Random Graph Coloring Evaluation

Entwurfsdokument

Jonas Kasper, Bernard Hohmann, Thomas Fischer, Christian Jung, Jonas Linßen

Inhaltsverzeichnis

1 Anmerkungen zum Pflichtenheft

1.1 Klarstellungen

1.2 Änderungen

1.2.1 GUI

Graphen-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.

Graphen-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.

Graphen-Editor Beim Graph-Editor kann man standardmäßig sowohl einen „Simple-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

2.1 Architektur

Das System basiert auf dem Model-View-Controller-Muster mit einer drei-Schichten-Architektur und einer durch JavaFX realisierten graphischen Benutzerschnittstelle.
Beim MVC-Muster wird das System in die drei Komponenten Modell, Präsentation und Steuerung aufgeteilt. Das Modell (Model) enthält und verarbeitet die Daten, welche dann von der Präsentation (View) dargestellt werden. Die Steuerung (Controller) steuert den Ablauf und das Verhalten der Anwendung. Dafür werden Benutzereingaben auf Modeländerungen und Ausführung von Berechnungen abgebildet. Weiterhin informiert das Model direkt oder über den Controller die View über Änderungen am Model (z.B. Ergebnisse von Berechnungen) und sorgt damit für eine Anpassung bzw. Aktualisierung der View.
In der hier verwendeten Architektur erfolgt die Kommunikation zwischen View und Model immer über den Controller. Dabei gibt es `FXController`-Komponenten, welche die Steuerung der View übernehmen, und `Controller`-Komponenten, welche als Schnittstelle zwischen den `FXControllern` und dem Model dienen.
Dadurch kann das System in drei logische Schichten unterteilt werden, eine Daten-, eine Logik- und eine Benutzerschicht 1.

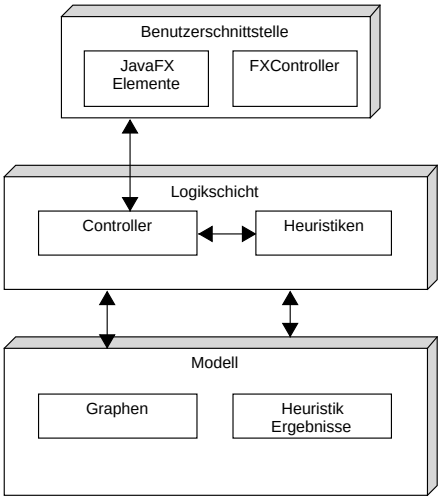

Abbildung 1: Die Architektur als drei-Schichtenmodell

2.2 Sequenzdiagramme

Durch Benutzereingaben initiierte Aktionen werden durch das JavaFX-Framework an die entsprechenden `FXController` weitergeleitet. (TODO wie werden Controller und Aktionen verbunden?). Dieses wird in den folgenden Diagrammen deshalb nicht dargestellt, sondern nur die anschließenden Interaktionen.

2.2.1 Heuristiken ausführen

Eine Benutzereingabe zum Ausführen der Ausgewählten Heuristiken wird durch den `FXTabController` verarbeitet und and den `TabController` weitergeleitet. Zu jeder ausgewählten Heuristik wird die Methode

`addToHeuristic` aufgerufen. Diese erzeugt zuerst ein Objekt der entsprechenden Heuristik. Diese wird zum `DataPool`, welcher Graphen mit darauf auszuführenden Heuristiken enthält, hinzugefügt. Dabei wird beim Hinzufügen die Heuristik über die `applyTo`-Methode auf alle Graphen im `DataPool` angewandt.
Ist die Anwendung aller Heuristiken abgeschlossen, wird eine Liste von Ergebnissen der Berechnungen vom Typ `HeuristicResult` vom `DataPool` zurückgegeben und über den `TabController` an den `FXTabController` weitergeleitet.

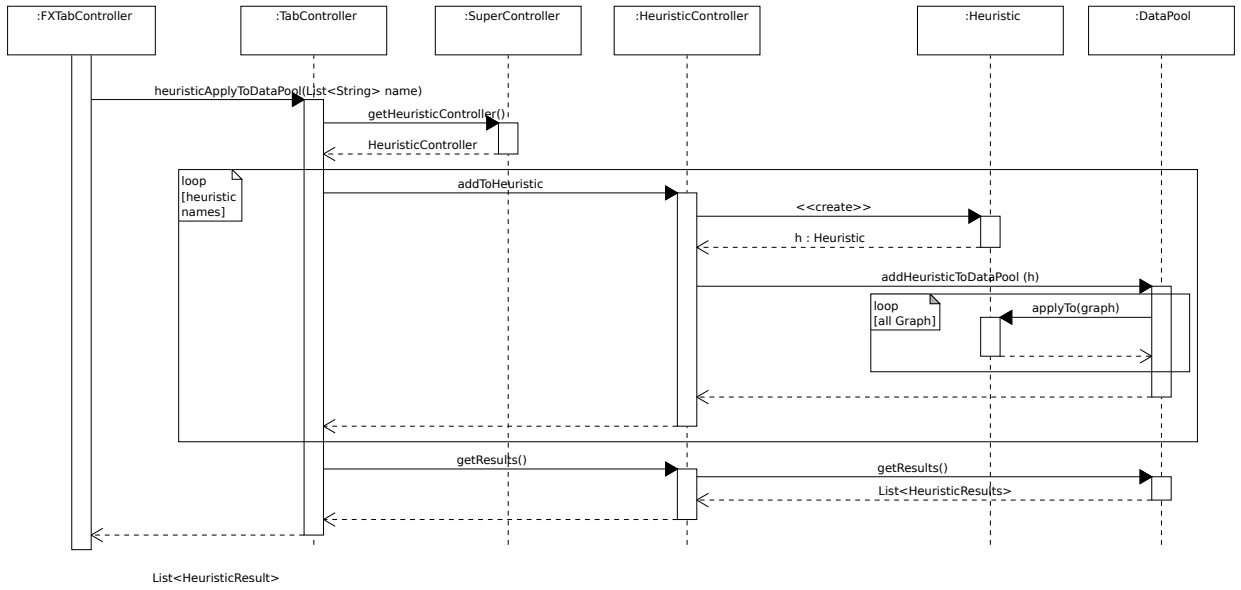

Abbildung 2: Sequenzdiagramm zum Ausführen von Heuristiken

2.2.2 Graphen generieren

Eine Benutzereingabe zum Generieren von Graphen wird vom `FXGraphGeneratorController` verarbeitet. Dieser delegiert durch den Aufruf der Methode `generate` an den `GraphGeneratorController`. Hier wird für alle n zu generierenden Graphen durch die Klasse `GraphBuilder` jeweils ein Objekt vom Typ `Graph` erzeugt und zu einem `DataPool` hinzugefügt. Jeder dieser Graphen wird anschließend durch die Klasse `GraphAdapter` in eine für die View benötigte Graphenstruktur umgewandelt.
Nach Generierung der Graphen wird ein neuer `TabController` erzeugt. Diesem wird der `DataPool` der neu generierten Graphen übergeben. Zum Schluss wird noch ein neuer `FXTabController` erzeugt, welcher für die Anzeige des Tabs für die generierten Graphen zuständig ist.

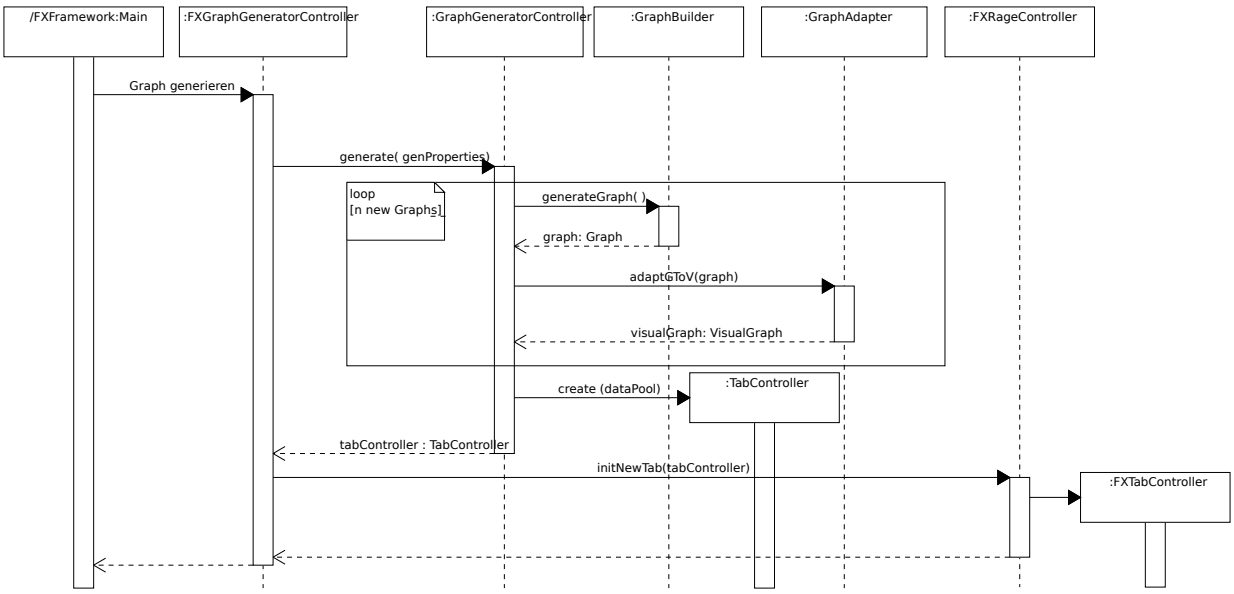

Abbildung 3: Sequenzdiagramm zum Generieren von Graphen

2.2.3 Graphen modifizieren

Das Modifizieren von Graphen erfolgt über die Detailansicht des Graphen. Dafür wird zuerst ein `FXDetailViewController` erzeugt. Bekommt dieser anschließend eine Benutzereingabe zum Modifizieren, dann wird ein `FXGraphEditorController`-Objekt erzeugt. Den dafür benötigten `GraphEditorConroller` erhält er über den Aufruf der Methode `getGEC` des `SuperControllers`.
Anschließende Editierbefehle werden direkt an den `FXGraphEditorController` weitergeleitet und von diesem verarbeitet. Bei Abschluss des Editieren durch den Benutzer wird der modifizierte Graph als neuer Graph durch den `GraphEditorController` zum `DataPool` hinzugefügt.

