Entwurfsheft

Gereon Recht Robin Link Cem Özcan Felix Köhler
Alexander Schorn Collin Lorbeer

7. Juni 2018

Inhaltsverzeichnis

1 Klarstellung Pflichtenheft

1.1 Filter

Die im Pflichtenheft genannte Möglichkeit, eine Aussage für ein bestimmtes Filterkriterium zu negieren, wird nicht realisiert. Grund dafür ist, dass die Funktionalität ebenso durch Invertierung des Relationsoperators erreicht werden kann.

1.2 Tabelle

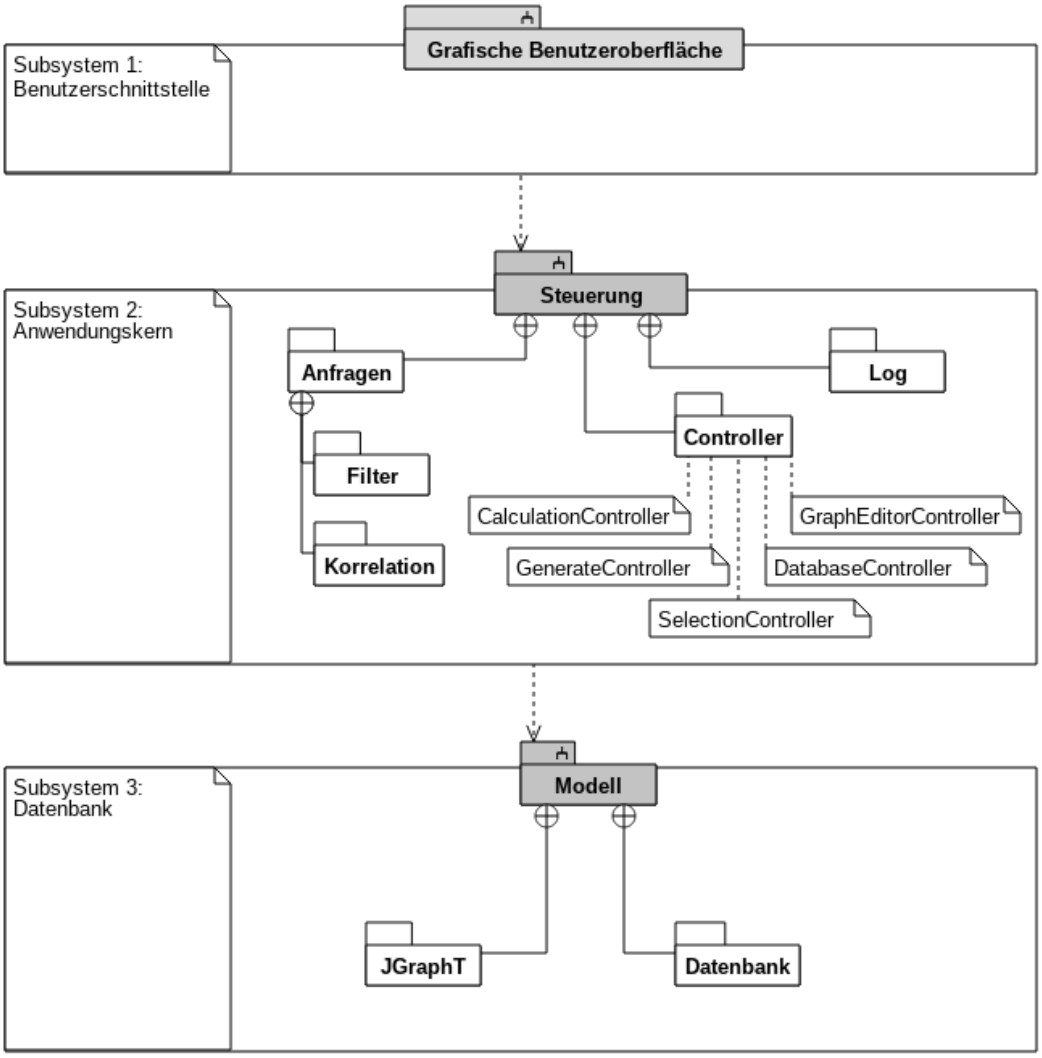
Im Pflichtenheft wurde spezifiziert, dass standardmäßig nach dem Namen der Graphen eine Sortierung der Graphen stattfindet. Diese Entscheidung wurde dahingehend geändert, dass nun standardmäßig nach der eindeutigen Identifikationsnummer der Graphen sortiert wird.

2 Einleitung

Im Rahmen des Moduls Praxis der Softwareentwicklung am Karlsruher Institut für Technologie wird das Software-Projekt *Grape* entwickelt. Das Programm soll dazu dienen, große Mengen von Graphen zu generieren, auszuwerten, und auf Korrelationen von Eigenschaften zu untersuchen. Die Auswertung erfolgt anhand von Filtern, mit denen Graphen mit bestimmten Eigenschaften ausgewählt werden können.
In diesem Dokument werden die Architektur sowie die Paketverteilung des Programms beschrieben. Über eine Benutzeroberfläche können sowohl Graphen durch den Benutzer generiert, gelöscht und auch modifiziert werden, als auch benutzerdefinierte Filter und Filtergruppen erstellt werden. Auch die visuelle Ansicht der Graphen in einem Graphen-Editor ist durch die Benutzeroberfläche möglich.
Die Datenbankanbindung findet über `java.sql` statt, und die Graphenbibliothek `JGraphT` wird als Basis für das Programm genutzt.
Dieses Dokument soll als Grundlage für die Implementierung von *Grape* dienen und die Strukturierung so festlegen, dass keine Entscheidungen dieser Art mehr getroffen werden müssen. Im Laufe des Dokumentes wird zunächst die Systemarchitektur von *Grape* beschrieben und danach die einzelnen Pakete genauer spezifiziert und dargestellt. Als Abschluss werden die wichtigsten Algorithmen, die *Grape* verwendet, als Pseudocode dargestellt.

2.1 Systemarchitektur

Grape baut sich auf einer 3-Schichten-Architektur auf. Die Benutzerinteraktion erfolgt über die grafische Benutzerschnittstelle, welche die Ereignisse an die Steuerung weiterleitet. Diese wiederum synchronisiert die Benutzeroberfläche mit dem Model. Dabei werden die verwendeten Daten im Modell gehalten.



3 Bibliothek JGraphT und Funktionalität

3.1 Beschreibung

JGraphT ist eine Java-Graphenbibliothek, welche graphentheoretische Objekte und Algorithmen bietet. Sie unterstützt verschiedene Graphentypen, wie beispielsweise gerichtete und ungerichtete Graphen.
Im Folgenden werden nur die Teile der Bibliothek in Diagrammen dargestellt, die erweitert werden. Weitere Details sind in der Dokumentation zu JGraphT nachzulesen (http://jgrapht.org/javadoc/).

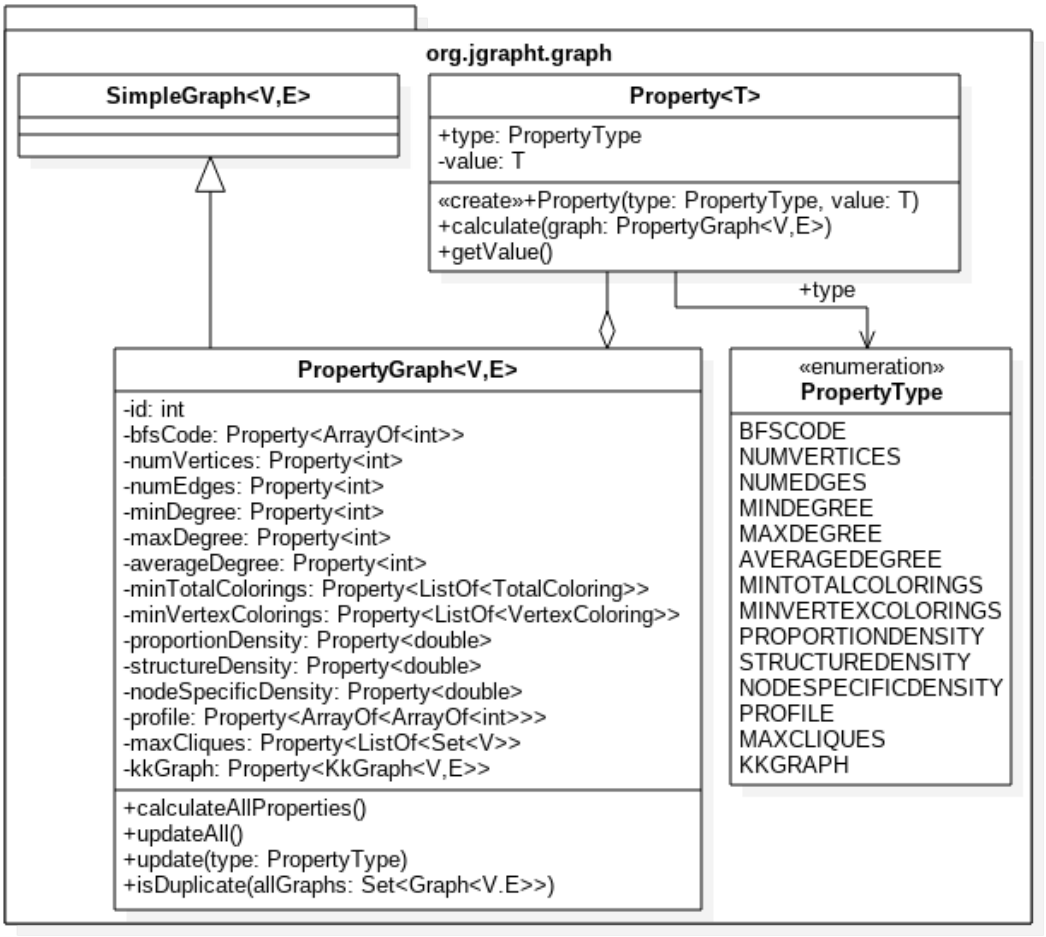
3.2 Paket org.jgrapht.graph

3.2.1 Beschreibung

Das Paket `org.jgrapht.graph` bietet verschiedene Graphenklassen, welche alle das Interface `Graph<V,E>` implementieren. Hier wird `SimpleGraph<V,E>` verwendet, was einen einfachen ungerichteten Graphen darstellt. Die Klasse wird mittels eines Dekorierers um weitere Funktionalität erweitert.

3.2.2 Interner Aufbau

Das Paket wird mit der Klasse `Property<T>`, welche eine Eigenschaft repräsentiert, und der Klasse `PropertyGraph<V,E>`, welche einen `SimpleGraph<V,E>` mit Eigenschaften darstellt, erweitert. `PropertyGraph<V,E>` erbt also von der Klasse `SimpleGraph<V,E>` und enthält `Property<T>`-Objekte.
Zwecks Übersichtlichkeit wurde auf die Getter-Methoden in `PropertyGraph<V,E>` verzichtet.



3.2.3 Feinentwurf

Class Property<T> A class that represents a property of a graph.

- Field Detail:
  - `type : PropertyType`
    The property's type
  - `value : T`
    The value of the property with type T
- Method Detail:

- `calculate(graph : PropertyGraph<V,E>)`
  Calculates the specific value of the graph property.
  `@param graph` - the graph whose property value is to be calculated

Class PropertyGraph<V,E> A `SimpleGraph<V,E>` that contains `Property<V,E>` objects.

- Field Detail:
  - `id : int` - The unique identifier of the graph
  - `numVertices : Property<int>` - the number of the vertices inside the graph
  - `numEdges : Property<int>` - the number of edges in the graph
  - `minDegree : Property< int>` - the minimal degree of the graph. No node has a lower degree but at least one node's degree is this value.
  - `maxDegree : Property<int>` - the maximal degree of the graph. No node has higher degree but at least one node's degree is this value.
  - `averageDegree : Property<int>` - the average degree of all vertices in the graph.
  - `minTotalColoring : Property<ListOf<TotalColoring>>` - A list of all total colorings. Every list entry contains a total coloring of the graph.
  - `minVertexColoring : Property<ListOf<VertexColoring>>` - A list of vertex colorings. Every list entry contains a vertex coloring of the graph.
  - `bfsCode : Property<ArrayOf<int>>` - The BFS Code represented as an array of integer values.
  - `proportionDensity : Property<double>` - the density of the graph calculated with the proportion density formula.
    Es sei G ein Graph mit n Kanten.

$$proportionDensity(G) := \frac{n}{n_{max}},$$

mit n_{max} := größtmögliche Anzahl von Kanten in G.

- `binomialDensity : Property<double>` - the density of the graph calculated with the binomial density formula.
  Es sei G ein Graph mit n Kanten und k Knoten.

$$binomialDensity(G) := \frac{n}{\binom{k}{2}},$$

- `structureDensity : Property<double>` - the density of the graph calculated with the structure density formula.
  Es sei G ein Graph mit n Kanten und k Knoten.

$$structureDensity(G) := \frac{\sum_{i=1}^{b} \frac{1}{g_i - f_i}}{\sum_{i=1}^{k-2} i \frac{1}{k-1-i}},$$

$$\text{mit } e_i = \{g_i, f_i\} \text{ eine } Backwarkante \text{ mit } g_i > f_i \, (1 \leq i \leq b)$$

- `nodeSpecificDensity : Property<double>` - the density of the graph calculated with the node specific density formula.
  Es sei G ein Graph mit n Kanten, k Knoten, und b Backwardkanten.

$$nodeSpecificDensity(G) := \frac{b}{n},$$

- `profile : Property<ArrayOf<ArrayOf<int>>>` - the profile of a graph.

- `maxCliques : Property<ListOf<Set<V>>>` - a list of all max cliques of the graph

- `kkGraph : Property<KkGraph<V,E>>` - the K_k graph of the graph

- Method Detail:

  - `calculateAllProperties() : void`
    calculate all properties of the graph. This method only calculates the value of a property if this value is `NULL`.

  - `updateAll() : void`
    updates all properties.

  - `update(type : PropertyType) : void`
    updates the value of the given property.
    `@param type` - the property's PropertyType

– `isDuplicate(allGraphs : Set<Graph<V,E>) : boolean`
determines whether the database already contains the graph
`@return` true, if the set already contains this graph

Enum PropertyType An `enum` containing the various possible types for `Property<T>`.

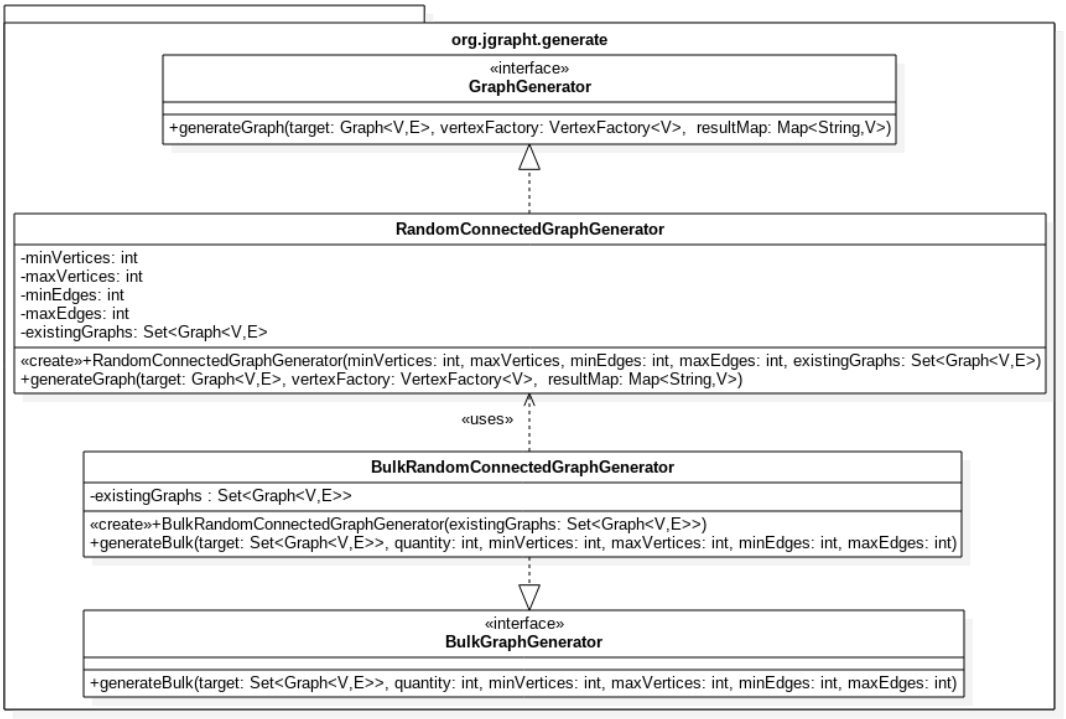
3.3 Paket org.jgrapht.generate

3.3.1 Beschreibung

Das Paket `org.jgrapht.generate` bietet verschiedene Generatoren für Graphen mit unterschiedlicher Struktur. Da jedoch kein Generator existiert, welcher einen zufälligen, streng zusammenhängenden Graphen generiert, wird dieses Paket um einen weiteren Graphgenerator `RandomConnectedGraphGenerator` erweitert. Des Weiteren werden dem Paket das Interface `BulkGraphGenerator` und die Klasse `BulkRandomConnectedGraphGenerator` hinzugefügt, wodurch eine Menge von Graphen generiert werden kann.

3.3.2 Interner Aufbau

Das Paket `org.jgrapht.generate` enthält das zentrale Interface `GraphGenerator`, welches alle Generatoren implementieren müssen. Dieses beinhaltet die Methode `generateGraph()`.
`RandomConnectedGraphGenerator` generiert die von *Grape* benötigten Graphen und implementiert das o.g. Interface.



3.3.3 Feinentwurf

Class RandomConnectedGraphGenerator<V,E>

Creates a random graph which is strongly connected, but does not create self-loops or multiple edges between the same two vertices.

- Constructor Detail:
  - `RandomConnectedGraphGenerator(minVertices : int, maxVertices : int, minEdges : int, maxEdges : int, existingGraphs : Set<Graph<V,E>>)`.
    Creates a new instance of `RandomConnectedGraphGenerator<V,E>`.
    `@param minVertices` - minimal number of vertices
    `@param maxVertices` - maximal number of vertices
    `@param minEdges` - minimal number of edges
    `@param maxEdges` - maximal number of edges
    `@param existingGraphs` set of graphs which already exist in a database
- Method Detail:

- `generateGraph(target : Graph<V,E> , vertexFactory : VertexFactory<V>, resultMap : Map<String,T>) : void`
  Generates a random, strongly connected graph.
  `@param target` - the target graph
  `@param vertexFactory` - the vertex factory
  `@param resultMap` - not used by this generator, can be `NULL`
  `@throws IllegalArgumentException` - Thrown if it is impossible to create a strongly connected graph with the restrictions passed in the constructor.

Interface BulkGraphGenerator<V,E>

A `BulkGraphGenerator` creates a set of graphs with certain restrictions. Just like in mathematics, sets do not contain duplicates.

