extends the VisualVertex Class.

This Vertex also contains a Colour-ID so that the Vertex can be coloured. **Dokumentation**

- **colourID : Integer**
  The ID of the Colour used by the Heuristic. This is like a Foreign-Key of the Colour.

  Remember: The actual colour of the specific Elements are not important because the User wants to see if the calculation of the Heuristic found a solution not what colour the Elements have. The Colour-ID can be associated with different drawing-colours for different draws without changing the statement of the Program.

+ **VisualVertexColoured(id : Integer, colourId : Integer)**
  The Constructor of this Class.

  It contains only the final-ID as Parameter to set. The Parameters of the JavaFx-Node will be set by the Layout if it calculates the Position of this Vertex.
  @param id The ID that will be set to this Vertex.
  @param coulorID The ID that will be set to this Vertex.

  If this Vertex is not coloured jet set the colour to null or use the other constructor.

+ **isColoured() : Boolean**
  Checks if this Vertex is Coloured.

  Therefore this Method checks if the ColourID is null or an actual Integer-Value.
  @return returns True if the ColourID-Varialbe is set and false if not.

+ **getColourID() : Integer**
  Get the ColourID of this Vertex.
  @return returns The Integer-Value of the ColourID of this Vertex.

+ **setColourID(colourID : Integer)**
  Set the ColourID of this Vertex.
  @param The Colour-ID this Vertex should be coloured with.

+ **toString() : String**
  This Method overwrites the standard toString-Method.
  @return returns It returns a String-Representation of this VisualVertexColoured. «ID>:<ColourID>"

Klasse VisualEdge

Beschreibung The Edge of an Visual-Graph. It is the Child of the JavaFx-Polygon Object so this Edge can be drawn. **Dokumentation**

- **connectedVerticesID : List<Integer>**
  This List contains all Vertices-ID's from the Vertices this Edge connects.

+ **VisualEdge(connectedVertices : List<VisualVertex>)**
  The Constructor of this Class.

  Set's the given List of by this Edge connected Vertices to the List of this Object.
  @param connectedVertices The List of by this Edge connected Vertices. This given List will be set to the List of this Edge-Object.

+ **connectsSame(compareEdge : VisualEdge) : Boolean**
  Checks if the given VisualEdge is an edge between the Same Vertices as this Edge.
  @param compareEdge The Edge of which the connected-Vertices should be checked with.
  @return returns If the two Edges are conections between the same Vertices it returns true, else false.

+ **getConnectedVertricesIDList() : List<Integer>**
  Get the List of the connected VerticesIDs.
  @return returns The List of the Vertices-ID's that this Edge connects.

+ **toString() : String**
  This Method overwrites the standard toString-Method.
  @return returns It returns a String-Representation of this VisualEdge. "<VertexID1>, ..."

Klasse VisualEdgeColour

Beschreibung Extends the VisualEdge Class. This Edge also contains a Colour-ID so that the Edge can be coloured. **Dokumentation**

- **colourID : Integer**
  The ID of the Colour used by the Heuristic. This is like a Foreign-Key of the Colour.

  Remember: The actual colour of the specific Elements are not important because the User wants to see if the calculation of the Heuristic found a solution not what colour the Elements have. The Colour-ID can be associated with different drawing-colours for different draws without changing the statement of the Program.

+ **VisualEdgeColoured(connectedVertices : List<VisualVertex>, colourId : Integer)**
  The Constructor of this Class.

  Set's the given List of by this Edge connected Vertices to the List of this Object.
  @param connectedVertices The List of by this Edge connected Vertices. This given List will be set to the List of this Edge-Object.
  @param coulorID The ColourID that will be set to this Edge.

  If this Edge is not coloured jet set the colour to null or use the other constructor.

+ **isColoured() : Boolean**
  Checks if this Vertex is Coloured.

Therefore this Method checks if the ColourID is null or an actual Integer-Value.
@return returns True if the ColourID-Varialbe is set and false if not.

+ **getColourID() : Integer**
Get the ColourID of this Edge.
@return returns The Integer-Value of the ColourID of this Edge.

+ **setColourID(colourID : Integer)**
Set the ColourID of this Edge.
@param The Colour-ID this Edge should be coloured with.

+ **toString() : String**
This Method overwrites the standard toString-Method.
@return returns It returns a String-Representation of this VisualEdge-Coloured. "<VertexID1>, ...:<ColourID>"

Klasse VisualGraph

Beschreibung This is the VisualGraph. It is the Graph-Construct that is used for the Drawing.