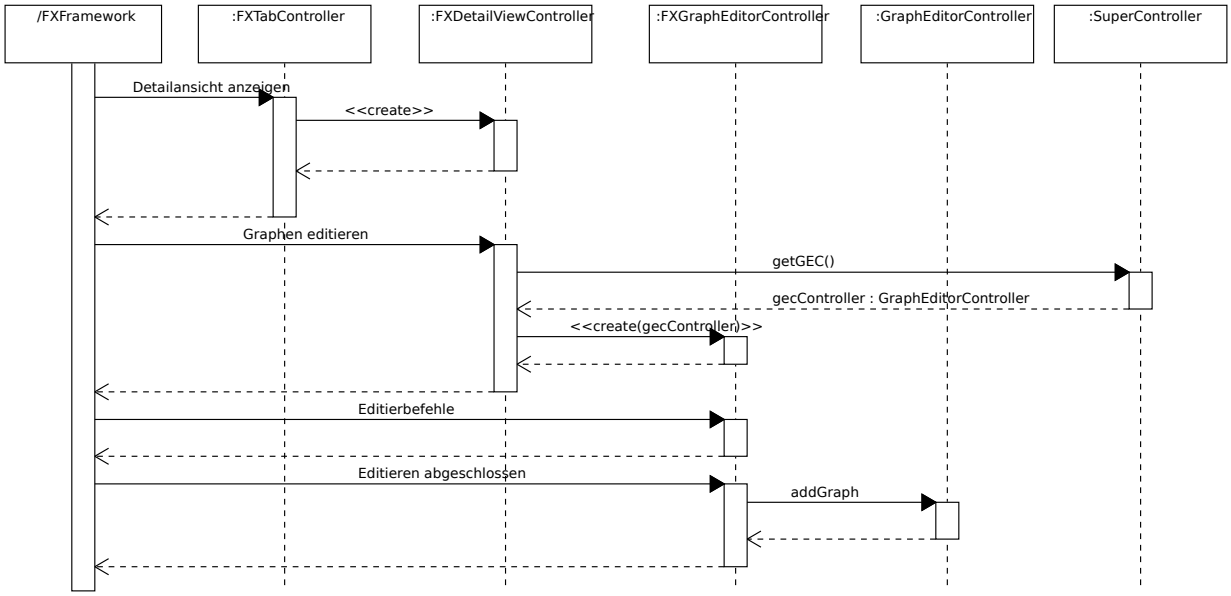

Abbildung 4: Sequenzdiagramm zum Modifizieren von Graphen

3 Model

Package graph

This package contains the interfaces for the interaction with graphs. In the subpackages concrete graph-types are implemented.

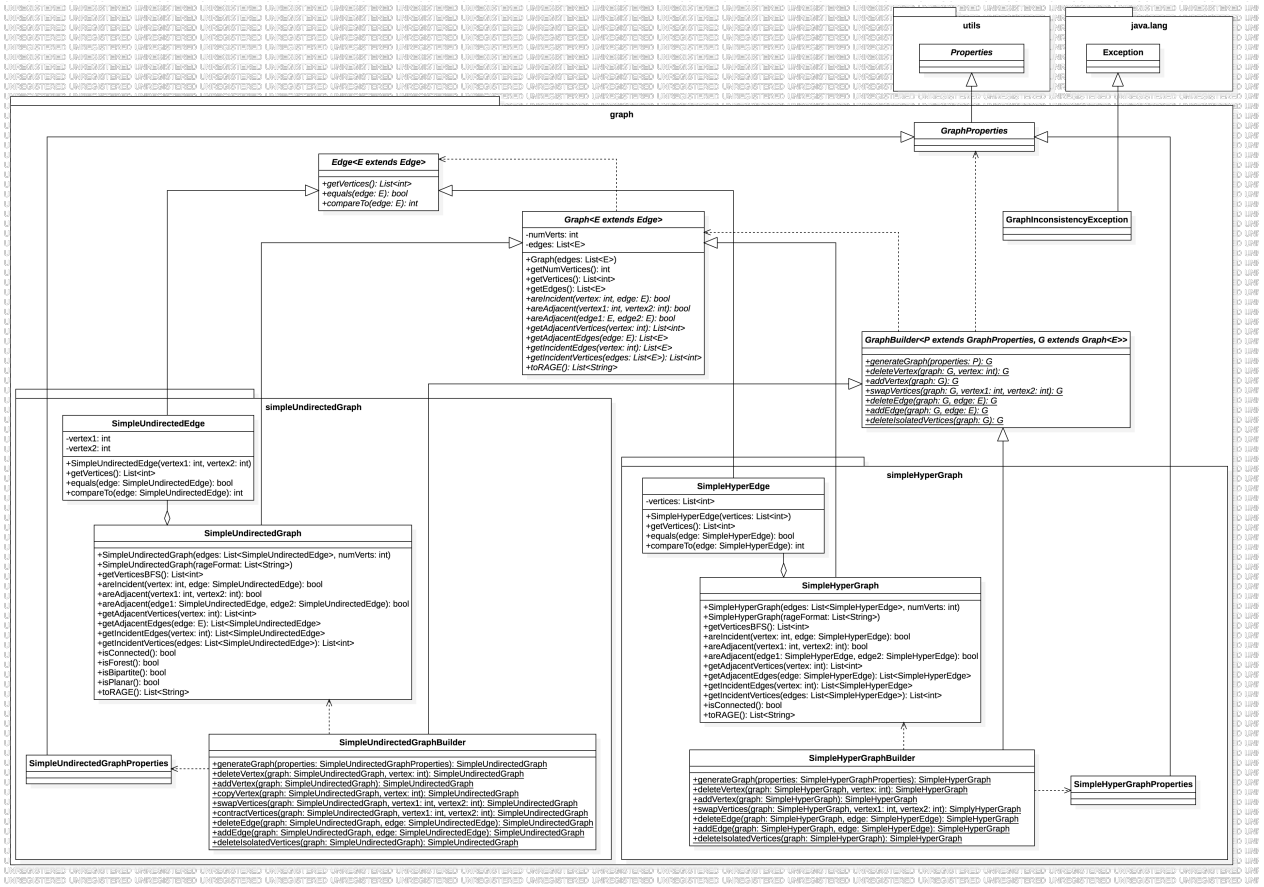

Abbildung 5: Das Paket graph

Class Graph

Description

This class describes the abstract structure of a graph. Each graph has (independent of its concrete type) a finite amount of vertices and edges, which define a relation of vertices. The type **E** of this edges defines the concrete graph type. The class has methods for retrieving the relations given by the edges. Vertices are identified with their unique index and thus are not saved explicitly.

Documentation

+ **Graph(edges: List<E>, numVertices: int)**
the constructor of this class
@param edges the edges belonging to this graph **@numVertices** the number of vertices this graph has

+ **getNumVertices(): int**
@return returns the number of vertices which the graph contains

+ **getVertices(): int**
convenience method for retrieving the list of vertex indices
@return returns the list [0 ... numVertices−1]

+ **getEdges(): List<E>**
@return returns the edges giving the graph its structure

+ **areIncident(vertex: int, edge: E): bool**
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@param edge an edge of the graph
@return returns **true** iff the vertex is incident to the given edge
@throws GraphInconstistencyException if **vertex** is an invalid vertex index or **edge** is not an edge of the graph

+ **areAdjacent(vertex1: int, vertex2: int): bool**
@param vertex1 the index of a vertex of the graph ie. in [0 ... numVertices−1]
@param vertex2 see **vertex1**
@return returns **true** iff there is an edge which is incident to both vertices
@throws GraphInconsistencyException if **vertex1** or **vertex2** is not a valid vertex index

+ **areAdjacent(edge1: E, edge2: E): bool**
@param edge1 an edge of the graph
@param edge2 another edge of the graph
@return returns **true** iff there is a vertex which is incident to both edges
@throws GraphInconsistencyException if **edge1** or **edge2** is not an edge of the graph

+ **getAdjacentVertices(vertex: int): List<int>**
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@return returns the list of all vertices which are adjacent to **vertex**
@throws GraphInconsistencyException if **vertex** is not a valid vertex index

+ **getAdjacentEdges(edge: E): bool**
@param edge an edge of the graph
@return returns the list of all edges which are adjacent to **edge**
@throws GraphInconsistencyException if **edge** is not an edge of the graph

+ **getIncidentEdges(vertex: int): List<E>**
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@return returns the list of all edges incident to **vertex**
@throws GraphInconsistencyException if **vertex** is an invalid vertex index

+ **getIncidentVertices(edges: List<E>): List<int>**
@param edges a list of edges of the graph
@return returns the list of all vertices which are incident to any of the edges in the list
@throws GraphInconsistencyException if there is an edge in **edges**, which is not an edge of the graph

+ ***toRAGE(): List<String>***
@return returns the line-by-line representation of the graph as specified in the RAGE-data format

Class Edge

Description

An edge always defines an adjacency-relation of the vertices incident to it. Moreover this class provides methods to compare edges.