- Method Detail:

  - `generateBulk(target: Set<Graph<V,E>>, quantity: int, minVertices: int, maxVertices: int, minEdges: int, maxEdges: int) : void`
    Generates a set of graphs with the given restrictions.
    `@param target` - the target graph
    `@param quantity` - the number of graphs to be generated.
    `@param minVertices` - minimal number of vertices
    `@param maxVertices` - maximal number of vertices
    `@param minEdges` - minimal number of edges
    `@param maxEdges` - maximal number of edges

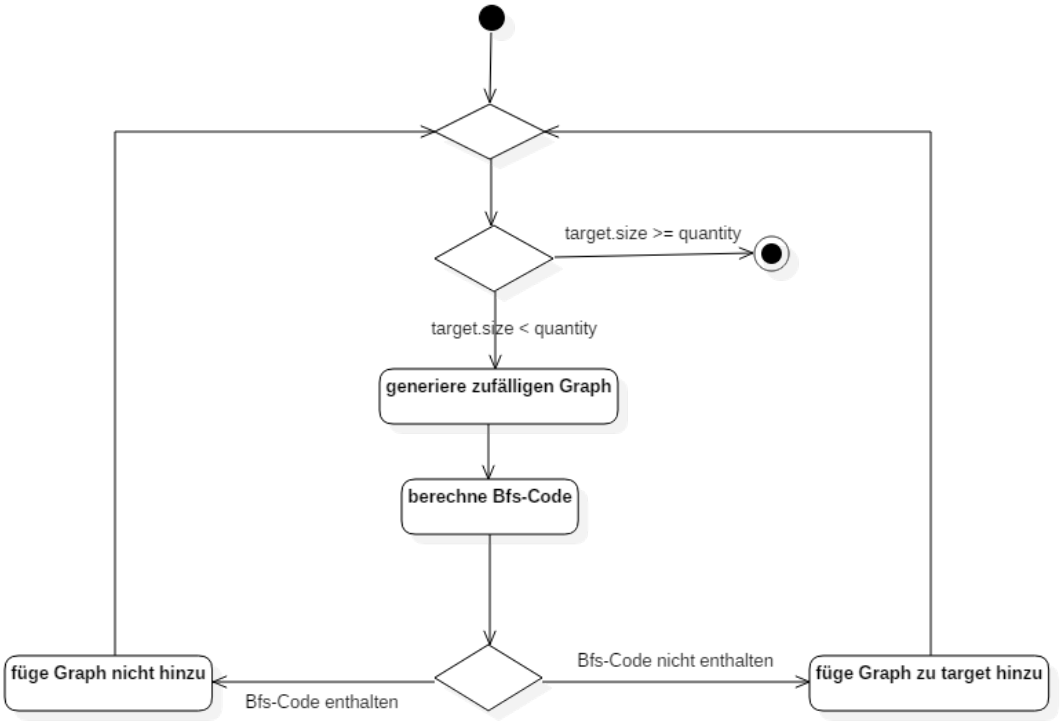
Class BulkRandomConnectedGraphGenerator<V,E>

A `BulkGraphGenerator` which creates a set of graphs that are strongly connected. The actual number of generated graphs may be lower than the specified one, as the given restrictions may limit the number of possible graphs.
This class uses `RandomConnectedGraphGenerator<V,E>` to create each individual graph and implements `BulkGraphGenerator<V,E>`.

- Constructor Detail:

  - `BulkRandomConnectedGraphGenerator(existingGraphs : Set<Graph<V,E>>)`
    Creates a new instance of `BulkRandomGraphGenerator`.
    `@param existingGraphs` set of graphs which already exist in a database

Das folgende Aktivitätsdiagramm zeigt den groben Ablauf der Graphgenerierung des `BulkRandomConnectedGraphGenerator`:



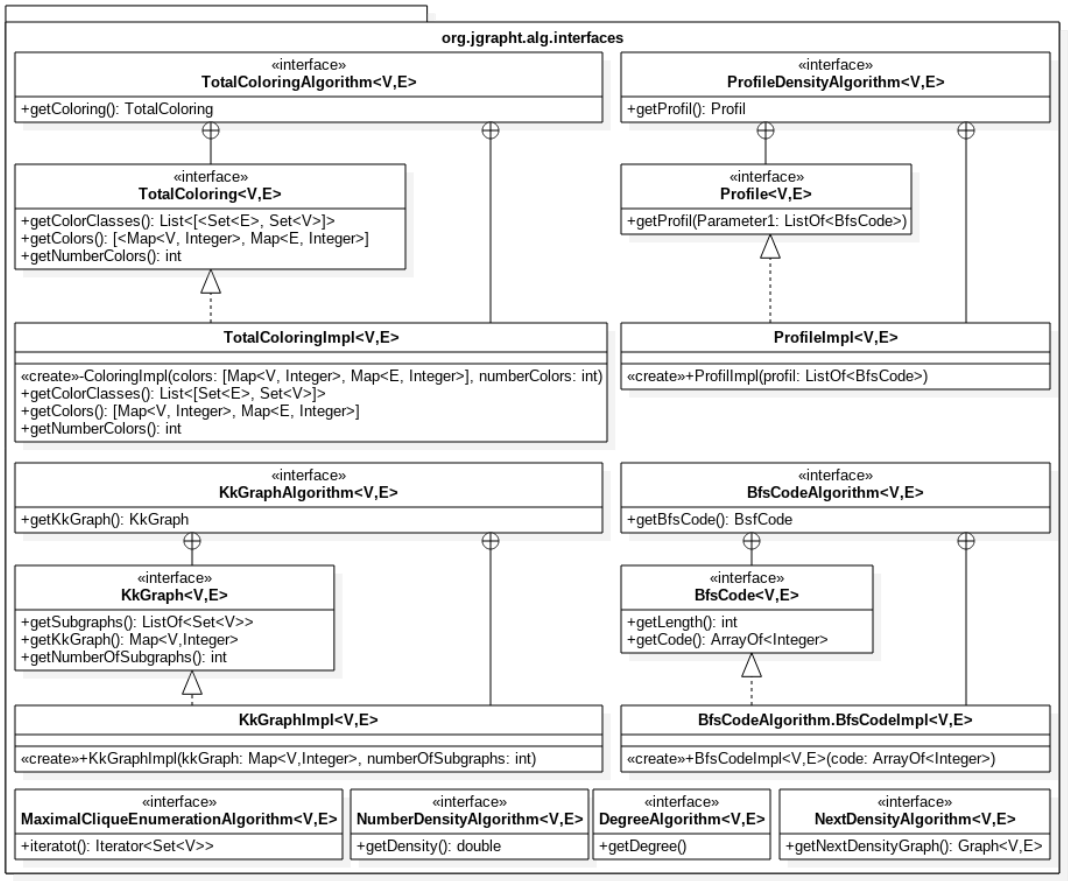
3.4 Paket org.jgrapht.alg.interfaces

3.4.1 Beschreibung

Das Paket `org.jgrapht.alg.interfaces` enthält alle für die verwendeten Algorithmen benötigten Interfaces und Standardimplementierungen von manchen dieser Schnittstellen. Des Weiteren verfügt dieses Paket auch über das Interface `VertexColoringAlgorithm.Coloring<V>`, welches eine Knotenfärbung darstellt. Zur Darstellung und Berechnung von Totalfärbungen muss dieses Paket um die Interfaces `TotalColoringAlgorithm<V,E>`, `TotalColoringAlgorithm.Coloring<V,E>`erweitert werden, sowie um `DensityAlgorithm<V,E>` und `DensityAlgorithm.Density<V,E>` als Repräsentation der Dichte eines Graphen. Außerdem werden Interfaces für die Eigenschaftsberechnungen des Knotengrads, des K_k-Graphen, und der Dichte hinzugefügt.

3.4.2 Interner Aufbau

Das Paket `org.jgrapht.alg.interfaces` enthält die zentralen Interfaces für Algorithmen in JGraphT.



3.4.3 Feinentwurf

Interface TotalColoringAlgorithm<V,E>

A total coloring algorithm.

- Method Detail:
  - `getColoring() : TotalColoring<V,E>`
    `@return` a TotalColoring of a graph.

Interface TotalColoringAlgorithm.TotalColoring<V,E>
A total coloring.

- Method Detail:
  - `getColorClasses() : List<Set<ColoredElement>>`
    `@return` sets of `ColoredElement` with the same color class.
  - `getColors() : Map<ColoredElement, Integer>`
    `@return` map of `ColoredElement` to their corresponding color, represented by an `Integer` value.
  - `getNumberColors() : int`
    `@return` the number of colors required for the coloring.

Class TotalColoringAlgorithm.ColoringImpl<V,E>
Default implementation of `TotalColoringAlgorithm.Coloring<V,E>`.

- Constructor Detail:
  - `ColoringImpl(colors:Map<ColoredElement,Integer>, numberColors:int)`
    `@return` new ColoringImpl<V,E> Object.

Interface BfsCodeAlgorithm<V,E>
A BFS Code algorithm.

- Method Detail:
  - `getBfsCode() : BfsCode`
    `@return` the BFS Code.

Interface BfsCodeAlgorithm.BfsCode<V,E>
A BFS Code.

- Method Detail:
  - `getLength() : int`
    `@return` the length of the code

- `getCode() : ArrayOf<Integer>`
  `@return` the code represented by an array of integers.

Interface BfsCodeAlgorithm.BfsCodeImpl<V,E>
Default implementation of `BfsCodeAlgorithm.BfsCode<V,E>`.

- Constructor Detail:

  - `BfsCodeImpl(code:ArrayOf<Integer>)`
    `@return` new BfsCodeImpl<V,E> Object.

Interface ProfileDensityAlgorithm<V,E>
A graph profile algorithm.

- Method Detail:

  - `getProfile() : Profile<V,E>`
    `@return` the profile of the graph.

Interface ProfileDensityAlgorithm.Profile<V,E>
A profile of a Graph.

- Method Detail:

  - `getProfile() : ListOf<BfsCode>`
    `@return` the profile in a list of .

Interface ProfileDensityAlgorithm.ProfileImpl<V,E>
Default implementation of `ProfileDensityAlgorithm.Profile<V,E>`.

- Constructor Detail:

  - `BfsCodeImpl(profile: List<BfsCode>)`
    `@return` new `ProfileImpl<V,E>` object.

Interface NumberDensityAlgorithm<V,E>
A graph density algorithm that returns a number.

- Method Detail:

  - `getDensity() : double`
    `@return` the density of the graph.

Interface DegreeAlgorithm<V,E>

An algorithm that handles degrees in graphs, such as determining the maximal or minimal degree.

- Method Detail:

  - `getDegree() : int`
    `@return` a degree

Interface NextDensityAlgorithm<V,E>

An algorithm that returns the closest new graph which has a higher density.

- Method Detail:

  - `getNextDensityGraph() : Graph<V,E>`
    `@return` the closest graph with a higher density.

Interface KkGraphAlgorithm<V,E>

A K_k-graph algorithm.

- Method Detail:

  - `getKkGraph() : KkGraph`
    `@return` the K_k-graph of a graph based on the Hadwiger Conjecture.

Interface KkGraphAlgorithm.KkGraph<V,E>

A K_k-graph.

- Method Detail:

  - `getSubgraphs() : ListOf<Set<V>>`
    `@return` the different subgraphs of the K_k-graph.

- `getKkGraph() : Map<V,Integer>`
  `@return` the map of the K_k-graph. Every node has an integer that represents the id of his subgraph.

- `getNumberOfSubgraphs() : int`
  `@return` the number of the different subgraphs of the K_k-graph.

Class KkGraphAlgorithm.KkGraphImpl<V,E>
Default implementation of `KkGraphAlgorithm.KkGraph<V,E>`

- Constructor Detail:

  - `KkGraphAlgorithm.KkGraphImpl(kkGraphl : Map<V,Integer>, numberOfSubgraphs : int)`
    `@return` new `KkGraphAlgorithm.KkGraphImpl<V,E>` object

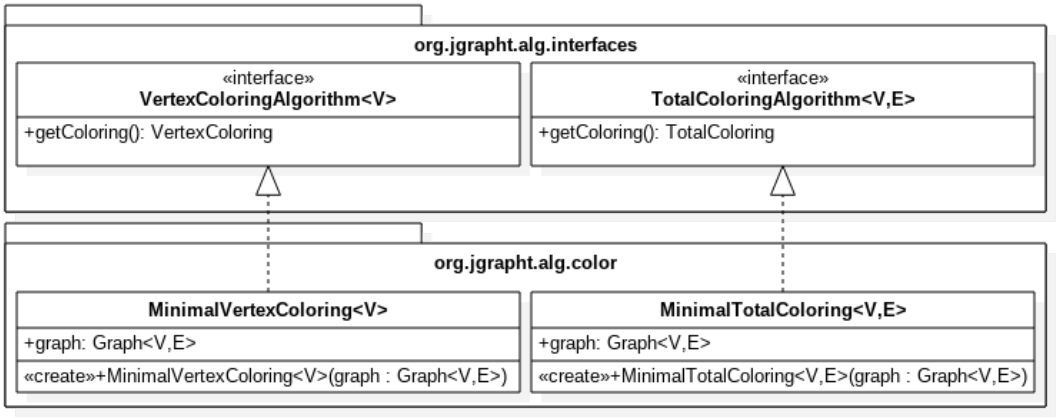
3.5 Paket org.jgrapht.alg.color

3.5.1 Beschreibung

Das Paket `org.jgrapht.alg.color` enthält Algorithmen bezüglich der Färbung von Graphen.

3.5.2 Interner Aufbau

Es wird ein Färbungsalgorithmus benötigt, der jeweils eine minimale Färbung eines Graphen zurückgibt. Da `org.jgrapht.alg.interfaces` dies nicht bietet, muss das Paket um weitere Klassen erweitert werden.
Hierzu wird das Paket um die Klassen `MinimalVertexColoring<V,E>`, welches eine minimale Knotenfärbung berechnet, und `MinimalTotalColoring<V,E>`, welches eine minimale Totalfärbung berechnet, ergänzt. Dabei implementiert `MinimalVertexColoring<V,E>` das Interface `VertexColoringAlgorithm<V>` und `MinimalTotalColoring<V,E>` das Interface `TotalColoringAlgorithm<V,E>`.



3.5.3 Feinentwurf

Class MinimalVertexColoring<V,E>
A minimal vertex coloring algorithm.

- Field Detail:
  - `protected Graph<V,E> graph`
    The input graph
- Constructor Detail:
  - `MinimalVertexColoring(Graph<V,E> graph)`
    Constructs a new coloring algorithm
    `@param graph` - the input graph
- Method Detail:
  - `getColoring() : VertexColoring<V>`
    `@return` a vertex coloring
    Computes a minimal vertex coloring.
    Specified by `getColoring()` in interface `VertexColoringAlgorithm<V>`

Class MinimalTotalColoring<V,E>
A minimal total coloring algorithm.

- Field Detail:
  - `protected Graph<V,E> graph`
    The input graph
- Constructor Detail:
  - `MinimalTotalColoring(Graph<V,E> graph`
    Constructs a new coloring algorithm
    `@param graph` - the input graph
- Method Detail:
  - `getColoring()`
    Computes a minimal total coloring.
    `@return` a total coloring
    Specified by by `getColoring()` in interface `TotalColoringAlgorithm<V>`

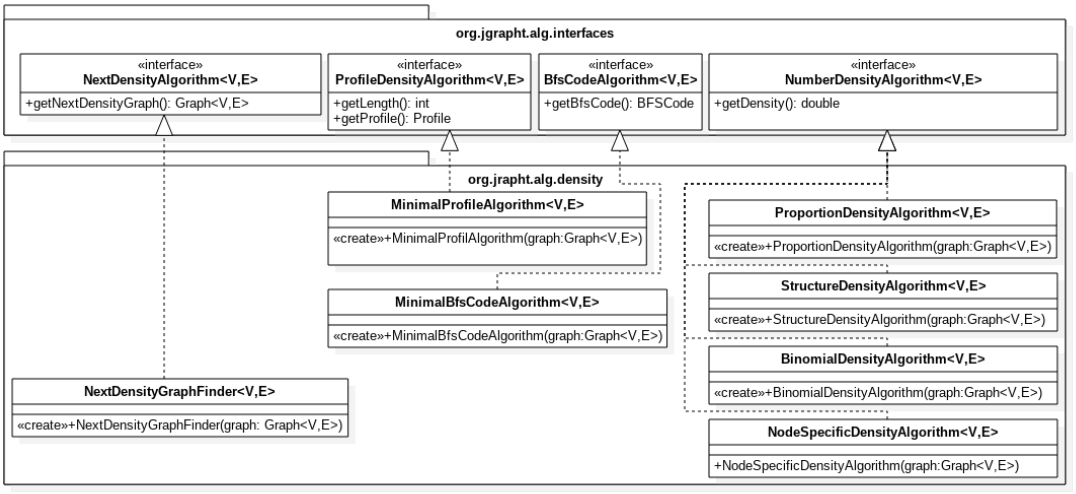
3.6 Paket org.jgrapht.alg.density

3.6.1 Beschreibung

Das Paket `org.jgrapht.alg.density` enthält Algorithmen für die Dichtenberechnung von Graphen.

3.6.2 Interner Aufbau

Das Paket besteht aus einzelnen Klassen, die jeweils einen Algorithmus zur Berechnung einer Dichtedefinition repräsentieren. Jede Klasse implementiert eines der im Paket `org.jgrapht.alg.interfaces` definierten Interfaces, welches der jeweiligen Dichtedefinition entspricht.
Dazu enthält das Paket die Klassen `PorportionDensityAlgorithm<V,E>`, `StructureDensityAlgorithm<V,E>`, `BinomialDensityAlgorithm<V,E>`, und `NodeSpecificDensityAlgorithm<V,E>`, welche jeweils das Interface `NumberDensityAlgorithm<V,E>` implementieren, da sie eine Zahl als Identifikator der Dichte berechnen.
Außerdem gibt es die Klassen `MinimalBfsCodeAlgorithm<V,E>`, welche das Interface `BfsCodeAlgorithm<V,E>` implementiert, und `MinimalProfileAlgorithm<V,E>`, die das Interface `ProfileDensityAlgorithm<V,E>` implementiert.



3.6.3 Feinentwurf

Class NodeSpecificDensityAlgorithm<V,E>

The node specific algorithm. It computes the density of a Graph with the S1 Density that is defined in the product requirements document.

- Constructor Detail:
  - `NodeSpecificDensityAlgorithm(Graph<V,E> graph`
    Constructs a new node specific density algorithm for a graph
    `@param graph` - the input graph
- Method Detail:
  - `getDensity() : double`
    Computes the density of the input graph with the specified density formula
    `@return`: a density number
    Specified by `getDensity()` in interface `NumberDensityAlgorithm<V,E>`

Class BinomialDensityAlgorithm<V,E>

The binomial algorithm. It computes the density of a Graph with the S2 Density that is defined in the product requirements document.

- Constructor Detail:

- `BinomialDensityAlgorithm(Graph<V,E> graph`
  Constructs a new binomial specific density algorithm for a graph
  `@param graph` - the input graph

- Method Detail:

  - `getDensity() : double`
    Computes the density of the input graph with the specified density formula
    `@return` a density number
    Specified by `getDensity()` in interface `NumberDensityAlgorithm<V,E>`

Class ProportionDensityAlgorithm<V,E>

The proportion algorithm. It computes the density of a Graph with the S3 Density that is defined in the product requirements document.

