- rde. Betrachten Sie dazu bitte den darüberliegenden Punkt zum Einbinden von Gradle. Gegebenenfalls ist danach noch ein Neustart von IntelliJ zu machen, falls JavaFX neu installiert wurde. Sollte dies das Problem nicht lösen, wenden Sie sich bitte an das Forum, die Fragestunde oder installieren Sie JavaFX von selbst per Hand. Hierzu sei auf Tutorials im Internet verwiesen.

Allgemeine Hinweise zur Umsetzung

- Es werden häufig Klassendiagramme gezeigt, die umzusetzen sind. Die Funktionalität der einzelnen Funktionen wird dabei entweder in der Angabe oder in zur Verfügung gestellten Interfaces detailliert beschrieben. Dies ist jedoch nicht der Fall für die folgenden Funktionen. Diese sind nicht im Klassendiagramm vorhanden, aber dennoch "wie üblich" umzusetzen:
  - Getter-Methoden und Konstruktoren: Diese können von IntelliJ automatisch generiert werden.
  - `String toString()`: Diese Methode ist mit einer Ihrer Meinung nach geeigneten Ausprägung zu überschreiben.
- Einzelne Klassen sind in den Klassendiagrammen farbig dargestellt. Dabei wird folgende Konvention benutzt:
  - Graue Klasse und Interfaces werdden im Template zur Verfügung gestellt. Sie brauchen nicht erneut implementiert zu werden.
  - Gelbe Klassen und Interfaces wurden bereits in vorherigen Aufgabenteilen implementiert. Sie müssen nicht erneut implementiert werden.
- Implementieren Sie während des Programmierpraktikum niemals ohne explizite Aufforderung einen Sortieralgorithmus. Es wird weder in der Aufgabe, noch in der

Klausur verlangt werden, eine Sortierung selbst zu implementieren! Nutzen Sie stattdessen `Collections.sort`.
- Testen Sie ihren eigenen Code regelmäßig mithilfe von main-Funktionen oder selbstgeschriebenen Tests! Die PABS-Tests sollen Ihnen nur Rückmeldung geben, ob Ihr Programm korrekt ist. Sie sind *nicht* zur Fehlersuche geeignet!
- Die Lines of Code (LoC) Angaben sollen Ihnen als Orientierung dienen. Sie zählen nur "richtige" Codezeilen. Leerzeilen, Zeilen, die nur Klammern oder Import-Befehle enthalten, werden nicht mitgezählt. Achten Sie darauf, dass Ihre Lösung nicht wesentlich länger als die Musterlösung ist! Ein häufiger Fehler bei der PP-Bearbeitung ist es, möglichst viel Aufwand hinein zu stecken. Eine kurze, einfache und wartbare Lösung erleichtert die Programmierung und das Bestehen des Praktikums enorm!

Teil 1: Model

Ein Graph besteht aus Knoten und Kanten, die diese verbinden. Knoten können dabei beliebige Objekte sein (z.B. Straßenkreuzungen), Kanten können zusätzliche Informationen (z.B. ein Gewicht) enthalten. Da wir analog zu der Collections-API nicht vorgeben wollen, welche Art von Daten im Graph gespeichert sind, werden wir Generics benutzen, mit denen der Nutzer diese Entscheidung selbst treffen kann.

Wir verwenden dabei folgende Konventionen und Abkürzungen zur Benennung von Generics:

- **N** (Node): Der Datentyp der zu verwaltenden Knoten
- **A** (Annotation): Der Datentyp der Zusatzinformationen, die in den Kanten gespeichert sind
- **G** (Graph): Der genaue Datentyp eines Graphens. Unterklasse von `Graph<N, A>`
- **F** (Format): Datentyp eines Formates, etwa zu I/O-Zwecken
- **E** (Edge): Datentyp einer Kante, ausgeschrieben `Edge<N, A>`

Technisch können Graphen auf verschiedene Art und Weisen implementiert werden. Beispiele hierfür wären eine Adjazenzmatrix oder eine Map, die zu jedem Knoten dessen Kanten enthält. Beide Implementierungen haben Vor- und Nachteile. So unterstützt eine Adijazenzmatrix keine mehrfachen Kanten und benötigt viel Speicher - kann aber auch die Distanz zwischen zwei Knoten schneller zurückgeben. Es kann daher sinnvoll sein, verschiedene Implementierungen eines Graphen zu haben und je nach Anwendungsfall die beste Option auszuwählen. Aus diesem Grund benutzen wir Interfaces, die mehrere Implementierungen des gleichen Datentyps ermöglichen.