Documentation

+ ***getVertices(): List<int>***
  @return returns the list of all indices of vertices incident to this edge
+ ***equals(edge: E): bool***
  @return returns **true** iff **edge** equals the edge this method is invoked upon. Note that the notion of equality depends on the concrete implementation.
+ ***compareTo(edge: E): int***
  @return returns **-1**/**0**/**1** if **edge** is greater/equal/smaller than the edge this method is invoked upon. Note that the notions of order and equality depend on the concrete implementation.

Class GraphProperties

Description

This class is required for exchanging data between controller and model, especially to signal the settings required to generate graphs. It assures that the following graph-properties can be retrieved and set at all times:

- "graphTypes" – a const list of strings, initialised with ["simpleUndirectedGraph", "simpleHyperGraph"]
- "type" – a string
- "numVertices" – a nonnegative integer

Class GraphBuilder

Description This class is a factory class to generate graphs of type **G** by given GraphProperties **G** as well as to modify graphs of this type.

Documentation

+ ***generateGraph(properties: P): G***
  @param properties the properties which the generated graphs will have
  @return returns a randomly generated graph satisfying the specified **properties**
+ ***deleteVertex(graph: G, vertex: int): G***
  @param graph the graph which is going to be modified
  @param vertex the index of a vertex of **graph**, which will be deleted
  @return returns a modified copy of **graph** in which the vertex with index **vertex** and all edges incident to it are deleted
  @throws GraphInconsistencyException if **graph** has no vertex with index **vertex**
+ ***addVertex(graph: G): G***
  @param graph the graph which is going to be modified
  @return returns a modified copy of **graph** which has precisely one isolated vertex more
+ ***swapVertices(graph: G, vertex1: int, vertex2: int): G***
  @param graph the graph which is going to be modified
  @param vertex1 the index of a vertex of **graph**
  @param vertex2 the index of another vertex of **graph**
  @return returns a modified copy of **graph** in which the vertices having index **vertex1** and **vertex2** swap indices. Note this results in a different but isomorphic graph to **graph**
  @throws GraphInconsistencyException if **graph** has no vertex with index **vertex1** or **vertex2**
+ ***deleteEdge(graph: G, edge: E): G***
  @param graph the graph which is going to be modified
  @param edge the edge which is going to be deleted
  @return returns a modified copy of **graph** in which **edge** is deleted, if it was an edge in **graph**. Otherwise it just returns **graph**

+ ***addEdge(graph: G, edge: E): G***
  @param graph the graph which is going to be modified
  @param edge the edge which is going to be inserted
  @return returns a modified copy of **graph** in which **edge** is inserted if it wasnt already an edge in **graph** otherwise it returns just **graph**. Note that the edge may contain vertices which are not in **graph**, since missing vertices will automatically be added

+ ***deleteIsolatedVertices(graph: G): G***
  @param graph the graph which is going to be modified
  @return returns a modified copy of **graph** in which all isolated vertices are deleted

Class GraphInconsistencyException

Description

This class extends the usual Java Exception to an exception specifically thrown when graphs are treated wrong.

Package graph.simpleUndirectedGraph

In this package **simple undirected graphs** (ie. graphs where edges always connect two distinct vertices x and y, where there is no distinction between edges xy and yx and where there is at most one edge xy) are implemented. It offers methods to generate, modify and distinct them by some (for simple undirected graphs well defined) criterions.

Class SimpleUndirectedGraph

Description

This class concretizes the abstract Graph class in the sense of simple undirected graphs. As mentioned such a graph does not contain any loops or multiedges. Besides incidence relations, this class offers methods to identify properties of simple undirected graphs.

Documentation

+ **SimpleUndirectedGraph(edges: List<SimpleUndirectedEdge>, numVertices: int)**
  a constructor for this class
  @param edges the edges contained in this graph
  @param numVertices the amount of vertices this graph being strictly greater than zero
  @throws GraphInconsistencyException if numVertices ≤ 0 or if there is an edge with a vertex $\geq$ **numVertices** or of there exists an edge more than once

+ **SimpleUndirectedGraph(rageFormat: List<String>)**
  another constructor for this class
  @param rageFormat the lines of the line by line representation as specified in the RAGE data-format.
  @throws GraphInconsistencyException if rageFormat is not a valid representation of SimpleUndirectedGraph

+ **getVerticesBFS(): List<int>**
  @return returns the list of vertices of the graph in the order of a breadth first search

+ **areIncident(vertex: int, edge: SimpleUndirectedEdge): bool**
  @param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @param edge an edge of the graph
  @return returns **true** iff the vertex is incident to the given edge
  @throws GraphInconstistencyException if **vertex** is an invalid vertex index or **edge** is not an edge of the graph

+ **areAdjacent(vertex1: int, vertex2: int): bool**
  @param vertex1 the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @param vertex2 see **vertex1**
  @return returns **true** iff there is an edge which is incident to both vertices
  @throws GraphInconsistencyException if **vertex1** or **vertex2** is not a valid vertex index

+ **areAdjacent(edge1: SimpleUndirectedEdge, edge2: SimpleUndirectedEdge): bool**
  @param edge1 an edge of the graph
  @param edge2 another edge of the graph
  @return returns **true** iff there is a vertex which is incident to both edges
  @throws GraphInconsistencyException if **edge1** or **edge2** is not an edge of the graph

+ **getAdjacentVertices(vertex: int): List<int>**
  @param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @return returns the list of all vertices which are adjacent to **vertex**
  @throws GraphInconsistencyException if **vertex** is not a valid vertex index

+ **getAdjacentEdges(edge: SimpleUndirectedEdge): bool**
@param edge an edge of the graph
@return returns the list of all edges which are adjacent to **edge**
@throws GraphInconsistencyException if **edge** is not an edge of the graph

+ **getIncidentEdges(vertex: int): List<SimpleUndirectedEdge>**
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@return returns the list of all edges incident to **vertex**
@throws GraphInconsistencyException if **vertex** is an invalid vertex index

+ **getIncidentVertices(edges: List<SimpleUndirectedEdge>): List<int>**
@param edges a list of edges of the graph
@return returns the list of all vertices which are incident to any of the edges in the list
@throws GraphInconsistencyException if there is an edge in **edges**, which is not an edge of the graph

+ **isConnected(): bool**
@return returns **true** iff the graph is connected ie. iff for any two vertices there is a sequence of edges where any two consecutive edges are adjacent

+ **isForest(): bool**
@return returns **true** iff the graph is a forest ie. acyclic

+ **isBipartite(): bool**
@return returns **true** iff the vertex set can be partitioned into two parts such that no two vertices from the same partition are adjacent

+ **isPlanar(): bool**
@return returns **true** iff the graph has an embedding into the plane such that no two edges intersect

+ **toRage(): List<String>**
@return returns the line-by-line representation of the graph as specified in the RAGE-data format

Class SimpleUndirectedEdge

Description

This class concretizes the class Edge in the sense of a simple undirected edge. It always relates two distinct vertices.

Documentation

+ **SimpleUndirectedEdge(vertex1: int, vertex2: int)**
a constructor for this class
@param vertex1 the index of the index of the first vertex this edge is incident to
@param vertex2 the index of the index of the second this edge is incident to
@throws GraphInconsistencyException if **vertex1** equals **vertex2**

+ **getVertices(): List<int>**
@return returns the list of all indices of vertices incident to this edge

+ **equals(edge: E): bool**
@return returns **true** iff both edges are adjacent to the same two vertices

+ **compareTo(edge: E): int**
The notion of order between edges (x, y) and (u, v) with $x \leq y$ and $u \leq v$ is defined by $(x, y) < (u, v)$ iff $x < u$ or $(x = u$ and $y < v)$
@return returns **-1**/**0**/**1** if **edge** is greater/equal/smaller than the edge this method is invoked upon

Class SimpleUndirectedGraphProperties

Description

This class is an extension of the GraphProperties class and serves as collection of data for exchange between controller and model, especially to signal the settings required for generating simple undirected graphs. It assures that the following properties can be retrieved and set at all times:

- "minDegree" – a nonnegative integer
- "maxDegree" – a nonnegative integer
- "connected" – a boolean
- "forest" – a boolean
- "bipartite" – a boolean
- "planar" – a boolean

Class SimpleUndirectedGraphBuilder

Description

This class concretizes the GraphBuilder class by offering methods for randomly generating simple undirected graphs after given SimpleUndirectedGraphProperties as well as modifying them.