- Constructor Detail:

  - `ProportionDensityAlgorithm(Graph<V,E> graph`
    Constructs a new proportion specific density algorithm for a graph
    `@param graph` - the input graph

- Method Detail:

  - `getDensity() : double`
    Computes the density of the input graph with the specified density formula
    `@return` a density number
    Specified by `getDensity()` in interface `NumberDensityAlgorithm<V,E>`

Class StructureDensityAlgorithm<V,E>

The structure algorithm. It computes the density of a Graph with the S4 Density that is defined in the product requirements document.

- Constructor Detail:

  - `StructureDensityAlgorithm(Graph<V,E> graph`
    Constructs a new structure-specific density algorithm for a graph
    `@param graph` - the input graph

- Method Detail:

  - `getDensity() : double`

Computes the density of the input graph with the specified density formula
`@return`: a density number
Specified by `getDensity()` in interface `NumberDensityAlgorithm<V,E>`

Class MinimalBfsCodeAlgorithm<V,E>

The minimal BFS Code algorithm. It computes the minimal BFS Code of a graph.

- Constructor Detail:
  - `MinimalBfsCodeAlgorithm(Graph<V,E> graph`
    Constructs a new BFS Code algorithm for a graph
    `@param graph` - the input graph
- Method Detail:
  - `getBfsCode() : BfsCode<V,E>`
    Computes the minimal BFS Code of the input graph
    `@return`: a BFS Code
    Specified by `getBfsCode()` in interface `BfsCodeAlgorithm<V,E>`

Class MinimalProfileAlgorithm<V,E>

The minimal profile algorithm. It computes the profile of a graph.

- Constructor Detail:
  - `MinimalProfileAlgorithm(Graph<V,E> graph`
    Constructs a new minimal profile algorithm for a graph
    `@param graph` - the input graph
- Method Detail:
  - `getProfile() : Profile<V,E>`
    Computes the profile of a graph
    `@return` the profile
    Specified by `getProfile()` in interface `ProfileDensityAlgorithm<V,E>`

Class NextDensityGraphFinder<V,E>

The next density algorithm finder. It computes the next density graph.

- Constructor Detail:

- `NextDensityGraphFinder(Graph<V,E> graph`
  Constructs a new next density graph finder for a given graph
  `@param graph` - the input graph

- Method Detail:

  - `getNextDensityGraph() : Graph<V,E>`
    Computes the profile of a graph
    `@return` the next density graph that is strongly connected
    Specified by `getNextDensityGraph()` in interface `NextDensityAlgorithm<V,E>`

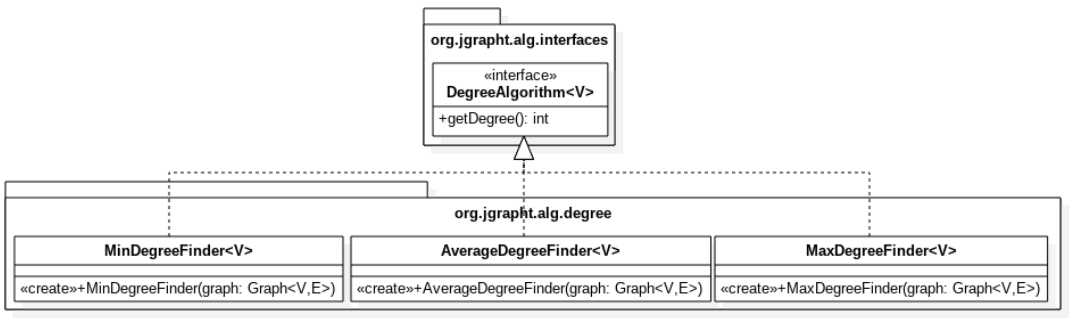
3.7 Paket org.jgrapht.alg.degree

3.7.1 Beschreibung

Das Paket `org.jgrapht.alg.degree` enthält Algorithmen für das Finden bestimmter Knotengrade in Graphen.

3.7.2 Interner Aufbau

Das Paket besteht aus drei Klassen, die bestimmte Knotengrade innerhalb eines Graphen finden können:
`MinDegreeFinder<V>`, `MaxDegreeFinder<V>` und `AverageDegreeFinder<V>`. Alle implementieren `DegreeAlgorithm<V>` aus dem Paket `org.jgrapht.alg.interfaces`.



3.7.3 Feinentwurf

Class MinDegreeFinder<V>
The minimal degree finder. It determines the smallest degree of all nodes within a graph.

- Constructor Detail:
  - `MinDegreeFinder(Graph<V> graph)`
    Constructs a new minimum degree finder for a given graph
    `@param graph` - the input graph
- Method Detail:
  - `getDegree() : int`
    Computes the minimum degree
    `@return` the degree
    Specified by `getDegree()` in interface `DegreeAlgorithm<V>`

Class MaxDegreeFinder<V>
The maximum degree finder. It determines the highest degree of all nodes within a graph.

- Constructor Detail:
  - `MaxDegreeFinder(Graph<V> graph`
    Constructs a new maximum degree finder for a given graph
    `@param graph` - the input graph
- Method Detail:
  - `getDegree() : int`
    Computes the maximum degree
    `@return` the degree
    Specified by `getDegree()` in interface `DegreeAlgorithm<V>`

Class AverageDegreeFinder<V>
The average degree finder. It computes the average degree of all nodes in a graph.

- Constructor Detail:

- `AverageDegreeFinder(Graph<V> graph)`
  Constructs a new average degree finder for a given graph
  `@param graph` - the input graph

- Method Detail:

  - `getDegree() : int`
    Computes the average degree
    `@return` the degree
    Specified by `getDegree()` in interface `DegreeAlgorithm<V>`

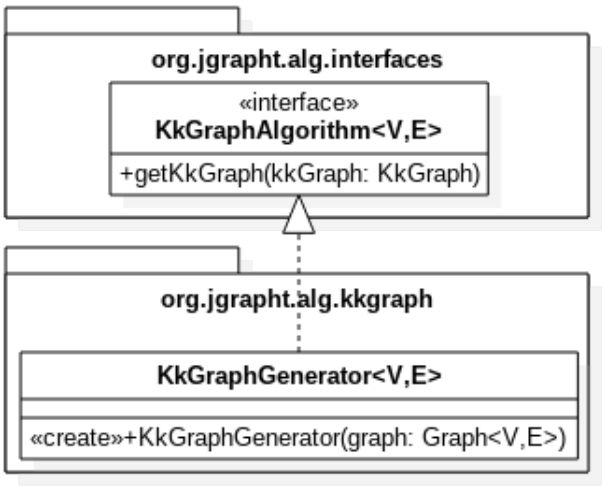
3.8 Paket org.jgrapht.alg.kkgraph

3.8.1 Beschreibung

Das Paket `org.jgrapht.alg.kkgraph` enthält Algorithmen für die Berechnung eines K_k-Graphen im Bezug auf die Hadwigervermutung.

3.8.2 Interner Aufbau

Dieses Paket enthält die Klasse `KkGraphGenerator`, welche `KkGraphAlgorithm<V,E>` aus `org.jgrapht.alg.interfaces` implementiert, und einen K_k-Graphen berechnet.

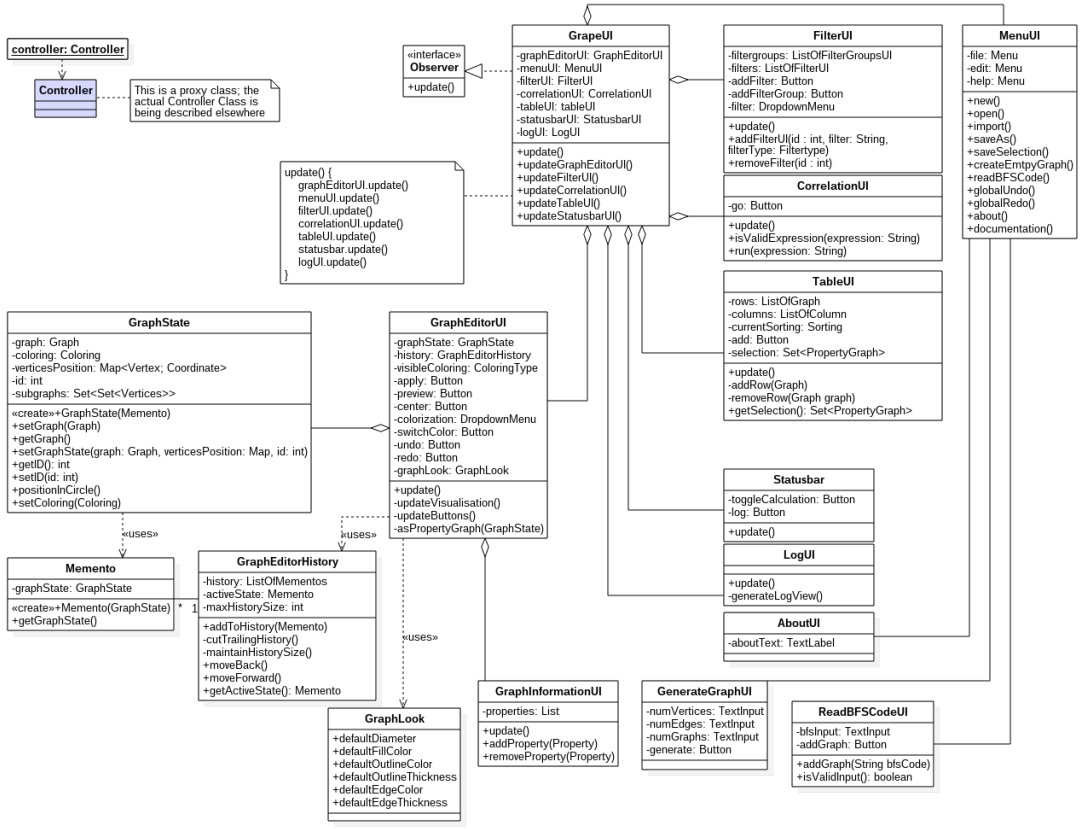


3.8.3 Feinentwurf

Class KkGraphGenerator<V,E>
The K_k-graph generator. It generates the K_k-graph for a given graph.

- Constructor Detail:
  - `KkGraphGenerator(Graph<V> graph`
    Constructs a new K_k-graph generator for a given graph
    `@param graph` - the input graph
- Method Detail:
  - `getKkGraph() : KkGraph<V,E>`
    `@return` the KkGraph
    Computes the kk graph
    Specified by `getKkGraph()` in interface `KkGraphAlgorithm<V,E>`

4 Grafische Benutzeroberfläche



Für die Implementierung der GUI wird die Java Swing Bibliothek verwendet, um Plattformunabhängigkeit zu gewährleisten. Swing verwendet intern `Components` zur Kapselung von einzelnen Bereichen der GUI, und `Panels`, zur Darstellung von Fenstern. In der Modellierung wird nicht beschrieben, welche Klassen der GUI von welcher Java Swing Klasse erben, da dies aus dem Pflichtenheft hervorgeht und nicht zu einer verständlichen Beschreibung beiträgt.
Die einzelnen Schaltflächen des jeweiligen Bereichs der GUI sind als Attribute modelliert. Für die Schaltflächen der GUI werden jeweils `ActionListener` implementiert; diese beschreiben die Aktion einer Schaltfläche bei Betätigung dieser. Schaltflächen sind entweder `Button`s, `Menu`s oder `MenuItem`s. Diese werden im Folgenden gleich behandelt.

Klasse MenuUI

Es werden im Folgenden die `ActionListener` der Menu-Einträge beschrieben.

- *file*-Menu MenuItems action detail:
  - Menuitem *new*
    Opens a file browser where the user specifies the filepath. Then the controller method `newDatabase(path)` is called.
  - Menuitem *open* `open(String path)`
    Opens a file browser where the user specifies the filepath to an existing database. Then the controller method `openDatabase(path)` is called.
  - Menuitem *import* `import(String path)`
    Opens a file browser where the user specifies the filepath. Then the controller method `mergeDatabase(String path)` is called.
  - Menuitem *saveAs* `saveAs(String path)`
    Opens a file browser where the user specifies the filepath. Then the controller method `saveDatabase(String path)` is called.
  - Menuitem *saveSelection* `saveAs(String path, List<Integer> graphIDs)`
    Opens a file browser where the user specifies the filepath. Then the controller method `saveSelection(path, graphIDs)` is called.
- *edit*-Menu MenuItems action detail:
  - Menuitem *generateGraphs*
    Opens a new window which allows to specifiy the parameters that are necessary for the controller method `generateGraphs(minVertices, maxVertices, minEdges, maxEdges)`.
  - Menuitem *createEmptyGraph*
    Calls the controller method `generateGraphs(0, 0, 0, 0, 1)`.
  - Menuitem *readBFSCode*
    Opens a window with a text input field that expects a BFS Code as an input. The controller method `generateBFSGraph(bfscode)` is then being called.
  - Menuitem *undo*
    Calls the controller method `globalUndo()`.

  - Menuitem *redo*
    Calls the controller method `globalRedo()`.

- *help*-Menu Menuitems action description:

  - Menuitem *info*
    Opens a window that shows information about Grape.

  - Menuitem *documentation*
    Opens Grape's documentation.

4.1 Graphen-Editor

Ein primärer Teil der GUI ist der Graphen-Editor. Für diesen ist zentral, wie Graphen aus der Tabelle bzw. aus der Datenbank geladen werden und wie diese nach Bestätigung der Änderung der Datenbank hinzugefügt werden. Außerdem steht eine dedizierte Historie zur Verfügung, welche es erlaubt, Änderungen am Graphen im Graphen-Editor rückgängig zu machen.

4.1.1 Laden, Speichern und Darstellen eines selektierten Graphen

Wie im Abschnitt zur Tabelle beschrieben, liegt der `Table`-Klasse eine Menge von Graphen vor, welche der aktuellen Auswahl entspricht. Ist die Anzahl der Elemente genau eins, so wird der aktuell ausgewählte Graph im Graphen-Editor angezeigt. Im Detail geschieht dies wie folgt:
Die `Table`-Klasse wird mittels ihrer Methode `update()` aktualisiert. Dabei wird auch die `update()` Methode des Graphen-Editors aufgerufen. Dieser überprüft nun, ob sich die Auswahl geändert hat. Dazu genügt ein Vergleich der ID des Graphen. Bei einer Änderung zu einer Auswahl von genau einem Graphen werden die Änderungen am aktuellen Graphen im Editor verworfen und der neue Graph angezeigt. Bei einer Änderung zu einer Auswahl, welche keinen oder mehrere Graphen enthält, werden die Änderungen verworfen und kein Graph angezeigt. In beiden Fällen wird die Historie des Graphen-Editors zurückgesetzt.
Während der Bearbeitung eines Graphen im Graphen-Editor wird intern der aktuelle Bearbeitungsstand des Graphen als `GraphState` instandgehalten. Erst bei Bestätigung der Änderung durch einen Klick auf den Button `apply` wird die Änderung an die Datenbank übergeben. Diese erhält wie oben beschrieben die `ID` eines Graphen und ein `PropertyGraph`.

Es folgt eine Beschreibung der Schaltflächen des Graphen-Editors.

- Button *apply*
  Calls the controller method
  `addEditedGraph(asPropertyGraph(graphState.getGraph), graphState.getID)`.

- Button *preview*
  Calls the controller method
  `getVertexColoring(asPropertyGraph(graphState.getGraph))` or
  `getTotalColoring(asPropertyGraph(graphState.getGraph))`
  depending on the currently shown type of coloring.

- Button *center*
  Rearranges the vertices in a circle.

- MenuItem *vertexColoring*
  Changes the shown coloring to a vertex coloring.

- MenuItem *totalColoring*
  Changes the shown coloring to a total coloring.

- Button *switchColor*
  Requests an alternative coloring of the graph from the controller.

- Button *undo*
  Undos the last action in the Graph-Editor

- Button *redo*
  Redos the last action in the Graph-Editor.

- Button *popout*
  Closes the Graph-Editor in the main-window and opens a new window with the Graph-Editor. This window has a *collapse* Button that reverses the action of the Button *popout*.

4.1.2 Historie des Graphen-Editors

Graphen in der Datenbank enthalten keine Information über die Position der Knoten eines Graphen. Auch weitere Informationen wie zum Beispiel das konkrete Aussehen eines Graphen sind nicht Teil der Datenbank. Diese müssen also unter Anderem durch die GUI gespeichert werden. Es wird deshalb die `GraphState`-Klasse eingeführt, welche diese Informationen bereithält.

Wie der Name `GraphState` bereits impliziert, entspricht eine Instanz dieser Klasse dem temporären Zustand eines Graphen im Graphen-Editor. Bei einer Änderung am Graphen, wie sie zum Beispiel beim Hinzufügen eines Knotens auftritt, wird eine neue Instanz der `GraphState` Klasse erstellt und der Historie hinzugefügt.
Die Historie selbst hält also eine zeitlich geordnete Liste von `GraphState`s instand. Dabei ist zu gewährleisten, dass das Attribut `activeState` dem aktuellen Zustand des Graphen im Graphen-Editor entspricht. Bei einem Aufruf von `update()` setzt die Klasse `GraphEditorUI` ihr Attribut `graphState` auf den aktuellen Zustand der Historie.
Zusätzlich stellt die Klasse `GraphEditorUI` zwei weitere Methoden zum aktualisieren der GUI-Elemente bereit. Die erste Methode `updateVisualisation()` aktualisiert nur die Darstellung des Graphen im Editor. Die zweite Methode `updateButtons()` überprüft mittels eines Aufrufs der `Controller`-Methode `isValidGraph(PropertyGraph)`, ob der aktuelle Graph ein gültiger Graph ist. Falls dies der Fall ist, wird die Schaltfläche `Anwenden` deaktiviert.

Beschreibung der Methoden der GraphEditorHistory.

- Method Detail:
  - `addToHistory(Memento memento)`
    Calls `cutTailingHistory()`, appends *memento* to *history* and sets *activeState* to the appended Memento.
  - `cutTrailingHistory(Memento memento)`
    Removes all history entries after *activeState*.
  - `cutTrailingHistory()`
    Removes all history entries after *activeState*.
  - `maintainHistorySize()`
    Removes oldest history entries util the number of history-entries matches *maxHistorySize*.
  - `moveBack()`
    If possible set *activeState* to the previous entry in *history*. Don't change *activeState* otherwise.
  - `moveForward()`
    If possible set *activeState* to the next entry in *history*. Don't change *activeState* otherwise.

4.2 Tabelle

Die wichtigste Methode der Klasse `Table` ist die `update()` Methode. Bei der Ausführung dieser wird die `Controller`-Methode `getFilteredAndSortedGraphs()` aufgerufen. Diese gibt unter Berücksichtigung der Filter und der aktuellen Sortierung Graphen zurück, die dann in der Tabelle angezeigt werden.
Bei einer Änderung der Sortierreihenfolge wird die `Controller`-Methode `getFilteredAndAscendingSortedGraphs(String attribute)` für aufsteigende bzw. `getFilteredAndDescendingSortedGraphs(String attribute)` für absteigende Sortierung, gefolgt von `update()` der Klasse `TableUI` aufgerufen. Bei einem Rechtsklick auf einen Graphen in der Tabelle wird ein Menü geöffnet, welches es erlaubt den nächstdichteren Graphen zu generieren. Es wird die Methode `getDenserGraph(PropertyGraph)` der Klasse `Controller` aufgerufen.
Außerdem kann in diesem Menü ausgewählt werden, ob alle Färbungen berechnet werden sollen.