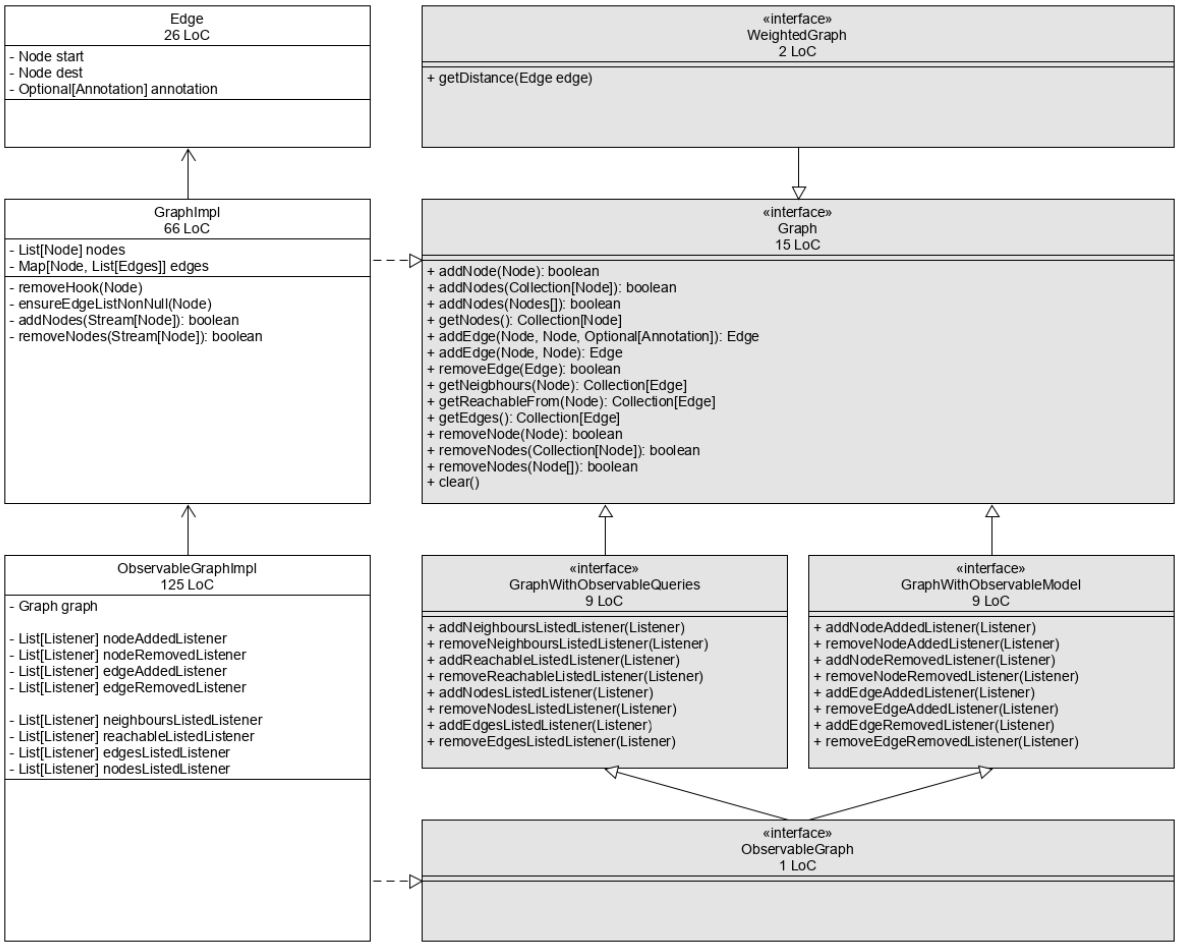
Um die Funktionalitäten eines Graphen und etwaige Use-Cases auf verschiedene Klassen zu verteilen, stehen 4 verschiedene Interfaces zur Verfügung:

- `Graph<N, A>`:
  Basisimplementierung eines Graphen
- `GraphWithObservableModel<N, A>`:
  Basisimplementierung, die zusätzlich Listener informiert, wenn sich etwas am Model geändert hat.
- `GraphWithObservableQueries<N, A>`:
  Basisimplementierung, die zusätzlich Listener informiert, wenn eine Anfrage an den Graphen stattgefunden hat.