Documentation

+ **generate(properties: SimpleUndirectedGraphProperties): SimpleUndirectedGraph**
@param properties the properties which the generated graphs will have
@return returns a randomly generated graph satisfying the specified **properties**
+ **deleteVertex(graph: SimpleUndirectedGraph, vertex: int): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param vertex the index of a vertex of **graph**, which will be deleted
@return returns a modified copy of **graph** in which the vertex with index **vertex** and all edges incident to it are deleted
@throws GraphInconsistencyException if **graph** has no vertex with index **vertex**
+ **addVertex(graph: SimpleUndirectedGraph): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@return returns a modified copy of **graph** which has precisely one isolated vertex more

+ **copyVertex(graph: SimpleUndirectedGraph, vertex: int): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param vertex the index of a vertex of **graph**, which will be copied
@return returns a modified copy of **graph** in which the vertex with index **vertex** is duplicated i.e. there is a new vertex which has precisely the same neighborhood
@throws GraphInconsistencyException if **graph** has no vertex with index **vertex**

+ **swapVertices(graph: SimpleUndirectedGraph, vertex1: int, vertex2: int): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param vertex1 the index of a vertex of **graph**
@param vertex2 the index of another vertex of **graph**
@return returns a modified copy of **graph** in which the vertices having index **vertex1** and **vertex2** swap indices. Note this results in a different but isomorphic graph to **graph**
@throws GraphInconsistencyException if **graph** has no vertex with index **vertex1** or **vertex2**

+ **contractVertices(graph: SimpleUndirectedGraph, vertex1: int, vertex2: int): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param vertex1 the index of a vertex of **graph**
@param vertex2 the index of another vertex of **graph**
@return returns a modified copy of **graph** in which the vertices having index **vertex1** and **vertex2** are contracted to a single vertex. Resulting loops will be deleted and multiedges will be reduced to one edge
@throws GraphInconsistencyException if **graph** has no vertex with index **vertex1** or **vertex2**

+ **deleteEdge(graph: SimpleUndirectedGraph, edge: SimpleUndirectedEdge): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param edge the edge which is going to be deleted
@return returns a modified copy of **graph** in which **edge** is deleted, if it was an edge in **graph**. Otherwise it just returns **graph**

+ **addEdge(graph: SimpleUndirectedGraph, edge: SimpleUndirectedEdge): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@param edge the edge which is going to be inserted
@return returns a modified copy of **graph** in which **edge** is inserted if it wasnt already an edge in **graph** otherwise it returns just **graph**. Note that the edge being added may contain vertices which are not in **graph**, since missing vertices will automatically be added

+ **deleteIsolatedVertices(graph: SimpleUndirectedGraph): SimpleUndirectedGraph**
@param graph the graph which is going to be modified
@return returns a modified copy of **graph** in which all isolated vertices are deleted

Package graph.simpleHyperGraph

In this package simple hypergraphs (i.e. graphs whose edges are sets of at least two distinct vertices and whose edges dont overlap in more than one vertex) are implemented. It offers the functionality to generate, modify and distinct them by some for simple hypergraph welldefined criterions.

Class SimpleHyperGraph

Description

This class concretizes the graph class in the sense of a simple hypergraphs. Besides incidence relations this class offers methods to identify some of their properties.

Documentation

+ **SimpleHyperGraph(edges: List<SimpleHyperEdge>, numVertices: int)**
  A constructor for this class
  @param edges the edges this graph contains
  @param numVertices the amount of vertices this graph has
  @throws GraphInconsistencyException if **numVertices** ≤ 0, if there is a hyperedge with a vertex $\geq$ **numVertices** or if the resulting hypergraph is not simple

+ **SimpleHyperGraph(rageFormat: List<String>)**
  A constructor of this class, assuring that this graph type can be loaded from harddrive
  @param rageFormat the line by line representation of the graph as specified in the RAGE data format

+ **getVerticesBFS(): List<int>**
  @return returns the list of vertices of the graph in the order of a breadth first search

+ **areIncident(vertex: int, edge: SimpleHyperEdge): bool**
  @param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @param edge an edge of the graph
  @return returns **true** iff the vertex is incident to the given edge
  @throws GraphInconstistencyException if **vertex** is an invalid vertex index or **edge** is not an edge of the graph

+ **areAdjacent(vertex1: int, vertex2: int): bool**
  @param vertex1 the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @param vertex2 see **vertex1**
  @return returns **true** iff there is an edge which is incident to both vertices
  @throws GraphInconsistencyException if **vertex1** or **vertex2** is not a valid vertex index

+ **areAdjacent(edge1: SimpleHyperEdge, edge2: SimpleHyperEdge): bool**
  @param edge1 an edge of the graph
  @param edge2 another edge of the graph
  @return returns **true** iff there is a vertex which is incident to both edges
  @throws GraphInconsistencyException if **edge1** or **edge2** is not an edge of the graph

+ **getAdjacentVertices(vertex: int): List<int>**
  @param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
  @return returns the list of all vertices which are adjacent to **vertex**
  @throws GraphInconsistencyException if **vertex** is not a valid vertex index

+ **getAdjacentEdges(edge: SimpleHyperEdge): bool**
  @param edge an edge of the graph
  @return returns the list of all edges which are adjacent to **edge**
  @throws GraphInconsistencyException if **edge** is not an edge of the graph

+ **getIncidentEdges(vertex: int): List<SimpleHyperEdge>**
@param vertex the index of a vertex of the graph ie. in [0 ... numVertices−1]
@return returns the list of all edges incident to **vertex**
@throws GraphInconsistencyException if **vertex** is an invalid vertex index

+ **getIncidentVertices(edges: List<SimpleHyperEdge>): List<int>**
@param edges a list of edges of the graph
@return returns the list of all vertices which are incident to any of the edges in the list
@throws GraphInconsistencyException if there is an edge in **edges**, which is not an edge of the graph

+ **isConnected(): bool**
@return returns **true** iff the graph is connected ie. iff for any two vertices there is a sequence of edges where any two consecutive edges are adjacent

+ **toRage(): List<String>**
@return returns the line-by-line representation of the graph as specified in the RAGE-data format

Class SimpleHyperEdge

Description

This class concretizes the class edge in the sense of a hyperedge. It always relates at least two distinct vertices.

Documentation

+ **SimpleHyperEdge(vertices: List<int>)**
A constructor for this class
@param vertices the vertices this edge sets in relation
@throws GraphInconsistencyException if the list is empty, contains just one vertex or any vertex twice

+ **getVertices(): List<int>**
@return returns the list of all indices of vertices incident to this edge

+ **equals(edge: E): bool**
@return returns **true** both edges are adjacent to the same vertices

+ **compareTo(edge: E): int**
The notion of order between edges $(x_1, ..., x_n)$ and $(y_1, ..., y_m)$ with $x_1 < ... < x_n$, $y_1 < ... < y_m$ and $n \leq m$ is defined by $(x1, ..., x_n) < (y_1, ..., y_n)$ iff $x_1 < y_1$ or ($x_1 = y_1$ and $x_2 < y_2$) or ... or ($x_1 = y_1$ and ... and $x_n = y_n$ and $n < m$)
@return returns **-1**/**0**/**1** if **edge** is greater/equal/smaller than the edge this method is invoked upon

Class SimpleHyperGraphProperties

Description

This class is an extension of the GraphProperties class and is likely meant for the exchange of data between controller and model, especially for transferring the settings required for generating simple hyper graphs. It assures that the following graph properties can be retrieved and set at all times:

- "uniform" – a nonnegative integer
- "minDegree" – a nonnegative integer
- "maxDegree" – a nonnegative integer
- "connected" – a boolean

Class SimpleHyperGraphBuilder

Description

This class concretizes the GraphBuilder class by offering methods for randomly generating simple hypergraphs after given SimpleHyperGraphProperties as well as modifying them.

Documentation

+ **generate(properties: SimpleHyperGraphProperties): SimpleHyperGraph**
  @param properties the properties which the generated graphs will have
  @return returns a randomly generated graph satisfying the specified **properties**
+ **deleteVertex(graph: SimpleHyperGraph, vertex: int): SimpleHyperGraph**
  @param graph the graph which is going to be modified
  @param vertex the index of a vertex of **graph**, which will be deleted
  @return returns a modified copy of **graph** in which the vertex with index **vertex** and all edges incident to it are deleted
  @throws GraphInconsistencyException if **graph** has no vertex with index **vertex**
+ **addVertex(graph: SimpleHyperGraph): SimpleHyperGraph**
  @param graph the graph which is going to be modified
  @return returns a modified copy of **graph** which has precisely one isolated vertex more
+ **swapVertices(graph: SimpleHyperGraph, vertex1: int, vertex2: int): SimpleHyperGraph**
  @param graph the graph which is going to be modified
  @param vertex1 the index of a vertex of **graph**
  @param vertex2 the index of another vertex of **graph**
  @return returns a modified copy of **graph** in which the vertices having index **vertex1** and **vertex2** swap indices. Note this results in a different but isomorphic graph to **graph**
  @throws GraphInconsistencyException if **graph** has no vertex with index **vertex1** or **vertex2**
+ **deleteEdge(graph: SimpleHyperGraph, edge: SimpleHyperEdge): SimpleHyperGraph**
  @param graph the graph which is going to be modified
  @param edge the edge which is going to be deleted
  @return returns a modified copy of **graph** in which **edge** is deleted, if it was an edge in **graph**. Otherwise it just returns **graph**

+ **addEdge(graph: SimpleHyperGraph, edge: SimpleHyperEdge): SimpleHyperGraph**
@param graph the graph which is going to be modified
@param edge the edge which is going to be inserted
@return returns a modified copy of **graph** in which **edge** is inserted if it wasnt already an edge in **graph** otherwise it returns just **graph**. Note that the edge being added may contain vertices which are not in **graph**, since missing vertices will automatically be added

+ **deleteIsolatedVertices(graph: SimpleHyperGraph): SimpleHyperGraph**
@param graph the graph which is going to be modified
@return returns a modified copy of **graph** in which all isolated vertices are deleted