4.3 Filter

Die Klasse `FilterUI` erlaubt das Generieren von GUI-Elementen zur Eingabe eines Filterausdrucks als Text. Bei der Generierung eines Filters wird dabei eine ID erzeugt, mittels welcher das `Filterpaket` die Filter identifiziert.
Nach jeder Eingabe wird für den aktuellen Ausdruck überprüft, ob ein gültiger Filter vorliegt. Es wird daher nach jedem eingegebenen Zeichen die Methode `checkFilterInput()` aufgerufen und das Eingabefeld entsprechend farbig hinterlegt. Nach der Bestätigung der Eingabe mittels *Enter-Taste* wird der Filter, sofern dieser gültig ist, dem Filterpaket als *String* übergeben.
Zudem stellt die `FilterUI` ein Dropdown Menü bereit mit den Optionen (i) *Filter Speichern* und (ii) *Filter Laden*:

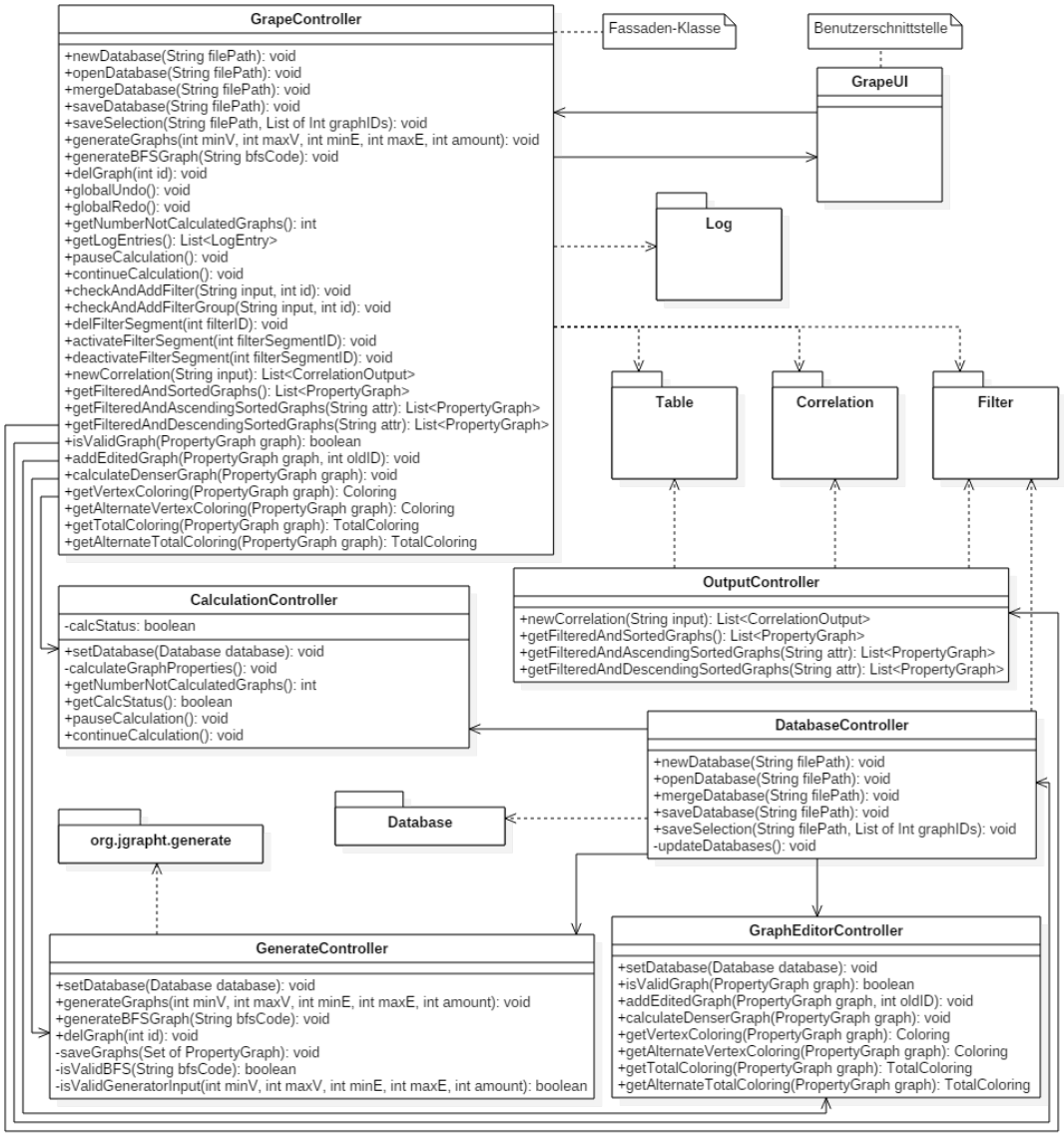
(i) Es öffnet sich ein Fenster, welches den Benutzer auffordert, einen Dateipfad festzulegen. Alle sichtbaren Filter werden in einer Textdatei gespeichert. Dabei wird in eckigen Klammern ([]) der Name der Filtergruppe dann der Filterausdruck gefolgt von einem Semikolon in die Datei geschrieben. Dies wird für jeden Filter wiederholt.

(ii) Wieder öffnet sich ein Fenster, in dem der Benutzer eine Datei mit Filterausdrücken auswählt. Für jeden Eintrag wird nun die Filtergruppe ermittelt und der Filter mit dem ermittelten Filterausdruck der GUI hinzugefügt. Das Laden von Filtern kann also wie das Hinzufügen von Filtern über den entsprechenden Button behandelt werden. Es werden demnach auch importierten Filtern IDs zugewiesen.

5 Controller

5.1 Beschreibung

Das Paket ist für die Steuerung von *Grape* verantwortlich. Die Steuerung verwaltet die Benutzerinteraktionen und aktualisiert das Modell. Sie gibt außerdem Änderungen der Modelldaten an die Präsentation weiter. Benutzerinteraktionen werden in *Grape* vor allem durch Mausklicks auf die Schaltflächen der Benutzeroberfläche ausgelöst.



5.2 GrapeController

5.2.1 Situierung

Der `GrapeController` ist die Fassade für das Controller-Paket. Er sorgt dafür, dass die GUI nur die Funktionen zur Verfügung gestellt bekommt, die sie benötigt. Dabei bleibt der interne Aufbau der Steuerung verborgen.

5.2.2 Blackbox-Beschreibung

Die Methoden von `GrapeController` delegieren die Funktionalität lediglich an die zuständigen Klassen und Pakete vom Controller.

5.2.3 Benötigte Schnittstellen

Der `GrapeController` benötigt den Zugriff auf alle vom Controller bereitgestellten Klassen und Pakete. Eine Datenbankanbindung ist nicht notwendig. Zur Kommunikation mit der GUI benötigt der `GrapeController` Zugriff auf die GrapeUI.

5.3 DatabaseController

5.3.1 Situierung

Der `DatabaseController` sorgt für das Öffnen, Schließen, Speichern und Importieren von Datenbanken und gibt die Informationen an die Controllerklassen und /-pakete weiter, die eine Datenbankverbindung benötigen.

5.3.2 Blackbox-Beschreibung

Bei der Auswahl einer neuen bzw. anderen Datenbank durch die GUI müssen die betroffenen Klassen ihre Datenbank aktualisieren. Der `DatabaseController` stößt diese Aktualisierungen mit der Methode `updateDatabases()` an. Hierbei wird bei allen betroffenen Klassen die Methode `setDatabase(database: Database)` ausgeführt. Es darf immer nur genau eine Instanz einer Datenbank existieren.

5.3.3 Feinentwurf

- `+ newDatabase(): void`
  triggers the database to open a new database table.

- `+ newDatabase(filePath: String): void`
  triggers the database to open the database table at the given file path.
  `@param filePath` the file path of the database.

- `+ mergeDatabase(filePath: String): void`
  triggers the database to merge the database table at the given file path with the current database table.
  `@param filePath` the file path of the Database.

- `+ saveDatabase(filePath: String): void`
  triggers the database to save the current database table at the given file path.
  `@param filePath` the file path of the Database.

- `+ saveSelection(filePath: String, graphIDs: List<Int>): void`
  triggers the database to save the current selected graphs in the table at the given file path.
  `@param filePath` the file path of the Database.
  `@param graphIDs` the GraphIDs to save.

- `- updateDatabases(): void`
  sets the database for all Controller classes that have a database connection.

5.3.4 Benötigte Schnittstellen

Der `DatabaseController` benötigt Zugriff auf die Interface `Connection` vom Paket Model. Dies ermöglicht Laden/Erzeugen von Datenbanken. Außerdem benötigt er Zugriff auf das Filterpaket, den `CalculationController`, den `GrapheditorController`, den `GenerateController` und das `Filterpaket`, um die Datenbankaktualisierung zu kommunizieren.

5.4 GenerateController

5.4.1 Situierung

Der `GenerateController` sorgt für das Anstoßen der Generierung von Graphen und Übersetzen des BFS Codes zu einer Graphinstanz. Die erzeugten und noch unberechneten Graphen werden zur Speicherung an die `Datenbank` geschickt und in der Menge der noch nicht berechneten Graphen gespeichert. Er sorgt außerdem für das Löschen eines Graphen aus der Tabelle.

5.4.2 Blackbox-Beschreibung

Die Generierung der Graphen geschieht über den `BulkRandomConnectedGraphGenerator`. Die generierten Graphen werden stets in der `Datenbank` gespeichert.

5.4.3 Feinentwurf

- `+ setDatabase(database: Database): void`
  replaces the old database with the given database.
  `@param database` the current database.

- `+ generateGraphs(minVertices: int, maxVertices: int, minEdges: int, maxEdges: int, amount: int): void`
  gives the graph generator the command to generate the graphs and saves them in the `Database`.
  `@param minVertices` lower bound of vertices.
  `@param maxVertices` upper bound of vertices.
  `@param minEdges` lower bound of edges.
  `@param maxEdges` upper bound of edges.
  `@param amount` the number of graphs

- `+ generateBFSGraph(bfsCode: String): void`
  calculates a graph with the BFS Code and saves it to the `Database`.
  `@param bfsCode` the BFS Code of the graph to save.

- `+ delGraph(id: int): void`
  Deletes the given graph from the GUI table.
  `@param id` the ID of the PropertyGraph<V,E>.

- `- saveGraphs(graphs: Set<PropertyGraph>): void`
  Saves the graphs in the Database in the list of not yet calculated graphs.
  `@param graphs` the set of PropertyGraph<V,E>.

- `- isValidBFS(bfsCode: String): Boolean`
  Checks if the given String is a correct BFS Code.
  `@param bfsCode` the BFS Code of a graph.
  `@return` True if the BFS Code is valid.

- `- isValidGeneratorInput(minVertices: int, maxVertices: int, minEdges: int, maxEdges: int, amount: int): Boolean`
  Checks if the input is valid for the graph generator.
  `@param minVertices` lower bound of vertices.
  `@param maxVertices` upper bound of vertices.
  `@param minEdges` lower bound of edges.
  `@param maxEdges` upper bound of edges.
  `@param amount` the number of graphs.
  `@return` true if the input is valid.

5.4.4 Benötigte Schnittstellen

Der `GenerateController` benötigt Zugriff auf das Unterpaket `org.jgrapht.generate` des `JGraphT` Pakets. Zur Speicherung der Graphen benötigt er noch Zugriff auf die Datenbank.

5.5 CalculationController

5.5.1 Situierung

Der `CalculationController` sorgt für die Berechnung der Graphenmerkmale. Es wird erst nach der vollständigen Berechnung, d.h. nachdem alle Merkmale eines Graphen berechnet wurden, an die Datenbank zur Speicherung zurückgegeben.

5.5.2 Blackbox-Beschreibung

Der `CalculationController` holt sich aus der Datenbank unberechnete Graphen, die sich in der Liste zur Berechnung befinden. Solange sich in dieser Liste Graphen befinden, wird die Methode `calculateGraphProperties()` ausgeführt. Sobald alle Merk-

male eines Graphen berechnet wurden, werden sie in der Datenbank gespeichert und der Graph aus der Menge der unberechneten Graphen entfernt. `calculateGraphProperties()` kann pausiert und wieder fortgesetzt werden.

5.5.3 Feinentwurf

- `+ setDatabase(database: Database): void`
  replaces the old database with the given database.
  `@param database` the current database.

- `- calculateGraphProperties(): void`
  induces the calculation of all properties of PropertyGraph<V,E> in the graphlist of the `database` and induces their saving in the `database`.

- `+ getNumberNotCalculatedGraphs(): int`
  `@return` the length of the graphlist of `CalculationController`.

- `+ getCalcStatus(): Boolean`
  checks if the current calculation is running.
  `@return` true if the calculation is running.

- `+ pauseCalculation(): void`
  pauses the method `calculateGraphProperties()`.

- `+ continueCalculation(): void`
  continues the method `calculateGraphProperties()`.

5.5.4 Benötigte Schnittstellen

Es wird lediglich die Schnittstelle zur Anbindung an die Datenbank benötigt.

5.6 GraphEditorController

5.6.1 Situierung

Der `GraphEditorController` stellt die Schnittstellen für den Grapheneditor in der grafischen Benutzeroberfläche bereit. Dies betrifft das Modifizieren von Graphen, das

Ermitteln von Total- beziehungsweise Knotenfärbungen, die Ermittlung des nächstdichteren Graphen und das Überprüfen von modifizierten Graphen auf Gültigkeit.

5.6.2 Blackbox-Beschreibung

Bereitgestellt wird einerseits die Methode `calculateDenserGraph(graph: PropertyGraph)`, mit welcher man den nächstdichteren Graphen ermitteln kann. Des Weiteren existieren die Methoden `getVertexColoring(graph: PropertyGraph)` und `getTotalColoring(graph: PropertyGraph)`, welche eine Färbung eines Graphen ermitteln und zurückgeben. Mit Hilfe der Methoden `getAlternateTotalColoring(graph: PropertyGraph)` bzw. `getAlternateVertexColoring(graph: PropertyGraph)`, kann man für einen bestimmten Graphen überprüfen, ob eine weitere, nicht äquivalente Färbung existiert. Falls dies der Fall ist, so wird eine andere Färbung zurückgegeben. Mit der Methode `isValidGraph(graph: PropertyGraph): boolean` kann überprüft werden, ob die Modifikationen des Benutzers dazu geführt haben, dass der Graph nicht mehr gültig ist. Ist der Graph auch nach den Modifikationen des Benutzers gültig, so kann er durch die Methode `addEditedGraph(graph: PropertyGraph, oldID: int)` den Graph zur unberechneten Graphenliste der Datenbank hinzugefügt werden.

5.6.3 Interner Aufbau

Da das Paket im Wesentlichen Methoden aus Unterpaketen aufruft, ist der interne Aufbau sehr einfach gehalten: Es existiert lediglich eine Klasse, welche die in der Blackbox-Beschreibung beschriebenen Methoden enthält.

5.6.4 Feinentwurf

- `+ setDatabase(database: Database): void`
  replaces the old database with the given database.
  `@param database` the current database.

- `+ addEditedGraph(graph: PropertyGraph, oldID: int): void`
  checks the given graph for duplicates then adds the graph to the not yet calculated graphlist of `CalculationController` and deletes the old graph from the database.
  `@param graph` the PropertyGraph<V,E> to add.
  `@param oldID` the id of the modified graph from the Grapheditor.

- `+ isValidGraph(graph: PropertyGraph): boolean`
  checks if the graph is valid
  `@param graph` the PropertyGraph<V,E> to check.
  `@return` true if the given graph is valid.

- `+ calculateDenserGraph(graph: PropertyGraph): void`
  triggers the calculation of the next denser graph for a specific graph
  `@param graph` the `PropertyGraph<V,E>` to calculate.

- `+ getVertexColoring(graph: PropertyGraph): Coloring`
  calculates a valid colorization for a specific graph.
  `@param graph` the `PropertyGraph<V,E>` to calculate.
  `@return` the graphcolorization.

- `+ getTotalColoring(graph: PropertyGraph): TotalColoring`
  calculates a valid colorization for a specific graph
  `@param graph` the `PropertyGraph<V,E>` to calculate.
  `@return` the graphcolorization.

- `+ getAlternateVertexColoring(graph: PropertyGraph): Coloring`
  calculates a coloring which is not equivalent to current coloring
  `@param graph` the `PropertyGraph<V,E>` to calculate.
  `@return` the next valid alternative Coloring.
  `@throws NoEquivalentColoringException` thrown if there is no equivalent colorization for a specific graph

- `+ getAlternateTotalColoring(graph: PropertyGraph): TotalColoring`
  calculates a coloring which is not equivalent to current coloring
  `@param graph` the `PropertyGraph<V,E>` to calculate.
  `@return` the next valid alternative Coloring.
  `@throws NoEquivalentColoringException` thrown if there is no equivalent colorization for a specific graph

5.6.5 Benötigte Schnittstellen

Es werden vor allem Schnittstellen zum Algorithmenpaket benötigt. Dort müssen Algorithmen zur Ermittlung des nächstdichteren Graphen und einer Färbung bereitgestellt werden. Auch von der Datenbank muss eine Schnittstelle bereitgestellt werden, über welche man einen Graphen hinzufügen kann.

5.7 OutputController

5.7.1 Situierung

Der `OutputController` ist dafür zuständig, Korrelations-, Filter- und Sortieranfragen entgegenzunehmen. Die entsprechenden Anfragen werden an die Unterpakete Filter, Tabelle und Korrelation weitergeleitet. Diese Unterpakete führen die eigentliche Berechnung durch. Die sortierten und gefilterten Graphen werden dann an die graphische Benutzerschnittstelle weitergeleitet.

5.7.2 Blackbox-Beschreibung

Für die grafische Benutzeroberfläche wird einerseits über die Fassade die Methode `newCorrelation(input: String): List<CorrelationOutput>` angeboten. Mittels dieser Methode kann die grafische Benutzeroberfläche Korrelationsanfragen stellen. Dazu leitet sie die Benutzereingabe über die Variable `input` weiter und erhält im Erfolgsfall eine Liste von `CorrelationOutput`-Objekten, welche alle notwendigen Informationen zu einer bestimmten Korrelation enthält.
Des Weiteren wird die Methode `+ getFilteredAndSortedGraphs(): List<PropertyGraph>` angeboten. Mit dieser werden Filter- und Sortieranfragen bereitgestellt. Die Methode filtert immer nach der eindeutigen Identifikationsnummer der Graphen. Möchte man zusätzlich noch angeben, nach welchem Attribut sortiert werden soll, so kann man das Attribut durch die Methoden `getFilteredAndAscendingSortedGraphs(attribute: String): List<PropertyGraph>` und `getFilteredAndDescendingSortedGraphs(attribute: String): List<PropertyGraph>` übergeben werden. Der Unterschied zwischen den beiden Methoden liegt lediglich darin, dass beim Aufruf der Methode `getFilteredAndAscendingSortedGraphs(attribute: String): List<PropertyGraph>` aufsteigend sortiert und beim Aufruf der Methode `getFilteredAndDescendingSortedGraphs(attribute: String): List<PropertyGraph>` absteigend sortiert wird.