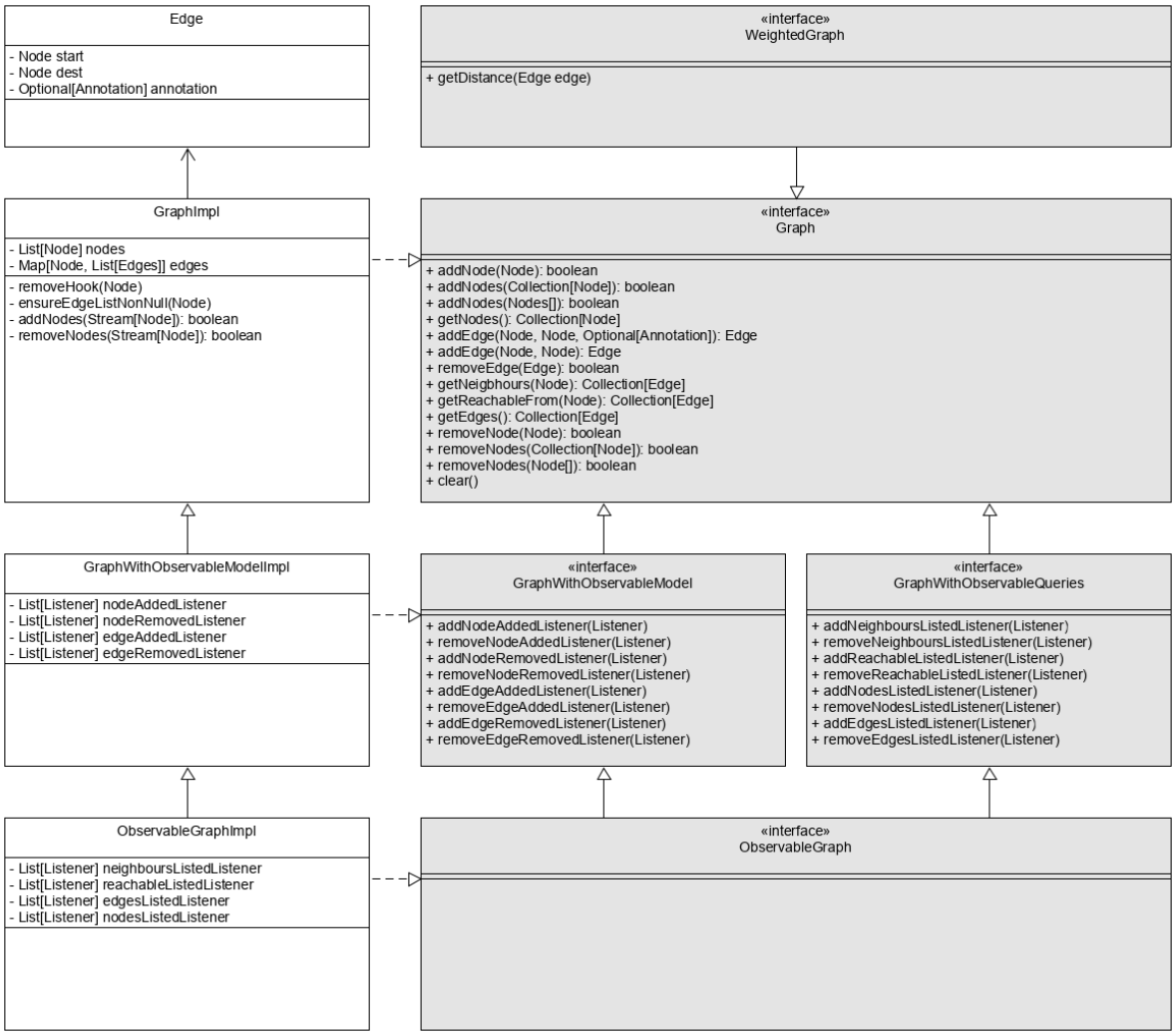
- `ObservableGraph<N, A>`:
  Basisimplementierung, die sowohl die Funktionalität von `GraphWithObservableModel<N, A>`, als auch von `GraphWithObservableQueries<N, A>` umsetzt.
- `WeightedGraph<N, A>`:
  Basisimplementierung, bei der zu jeder Kante mindestens das Gewicht gespeichert werden muss.

Teil 1 Aufgabe 1: Implementierung des Models

Implementieren sie das Model mithilfe der Funktiosbeschreibung in den bereits vorhandenen Interfaces. Sofern nicht anders angegeben sind alle Klassen im Paket `de.jpp.model` zu implementieren. Ihr Graph muss Schleifen (= Kante, bei der Start- und Zielknoten identisch sind) und Doppelkanten (= Zwei Kanten, die die gleichen Knoten verbinden) nicht unterstützen können. Für die Struktur können Sie beispielsweise folgendes Klassendiagramm verwenden. Diese setzt die Observer-Funktionalitäten mithilfe des Decorator-Patterns um, das Sie bereits aus der Vorlesung Softwaretechnik kennen sollten:



Es ist aber auch auf andere Art und Weisen möglich, die geforderte Funktionalität umzusetzen. Eine mit Vererbung implementierte Version könnte folgendermaßen aussehen:



Implementieren sie jedoch *auf keinen Fall* alle Funktionalitäten in einer Klasse. Dies führt zu großen Klassen, die fehleranfällig und schlecht wartbar sind! Achten Sie auch darauf - wo möglich - Codblöcke wiederzuverwenden. Schreiben Sie *auf keinen Fall* zwei unabhängige Implementierungen. Doppelt vorhandener Code ist ebenfalls fehleranfällig ("oh, da muss ich das ja nochmal ändern") und schlecht wartbar.