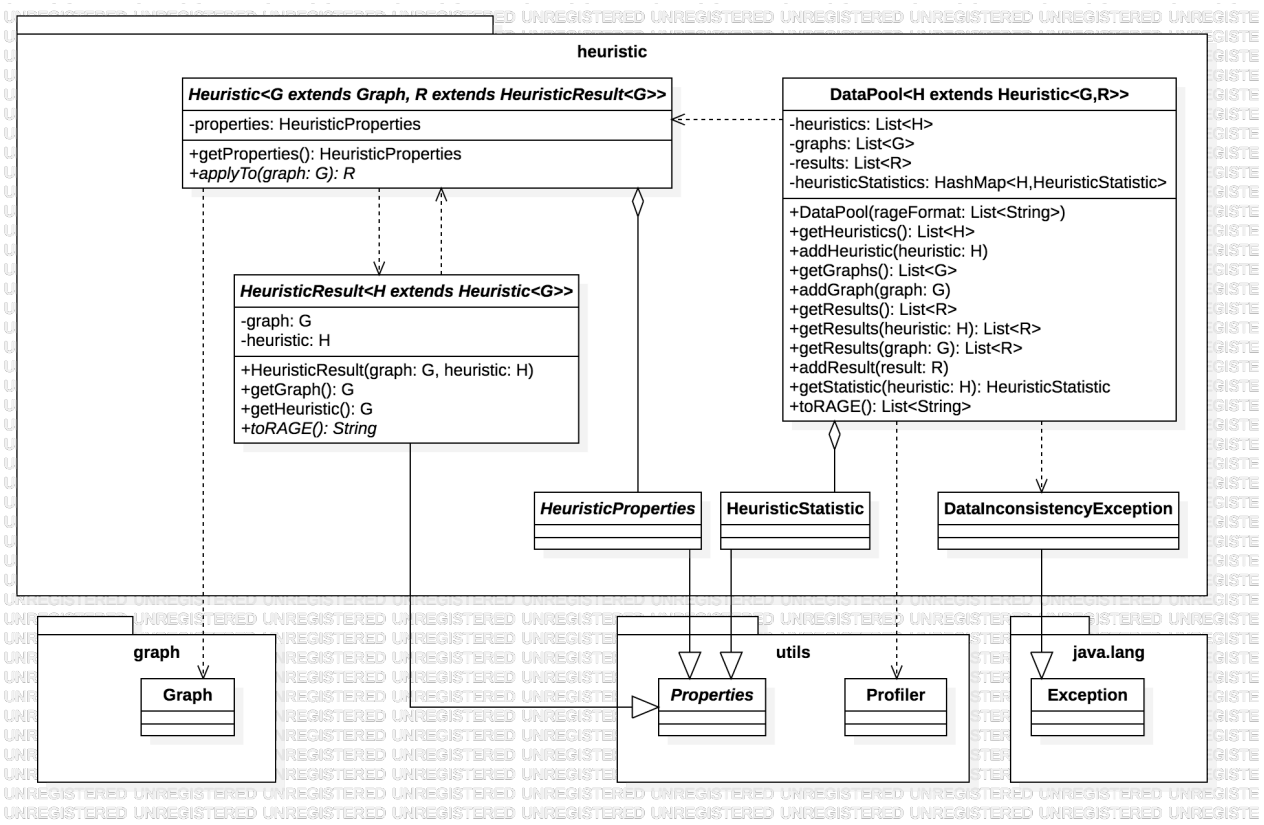

Abbildung 6: Das Paket heuristic

Package heuristic

The package contains the interface for implementing heuristics. In the subpackages some heuristics for the total coloring conjecture as well as for the Erdös-Faber-Lovasz conjecture are implemented.

Class Heuristic

Description

The class is the abstract interface of a heuristic which is applied to a graph of typ **G** which has a result of type **R**.

Documentation

+ **Heuristic(properties: HeuristicProperties)**
  A constructor for this class
  @param properties the properties defining this heuristic
+ **getProperties(): HeuristicProperties**
  @return returns the properties of this heuristic
+ ***applyTo(graph: G): R***
  @param graph the graph of type **G** on which the heuristic will be applied
  @return returns the result of the heuristic application

Class HeuristicResult

Description

This class is the abstract interface of the result of a specific calculation of an heuristic **H** on a specific graph of type **G**.

Documentation

+ **HeuristicResult(graph: G, heuristic: H)**
  The constructor of this class
  @param graph the graph this heuristic was calculated upon
  @param heuristic the heuristic by which the result was calculated
+ **getGraph(): G**
  @return returns the graph this result was calculated upon
+ **getHeuristic(): H**
  @return returns the heuristic by which this result was calculated
+ ***toRAGE(): List<String>***
  @return returns the line-by-line representation of this heuristic result as specified in the RAGE data format

Class HeuristicProperties

Description

This class serves as collection of data for exchange between controller and model, especially to transfer properties of heuristics. It assures that the following properties may be retrieved and set at any time:

- ”name” – ein String
- "valid- ein Boolean

Class DataPool

Description

The class manages the application of heuristics of type **H** on graphs of type **G** which results have type **R**. It assures that every heuristic stored in the pool is applied to every graph stored in the pool. Moreover it gathers statistics over this applications.

Documentation

+ **DataPool(rageFormat: List<String>)**
  A constructor for this class, assuring that the datapool can be loaded from harddrive
  @param rageFormat the line by line representation of a datapool as specified in the RAGE data format.
+ **getHeuristics(): List<H>**
  @return returns the list of heuristics currently in this data pool
+ **addHeuristic(heuristic: H)**
  @param heuristic the heuristic to be added to data pool, which then will be applied to every graph in the data pool
  @throws DataInconsistencyException if heuristic may not be applied on graphs of type **G** or does not has results of type **R**
+ **getGraphs(): List<G>**
  @return returns the list of graphs currently in this data pool
+ **addGraph(graph: G)**
  @param graph the graph to be added to the data pool, on which then all heuristics in the data pool will be applied
  @throws DataInconsistencyException if heuristics of type **H** may not be applied on this graph
+ **getResults(): List<R>**
  @return returns the list of all results calculated on graphs by heuristics in this data pool
+ **getResults(heuristic: H): List<R>**
  @param heuristic the heuristic the results were calculated by
  @return returns all results calculated by **heuristic** on graphs in this data pool
+ **getResults(graph: G)**
  @param graph the graph the results were calculated upon
  @return returns all results calculated on **graph** by heuristics in this data pool

+ **getStatistics(heuristic: H): HeuristicStatistic**
@param heuristic the heuristic whose statistics are requested
@return returns the statistic gathered for **heuristic**
@throws DataInconsistencyException if **heuristic** is not a heuristic of this data pool

+ **toRAGE(): List<String>**
@return returns the line by line representation of this data pool as specified in the RAGE data format

Class HeuristicStatistic

This class collects some statistics overt the applications of a specific heuristic within a data pool. It assures that the following properties may be retrieved at any time:

- "minRuntime" – a floating point number
- "avgRuntime" – a floating point number
- maxRuntime- a floating point number
- numApplications- a nonnegative integer
- numSuccesses- a nonnegative integer

Class DataInconsistencyException

Description

This class extends the usual Java Exception to an exception specifically thrown when data pools are treated wrong.

Package heuristic.totalColoring

In this package and its subpackages some heuristics for the **total coloring conjecture** (ie. any simple undirected graph with maximal degree Δ has a total coloring with $\Delta + 2$ colors) are implemented.

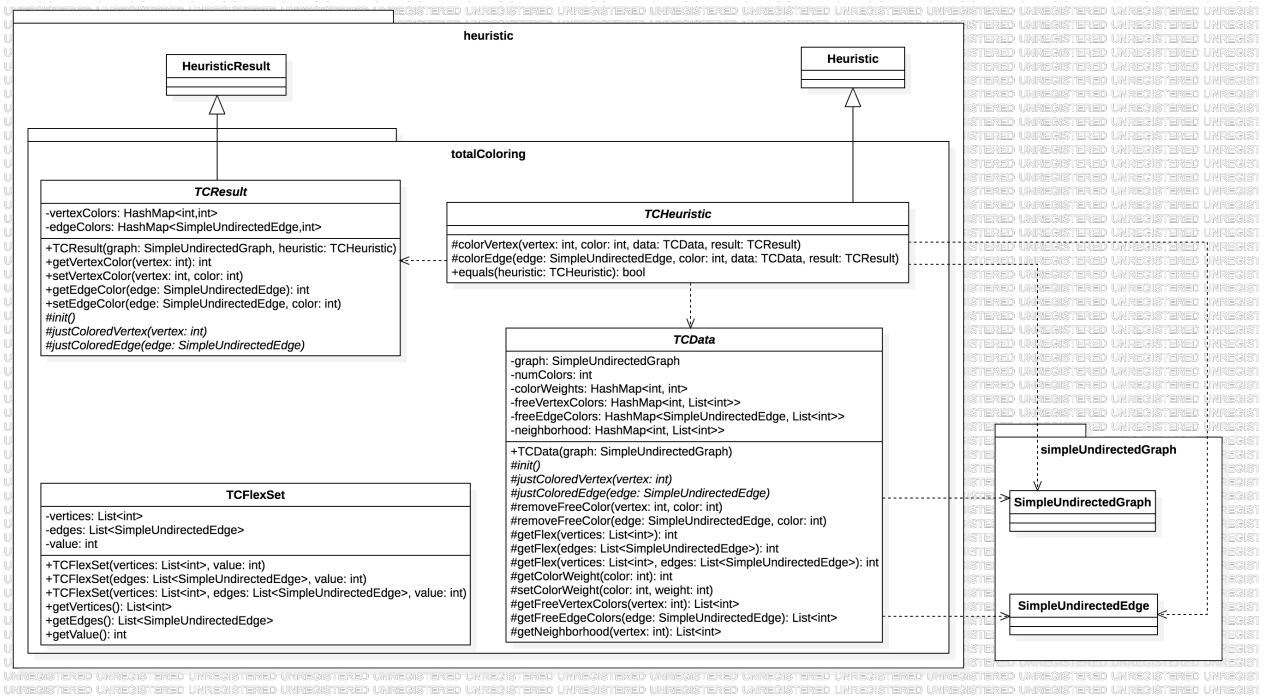

Abbildung 7: totalColoring

Class TCHeuristic

Description

This abstract class is the abstract interface for a total coloring heuristic. It assures that any total coloring heuristic is calculated on SimpleUndirectedGraphs and returns a TCResult as result. It provides some methods, which any total coloring heuristic needs, such as coloring vertices and edges.