5.7.3 Interner Aufbau

Der `OutputController` besteht aus den Methoden `newCorrelation(String input): List<CorrelationOutput>`, `getFilteredAndSortedGraphs(): List<PropertyGraph>`, `getFilteredAndAscendingSortedGraphs(attribute: String): List<PropertyGraph>` und `getFilteredAndDescendingSortedGraphs(attribute: String): List<PropertyGraph>`, welche die in der Blackbox-Beschreibung beschriebene Funktionalität bereitstellen. Erwähnt sei hier noch der Aufbau der Methode `getFilteredAndSortedGraphs(): List<PropertyGraph>`: Diese Methode filtert immer erst alle Graphen der aktuellen

Datenbank nach den aktuellen Filterkriterien und führt dann stets eine Sortierung nach der eindeutigen Identifikationsnummer der Graphen durch. Möchte man nach einem anderen Attribut sortieren, so verwendet man die Methoden `getFilteredAndAscendingSortedGraphs(attribute: String): List<PropertyGraph>` und `getFilteredAndDescendingSortedGraphs(attribute: String): List<PropertyGraph>`. Diese filtern ebenfalls zunächst alle Graphen in der aktuellen Datenbank nach den aktuellen Filterkriterien. Im Anschluss wird die so entstandene Graphenmenge nach dem übergebenen Attribut auf- beziehungsweise absteigend sortiert.

5.7.4 Feinentwurf

- `+ newCorrelation(input: String): List<CorrelationOutput>`
  gets all filtered and sorted graphs, checks the input for the correlation, creates a new instance of `CorrelationRequest` and finally executes `use(in graphs:List<PropertyGraph<V,E»): List<CorrelationOutput>` on the `CorrelationRequest` instance with the graph list.
  `@param input` the input for the correlation.
  `@return` a list of `CorrelationOutput`.

- `+ getFilteredAndSortedGraphs(): List<PropertyGraph>`
  gets all graphs that fulfill the filter requirements and sorts these graphs by graphID.
  `@return` a list of `PropertyGraph<V,E>`.

- `+ getFilteredAndAscendingSortedGraphs(attribute: String): List<PropertyGraph>`
  gets all graphs that fulfill the filter requirements and sorts these graphs ascending by a specific attribute
  `@param attribute` the attribute to sort after.
  `@return` a list of `PropertyGraph<V,E>`.

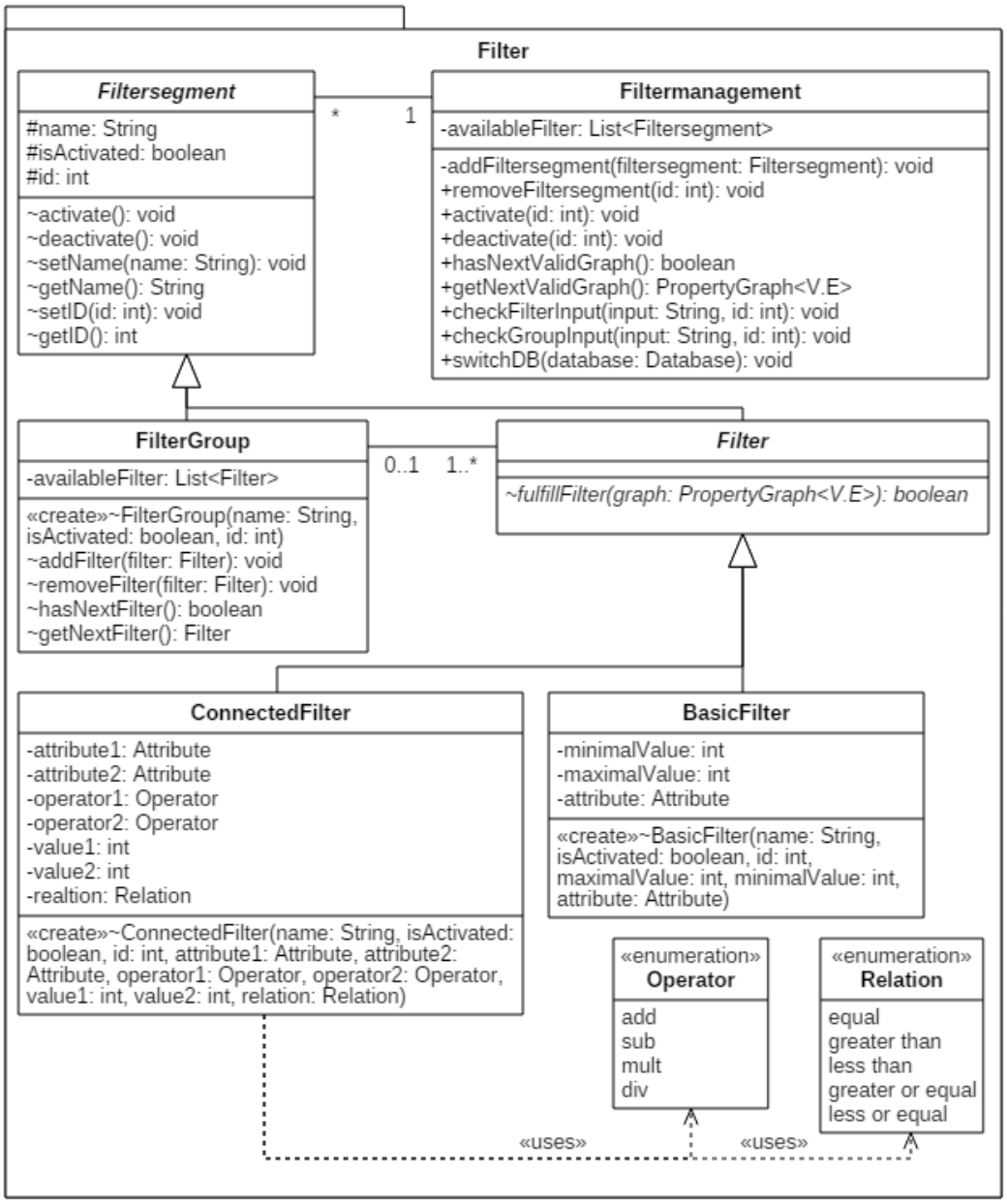
- `+ getFilteredAndDescendingSortedGraphs(attribute: String): List<PropertyGraph>`
  gets all graphs that fulfill the filter requirements and sorts these graphs descending by a specific attribute
  `@param attribute` the attribute to sort after.
  `@return` a list of `PropertyGraph<V,E>`.

5.7.5 Benötigte Schnittstellen

Schnittstellen werden zu den Paketen Filter, Korrelation und Tabelle benötigt. Zum Paket Filter wird eine Schnittstelle benötigt, über welche man alle Filter der Daten-

bank, die die aktuellen Filterkriterien erfüllen, abrufen kann.

5.8 Filter



5.8.1 Situierung

Das Paket Filter ist dafür zuständig, die Filteroptionen des Benutzers entgegenzunehmen. Darüber hinaus verwaltet es elementare Filter sowie Filtergruppen. Die vom Benutzer gewünschten Filteroptionen werden bei jeder Änderung mit den gespeicherten Filteroptionen in der Datenbank synchronisiert. Dazu leitet das Paket Filter alle Änderungen des Benutzers sofort an die Datenbank weiter. Zudem wird eine Schnittstelle bereitgestellt, über welche man alle Graphen der Datenbank, welche die gerade eingestellten Filterkriterien erfüllen, abrufen kann.

5.8.2 Blackbox-Beschreibung

Die wichtigste bereitgestellte Methode ist die Methode `getNextValidGraph(): PropertyGraph<V,E>`, welche den nächsten Graphen in der Datenbank ausgibt, der die eingestellten Filterkriterien erfüllt. Des Weiteren lassen sich durch Aufruf der Methode `checkFilterInput(input: String, id: int): void` neue Filter zur Filtermenge hinzufügen. Neue Filtergruppen können durch die Methode `checkFilterInput(input: String, id: int): void` der Filtermenge hinzugefügt werden. Sowohl die Möglichkeit der Filteraktivierung als auch die Möglichkeit der Filterdeaktivierung wird ebenfalls durch das Paket unterstützt: Zum Aktivieren eines bereits vorhandenen Filters verwendet man die Methode `activate(id: int): void`; zum Deaktivieren die Methode `deactivate(id: int): void`. Filtersegmente werden von außen immer über eine eindeutige Identifikationsnummer angesprochen.

5.8.3 Interner Aufbau

Das Paket besteht aus einer zentralen Klasse Filterverwaltung, welche koordiniert, welche Filter gerade vorhanden beziehungsweise ausgewählt sind. Die Kommunikation nach außen findet ausschließlich über diese Klasse statt. Zu dieser zentralen Klasse können nur Objekte der abstrakten Klasse Filtersegment hinzugefügt werden. Ein Filtersegment besteht entweder aus einer Filtergruppe (diese enthält mehrere Objekte der abstrakten Klasse Filter) oder aus einem einzelnen Objekt der Klasse Filter. Filter können entweder elementar (damit kann beispielsweise der Ausdruck *Kanten = 10* realisiert werden) oder eine Verknüpfung zweier Filterobjekte sein (womit sich zum Beispiel der Ausdruck *Farben <= maxGrad + 2* realisieren lässt).

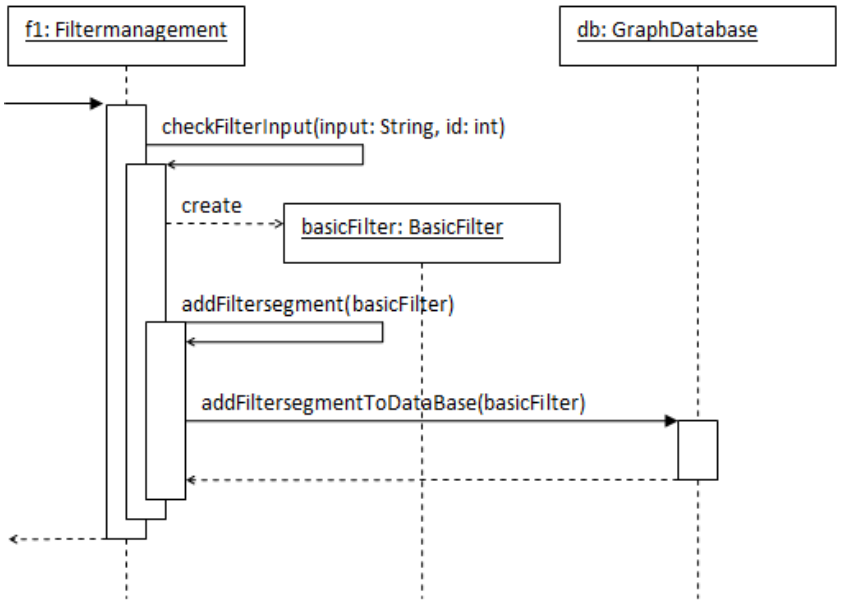


Abbildung 1: Sequenzdiagramm zu einer Anfrage des Benutzers, einen BasicFilter hinzuzufügen

5.8.4 Feinentwurf

Filtermanagement
Die Klasse Filtermanagement dient der Verwaltung aller vorhandenen Filter beziehungsweise Filtergruppen. Sie enthält eine Liste, welche alle derzeit verfügbaren Filter enthält. Es existiert immer nur eine Instanz dieser Klasse. Die Klasse stellt folgende Methoden bereit:

- `- addFiltersegment(filtersegment: Filtersegment): void`
  removes a filtersegment out of the list of class `Filtermanagement`
  `@param filtersegment` filtersegment which should be added
  adds a filtersegment to the list of class `Filtermanagement`

- `+ removeFiltersegment(id: int): void`
  `@param id` unique identifier of the filtersegment which should be removed

- `+ activate(id: int): void`
  enables a filtersegment which means that the criteria of the fitersegment are now used to filter graphs
  `@param id` unique identifier of the filtersegment which should be enabled

- `+ deactivate(id: int): void`

disables a filtersegment which means that the criteria of the filtersegment are ignored while filtering graphs
`@param id` unique identifier of the filtersegment which should be disabled

- `+ hasNextValidGraph(): boolean`
  allows to prevent a `NullPointerException` by asking if there is a graph to return
  `@return` returns true if there is a graph which meets the current filter criteria but was not returned yet

- `+ getNextValidGraph(): PropertyGraph<V,E>`
  allows to get only graphs from database which meet the current filter criteria
  `@return` graph which meets all current filter criteria
  `@throws NullPointerException` this exception is thrown if all graphs of database which meet the current criteria were already returned

- `+ checkFilterInput(input: String, id: int): void`
  checks whether the input string codes a valid filter. In case of success the method `addFiltersegment(filtersegment: Filtersegment): void` is called and a new filter is added to the list of class `Filtermanagement`
  `@param input` string which might code a filter
  `@param id` unique identifier of the new filterobject
  `@throws InvalidInputException` this exception is thrown if the input string does not code a valid filter
  `@throws DoubledIdentifierException` this exception is thrown if there is already a filter with same identifier

- `switchDB(database: Database): void`
  used when initializing *Grape* or switching a database. The methode clears the current list of filtersegments and calls the methode `addFiltersegment(filtersegment: Filtersegment): void` for every filter element of the new database
  `@param database` new database which should be used in future

Filtersegment
Die Klasse `Filtersegment` ist abstrakt. Sie enthält die Unterklassen `Filtergroup` und die abstrakte Klasse `Filter`. Durch die Klasse `Filtersegment` wird dafür gesorgt, dass Filtergruppen und Filter von der Klasse `Filtermanagement` einheitlich behandelt werden können. Folgende Methoden sind enthalten:

- `activate(): void`
  allows to enable a filtergroup or a filter. If a filter gets enabled, the criteria of the filter are now used to filter graphs. If a filtergroup gets enabled every enabled filter in this group is now used to filter graphs. Every filter of a filtergroup which

was disabled stays disabled even if the filtergroup gets enabled.

- `deactivate(): void`
  allows to disable a filtergroup or a filter. If a filter gets disabled, the criteria of the filter are now ignored when filtering graphs. If a filtergroup gets disabled, every filter of the group gets disabled even if the filter of the group are enabled.

- `setName(name: String): void`
  sets a name for a specific filter. The name should be identical to the user input. If the user wrote *Kanten = 10*, the name of the filtersegment should be *Kanten = 10*
  `@param name` name of the filtersegment

- `getName(): String`
  allows to get the name of a specific filtersegment
  `@return` name of filtersegment

- `setID(id: int): void`
  sets an unique identifier to a specific filtersegment
  `@param id` identifier of the filtesegment
  `@throws DoubledIdentifierException` this exception is thrown if there is already a filter with same identifier

- `getID(): int`
  allows to get the identifier of a specific filtersegment
  `@return` identifier of a specific filtersegment

Filtergroup
Mit der Klasse `Filtergroup` wird es möglich, mehrere logisch zusammenhängende Objekte der Klasse `Filter` zusammenzufassen. Dadurch ist es insbesondere möglich, alle Filterobjekte einer Filtergruppe durch einen Funktionsaufruf aktiv oder inaktiv zu setzen. In dieser Klasse lassen sich folgende Methoden finden:

- `Filtergroup(name: String, isActivated: boolean, id: int)`
  Constructor of class Filtergroup
  `@param name` name of the filtergroup (name of filtergroup should be equal to user input)
  `@param isActivated` true if the filter of this group are currently used to filter graphs
  `@param id` unique identifier of the filter group

- `addFilter(filter: Filter): void`
  removes a filter from a specific filtergroup
  `@param filter` filter which should be added to a filtergroup

adds a filter to a specific filtergroup

- `removeFilter(filter: Filter): void`
  `@param filter` filter which should be removed from a specific filtergroup

- `hasNextFilter(): boolean`
  helps to prevent a `NullPointerException` by checking if there is any filter left
  `@return` returns true if there is a filter which was not returned yet

- `getNextFilter(): Filter`
  used to get all filter of a specific filtergroup
  `@return` returns a filter which was not returned yet
  `@throws NullPointerException` this exception is thrown if all filter of the specific group were already returned

Filter
Die Klasse Filter ist abstrakt. Sie enthält zwei Unterklassen: Die Klasse BasicFilter sowie die Klasse ConnectedFilter. Von den Unterklassen wird gefordert, dass sie folgende Methode implementieren:

- `fulfillFilter(graph: PropertyGraph<V,E>): boolean`
  checks if a specific graph meets every criteria of the current filter
  `@param graph` graph which should be checked for the criteria of the filter
  `@return` returns true if the graph meets every criteria of the filter

BasicFilter
Die Klasse `BasicFilter` ist eine Unterklasse der Klasse `Filter`. Objekte dieser Klasse können einen Graphen auf genau eine Bedingung hin überprüfen. Sie realisieren also Filteranfragen wie zum Beispiel folgende Bedingung: Knoten = 10. Es kann hierbei stets nur ein Attribut betrachtet werden. Folgender Konstruktor wird bereitgestellt:

- `BasicFilter(name: String, isActivated: boolean, id: int, maximalValue: int, minimalValue: int, attribute: Attribute)`
  Constructor of class BasicFilter
  `@param name` name of the BasicFilter (the name should be identical to the user input)
  `@param isActivated` boolean which shows if the specific filter is activated
  `@param id` unique identifier for this specific connected filter
  `@param maximalValue` largest value which an attribute of a graph can have to meet the criteria of the filter
  `@param minimalValue` smallest value which an attribute of a graph can have to meet the criteria of the filter
  `@param attribute` attribute which should be checked by the filter