Einige Hinweise zum genauen Aussehen von Methoden:

- `de.jpp.model.graph.getNeighbours(Node)`, `de.jpp.model.graph.getReachableFrom(Node)`, `de.jpp.model.graph.getEdges()`, `de.jpp.model.graph.getNodes()`:
  Die zurückgegebene `Collection` soll **nicht** modifizierbar sein.
- `de.jpp.model.graph.removeEdge(Edge)`:
  Achten Sie darauf, dass das übergebene `Edge`-Objekt auch `null` sein kann und behandeln Sie diesen Fall entsprechend der Dokumentation.

Detailbeschreibung neu auftauchender Methoden:

- `de.jpp.model.Edge(Node, Node, Optional[Annotation])`: Der Konstruktor soll zuerst den Startknoten, dann den Zielknoten, dann eine ggf. vorhandene Annotation übergeben bekommen. `start` und `dest` dürfen nicht null sein. Das private Attribut `annotation` darf nach Aufruf des Konstruktors nicht `null`, sehr wohl aber das leere Optional sein. Vergessen Sie nicht, die `equals()` und `hashCode()`-Methoden zu überschreiben. Zwei `Edge`s sind gleich, wenn sie den selben Startknoten, Zielknoten und die selbe Annotation haben.
- `de.jpp.model.GraphImpl.removeHook(Node)`:
  Optionale Hilfsmethode die sicherstellt, dass alle Kanten gelöscht werden, die den übergebenen Knoten als Start oder Ziel enthalten.
- `mode.GraphImpl.ensureEdgeListNonNull(Node)`:
  Hilfsmethode die sicherstellt, dass der Aufruf von `edges.get(Node)` nicht `null` zurückliefert, sondern ggf. eine leere Liste.

Fügen sie anschließend folgende Funktionen zur Klasse `de.jpp.factory.GraphFactory` hinzu:

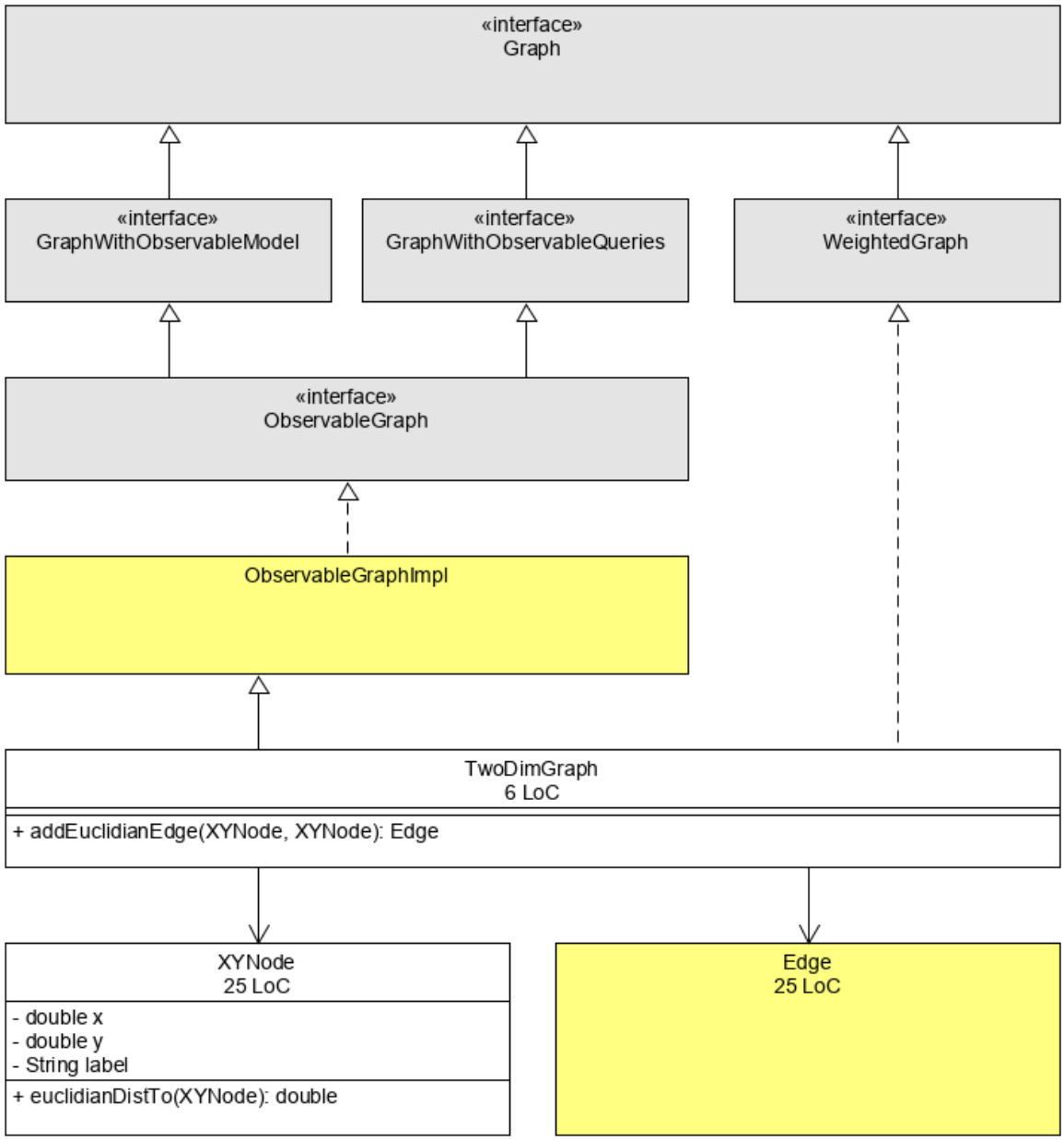
- `public static <N, A> Graph<N, A> createNewGraph()`:
  Erstellt einen neuen Graph. Dieser soll explizit *nicht* eine Instanz der Klasse ObservableGraph sein.
- `public static <N, A> ObservableGraph<N, A> createNewObservableGraph()`:
  Erstellt einen neuen ObservableGraph.