Documentation

\# **colorVertex(vertex: int, color: int, data: TCData, result: TCResult)**
@param vertex the vertex to be colored
@param color the color which will be assigned to the vertex
@param data the data required for the calculation of a total coloring
@param result the resulting total coloring

\# **colorEdge(edge: SimpleUndirectedEdge, color: int, data: TCData, result: TCResult)**
@param edge the edge to be colored
@param color the color which will be assigned to the edge
@param data the data required for the calculation of a total coloring
@param result the resulting total coloring

\+ **equals(heuristic: TCHeuristic): bool**
@param heuristic another TCHeuristic this will be compared to
@return returns **true** iff the other TCHeuristic of of the same type and has exactly the same properties

Class TCResult

Description

This class represents a total coloring of a simple undirected graph ie. a coloring of vertices and edges, such that no two adjacent or incident objects share the same color. Colors are represented as integers.

Documentation

+ **TCResult(graph: SimpleUndirectedGraph, heuristic: TCHeuristic)**
  A constructor for this class **@param graph** the graph this result was calculated upon
  @param heuristic the heuristic this result was calculated by

+ **getVertexColor(vertex: int): int**
  @param vertex the vertex whose color is requested
  @return returns the color of **vertex**
  @throws DataInconsistencyException if vertex has no color

+ **setVertexColor(vertex: int, color: int)**
  @param vertex the vertex to be colored
  @param color the color to color **vertex** with

+ **getEdgeColor(edge: SimpleUndirectedEdge): int**
  @param edge the edge whose color is requested
  @return returns the color of **edge**
  @throws DataInconsistencyException if edge has no color

+ **setEdgeColor(edge: SimpleUndirectedEdge, color: int)**
  @param edge the edge to be colored
  @param color the color to color **edge** with

Class TCData

Description

This abstract class encapsulates the data required temporarly to calculate a total coloring, such as the lists of **free colors** of uncolored vertices and edges (ie. the colors which are not used by other objects adjacent / incident to them). Moreover it stores the weighted (vertex vs. edges) sum of how often colors are used.

Documentation

TCData(graph: SimpleUndirectedGraph)
  A constructor of this class
  @param graph the graph the heuristic is running at

init()
  May be implemented to (re-)initialize the data at any time within the running heuristic

justColoredVertex(vertex: int)
May be implemented to update data anytime when a vertex was colored
@param vertex the vertex which was just colored

justColoredEdge(edge: SimpleUndirectedEdge)
May be implemented to update data anytime when an edge was colored
@param edge the edge which was just colored

removeFreeColor(vertex: int, color: int)
@param vertex the vertex which will have one free color less
@param color the color which **vertex** mustnt use

removeFreeColor(edge: SimpleUndirectedEdge, color: int)
@param edge the edge which will have one free color less
@param color the color which **edge** mustnt use

getFlex(vertices: List<int>): int
@param vertices the set of vertices whose flexibility should be calculated
@return returns the flexibility of these vertices ie. # of colors free for all vertices – # of **vertices**

getFlex(edges: List<SimpleUndirectedEdge>): int
@param vertices the set of vertices whose flexibility should be calculated
@return returns the flexibility of these vertices ie. # of colors which are free for all edges – # of **edges**

getFlex(vertices: List<int>, edges: List<SimpleUndirectedEdge>): int
@param vertices a set of vertices
@param edges a set of edges
@return returns the flexibility of these objects ie. # of colors which are free for all objects – # of **objects**

getColorWeight(color: int): int
@param color the color whose weight is requested
@return returns the weight of this color ie. how often it was used weighted differently by vertices and edges

setColorWeight(color: int, weight: int)
@param color the color whose weight will be updated
@param weight the new weight of **color**

getFreeVertexColors(vertex: int): List<int>
@param vertex the vertex whose free colors are requested
@return returns the list of free colors of **vertex**

getFreeVertexColors(edge: SimpleUndirectedEdge): List<int>
@param edge the edge whose free colors are requested
@return returns the list of free colors of **edge**

Class TCFlexSet

Description

This class represents a subset of vertices and edges of a graph with a given **flexibility value** (ie. # colors free for all objects – # objects) used heavily in some TCHeuristics.

Documentation

TCFlexSet(vertices: List<int>, value: int)
A constructor of this class
@param vertices some vertices
@param value the flexibility value of **vertices**

TCFlexSet(edges: List<SimpleUndirectedEdge>, value: int)
A constructor of this class
@param edges some edges
@param value the flexibility value of **edges**

TCFlexSet(vertices: List<int>, edges: List<SimpleUndirectedEdge>, value: int)
A constructor of this class
@param vertices some vertices
@param edges some edges
@param value the flexibility value of the set of objects in **vertices** and **edges**

getVertices(): List<int>
@return returns the vertices in this flex set

getEdges(): List<SimpleUndirectedEdge>
@return returns the edges in this flex set

getValue(): int
@return returns the flexibility value of this set of objects

Package heuristic.totalColoring.greedy

In this package some greedy heuristics for the total coloring conjecture are implemented. They all have in common, that the vertices are colored first and the edges are colored afterwards. The heuristics differ in the way the edges are colored.

Class TCGreedyData

Description

Since TCData is abstract this class is required such that the TCGreedy heuristic has its own data class, even if with respect to TCData no additional attributes or methods are added.

Class TCGreedy

Description

This class implements the TCGreedy heuristic which tries to calculate a total coloring.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
  implements the following heuristic

```
for every vertex v in order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

for every vertex v in order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

  @param graph the graph this heuristic will be applied on
  @return returns the calculated coloring

Class TCGreedyOneData

Description

This class stores all uncolored edges with exactly one free color temporarily.

Documentation

init()
initializes the list of all uncolored edges with exactly one free color

justColoredEdge(edge: SimpleUndirectedEdge)
updates the list of edges with exactly one free color
@param edge the edge which was just colored

- **calcSingularEdges()**
updates the list of edges with exactly one free color

getMinimalSingularEdge(): SimpleUndirectedEdge
@return returns the minimal edge with exactly one free color with respect to the order defined on edges

Class TCGreedyOne

Description

This class implements the TCGreedyOne heuristic which tries to calculate a total coloring.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
implements the following heuristic

```
for every vertex v in the order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

for every vertex v in the order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        while there are uncolored edges with exactly one free color
            get minimal uncolored edge with exactly one free color f
            get minimally used free color c of f with respect to the color weights
            color f with color c
        if e is colored already
            continue
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Class TCGreedyFewData

Description

This class stores all uncolored edges sorted first by their amount of free colors and then by the order defined on edges.

Documentation

- # **init()**
  initializes the list of uncolored edges
- # **justColoredEdge(edge: SimpleUndirectedEdge)**
  updates the list of uncolored edges
- # **getMinimalUncoloredEdge(): SimpleUndirectedEdge**
  @return returns the minimal uncolored edge with respect to the number of free colors and the order defined on edges

Class TCGreedyFew

Description

This class implements the TCGreedyFew heuristic, which tries to calculate a total coloring.

Documentation

- \+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
  implements the following heuristic

```
for every vertex v in the order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

for every vertex v in the order of a breadth first search
    for every uncolored edge e incident to v in the order defined on edges
        while there are uncolored edges with less free colors than e and lower order than e
            get minimal uncolored edge f
            if f cannot be colored
                return incomplete coloring
            get minimally used free color c of f with respect to the color weights
            color f with color c
        if e is colored already
            continue
        if e cannot be colored
            return incomplete coloring
        get minimally used free color c of e with respect to the color weights
        color e with color c

return complete coloring
```

  @param graph the graph this heuristic will be calculated on
  @return returns the calculated coloring

Class TCGreedySetData

Description

This class stores for any vertex v the subset of all uncolored edges incident to v which has the lowest flexibility value (ie. # of colors which are free for every edge in this set − # of edges in the set) and is the lowest with respect to lexicographic ordering using the order defined on edges. These sets are from now on referred to as minimal flex sets

Documentation

init()
initializes the minimal flex sets

justColoredEdge(edge: SimpleUndirectedEdge)
updates the minimal flex sets of the vertices incident to edge

- **calcMinimalFlexSet(vertex: int)**
calculates the minimal flex set of **vertex**
@param vertex the vertex whose minimal flex set is calculated

getMinimalFlexSet(): TCFlexSet
@return returns the minimal flex set belonging to the vertex with minimal index

Class TCGreedySet

Description

This class implements the TCGreedySet heuristic, which tries to calculate a total coloring.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
implements the following heuristic

```
for every vertex v in the order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

while there is a set with minimal flexibility
    find the set X of minimal flexibility belonging to the vertex v with lowest index
        if X has negative flexibility
            return incomplete coloring
        for every edge e of X in the order defined on edges
            if e cannot be colored
                return incomplete coloring
            get minimally used free color c of e with respect to the color weights
            color e with color c

return complete coloring
```

@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Class TCGreedyConData

Description

This class stores the list of uncolored edges temporarily to compute connected subsets of uncolored edges up to a specific size.