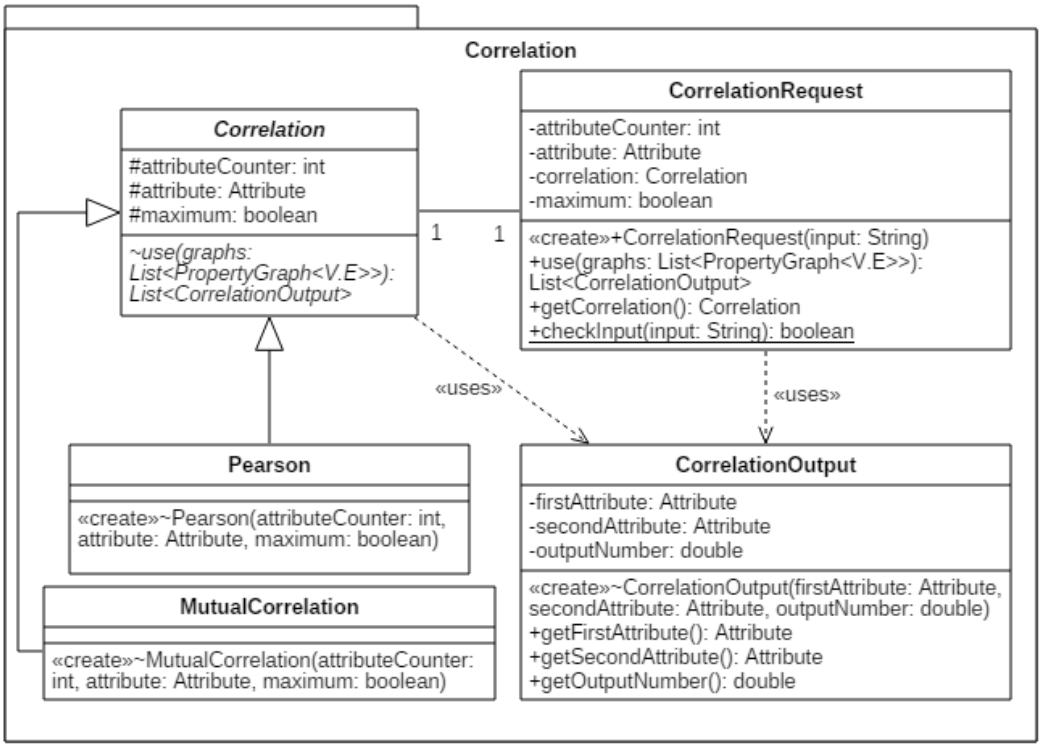
ConnectedFilter
Eine Instanz der Klasse `ConnectedFilter` besteht aus zwei Objekten der Klasse `BasicFilter`. Diese beiden Objekte werden durch die Klasse `ConnectedFilter` mittels einer Relation miteinander verknüpft. Die Relationen werden in einem Enum (genannt Relation) bereitgestellt. Das Enum Relation bietet genau die zuvor beschriebenen Relationen an. Zudem können Integerwerte die Relation verändern. Man kann Integerwerte aufaddieren, subtrahieren, multiplizieren oder auch dividieren. Diese Operatoren sind in einem weiteren Enum (genannt Operator) bereitgestellt. Durch die Klasse wird der folgende Konstruktor bereitgestellt:

- `ConnectedFilter(name: String, isActivated: boolean, id: int, attribute1: Attribute, attribute2: Attribute, operator1: Operator, operator2: Operator, value1: int, value2: int, relation : Relation)`
  Constructor of class ConnectedFilter
  `@param name` name of the connected filter
  `@param isActivated` boolean which shows if the specific filter is activated
  `@param id` unique identifier for this specific connected filter
  `@param attribute1` first attribute who is part of the relation
  `@param attribute2` second attribute who is part of the relation
  `@param operator1` first operator which modifies an attribute value of f1
  `@param operator2` second operator which modifies an attribute value of f2
  `@param value1` integer value which modifies an attribute value of f1
  `@param value2` integer value which modifies an attribute value of f2
  `@param relation` relation which connects the two filter

5.8.5 Benötigte Schnittstellen

Das Paket benötigt lediglich einen Zugriff zur Datenbank, um auf bereits in der Datenbank vorhandene Filter und Filtergruppen zugreifen zu können. Des Weiteren ist ein Zugriff auf die in der Datenbank enthaltenen Graphen notwendig.

5.9 Korrelation



5.9.1 Situierung

Das Korrelationspaket ist dafür zuständig, Korrelationsanfragen entgegenzunehmen und auszuwerten. In der nicht erweiterten Ausgabe des Programmes werden lediglich die Pearson-Korrelation sowie die MutualCorrelation unterstützt. Das Paket ist aber so aufgebaut, dass weitere Korrelationen einfach hinzugefügt werden (diese fügt man einfach als Unterklassen der Klasse Correlation hinzu).

5.9.2 Blackbox-Beschreibung

Das Paket stellt den Service bereit, Korrelationsanfragen des Benutzers entgegenzunehmen und Ergebnisse der vom Benutzer ausgewählten Korrelation zurückzugeben. Für einen neue Anfrage muss eine Instanz der Klasse `CorrelationRequest` erstellt werden. Die Ergebnisse der Berechnung erhält man dann durch Aufruf der Methode `use()`:

`CorrelationOutput`. Die Ergebnisse werden als eine Liste von Tripeln ausgegeben: In einem Tripel befinden sich stets zwei Attribute (die miteinander in Korrelation stehen) sowie ein Fließkommawert, der angibt, wie stark die beiden Attribute bezüglich dieser Korrelationen in Verbindung stehen. Bevor eine neue Instanz der Klasse `CorrelationRequest` erstellt wird, sollte man stets mit Hilfe der Methode `checkInput(input: String): boolean` überprüfen, ob die eingegebene Zeichenkette des Benutzers eine gültige Korrelation codiert.

5.9.3 Interner Aufbau

Das Paket besteht einerseits aus der Klasse `CorrelationRequest`, welche für die Kommunikation nach außen zuständig ist. Jedes Mal, wenn das Paket durch ein anderes Paket benutzt wird, wird eine neue Instanz der Klasse `CorrelationRequest` angelegt. Jede Instanz der Klasse `CorrelationRequest` steht in Verbindung mit genau einer Instanz der Klasse `Correlation`. Die Klasse `Correlation` ist für die eigentliche Berechnung zuständig. Sie besteht aus mehreren Unterklassen (in der nicht erweiterten Version sind dies die Klassen `Pearson` und `MutualCorrelation`), welche ihre jeweilige Korrelation realisieren. Die Ausgabe mit Hilfe einer Liste von Tripeln wird durch die Klasse textttCorrelationOutput realisiert: Die Ausgabeliste enthält lediglich Objekte dieser Klasse. Daher enthält die Klasse textttCorrelationOutput die beiden in Korrelation stehenden Attribute sowie einen Fließkommawert, der aussagt, wie stark die Korrelation ist.

5.9.4 Feinentwurf

CorrelationRequest
Durch diese Klasse wird die Schnittstelle zu anderen Paketen realisiert. Folgende Methoden werden durch diese Klasse bereitgestellt:

- `+ CorrelationRequest(input: String)`
  Constructor of class CorrelationRequest
  `@param input` string which represents a specific correlation
  `@throws IllegalInputException` if the input string does not code a specific correlation

- `+ use(graphs: List<PropertyGraph<V,E»): List<CorrelationOutput>`
  used to delegate the calculation of the correlation to a subclass of class `Correlation`
  `@param graphs` list of elements of class PropertyGraph which should be obscured regarding a specific correlation

`@return` list which inherits all necessary information about the result of the correlation calculation

- `+ getCorrelation(): Correlation`
  used to check if the user input codes a valid correlation. This method should be used before using the constructor of class CorrelationRequest to prevent an `IllegalInputException`.
  `@return` returns the correlation which belongs to the specific correlation request
  used to get the belonging correlation of a specific correlation request

- `+ checkInput(input: String): boolean`
  `@param input` string (usually from user) which might code a specific correlation
  `@return` returns true if the input string codes a valid correlation

Correlation
Abstrakte Klasse, welche alle denkbaren Korrelationen als Unterklasse enthält. In der nicht erweiterten Version existieren nur die Unterklassen `Pearson` und `MutualCorrelation`. Jede Unterklasse muss folgende Methode implementieren:

- `use(graphs: List<PropertyGraph<V,E>>): List<CorrelationOutput>`
  used for correlation calculation
  `@param graphs` list of graphs who should be checked for a specific correlation
  `@return` list which inherits all necessary information about the result of the correlation calculation

Pearson
Klasse, welche für die Berechnung der Pearson-Korrelation zuständig ist. Sie ist Unterklasse der Klasse `Correlation`. Neben der Implementierung der Methode `use(graphs: List<Graph>): List<CorrelationOutput>` hält sie folgenden Konstruktor bereit:

- `Pearson(attributeCounter: int, attribute: Attribute, maximum: boolean)`
  Constructor of class Pearson
  `@param attributeCounter` number of pairs of attributes
  `@param attribute attribute` to which a correlation is to be found (set this value to `NULL` if there is no such attribute)
  `@param maximum` set this parameter `true` to search for strong correlations, `false` to search for weak correlations.

MutualCorrelation
Klasse, welche für die Berechnung einer weiteren Korrelation zuständig ist. Sie ist Unterklasse der Klasse `Correlation`. Neben der Implementierung der Methode `use(graphs: List<Graph>): List<CorrelationOutput>` hält sie folgenden Konstruktor bereit:

- `MutualCorrelation(attributeCounter: int, attribute: Attribute, maximum: boolean)`
  Constructor of class MutualCorrelation
  `@param attributeCounter` number of pairs of attributes
  `@param attribute` attribute to which a correlation is to be found (set this value to `NULL` if there is no such attribute)
  `@param maximum` set this parameter `true` to search for strong correlations, `false` to search for weak correlations.

CorrelationOutput
Klasse, welche die Ausgabe der Methode `use() : List<CorrelationOutput>` in ein sinnvolles Format bringt. Die Klasse ist ausschließlich für diesen Zweck gedacht. Es existieren die folgenden Methoden:

- `CorrelationOutput(firstAttribute: Attribute, secondAttribute: Attribute, outputNumber: double)`
  Constructor of class CorrelationOutput
  used to get a double which shows how strong a specific correlation is
  `@param firstAttribute` first attribute of a specific correlation
  `@param secondAttribute` second attribute of a specific correlation
  `@param outputNumber` shows how strong a specific correlation is

- `+ getFirstAttribute(): Attribute`
  used to get the first attribute of a specific correlation
  `@return` first attribute of a specific correlation

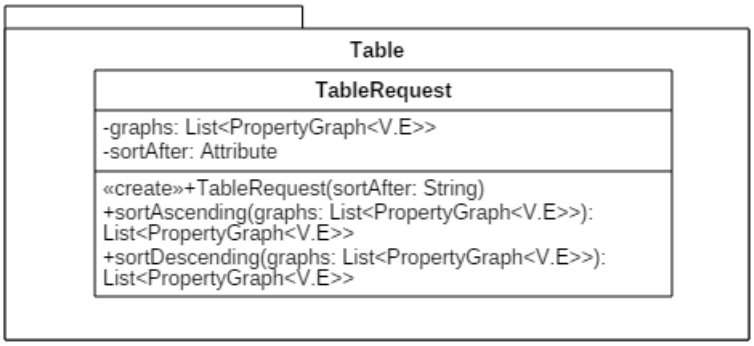
- `+ getSecondAttribute(): Attribute`
  used to get the second attribute of a specific correlation
  `@return` second attribute of a specific correlation

- `+ getOutputNumber(): double`
  used to get the double which shows how strong a specific correlation is
  `@return` double which shows how strong a specific correlation is

5.9.5 Benötigte Schnittstellen

Das Paket benötigt lediglich einen Zugriff zur Datenbank, um auf die in der Datenbank enthaltenen Graphen zugreifen zu können.

5.10 Tabelle



5.10.1 Situierung

Das Tabellenpaket übernimmt die Sortierung von Einträgen in der Tabelle.

5.10.2 Blackbox-Beschreibung

Durch das Anlegen einer neuen Instanz der Klasse `TableRequest` teilt man dem Paket mit, welche Graphen nach welchem Merkmal zu sortieren sind. Zurückgegeben wird eine sortierte Liste, welche alle in der Datenbank vorhandenen Graphen in sortierter Reihenfolge enthält.

5.10.3 Interner Aufbau

Das Paket besteht lediglich aus einer einzigen Klasse; der Klasse `TableRequest`. Eine Sortieranfrage wird wie folgt realisiert: Eine Methode aus einem anderen Paket legt eine neue Instanz der Klasse `TableRequest` an und ruft dann die Methode `sortAscending(graphs:List<PropertyGraph<V.E»): List<PropertyGraph<V,E>>` beziehungsweise `sortDescending(graphs:List<PropertyGraph<V.E»): List<PropertyGraph<V,E>>` auf.

5.10.4 Feinentwurf

TableRequest
Die Klasse TableRequest realisiert die gesamte Funktionalität des Pakets `Table`. Die Klasse enthält folgende Methoden:

- `+ TableRequest(sortAfter: String)`
  Constructor of class TableRequest
  `@param sortAfter` attribute by which the graphs are to be sorted
  `@throws IllegalInputException` thrown if the input string does not code an valid attribute

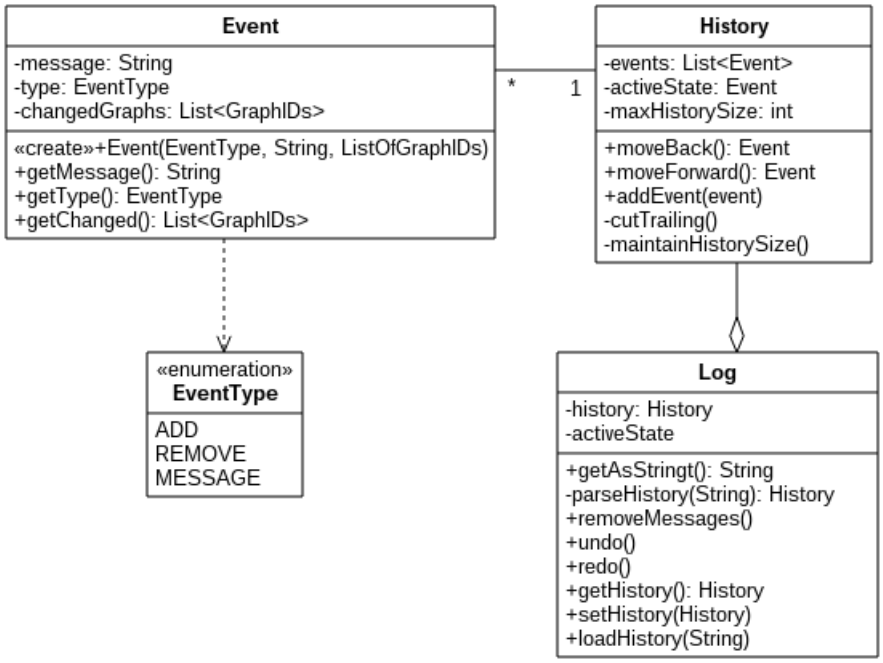
- `+ sortAscending(graphs: List<PropertyGraph<V.E»): List<PropertyGraph<V,E>>`
  sorts the graphs of database by a specific attribute in ascending order
  `@param graphs` list of graphs that should be sorted ascending
  `@return` List which inherits every graph of database sorted by a specific attribute

- `+ sortDescending(graphs: List<PropertyGraph<V.E»): List<PropertyGraph<V,E>>`
  sorts the graphs of database by a specific attribute in descending order
  `@param graphs` list of graphs that should be sorted descending
  `@return` List which inherits every graph of database sorted by a specific attribute

5.10.5 Benötigte Schnittstellen

Das Paket benötigt lediglich einen Zugriff zur Datenbank, um auf die in der Datenbank enthaltenen Graphen zugreifen zu können.

5.11 Log



5.11.1 Situierung

Das Log stellt einerseits die Funktionalität der Historie bereit und andererseits Methoden zum Laden und Speichern einer solchen Historie in Textform. Dabei ist von der LogUI zu unterscheiden, welche nur die Daten der Log-Historie verwendet, um daraus ein GUI-Element zu generieren.

5.11.2 Blackbox-Beschreibung

Für die Historie sind `Events` vom `EventType` `ADD/REMOVE` relevant; `MESSAGE-Events` sind nur für die Generierung der Logansicht in der GUI von Bedeutung. Letztere werden demnach beim Rückgängig machen bzw. Wiederholen eines `Events` ignoriert.

5.11.3 Interner Aufbau

Es wird eine angepasste Version des Memento-Entwurfsmusters verwendet.

5.11.4 Feinentwurf

Eine Beschreibung der relevanten Methoden der Klasse `Log` folgen.

- `getAsString() : String`
  `@return` a string with the format [EventType] “Event Message"GraphID-1, GraphID-2, ... \n
  Therefore each line of the string corresponds to one log entry.

- `parseHistory(string : String)`
  `@param string` parses the given text into a history.

- `undo()`
  Move back in the historie

Eine Beschreibung der relevanten Methoden der Klasse `Log` folgen.

- `moveBack() : Event`
  `@return` the most previous *Event* for which the `EventType` is not `MESSAGE`.

- `moveForward() : Event`
  `@return` the next *Event* for which the `EventType` is not `MESSAGE`.

- `addEvent(event : Event)`
  `@param event` this event will be added to the end of the history.
  Before this happens `cutTrailing()` is called. After the event is added `maintainHistorySize()` is called.

- `cutTrailing()`
  Remove all `Event`s after *activeState*.

- `maintainhistorySize()`
  Remove oldest `Event`s from the history until it's size matches *maxHistorySize*.

6 Model

6.1 Database

6.1.1 Situierung

Das Paket Datenbank ist in erster Linie für die Kommunikation zur MySQL-Datenbank verantwortlich. Dabei wird eine Verbindung zu einer bereits existenten MySQL-Datenbank aufgenommen, um die von der Steuerung übergebenen Daten zu speichern, zu bearbeiten und zu laden. Die zum Laden einer Graphendatenbank benötigten Informationen werden nach dem ersten Erstellen in einer.txt-Datei gespeichert, die ebenfalls vom Paket Database erstellt wird. Eine Graphendatenbank besteht dabei aus zwei MySQL-Tabellen, eine zum speichern der Graphen und die andere zum speichern der Filter, die zur selben MySQL-Datenbank gehören. Instanzen der Klasse `GraphDatabase` repräsentieren eine solche Graphendatenbank.