Hinweise :

- Die `<N, A>` aus der Funktionsdeklaration erlauben es Ihnen, eine Typdeklaration an den Konstruktor zu übergeben. Machen sie das nicht, zeigt IntelliJ eine Warning `UncheckedAssignment` an. Beim Aufruf der Funktion ist nichts weiter zu beachten, Sie können etwa folgendermaßen einen Graph erstellen, dessen Knoten Texte sind und deren Kante ein Gewicht (als Kommazahl) enthalten: `Graph<String, Double> graph = createNewGraph();`.
- Das Klassendiagramm zeigt nur schematische Typennamen, die keinen direkten Rückschlüss auf die Implementierung erlauben. Sie müssen sich selbst überlegen, wie sie die geforderten Typen mit Generics wählen, um die Anforderung zu erfüllen!
- Die Tests können erst ein sinnvolles Ergebnis anzeigen, nachdem die Factory-Methoden korrekt implementiert worden sind. Dies ist leider unvermeidbar, da Sie Teile der Klassen selbst vorgeben. Testen Sie ihren Code selbstständig!
- Vergessen Sie nicht, die `equals()` und `hashCode()`-Methoden für Ihre Graph-Implementierung (z.B. `GraphImpl`) zu überschreiben. Ansonsten können ihre Graphen nicht korrekt verglichen werden und die Testausgaben können abstrakt wirken. Achten Sie dabei besonders darauf, dass sie Vergleiche auf `Collection`s passend implementieren. Je nach verwendeter Klasse kann es nicht ausreichen, lediglich `equals()` auf diesen aufzurufen. Achten Sie dabei insbesondere auf den API-Vertrag)!
- Für die Klassen, die von Ihrer Graph-Implementierung erben, reicht die nun geschriebene `equals()`-Methode bereits aus. Sie muss - sofern nicht anders angegeben - nicht mehr überschrieben werden. Das gilt insbesondere auch für `ObservableGraphImpl`.

Teil 1 Aufgabe 2: Implementierung einer 2D-Ausprägung

Unser bisheriges Framework ist ziemlich allgemein gehalten. Es enthält keine Informationen zu Knoten oder Kanten. Wir wollen dies im Folgenden ändern und eine konkrete Ausprägung implementieren, die später graphisch angezeigt wird. Zu Programmieren ist ein zweidimensionaler Graph `TwoDimGraph`, der vom Typ `Graph<XYNode, Double>` ist und zusätzlich die Interfaces `WeightedGraph` und `ObservableGraph` implementiert. `XYNode` ist dabei wie im Klassendiagramm definiert. Jede Kante enthält nur ein Gewicht (gespeichert als `Double` als Annotation in den Kanten) und keine weiteren Informationen:



Detailbeschreibung neu auftauchender Methoden:

- `de.jpp.model.XYNode.XYNode(String, double, double)`:
  Der Konstruktor soll zuerst das Label, dann den x-Wert, dann den y-Wert übergeben bekommen. Das label darf nicht `null`, sehr wohl aber der leere String sein. Vergessen Sie nicht, die `equals()` und `hashCode()`-Methoden zu überschreiben. Zwei `XYNode`s sind gleich, wenn x und y Wert gleich sind und das Label ebenfalls.