Remember: VisualGraph is Generic VisualGaph<V extends VisualVertex, E extends VisualEdge> This is necessary so that the Graph can differentiate between the uncoloured and the coloured Elements.

This separate Graph-Representation for the View is necessary because the Model and the View of the Rage-Program should be strictly separated and therefore the View could not use the same Graph-Object. As well this Graph-Representation uses special Nodes and Edges as Elements that could be drawn. **Dokumentation**

- **vertices : List<VisualVertex>**
This is a List of all Vertices (=Node's) of this Graph.

  Remenber: At any further Point the Nodes"will be named Vertex/Vertices because of the confusion with JavaFx-Nodes that would otherwise occur.

- **edges : List<VisualEdge>** This is a List of all Edge's of this Graph.

+ **VisualGraph()**
The Empty-Constructor of this Class.

+ **isColoured()**
Checks if the Graph is made out of VisualVertexColoured and VisualEdgeColoured and if so if the ColouredID's of all Objects are set.
@return returns If they are set it returns true, and if not false.

+ **addVertex()**
Add a new Vertex to the List of Vertices of this Graph.

  If the List is not instanciated yet this will be done.

  To add a new Vertex this Method searches for the next unused Integer-ID that could be used for a new Node and created the VisualVertex-Object with this Parameter. This created Object will be added to the List.

+ **addVertex(vertex : VisualVertex) : Boolean**
Add the given Vertex to the List of Vertices.

If the List is not instanciated yet this will be done.

Also it is checked that the Vertex-ID is not already used by another Vertex. If so the given Vertex will not be added.
@param vertex The Vertex that should be added to this Graph.
@return returns If the Vertex-ID was added this Method returns true, otherwise false.

+ **addVertex(vertices : List<VisualVertex>)**
Add a whole List of Vertices to this Graph.

This is done by calling the addVertex-Method multiple times.
@param vertices The List of Vertices that should be added to the List.

+ **addVertex(amount : Integer)**
Add the given amount of Vertices to the Graph.

This is done by calling the addVertex-Method multiple times.
@param amount The amount of Vertices the user wants to add to this Graph.

+ **addEdge(edges : VisualEdges)**
Add the given Edge to the Graph.

If the List is not instanciated yet this will be done.

Also it is checked if this Edge has the exact same connected Vertices as any other Edge of this Graph. This is done by calling the connectSame-Method of the given Edge.

Also it is checked that the given Edge is valid. That means that this method checks if all connected-Vertices that are given by ID are Vertices of this Graph. If there is an unexisting Vertex this Vertex will be created and added to the Graph by calling the addVertex(VisualVertex)-Method.
@param edge The Edge that should be added to this Graph.

Check if this Edge contains valid VertexID's and if it only connects Vertices that are not currently connected.

+ **addEdge(vertices : List<VisualEdges>)**
Add a whole List of Edges to this Graph.

This is done by calling the addEdge-Method multiple times.
@param edges A List of Edges that should be added.

+ **addEdge(verticeIDs : List<Integer>)**
Add the Edge, that is given by the List of Vertice-ID's, to the graph.

This is done by creating an new VisualEdge-Object with the given List as Parameter and then calling the addEdge-Method.
@param verticeIDs The List of Vertice-ID's that should be connected by Edge that should be added.

+ **removeVertex(vertex : VisualVertex)**
Remove the given Vertex from the Graph.

If an Edge was connected to this Vertex and it only contains one other Vertex after the deletion, the Edge will be removed too.

@param vertex The Vertex that should be removed.

+ **removeVertex(vertexID : Integer)**
Remove the Vertex, by the given ID, from the Graph.

This is done by calling the removeVertex-Method. (The Vertex that should be deleted can be found at the Vertices-List by the given ID).
@param vertexID The Vertex-ID from the Vertex that should be removed from the Graph.

+ **removeEdge(edge : VisualEdge)**
Remove the given Edge from the Graph.
@param edge The Edge of the VisualGraph that should be removed.

+ **removeEdge(verticesIDs : List<Integer>)**
Remove the Edge between the given Vertrice.
@param verticesIDs The List of the Vertices-ID's that the Edge is between, that should be removed.

+ **duplicateVertex(vertexID : Integer)**
Duplicate the given Vertex so that a new Vertex is at the Graph with exactly the same neighbourhood.
@param vertexID The Vertex-ID of the Vertex that should be duplicated.

+ **contractVertices(verticesIDs : Integer)**
Contract the given Vertices to one Vertex.