6.1.2 Blackbox-Beschreibung

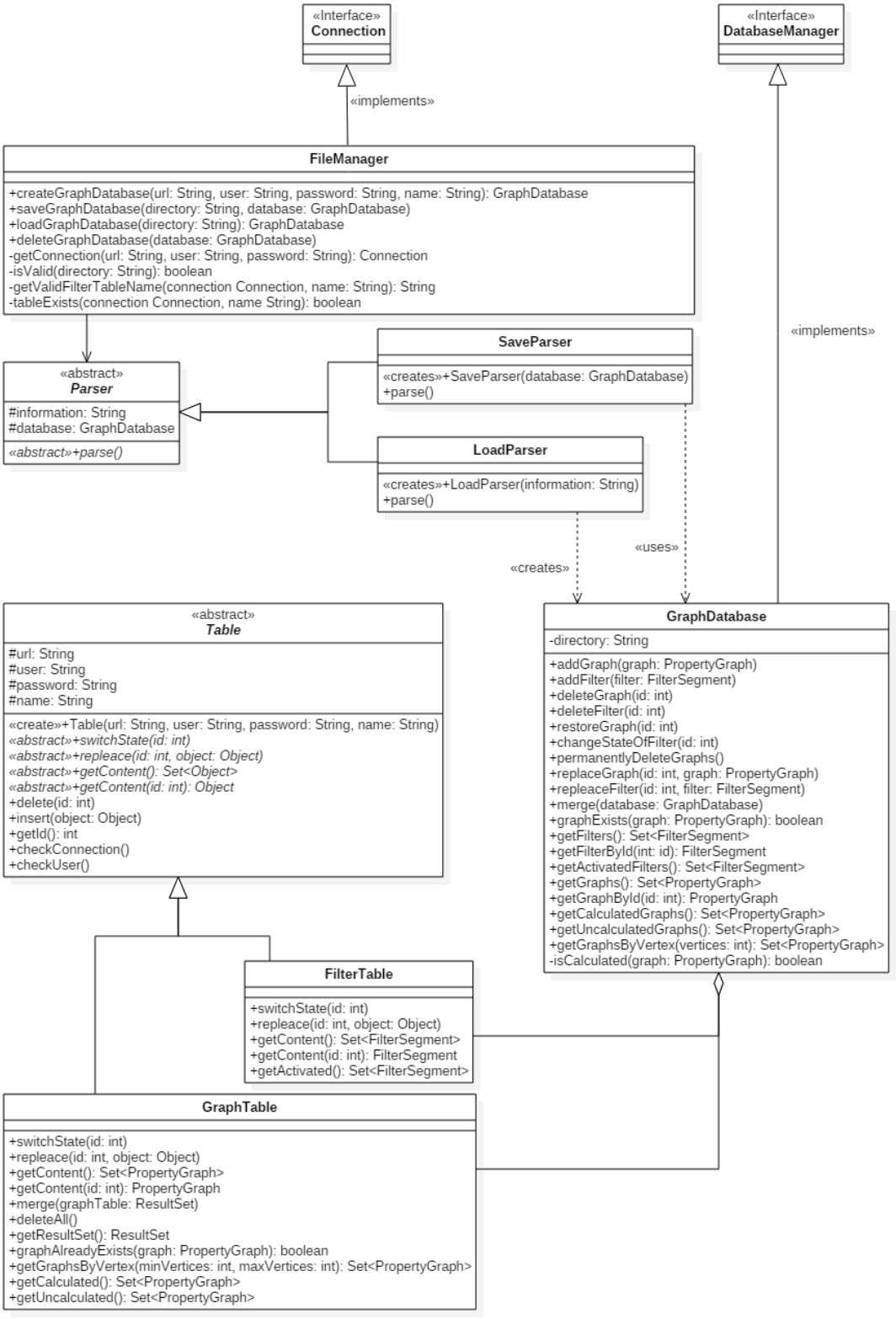
Die für die Steuerung bereitgestellten Methoden sind für die Realisierung der datenbankbezogenen Eingaben zuständig. Dabei sind vor allem zwei Schnittstellen von Bedeutung:

- `«Interface» Connection` :
  Diese Schnittstelle stellt die Funktionalität zur Verfügung, die benötigt wird, um auf eine Datenbank zugreifen zu können. Dabei sind insbesondere die Methoden `createGraphDatabase(url: String, user: String, password: String, name: String, directory: String): GraphDatabase` und `loadGraphDatabase(directory: String): GraphDatabase` für die Steuerung interessant, da diese durch die Rückgabe eines `GraphDatabase`-Objekts den indirekten zugriff auf die entsprechende Graphendatenbank erlauben.

- `«Interface» DatabaseManager` :
  Die Schnittstelle ermöglicht es der Steuerung, den Inhalt der Graphendatenbank sowie dessen Filter zu verwalten. Dabei fungiert diese Schnittstelle als Fassade, die die entgegengenommenen Befehle an das darunterliegende Subsystem delegiert, wo diese dann in SQL-Queries umgewandelt werden.

6.1.3 Interner Aufbau

Das Paket Database besteht aus zwei lose gekoppelten Subsystemen: Über die Schnittstelle «Interface» Connection wird der Steuerung die Möglichkeit geboten, eine neue Datenbakverbindung herzustellen. Soll dabei eine neue Graphendatenbank erstellt werden, so werden zwei MySQL-Tabellen, die von den Klassen GraphTable und FilterTable repräsentiert werden, in der gegebenen MySQL-Datenbank erstellt. Die Steuerung bekommt dann das GraphDatabase-Objekt übergeben, das die soeben erstellte Graphendatenbank repräsentiert. Über die Klasse SaveParser werden die zum laden der Graphendatenbank nötigen Informationen in einer Textdatei gespeichert, sodass die Klasse LoadParser alleine den Inhalt dieser Datei benötigt, um eine bereits erstellte Graphendatenbank zu laden.


«Interface»
Connection
«Interface»
DatabaseManager
«implements»
FileManager
+createGraphDatabase(url: String, user: String, password: String, name: String): GraphDatabase
+saveGraphDatabase(directory: String, database: GraphDatabase)
+loadGraphDatabase(directory: String): GraphDatabase
+deleteGraphDatabase(database: GraphDatabase)
-getConnection(url: String, user: String, password: String): Connection
-isValid(directory: String): boolean
-getValidFilterTableName(connection Connection, name: String): String
-tableExists(connection Connection, name String): boolean
«implements»
SaveParser
«creates»+SaveParser(database: GraphDatabase)
+parse()
«abstract»
Parser
#information: String
#database: GraphDatabase
«abstract»+parse()
LoadParser
«creates»+LoadParser(information: String)
+parse()
«uses»
«creates»
«abstract»
Table
#url: String
#user: String
#password: String
#name: String
«create»+Table(url: String, user: String, password: String, name: String)
«abstract»+switchState(id: int)
«abstract»+repleace(id: int, object: Object)
«abstract»+getContent(): Set<Object>
«abstract»+getContent(id: int): Object
+delete(id: int)
+insert(object: Object)
+getId(): int
+checkConnection()
+checkUser()
GraphDatabase
-directory: String
+addGraph(graph: PropertyGraph)
+addFilter(filter: FilterSegment)
+deleteGraph(id: int)
+deleteFilter(id: int)
+restoreGraph(id: int)
+changeStateOfFilter(id: int)
+permanentlyDeleteGraphs()
+replaceGraph(id: int, graph: PropertyGraph)
+repleaceFilter(id: int, filter: FilterSegment)
+merge(database: GraphDatabase)
+graphExists(graph: PropertyGraph): boolean
+getFilters(): Set<FilterSegment>
+getFilterById(int: id): FilterSegment
+getActivatedFilters(): Set<FilterSegment>
+getGraphs(): Set<PropertyGraph>
+getGraphById(id: int): PropertyGraph
+getCalculatedGraphs(): Set<PropertyGraph>
+getUncalculatedGraphs(): Set<PropertyGraph>
+getGraphsByVertex(vertices: int): Set<PropertyGraph>
-isCalculated(graph: PropertyGraph): boolean
FilterTable
+switchState(id: int)
+repleace(id: int, object: Object)
+getContent(): Set<FilterSegment>
+getContent(id: int): FilterSegment
+getActivated(): Set<FilterSegment>
GraphTable
+switchState(id: int)
+repleace(id: int, object: Object)
+getContent(): Set<PropertyGraph>
+getContent(id: int): PropertyGraph
+merge(graphTable: ResultSet)
+deleteAll()
+getResultSet(): ResultSet
+graphAlreadyExists(graph: PropertyGraph): boolean
+getGraphsByVertex(minVertices: int, maxVertices: int): Set<PropertyGraph>
+getCalculated(): Set<PropertyGraph>
+getUncalculated(): Set<PropertyGraph>

6.1.4 Feinentwurf

- «Interface» Connection
  This interface allows access to a FileManager-Object.

  + createGraphDatabase(url: String, user: String, password: String, name: String): GraphDatabase
  Creates and returns a new GraphDatabase-Object.
  @param url localizes the MySQL-Database in which the data should be stored.
  @param user username of the MySQL-Database user.
  @param password password of the user.
  @param name determines the name of the MySQL-Table in which the Graphs will be stored.
  @return GraphDatabase-Object will be created and returned.
  @throws WrongUserException if the user has no access to the current MySQL-Database.
  @throws UserNotPrivilegedException if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.
  @throws InvalidNameException if the given name cannot be used.
  @throws InvalidConnectionException if the connection to the MySQL-Database cannot be established.

  + saveGraphDatabase(directory: String, database: GraphDatabase)
  Saves the information of the given GraphDatabase-Object in a text file.
  @param directory localizes where the text file will be saved.
  @param database GraphDatabase that will be saved as text file.
  @throws InvalidDirectoryException if the text file could not be saved in the given directory.

  + loadGraphDatabase(directory: String): GraphDatabase
  Creates and returns a GraphDatabse-Object from the information contained by the given file.
  @param directory localizes the text file from which a GraphDatabase-Object can be created.
  @return GraphDatabase-Object that contains the connection to an already used MySQL-Database.
  @throws FileNotAsExpectedException if the content of the given File was not as expected.

  + deleteGraphDatabase(database: GraphDatabase)
  Deletes given database by deleting its MySQL-Tables and the text file that belongs to it.

@param database GraphDatabase-Object that should be deleted.

- FileManager
  This class creates GraphDatabase-Objects and saves them by creating text files.

  Implements «Interface» Connection. Descriptions can be found above.

  - getConnection(url: String, user: String, password: String): Connection
  Creates and returns a Connection-Objcect.
  @param url localizes the MySQL-Database in which the data should be stored.
  @param user username of the MySQL-Database user.
  @param password password of the user.
  @return the Connection to the MySQL-Database.
  @throws InvalidConnectionException if connection could not be established.
  @throws WrongUserException if the user has no access to the current MySQL-Database.
  @throws UserNotPrivilegedException if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

  - isValid(directory: String): boolean
  Checks whether a directory is valid or not.
  @param directory localizes where the text file will be saved.
  @return true if text file can be saved in given directory.

  - getValidFilterTableName(name: String): String
  Returns a name for a MySQL-Table that does not already exist.
  @param name name of the GraphTable-Object.
  @return a Valid name that can be used to create a FilterTable-Object.

  - tableExists(connection: Connection, name: String): boolean
  Checks if a MySQL-Table with the given name already exists.
  @param connection the Connection to a MySQL-Database.
  @param name name of a MySQL-Table.
  @return true if there already is a MySQL-Table with the given name.

- «abstract» Parser

  «abstract» + parse():
  creates a GraphDatabase-Object or the content of the text file that should be created by the FileManager

- `SaveParser`
  Stores all the information that is necessary to load the given `GraphDatabase`-Object into a String.

  `«creates» SaveParser(database: GraphDatabase)`
  Creates a `SaveParser`-Object and sets the given `database` as attribute.
  `@param database` contains all the information that should be saved in the text file.

  `+ parse()`:
  Parses the given `database` into a String and saves its value as attribute.

- `LoadParser`
  Creates a `GraphDatabase`-Object by setting its `Table`-Objects and the directory where its text file is saved.

  `«creates» LoadParser(information: String)` Creates a `LoadParser`-Object and sets the given `information` as attribute.
  `@param information` contains all the information that should be saved in the text file.

  `+ parse()`:
  Parses the given information into a `GraphDatabase`-Object and sets its value as attribute.
  `@throws FileNotAsExpectedException` if the content of the File is not as expected.
  `@throws InvalidConnectionException` if connection could not be established.
  `@throws WrongUserException` if the user has no access to the current MySQL-Database.
  `@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

- `«Interface»DatabaseManager`
  This interface allows access to the currently used `GraphDatabase`-Object.

  `+ addGraph(graph: PropertyGraph)`
  Inserts `graph` into the MySQL-Table that belongs to the current MySQL-Database, sets its automatically generated id and determines whether it should be marked as calculated or not.
  `@param graph` object that should be inserted into the current MySQL-Database.
  `@throws WrongUserException` if the user has no access to the current MySQL-Database.

`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ addFilter(filter: FilterSegment)`
Inserts `filter` into the MySQL-Table belonging to the current MySQL-Database.
`@param filter` object that should be inserted into the current MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database..
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ deleteGraph(id: int)`
`PropertyGraph`-Object with the given `id` will be marked as deleted.
`@param id` identifies a `PropertyGraph`-Object in the current MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ deleteFilter(id: int)`
`FilterSegment`-Object with the given `id` will be deleted.
`@param id` identifies a `FilterSegment`-Object in the current MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ restoreGraph(id: int)`
`PropertyGraph`-Object with the given `id` will be restored (unmarked as deleted).
`@param id` identifies `PropertyGraph`-Object in the current MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database..
`@throws ConnectionLostException` if the connection to the Current MySQL-

Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ changeStateOfFilter(id: int)`
The state (determines whether a `FilterSegment`-Object is activated or not) of the given `FilterSegment`-Object will be changed.
`@param id` identifies a `FilterSegment`-Object.
`@throws WrongUserException` if the user has no access to the current MySQL-Database..
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ permanentlyDeleteGraphs()`
Every `PropertyGraph`-Object that is marked as deleted will be removed irreservably from the current MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ replaceGraph(id: int, graph: PropertyGraph)`
`graph` replaces the `PropertyGraph`-Object identified by the given `id`. Additionally it will be checked whether the new `graph` is already calculated or not.
`@param id` identifies a `PropertyGraph`-Object that will be replaced.
`@param graph` new object that should replace the old one.
`@throws WrongUserException` if the user has no access to the current MySQL-Database..
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ replaceFilter(id: int, filter: FilterSegment)`
`filter` replaces the `FilterSegment`-Object identified by the given `id`.
`@param id` identifies a `FilterSegment`-Object that will be replaced.
`@param filter` new object that should replace the old one.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.

`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ merge(database: GraphDatabase)`
Every `PropertyGraph`-Object in `databese` that does not already exist in the current MySQL-Database, will be inserted to the current MySQL-Database.
`@param database GraphDatabase`-Object that should be merged with the current MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ graphExists(graph: PropertyGraph): boolean`
Checks whether a `propertyGraph`-Object already exists.
`@param graph` a `PropertyGraph`-Object
`@return` true if the given `graph` is isomorphic to another `PropertyGraph`-Object in the current MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ getFilters(): Set<FilterSegment>`
Returns all `FilterSegment`-Objects.
`@return` all `FilterSegment`-Objects in the current MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ getFilterById(id: int): FilterSegment`
Identifies a `FilterSegment`-Object and returns it.
`@param id` identifies a `FilterSegment`-Object.
`@return` identified `FilterSegment`-Object in the MySQL-Database.
`@throws WrongUserException` if the user has no access to the current

MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ getActivatedFilters(): Set<FilterSegment>`
Identifies all activated `FilterSegment`-Objects and returns them.
`@return` every `FilterSegment`-Object in the MySQL-Database that is marked as activated.
`@throws WrongUserException` if the user has no access to the current MySQL-Databases.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ getGraphs(): Set<PropertyGraph>`
Returns all `PropertyGraph`-Objects that are not marked as Deleted.
`@return` all `PropertyGraph`-Objects in the MySQL-Database that are not marked as deleted.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ getGraphById(id: int): PropertyGraph`
Identifies a `PropertyGraph`-Object and returns it.
`@param id` identifies a `PropertyGraph`-Object.
`@return` identified `PropertyGraph`-Object in the MySQL-Database.
`@throws WrongUserException` if the user has no access to the current MySQL-Database..
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ getGraphsByVertexNumber(vertices: int): List<PropertyGraph>`
Returns all `PropertyGraph`-Objects with the given number of vertices.
`@param vertices` number of Vertices

`@return` all `PropertyGraph`-Objects in the MySQL-Database with the given number of `vertices`.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ getCalculatedGraphs(): Set<PropertyGraph>`
`@return` all `PropertyGraph`-Objects in the MySQL-Database that are marked as calculated.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

`+ getUncalculatedGraphs(): Set<PropertyGraph>`
`@return` all `PropertyGraph`-Objects in the MySQL-Database that are marked as uncalculated.
`@throws WrongUserException` if the user has no access to the current MySQL-Database.
`@throws ConnectionLostException` if the connection to the Current MySQL-Database was lost.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

- `GraphDatabase`
  This class represents a graphdatabase in which `PropertyGraph`-Objects and `FilterSegments` are saved.

  Implements `«Interface»DatabaseManagement`. Descriptions can be found above.

  `- isCalculated(graph: PropertyGraph): boolean`
  Checks whether every Property of a `PropertyGraph`-Object is calculated or not.
  `@param graph` the `PropertyGraph`-Object.
  `@return` true if the given `graph` is fully calculated.

- «abstract» Table
  This class represents a MySQL-Table.

  «create»Table(url: String, user: String, password: String, name: String)
  Creates a new Table.
  @param url location of the MySQL-Database that contains the MySQL-Table which is represented by a subclass of Table.
  @param user username of the MySQL-Database user.
  @param password password of the user.
  @param name name of the MySQL-Table which is represented by a subclass of Table.

  «abstract» + switchState(id: int)
  Functionality will be described in the subclasses.
  @param id identifies a row in the represented MySQL-Table.

  «abstract» + repleace(id: int, object: Object)
  Repleaces an Object in the represented MySQL-Table.
  @param id identifies a row in the represented MySQL-Table.
  @param object Java object that will be inserted.
  Table entry of the identified row will be repleaced by the given object.

  «abstract» + getContent(): Set<Object>
  @return every object in the represented MySQL-Table.

  «abstract» + getContent(id: int): Object
  @param id identifies a row in the represented MySQL-Table.
  @return the object that is saved in the identified row.

  + delete(id: int)
  Deletes the identified row from the represented MySQL-Table.
  @param id identifies a row in the represented MySQL-Table.

  + insert(object: Object)
  Inserts the given object into the MySQL-Table. @param object Java object that should be inserted into the represented MySQL-Table.

  + getId(): int
  @return Value of the next free id.

`+ checkConnection()`
Checks if the current connection is still valid.
`@throws WrongUserException` if access is denied for the current user.
`@throws ConnectionLostException` if the connection was lost.

`+ checkUser()`
Checks if current user is privileged to edit MySQL-Tables.
`@throws UserNotPrivilegedException` if the current user is not privileged to edit MySQL-Tables in the current MySQL-Database.

- `GraphTable`
  This class represents a MySQL-Table where `PropertyGraph`-Objects will be stored.

  `+ switchState(id: int)`
  State (defines whether a `PropertyGraph`-Object is deleted or not) of the identified `PropertyGraph`-Object will be switched.
  `@param id` identifies a `PropertyGraph`-Object contained by the represented MySQL-Table.

  `+ repleace(id: int, object: Object)`
  Already described in (`abstract Table`).

  `+ deleteAll()`
  All `PropertyGraph`-Objects that are marked as deleted will be removed from the represented MySQL-Table.

  `+ merge(graphTable: ResultSet)`
  The given content will be inserted into the represented MySQL-Table.
  `@param graphTable` content of another MySQL-Table.

  `+ graphAlreadyExists(graph: PropertyGraph): boolean`
  Checks if a `PropertyGraph`-Object already Exists.
  `@param graph PropertyGaph`-Object.
  `@return` true if the given `graph` already exists in the represented MySQL-Table.

  `+ getResultSet(): ResultSet`
  `@return` the content of the represented MySQL-Table as ResultSet.

  `+ getContent(): Set<PropertyGraph>`
  `@return` every `PropertyGraph`-Object in the represented MySQL-Table that is not marked as deleted.

`+ getContent(id: int): PropertyGraph`
`@param id` identifies a `PropertyGraph`-Object.
`@return` identified `PropertyGraph`-Object in the represented MySQL-Table.

`+ getGraphsByVertexNumber(vertices: int): List<PropertyGraph>`
`@param vertices` number of vertices.
`@return` every `PropertyGraph`-Object that has the given number of vertices but is not marked as deleted.

`+ getCalculated(): Set<PropertyGraph>`
`@return` all `PropertyGraph`-Objects in the represented MySQL-Table that are marked as calculated.