Documentation

init()
initializes the list of uncolored edges

justColoredEdge(edge: SimpleUndirectedEdge)
updates the list of uncolored edges

getMinimalFlexSet(): TCFlexSet
@return returns the connected set of uncolored edges with minimal flexibility value (# of colors which are free for all edges – # of edges) and minimal lexicographic order using the order defined on edges.

Class TCGreedyCon

This class implements the TCGreedyCon heuristic, which tries to calculate a total coloring.

Documentation

+ **applyTo(graph: SimpleUndirectedGraph): TCResult**
implements the following heuristic

```
for every vertex v in the order of a breadth first search
    if v cannot be colored
        return incomplete coloring
    get minimally used free color c of v with respect to the color weights
    color v with color c

while there is a set with minimal flexibility
    find the set X of minimal flexibility which has the lowest lexicographic order
        if X has negative flexibility
            return incomplete coloring
        for every edge e of X in the order defined on edges
            if e cannot be colored
                return incomplete coloring
            get minimally used free color c of e with respect to the color weights
            color e with color c

return complete coloring
```

@param graph the graph this heuristic will be calculated on
@return returns the calculated coloring

Package heuristic.totalColoring.mixedGreedy

In this package some heuristics for the total coloring conjecture are implemented. In comparison to the greedy heuristics, these heuristics do not seperate the coloring of vertices and edges strictly but rather alternate between them.

Class TCMixedGreedyData

Class TCMixedGreedy

Class TCMixedGreedyOneData

Class TCMixedGreedyOne

Class TCMixedGreedyFewData

Class TCMixedGreedyFew

Class TCMixedGreedySetData

Class TCMixedGreedySet

Class TCMixedGreedyConData

Class TCMixedGreedyCon

Package heuristic.erdosFaberLovasz

Class EFLHeuristic

Class EFLResult

Package heuristic.erdosFaberLovasz.greedy

Class EFLGreedyData

Class EFLGreedy

Class EFLGreedyOneData

Class EFLGreedyOne

Class EFLGreedyFewData

Class EFLGreedyFew

Class EFLGreedySetData

Class EFLGreedySet

Class EFLGreedyConData

Class EFLGreedyCon

4 View

4.1 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.

1. Standart Widget Toolkit (SWT)
2. Abstract Widget Toolkit (AWT)
3. Swing
4. 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.

4.1.1 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.

4.1.2 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).

4.2 Entwurf

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

1. Graphic
2. Drawer
3. Sound

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

Package Graphic

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

Package Graphic.UIElements

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

Class 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.

Package 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.

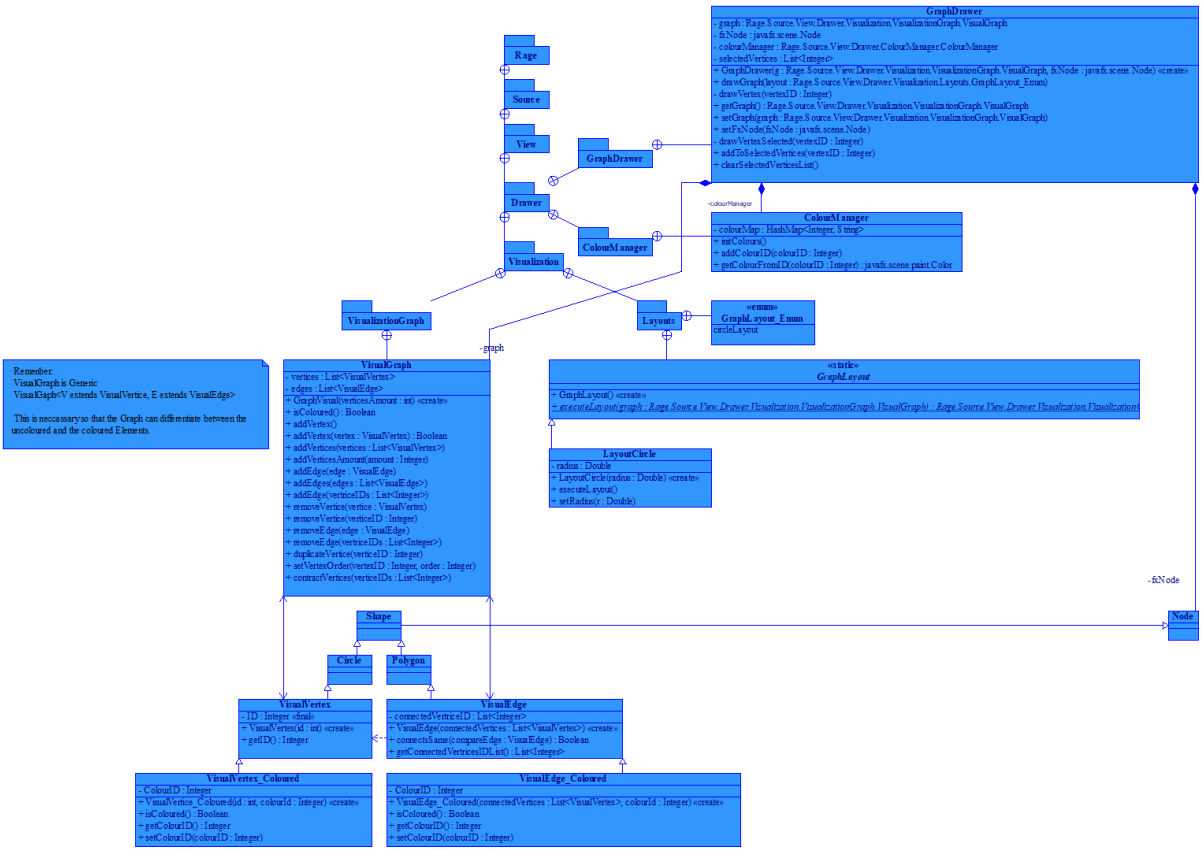

Abbildung 8: ViewDrawer

Package 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.

Class 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.

Package 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.

Class 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.

Package Drawer.Layouts

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

Class 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.

Class 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.

Class 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.

Package Drawer.Visualization.VisualizationGraph

Class 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>"

Class 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>"

Class 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>, ..."

Class 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>"

Class 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.

Package Sound

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

Class 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 FXController

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

5.2.1 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.

5.2.2 Entwurf

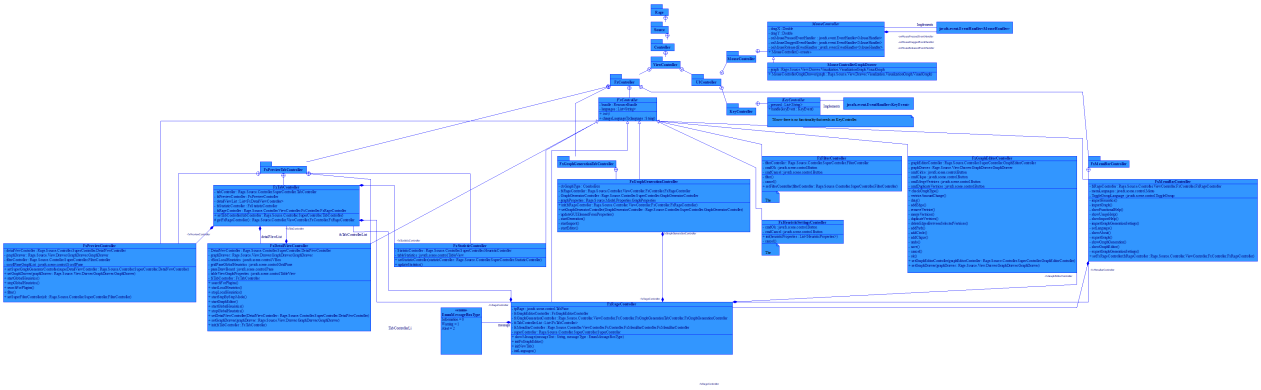

Abbildung 9: ViewController

6 Resources

6.1 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.

6.2 Entwurf

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

Package 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.

7 Controller

Package Controller

Manages interaction with the user and asks the model to execute tasks.

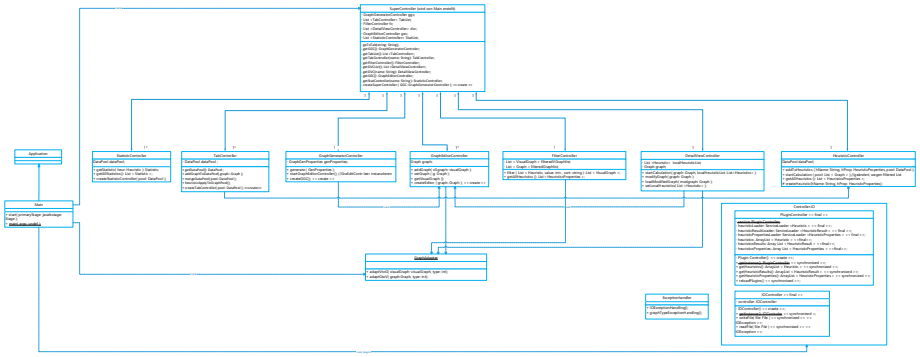

Abbildung 10: Controller

Class SuperController

Description

The SuperController has one or more instances of theGrapgGeneratorController, List of TabController, GraphEditorController.
Documentation

+ **SuperController**
  Constructor: Creates a SuperController and gives him immedeately a GraphGeneratorController instance.
  @param GGC The Param GGC is the instance of a GraphGeneratorController.