- `de.jpp.model.XYNode.euclidianDistTo(XYNode other)`:
  Diese Funktion berechnet die euklidische Distanz zu einem anderen Knoten und gibt sie zurück.
- `de.jpp.model.TwoDimGraph.addEuclidianEdge(XYNode, XYNode)`:
  Diese Funktion fügt eine neue Kante in den Graphen ein, dessen Gewicht automatisch auf die euklidische Distanz der Knoten gesetzt wird.
  Wie bei der normalen Einfügeoperation sollen nicht vorhandene Knoten ggf. in den Graph aufgenommen werden.

Implementieren Sie die Klasse TwoDimGraph mit den oben beschriebenen und vom Interface geforderten Methoden. **Wichtig**: Damit das Template von Anfang an kompiliert ist `TwoDimGraph` als abstrakt gekennzeichnet. Entfernen Sie diesen Tag und implementieren Sie es als normale Klasse. Fügen Sie anschließend folgende Methode zur Klasse `de.jpp.factory.GraphFactory` hinzu:

- `public static TwoDimGraph createNewToDimGraph()`:
  Erstellt einen neuen und leeren TwoDimGraph.

Teil 2: Graphen erzeugen, speichern und lesen

In diesem Abschnitt sollen Eingabe/Ausgabe-Funktionalität implementiert werden, um unseren Graphen auf der Festplatte zu speichern. Wir benutzen dafür das `gxl` und das `dot` Format. Beispieldateien zum lokalen Testen finden sie in in diesem Archiv. Orientieren Sie sich an diesen und stellen Sie sicher, dass sie korrekt funktionieren. Sparen Sie sich auch, auf Eigenschaften einzugehen, die nicht benötigt werden wie beispielsweise ungerichtete Graphen.

Hierfür definieren wir zunächst zwei Schnittstellen: `GraphWriter` definiert Funktionalität, um einen Graphen `Graph<N, A>` in ein nutzerdefiniertes Format `F` umzuwandeln. Analog dazu ist ein `GraphReader` dafür zuständig, den Graph `Graph<N, A>` aus dem Dateiformat `F` zu rekonstruieren. Da beide benutzten Formate Stringbasiert sind, ist `F` in unserem Fall zunächst immer ein String. Es wäre aber auch etwa möglich, einen `OutputStream` bzw. `InputStream` als Zielformat zu verwenden.