`+ getUncalculatedGraphs(): Set<PropertyGraph>`
`@return` all `PropertyGraph`-Objects in the represented MySQL-Table that are marked as uncalculated.

- `FilterTable`
  This class represents a MySQL-Table where `FilterSegment`-Objects will be stored.

  `+ switchState(id: int)`
  `@param id` identifies a `FilterSegment`-Object contained by the represented MySQL-Table
  State (defines whether a `FilterSegment`-Object is activated or not) of the identified `FilterSegment`-Object will be switched.

  `+ repleace(id: int, object: Object)`
  Already described (`abstract Table`).

  `+ getContent(): Set<FilterSegment>`
  `@return` every `FilterSegment`-Object in the represented MySQL-Table.

  `+ getContent(id: int): FilterSegment`
  `@param id` identifies a `FilterSegment`-Object.
  `@return` identified `FilterSegment`-Object in the represented MySQL-Table.

  `+ getActivated(): Set<FilterSegment>`
  `@return` all `FilterSegment`-Objects in the represented MySQL-Table that are marked as activated.

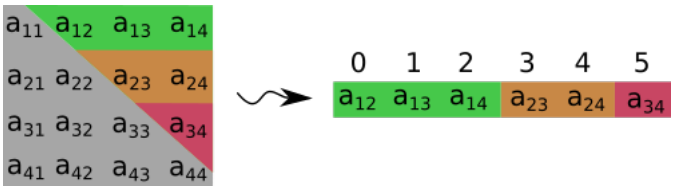
6.1.5 Benötigte Schnittstellen

Da das Model die letzte Schicht der Systemarchitektur von Grape darstellt, hat das Paket Database lediglich zugriff auf die Pakete im Model. Des Weiteren greift das Paket Database auf das Paket java.sql in der Java Standardbibliothek zu, um auf MySQL-Datenbanken zugreifen zu können.

7 Algorithmen Pseudocode

7.1 Graphen-Generierung

Für die folgenden Generierungs-Algorithmen wird eine Graphendarstellung vorgeschlagen, welche auf der bekannten Matrix-Darstellung für Graphen basiert.
Wird die Matrixdarstellung eines ungerichteten, schlingenfreien Graphen betrachtet, so stellen wir fest, dass (*i*) die Matrix symmetrisch ist, und (*ii*) die Diagonaleinträge Null sind. Da somit die grauen Einträge implizit durch die farbig markierten Einträge bestimmbar sind, kann auf diese verzichtet werden. Zur effizienten Speicherung wird die Matrix exemplarisch wie folgt „aufgeklappt“.



Für n Knoten hat die Matrixdarstellung n^2 viele Einträge. Im Vergleich dazu hat die aufgeklappte Darstellung $\frac{n(n-1)}{2}$ viele Einträge.
Um zu überprüfen, ob zwei Knoten $k_1 > k_2$ durch eine Kante verbunden sind, kann die folgende Formel verwendet werden $\frac{k_1(k_1-1)}{2} - \frac{k_2(k_2-1)}{2} + k_2 - k_1 - 1$. Das Ergebnis dieser ist die Stelle der aufgeklappten Matrix, an welcher der Eintrag $a_{k_1 k_2}$ steht.

Für die Graphen Generierung wird ein Algorithmus bereitgestellt, welche garantiert, dass ein zufälliger, streng-zusammenhängender Graph zurückgegeben wird, welcher noch nicht Teil einer Vergleichsmenge ist. Falls bereits alle Graphen mit n-Knoten Teil der Vergleichsmenge sind, wird *null* zurückgegeben. Dazu wird die oben vorgestellte Darstellung für Graphen verwendet.
Obwohl der Algorithmus bis zu $2^n - 1$ Schleifendurchläufe benötigt ist er in der Praxis deutlich schneller. Die Geschwindigkeit hängt hauptsächlich von der Anzahl der bereits gefundenen Graphen in der Vergleichsmenge ab. Ist diese beispielsweise leer, dann wird der erste gefundene streng-zusammenhängende Graph zurückgegeben.

```
1  Graph generateNextRandomGraph(int numVertices, Set<Graph>
       comparisonSet) {
2      if comparisonSet.size() = 2^numVertices { return null; }
3
4      graph := random base-2 number of length n(n-1)/2;
5
```

```
6       while(!isStronglyConnected(graph) or comparisonSet.contains(
            graph)) {
7           graph++;
8       }
9
10      return graph;
11  }
```

Der vorgeschlagene Algorithmus garantiert nur die Anzahl an Knoten, die Anzahl Kanten kann dabei nicht festgelegt werden. Aus diesem Grund ist ein zweiter Algorithmus gegeben, welcher es erlaubt Knoten- und Kantenanzahl festzulegen. Auch dieser Algorithmus garantiert, dass ein neuer, zufälliger, streng-zusammenhängender Graph gefunden wird, sofern noch nicht alle Graphen mit der gegebenen Anzahl an Knoten und Kanten in der Vergleichsmenge sind. In letzterem Fall gibt der Algorithmus *null* zurück.

```
1   Graph generateNextRandomGraph
2           (int numVertices, int numEdges Set<Graph> comparisonSet)
                {
3
4       if comparisonSet.size() == (n(n-1)/2)! { return null }
5
6       graph = random base-2 number of length n(n-1)/2
7                 where exactly numEdges-digits are one
8
9       i := 0
10      while(!isStronglyConnected(graph)
11                      && comparisonSet.contains(graph)) {
12          if c[i] < i {
13              if isEven(i) {
14                  swap(graph[0], graph[i])
15              } else {
16                  swap(graph[c[i]], graph[i])
17              }
18          } else {
19              c[i] = 0
20              i++
21          }
22
23          if isValidVertexColoring(graph[i - maxDegree]) {
24              add graph[i - maxDegree] to validMinimalColorings
25          }
26
27      return graph;
```

```
28      }
```

7.2 Färbungen

7.2.1 Knotenfärbungen

Im Folgenden werden zwei Algorithmen vorgestellt, welche das Finden von allen möglichen Knotenfärbungen und deren Validierung als tatsächliche ermöglichen. In beiden Algorithmen werden Färbungen (*Colorings*) verwendet. Diese werden jeweils als Zahl f zur Basis k aufgefasst, die Farben sind daher $\{0, ..., k-1\}$. Die $i-te$ Stelle von f ist die Farbe des $i-ten$ Knotens. Diese Entscheidung ist in erster Linie entscheidend für den *isValidVertexColoring*-Algorithmus und ermöglicht effizienten Zugriff auf die jeweilige Farbe. Auch einzelne Färbungen können potenziell in einem primitiven Datentypen gespeichert werden, was ebenfalls einen Geschwindigkeits- und Speichervorteil zu Folge hat.

Es wird ein Algorithmus vorgestellt, welcher alle minimalen Knotenfärbungen eines Graphen in $\mathcal{O}(n!)$ Zeit findet. Auch die Ausgabe hat Größe $\mathcal{O}(n!)$. Falls nur eine Färbung gewünscht ist, kann der Algorithmus leicht angepasst werden, indem die erste gültige Färbung zurückgegeben wird. In diesem Fall ist der Speicherbedarf in $\mathcal{O}(n)$.

```
1  Array of Colorings getAllMinimalVertexColorings(Graph graph) {
2      maxDegree := graph.getMaxDegree()
3      maxColors := graph.getNumVertices()
4      initialColorings := Array of Colorings
5                                  of length (maxColors - maxDegree)
6      validMinimalColorings := Array of Colorings
7
8
9      // Generate an inital possible coloring with i different
           colors
10     for i := maxDegree,...,maxColors {
11         for j := 1,...,i {
12             Of coloring initialColorings[i - maxDegree]
13                                             set j-th digit to
                                                   color j
14         }
15     }
16
17     // Test all permutations of the i-th coloring
18     c := integer Array of length maxColors
19
20     for i := 1,...,maxColors {
21         if isValidVertexColoring(initialColorings[i - maxDegree
               ]) {
```

```
22                add initialColorings[i - maxDegree]
23                                                to
                                                      validMinimalColorings

24            }
25
26            i := 0
27            while i < maxColors {
28                if c[i] < i {
29                    if isEven(i) {
30                        swap(initialColorings[0], initialColorings[i
                              ])
31                    } else {
32                        swap(initialColorings[c[i]],
                              initialColorings[i])
33                    }
34                } else {
35                    c[i] := 0
36                    i++
37                }
38
39                if isValidVertexColoring(initialColorings[i -
                      maxDegree]) {
40                    add initialColorings[i - maxDegree]
41                                                    to
                                                          validMinimalColorings

42                }
43            }
44
45            // this case is satisfied for k = n colors,
46            // therefore the algorithm always terminates
47            if validMinimalColorings contains > 0 colorings {
48                return validMinimalColorings
49            }
50        }
51    }
```

Behauptung: $getAllMinimalVertexColorings \in \mathcal{O}(n!)$
Beweis: Die erste Schleife generiert eine Menge von Anfangsfärbungen. Die Laufzeit dieser Schleife ist durch $\mathcal{O}(maxColors^2)$ gegeben, da für jeden Schleifendurchlauf gilt $i, j \leq maxColors$. Die zweite Schleife überprüft für $k = 1, ..., maxColors$ Farben, für alle $k!$ Permutationen, ob eine gültige Permutation vorliegt. Wir definieren $m_D := 1$

und $m_C := maxColors$. Damit ist die Laufzeit dieser Schleife durch $\mathcal{O}(m_D! + (m_D + 1)! + ... + m_C!) =: \mathcal{O}(k! + (k-1)! + ... + (k-i)!)$ gegeben. Hierbei wird der konstante Faktor für die Überprüfung, ob eine korrekte Färbung vorliegt ignoriert.
Allgemein beobachten wir, dass für $n, i_1, i_2 \in \mathbb{N}$ mit $i_1 > i_2$ gilt $(n-i_1)! \in \mathcal{O}((k-i_2)!)$. Dies folgt aus $i_1 - i_2 > 0$ und $\lim_{n\to\infty} \frac{(n-i_1)!}{(n-i_2)!} = \lim_{n\to\infty} \frac{1}{n^{\underline{i_1-i_2}}} = 0$. Hierbei ist $n^{\underline{k}} := n * (n-1) * ... * (n-k+1)$.
Somit ist also $\mathcal{O}(k!+(k-1)!+...+(k-i+1)+(k-i)!) = O(k!+(k-1)!+...+(k-i+1)!)$. Induktiv folgt somit die Behauptung.

■

Der folgende Algorithmus entscheidet in $\mathcal{O}(e)$, ob für einen Graphen eine gültige Knotenfärbung vorliegt.

```
1  Boolean isValidVertexColoring(Coloring coloring, Graph graph) {
2
3      k is the number of colors in coloring
4
5      for each edge e = (v, w) in graph {
6          if (coloring / k^v) % k == (coloring / k^w) % k {
7              return false
8          }
9      }
10
11     return true;
12 }
```

Hinweis: Der Algorithmus überprüft nicht, ob die gegebene Färbung minimal ist.

7.2.2 Totalfärbung

Um eine Totalfärbung zu berechnen definieren wir für einen Graphen $G = (V, E)$ eine Bijektion auf einen erweiterten Graphen G' und verwenden den obigen Knotenfärbungs-Algorithmus zur Berechnung der Totalfärbung. Die gegebene Transformation benötigt polynomiell viel Aufwand, womit auch für diesen Algorithmus die Laufzeit von $\mathcal{O}(n!)$ (für $n = |V|$) gilt.
Konstruiere $G' = (V', E')$ wie folgt.

- Setze $V' := V \cup E$. Die Knoten aus E werden nun als Kanten-Knoten bezeichnet.
- Betrachte alle Kanten-Knoten $\{v, w\}$. Verbinde nun $\{v, w\}$ mit den Knoten v, w und mit den Kanten-Knoten $\{v', w'\}$, falls $v = v'$ gilt.

Eine minimale Knotenfärbung von G' ist eine minimale Totalfärbung von G.

7.2.3 Äquivalenz von Färbungen

Zuletzt wird ein Algorithmus präsentiert, welcher für zwei minimale Färbungen entscheidet, ob diese nach folgender Definition äquivalent sind. Diese ist aus dem Pflichtenheft entnommen.
Um zu überprüfen, ob zwei Totalfärbungen äquivalent sind, geht man wie folgt vor: Zunächst werden alle Knoten und Kanten, die die gleiche Färbung bzgl. der ersten Totalfärbung besitzen, zu einer Menge zusammengefasst. Es entstehen n Mengen $K_1, K_2, ..., K_n$. *Anschließend fasst man alle Knoten und Kanten, die die gleiche Färbung bzgl. der zweiten Totalfärbung haben, ebenfalls zu einer Menge zusammen. Es entstehen n Mengen* $G_1, G_2, ..., G_n$. *Falls die beiden Totalfärbungen nicht äquivalent sind, so muss gelten:*
$\exists K_h(k \in \{1, ..., n\}) \forall G_i(i \in \{1, ..., n\}) : K_h \backslash G_i \neq \emptyset$

Diese Definition lässt sich leicht als Algorithmus umsetzen, davor ist es allerdings nötig das Graphen-Isomorphie Problem zu lösen. Dies kann selbst implementiert werden. Hier verwenden wir die von `JGraphT` bereitgestellte Funktionalität.

```
1  boolean equivalentColoring(Coloring col_1, col_2) {
2
3      if numColors(col_1) != numColors(col_2) return false
4
5      K : Array of Set of Vertices of length numColors(col_1)
6      G : Array of Set of Vertices of length numColors(col_2)
7
8      for (Set of Vertices k : K) {
9
10         foundMatching := false
11
12         for (Set of Vertices g : G) {
13             if k = g {
14                 foundMatching := true
15             }
16         }
17
18         if foundMatching is false {
19             return false
20         }
21     }
22
23     return true
24 }
```

Anmerkungen

Die Algorithmen zur Berechnung eines zufälligen Graphen und zum Finden aller Knotenfärbungen basieren beide auf der Generierung von Permutationen. Beide Algorithmen basieren auf der nicht-rekursiven Version von Heap's Algorithmus[1].

[1] Sedgewick, R. (1977) "Permutation Generation Methods"`http://www.cs.princeton.edu/~rs/talks/perms.pdf`

7.3 BFS Code Berechnung

Es folgt ein Algorithmus, welcher den minimalen BFS Code eines Graphen $G = (V, E)$ findet. Bei diesem Algorithmus gehen wir von einer gegebenen Knotenmenge $V = \{A, B, C, ...\}$ aus. Diese erste Menge wird nur verwendet, um den Knoten eine ursprüngliche Benennung zu geben und wird durch den Algorithmus nach und nach durch eine Nummerierung aus natürlichen Zahlen ersetzt. Dazu definieren wir eine Abbildung $map : G* \mapsto V*$ mit $G* = V \cup E$ und $V* = \{1, ..., |V|\} \cup \{FORWARD, BACKWARD\}$. Diese Abbildung ordnet den Knoten von G eine eindeutige natürliche Zahl zu und markiert alle Kanten von G mit $FORWARD$ oder $BACKWARD$. Es ist also $map(v) \in \mathbb{N}$ $(\forall v \in V)$ die für den BFS Code benötigte Nummerierung und $map(e)$ für $e \in E$ gibt an, um welche Art von Kante es sich handelt. Die Abbildung map definiert also genau einen BFS Code.
Die erste vorgestellte Methode verwendet jeden Knoten $v \in V$ als Startknoten und überprüft, ob ein minimaler BFS Code vorliegt.

```
1  findMinimalBFSCode(Graph G = (V,E)) : BFSCode {
2      minimalBFSCode = [inf, ..., inf, 0]
3
4      for (Vertex v : V) {
5          minimalBFSCode =
6                  findMinimalBFSCode(minimalBFSCode, G,
7                                  {(v)}, new Map<G*,V*>, 1)
8      }
9
10     return minimalBFSCode
11 }
```

Die nächste Methode übernimmt das tatsächliche Finden eines minimalen BFS Codes für den jeweiligen Startknoten. Dazu wird eine mögliche Definition von map beschrieben und der zugehörige BFS Code auf Minimalität überprüft. Falls ein Präfix des BFS Codes größer ist, ist auch der gesamte BFS Code größer, es kann daher mit einer anderen Permutation fortgefahren werden.

```
1  BFSCode findMinimalBFSCode(
2          BFSCode bfsCode, Graph G = (V, E),
3          Permutations V', Map<G*,V*> map, n Integer) {
4
5      minimalBFSCode := bfsCode
6
7      E' := {}
8
9      p' := arbitrary permutation in V'
10
```

```
11        for (Vertex u : p') {
12            for (Edge e = {u, v} : E) {
13                if (map(v) = undefined) {
14                    E' := E'.put(v)
15                }
16            }
17        }
18
19        V'' := {p : p is permutation of E'}
20
21        for (Permutation p : V') { // for all stage n permutations
22            for (Permutation p' : V'') { // for all stage n+1
                  permutations
23                for (Vertex u : p) { // work through vertices u in
                      ascending order defined by V' and V''
24                    for (Edge e = {u, v} in E) { // u in V' and v in
                          V''
25                        if (map(v) = undefined) {
26                            map(e) := FORWARD
27                            map(v) := ++n
28                        } else {
29                            map(e) := BACKWARD
30                        }
31                    }
32                }
33
34                newBFSCode := readBFSCode(G, map) // read currernt
                      BFS Code
35
36                if (newBFSCode > bfsCode) {   // if prefix is larger
                       then no BFS Code resulting from this can be
                      smaller continue with original BFS Code
37                    return bfsCode
38                }
39
40                minimalBFSCode := findMinimalBFSCode(newBFSCode, G,
                      V'', map, n)
41            }
42        }
43
44        return minimalBFSCode
45    }
```

Die das Übersetzen der Abbildung *map* zu einem BFS Code erfolgt mittels der letzten Methode.

```
1  readBFSCode(Graph G = (V,E), Map<G*, V*> map) {
2
3      bfsCode := () // empty BFS Code
4
5      for (Edge e = {u,v} in E) {
6          u' := map(u)
7          v' := map(v)
8          bfsCode.append(map(e), min{u',v'}, max{u',v'})
9      }
10
11     bfsCode := sort(bfsCode) // sorts the bfsCode in ascending
           order
12     return bfsCode.appendZero()
13 }
```

Ein Aufruf von *sort(bfsCode)* sortiert den jeweiligen BFS Code so, dass es keine kleinere Sortierung gibt. Für den Vergleich zweier BFS Code wird die Definition der lexikographischen Sortierung verwendet:
Für zwei BFS Codes $c_1 = [c_1^1, c_1^2, ...]$ und $c_2 = [c_2^1, c_2^2, ...]$ gilt $c_1 < c_2$, wenn eines der folgenden Axiome erfüllt ist.

(1) $\exists k \leq min(|c_1|, |c_2|)$ für das

- $\forall j < k : c_1^j = c_2^j$
- $c_1^k < c_2^k$

(2) c_1 ist Präfix von c_2

Diese Definition lässt sich leicht als Algorithmus umsetzen, weshalb auf die Angabe von Pseudocode verzichtet wird.
Es ist außerdem anzumerken, dass im Falle des obigen Algorithmus der BFS Code immer volle Länge hat, anders gesagt: Fall (2) tritt nie ein.