+ **getGGC**
  @return GraphGeneratorController.

+ **createGEC**
  Creates a new GraphEditorController with(out) a graph to display.
  @param pool The DataPool, where the created graph from the user will be added.
  @param graphl The Graphl, that should be modified.

+ **getTabList**
  @return List <TabController>.

+ **getTabController**
  @param name name is the PreviewTab identifier, with it, the SuperController can identify the current TabController, the User is working on.
  @return TabController.

+ **getGEC**
  @return GraphEditorController.

+ **createTabController**
  Creates a new preview tab, with a graph liast and a heuristics list.
  @param graphList List of graphs that should be taken to the new tab.
  @param heurList List of heuristics that should be taken to the new tab.
  @return TabController.

+ **createTabController**
  Creates a new preview tab with its own DataPool, and calls the GrapgGeneratorController to generate graphs for the DataPool and it will show the graphs in the preview Tab.
  @param GgenPropertiest The properties, that dictates how the random graph generation generates graphsö.
  @return TabController.

- **PRIVATEMETHODE** etc

Class StatisticController

Description

Reads the statistics for a heuristic out of the Model and collects them to show it to the View.

Documentation

+ **StatisticController**
  Constructor: Creates a StatisticController and gets himself a DataPool. **@param pool** The DataPool, that the StatisticController belongs to.

+ **getAllStatistics**
  @return List <Statistic >.

+ **getStatistic**
  @param heur heur is the Name of the Heuristic, that you want the statistics from.
  @return Statistic.

- **PRIVATEMETHODE** etc

Class TabController

Beschreibung

The Controller of exactly one Preview Tab in the View, that manages the DataPool of this Preview Tab.

Dokumentation

+ **TabController**
  Constructor: Creates a new TabController and connects it with an own DataPool, it also creates an own StatisticController.
  @param tabnamel The name of this TabController.
  @param pool The DataPool, that belongs to the TabController.

+ **getDVCList**
  @return List <DetailViewController>.

+ **getDVC**
  @param name The name is the DetailViewController identifier, with it, theTabController can identify the current DetailViewController, the User is working on.
  @return DetailViewController.

+ **getDataPool**
  @return DataPool

+ **addGraphToDataPool**
  Adds one Graph to the DataPool, that belongs to the TabController instance.
  @param graph The graph that should be added to the DataPool.
  @throws EXCEPTION if the type of the Graph is not of the same graph type in the DataPool.

+ **mergeDataPool**
  Merges two DataPools under one of the two TabController. The other TabController with its DataPool remains untouched.
  @param pool The DataPool, that should be copied.
  @throws EXCEPTION if the graph type of both DataPools is not equal.

+ **getStatisticController**
@return StatisticController

+ **getHeuristicController**
@return HeuristicController

+ **getFilterController**
@return FilterController

+ **heuristicApplyToDataPool**
Calls the HeuristicController to collor the graphs.

+ **createHeuristicController**
Instanciates a HeuristicController and gives him a DataPool. **@param pool** The given DataPool.

+ **createDetailViewController**
Instanciates a DetailViewController and gives him a graph to display with all heuristics, that tried to collor it. **@param graphPositionl** The position of the graph in the graph list in the given DataPool.

+ **createFilterController**
Instanciates a FilterController and gives him a DataPool. **@param pool** The given DataPool.

- **PRIVATEMETHODE** etc

Class GraphGeneratorController

Beschreibung

The controller for the graph generation communication between the view and the GraphBuilder in the Model.

Dokumentation

+ **GraphGeneratorController**
Constructor: Creates a GraphGeneratorController.

+ **generate**
Commands the GraphBuilder to create random graphs with specific properties.
@param genProperties The properties, that restrict the randomnes of the GraphBuilder.

+ **createManuallyGraph**
Creates an empty GraphEditorController, that adds that manually generated graph from a user.
It calls the SuperController to start the method createGEC without a DataPool and without a graph.

- **PRIVATEMETHODE** etc

Class GraphEditorController

Beschreibung

Manages the manipulated or created graph by the user and adds it to the right DataPool.

Dokumentation

+ **GraphEditorController**
  Constructor: Creates a GraphEditorController and it will get a DataPool instance. When it was created by the GraphGeneratorController, it will create an GraphEditorController without a graph.
  If it was created by the DetailViesController, it will get a graph to the new instance.

+ **setGraph**
  Sets the graph of this instance.
  @param g The graph, that belongs to this instance.

+ **addGraph**
  Adapts the created visualGraph to a Graph and adds the created graph to the DataPool. If there is no DataPool, it will create a new one.
  @param vGraph The created visualGraph.

+ **getVisualGraph**
  Returns the Graph of this instance as a visualGraph.

- **PRIVATEMETHODE** etc

Class FilterController

Beschreibung

Controlls the filter set by the user and manages the filtered graph pool and the graph pool in the DataPool.

Dokumentation

+ **createFilterController**
  Constructor: Creates a FilterrController and it will get a DataPool instance.

+ **filter**
  Filters the graph list from the DataPool.
  @param List <Heuristic, value: int> It determines how the list will be filtered.
  @param sort It determines how list will be sorted (decending, ascending ...).
  @return returns List <VisualGraph>, the DataPool remains untouched.
  @throws EXCEPTION if filter and sort are contradictory.

+ **getAllHeuristics**
  Returns all properties of the used heuristics and the heuristic name is also a heuristic property.
  @return returns List <HeuristicProperties>

- **PRIVATEMETHODE** etc

Class DetailViewController

Beschreibung

Manages the chosen graph and the heuristics that only apply to this graph, the so called local heuristics.

Dokumentation

+ **startCalculation**
  Applies the local Heuristics to the one graph in the DetailView.
  @param graph The graph that get colloerd by the local heuristics.
  @param localHeuristicList The list of local heuristics, that collors the one graph.

+ **modifyGraph**
  Creates a GraphEditorController instance with the one graph.
  @param graph The graph that should be modified.

+ **loadModifiedGraph**
  Loads the modified graph into the DataPool and in to the DetailViewController.
  @param modgraph The modified graph.

+ **addLocalHeuristics**
  Adds the local heuristics chosen by the user to the localHeuristics.
  @param List <Heuristic> The local Heuristics.

+ **DetailViewController**
  Creates a new Detail View Controller and creates an empty localHeuristicsList.
  @param graph The graph, thatshouldbe loaded in to the DetailViewController.

- **PRIVATEMETHODE** etc

Class HeuristicController

Beschreibung

Manages the chosen graph and the heuristics that only apply to this graph, the so called local heuristics.

Dokumentation

+ **HeuristicController**
  Creates a HeuristicController and gives him a DataPool.
  @param pool The DataPool to give.

+ **addToHeuristics**
  Applies the chosen heuristics to the heuristic pool in the DataPool. Calls the createHeuristics method.
  @param hName The name of the heuristic.
  @param hProp The properties of the heuristic.
  @param pool The DataPool, where the heuristics belong.

+ **startCalculation**
  Commands the Heuristics to calculate their results on the graph pool.
  @param pool The graph list.
  @param hpool The heuristicList of the DataPool, that should calculate the result on the graph list.

+ **getAllHeuristics**
Returns all possible Heuristics.
@return returns List <HeuristicsProperties>.

#

+ **createHeuristics**
Get invoked by addToHeuristics and commands the model to create a new heuristic.
@param hName The heuristic name.
@param hProp The heuristic properties.

8 Input-Output

Package IO

This package contains classes for input, output and plugin loading.

Class PluginController

Beschreibung

Loads all Heuristic, HeuristicResult and HeuristicProperties classes using the ServiceLoader class. It uses the singelton design pattern.

Dokumentation

+ **getInstance():PluginController**
This method is the only way to access the PluginController. Creates a new PluginController if it does not exist.
@return returns the PluginController itself.

+ **getHeuristics():ArrayList<Heuristic>**
Loads all Heuristic classes if they are not already loaded.
@return returns a list with all Heuristics.

+ **getHeuristicResults():ArrayList<HeuristicResult>**
Loads all HeuristicResult classes if they are not already loaded.
@return returns a list with all HeuristicResult classes.

+ **getHeuristicProperties():ArrayList<HeuristicProperties>**
Loads all HeuristicProperties classes if they are not already loaded.
@return returns a list with all HeuristicProperties classes.

+ **reloadPlugins()**
Clears the pluginlists and then loads all plugins.

Class IOController

Beschreibung

Saves and loads the data of a single view tab. The file has the extension „.RAGE“. It uses the singelton design pattern.

Dokumentation

+ **getInstance():IOController**
  This method is the only way to access the IOController. Creates a new IOController if it does not exist.
  @return returns the IOController itself.

+ **writeFile(File file)**
  Writes a RAGE file to the disk.
  @param file Information about the file.
  @throws IOException if saving fails print: „Error while saving the file.“.

+ **readFile(File file)**
  Reads a RAGE file from the disk and sends the content to the model.
  @param file Information about the file.
  @throws IOException if loading fails print: „Error while loading the file.“.

9 Utils

10 Addendum: Heuristiken

11 Addendum: RAGE-Datenformate