Teil 2 Aufgabe 1: Grundlegende Ein- und Ausgabefunktionalität

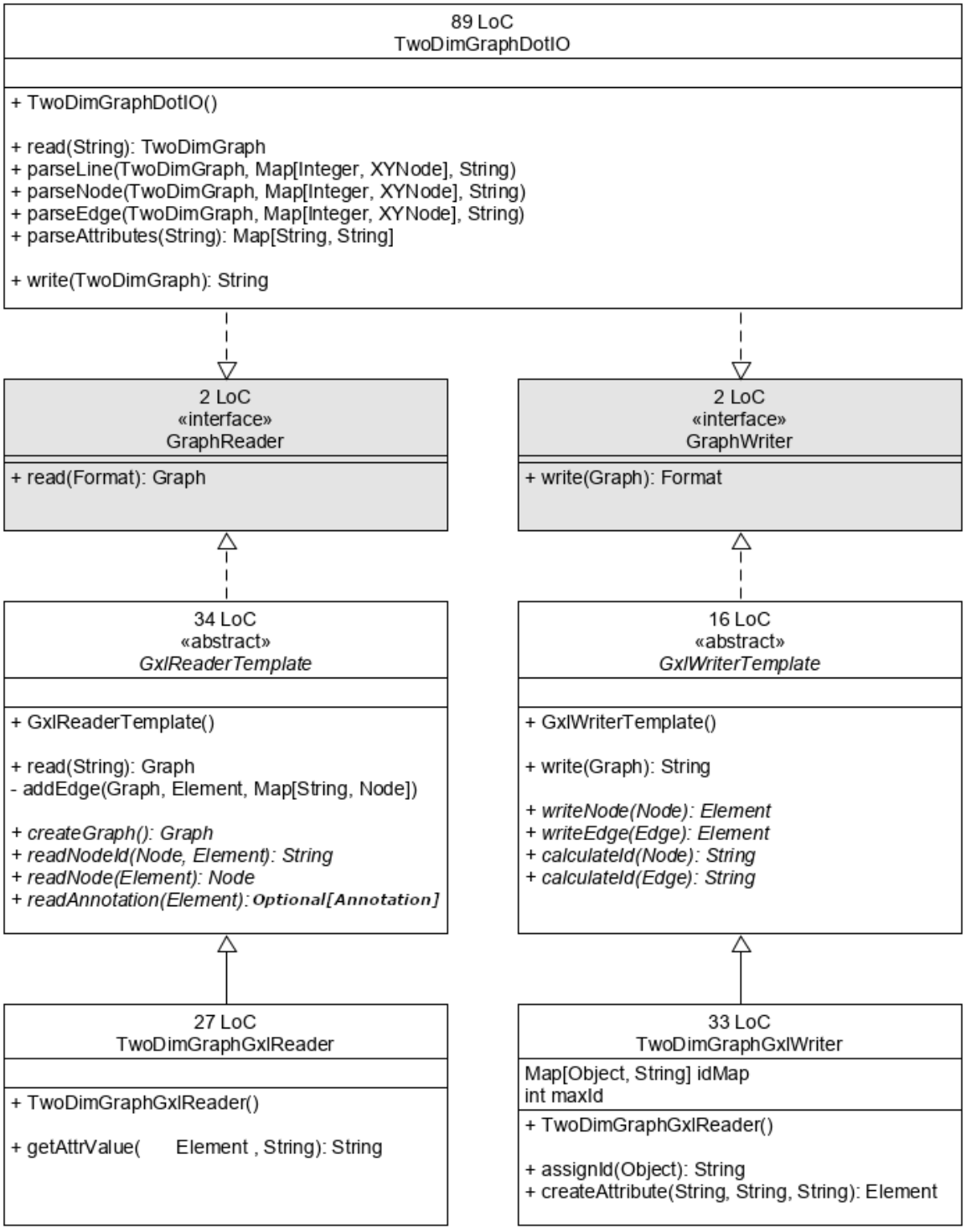
Wir wollen nun einen TwoDimGraph laden und speichern können. Stellen Sie hierfür folgende Methoden in der Klasse `de.jpp.factory.IOFactory` zur Verfügung:

- `public GraphReader<XYNode, Double, TwoDimGraph, String> getTwoDimGxlReader()`:
  Erstellt einen GraphReader, der einen TwoDimGraph aus einem String im gxl-Format rekonstruiert.
- `public GraphReader<XYNode, Double, TwoDimGraph, String> getTwoDimDotReader()`:
  Erstellt einen GraphReader, der einen TwoDimGraph aus einem String im dot-Format rekonstruiert.
- `public GraphWriter<XYNode, Double, TwoDimGraph, String> getTwoDimGxlWriter()`:

Erstellt einen GraphWriter, der einen TwoDimGraph in einen String im gxl-Format umwandelt.

- `public GraphWriter<XYNode, Double, TwoDimGraph, String> getTwoDimDotWriter()`:
  Erstellt einen GraphWriter, der einen TwoDimGraph in einen String im dot-Format umwandelt.

Folgendes Klassendiagramm könnte dabei herauskommen. Sie können natürlich aber auch eine eigene Klassenstruktur entwerfen:



Erklärung zusätzlich auftauchender Methoden aus der Klasse `de.jpp.io.TwoDimGraphDotIO`:

- `parseLine`:
  Fügt das Element der Zeile zum Graphen hinzu
- `parseNode`:
  Fügt das Element der Zeile zum Graphen hinzu, falls es sich um eine Zeile handelt, die einen Knoten definiert
- `parseEdge`:
  Fügt das Element der Zeile zum Graphen hinzu, falls es sich um eine Zeile handelt, die eine Kante definiert
- `parseAttributes`:
  Liest alle `key=value`-Werte die in `[ ]` der übergebenen Zeile ein und gibt sie als Map zurück.

Hinweis: Benutzen Sie geeignete Reguläre Ausdrücke um die Zeilen zu parsen. Nur mit `String.split` werden Sie nicht sehr weit kommen!

`gxl` ist eine Untersprache von `XML`, daher können Sie sich auf der Suche nach Tutorials auch darauf fokussieren und müssen nicht speziell nach `gxl` suchen. Die Methoden benutzten dabei die `org.jdom2`-API, die sie ebenfalls verwenden dürfen. PABS ist entsprechend eingerichtet, sodass diese Bibliothek zur Verfügung steht. Die API ist sehr umfangreich, lesen Sie sich daher bevor sie mit der Implementierung beginnen in die entsprechenden Abschnitte ein, die sie benötigen, um zu vermeiden, dass sie später alles wieder ändern müssen.