Multiple Edges between the same destinations will be removed, so that only one of these Edges is in the Graph. Edge-Loops will be removed.
@param verticesIDs The List of the given VertricesID's.

+ **setVertexOrder(vertexID : Integer, order : Integer)**
Set the Vertex to the given Order.

The Vertex that was at this Position of the List earlier will be put behind the set Vertex.
@param vertexID The ID of the Vertex that should be moved to a different Order.
@param vertexID The order the Vertex should be set to.

Paket Sound

The Sound-Package contains everything that has to do with the Sounds. It separates the SoundHandler from the other parts.

Klasse SoundHandler

Beschreibung The Sound Handler that manages the different Sounds the Program can make. Including the Error and finish Sound.

Dokumentation

- **soundList : List<String>**
The List of all paths to the Audio-Files.

- **player : javafx.scene.media.MediaPlayer**
  The MediaPlayer that plays the given Music.

+ **SoundHandler()**
  The Constructor of this Class. Has no parameters so it only sets the List to an Empty List so that the User can add File-paths to the playable Sounds later.

+ **SoundHandler(sounds : List<String>)**
  The Constructor of this Class. The List of Strings should contain path to the Sound-Files the Player should play. The given List will be set at the soundList of this Class.
  @param sounds The Path-List that the SoundHandler should use as soundList.

+ **addSound(filepath : String)**
  Add a new Sound-Filepath to the soundList.
  @param filepath The Filepath that should be added.

+ **playSound()**
  Starts the MediaPlayer with a random Sound of the given List.

  Therefore it calls the playSound(listPosition)-Method with an randomly choosen Value.

+ **playSound(listPosition : Integer)**
  Starts the MediaPlayer with the Sound at the given position of the soundList of this Class.

  Therefore it checs the given position if it is valid. Then it loads the File from the path that is stored at the soundList at the given Position. If the File could not be loaded the Method stops. Else the loaded File will be passed on to the MediaPlayer of this class. The MediaPlayer will be started, so that the Sound is played.
  @param listPosition The position of the Sound at the soundList that should be played.

+ **stopSound()**
  Stops the playing of the MediaPlayer.

5 Controller

Dieser Abschnitt beschäftigt sich wie der Titel andeuten lässt mit dem Controller des Projektes. Dieser ist wiederum in zwei Hauptbestandteile unterteilt. Zum einen natürlich den üblichen Controller, zum anderen aber auch einem Graphic-Controller, der sich spezifisch mit dem Controlling der View beschäftigt.

5.1 Super-Controller

5.2 View-Controller

Allgemein Der Graphic-Controller oder unter JavaFx üblicherweise auch FxController ist der Teil eines JavaFx-Programms der direkt mit dem von der FXML-Datei bereitgestellten GUI verknüpft ist. Der FxController ist somit ein Separater Teil des Controllers, der sich lediglich mit der GUI beschäftigt und die getätigten Eingaben an die richtigen Stellen im allgemeinen Controller weitergibt. Dies bringt den Vorteil, dass der allgemeine Controller keine Kenntnisse über die GUI benötigt und losgelöst von dieser funktionieren kann. Dadurch ist auch die Modularität in diesem Teil des Entwurfs gewährleistet.

Entwurf

6 Resources

Allgemein Die Ressourcen sind alle Dateien, die nicht in direktem Zusammenhang mit der Funktionalität und des Programms stehen und keinen Einfluss auf den Ablauf haben. Hierunter fallen meist Bilder, wie Icons, oder auch andere Mediendateien und vieles mehr. Diese Dateien muss unser Programm aus externen Stellen ziehen.

Entwurf Diese Daten werden getrennt vom Programmcode abgelegt und dann bei Bedarf aus der vordefinierten Stelle vom Programm eingeladen.

Paket Resources

This contains all the Resources that are needed for the Project.

* **FXML**
  This contains all the FXML files for the GUI. They are arranged in different Sub-Folders to separate.

  Main

  StartTab

  Preview

  GraphGeneration

  MenuBar

  Editor

  Popups

* **Pictrues**
  This contains all the Pictures used at the GUI organized by sub-Folders.

  Icons
  This contains all Icons for the Buttons, ... of the GUI.

  Logo
  This contains all Logos used at the GUI.

* **Sound**
  This contains all the Sounds that can be played by default.

* **StyleSheets**
  This Contains all the CSS-Files for the GUI.

* **Plugins**
  This Contains all the Plugins the User could add to the Rage-Program. By Default, there are the Plugins for the TC and EFL that we should implement.

* **Log**
  Contains the Log-Files.