Erklärung zusätzlich auftauchender Methoden aus der Klasse `de.jpp.io.GxlReaderTemplate`:

- `abstract readNodeId`:
  Liest die ID des Knoten aus der GXL-Darstellung des Knoten (übergeben als `org.jdom2.Element`) aus. In den Beispielen des `TwoDimGraph` ist die ID im GXL-Element als Attribut "id" gespeichert und kann von dort ausgelesen werden.
- `abstract readNode(Element)`:
  Erzeugt das entsprechende Knotenobjekt aus dem `org.jdom2.Element` Objekt. Im Falle des `TwoDimGraph` werden hier x- und y-Koordinate des Knoten ausgelesen und das entsprechende Objekt damit erstellt. Sollte es ein Label geben, so befindet sich dieses im `description`-Attribut.
- `abstract readAnnotation(Element)`:
  Erzeugt die entsprechende Annotationa us dem `org.jdom2.Element` Objekt. Im Falle des `TwoDimGraph` werden hier die Kosten der Kante ausgelesen und zurückgegeben.
- `abstract createGraph`:
  Erzeugt einen neuen, leeren Graphen in den die neuen Objekte hinzugefügt werden können. Dies muss abstrakt sein, da wir noch nicht wissen, welchen Typ von Graphen wir erzeugen. Im Falle des `TwoDimGraph` wird ein leerer `TwoDimGraph` erstellt.

Erklärung zusätzlich auftauchender Methoden aus der Klasse `de.jpp.io.GxlWriterTemplate`:

- `abstract writeNode`:
  Erstellt aus dem übergebenen Knoten ein `org.jdom2.Element`-Objekt.
- `abstract writeEdge`:
  Erstellt aus der übergebenen Kante ein `org.jdom2.Element`-Objekt.
- `abstract calculateId`:
  Weist dem übergebenen Knoten / der übergebenen Kante eine einzigartige ID zu. Das ist notwendig, da die Kanten nur die ID der Knotenobjekte als Start- bzw. Endpunkt speichern sollen. Im einfachsten Fall fangen sie mit der ID `1` an und weisen den Objekten nacheinander höhere Zahlen als IDs zu.

Erklärung weiterer Elemente:

- `de.jpp.io.TwoDimGraphGxlReader`:
  Liest ein bestimmtes Attribut aus einer Liste an Attributelementen aus.
  Beispiel: `getAttrValue("<edge from="id2" id="id19" to="id1">", "from")` => `id2`
- `de.jpp.io.TwoDimGraphGxlWriter.createAttribute`:
  Erstellt ein `org.jdom2.Element`-Objekt, das ein Attribut speichert.
  Beispiel: `createAttribute("x", "float", "1.0")` => `"<attr name = "x"> <float> 1.0 </float> </attr>"`
- `de.jpp.io.TwoDimGraphGxlWriter.idMap` (Attribut):
  Speichert die ID jedes Objektes (Knoten und Kante), um ggf. später wieder darauf zugreifen zu können.

Ihre Parser sollten die Beispieldateien erfolgreich verarbeiten können. Testen Sie dies ausführlich, bevor sie auf PABS committen! Auch die Tests arbeiten im wesentlichen mit diesen Dateien. Für das Einlesen der Dateien sind folgende Hinweise zu beachten:

- Implementieren Sie die Klasse ParseException, so dass diese die entsprechenden Exceptions mit den übergebenen Parametern erzeugt. Verwenden Sie den `super(...)`-Konstruktor und schauen Sie sich an, wie Exceptions in Java typischerweise implementiert werden.
- Enthält ein Knoten keine x-Koordinate, so soll eine ParseException geworfen werden.
- Enthält ein Knoten keine y-Koordinate, so soll eine ParseException geworfen werden.
- Enthält ein Knoten kein Label, so soll KEINE ParseException geworfen werden.
- Enthält eine Kante kein Gewicht, so soll eine ParseException geworfen werden.
- Ist Start- oder Endknoten einer Kante nicht spezifiziert oder vorhanden, so soll eine ParseException geworfen werden.
- Erfüllt das Dokument nicht die vorgegebene Formatspezifikation(gxl oder dot), so soll eine ParseException geworfen werden. *Hinweis*: Ein `<attr>`-Tag, der keine Kinder hat verletzt auch die `gxl`-Spezifizierung.

Teil 2 Aufgabe 2: Erweiterte Ein- und Ausgabefunktionalität

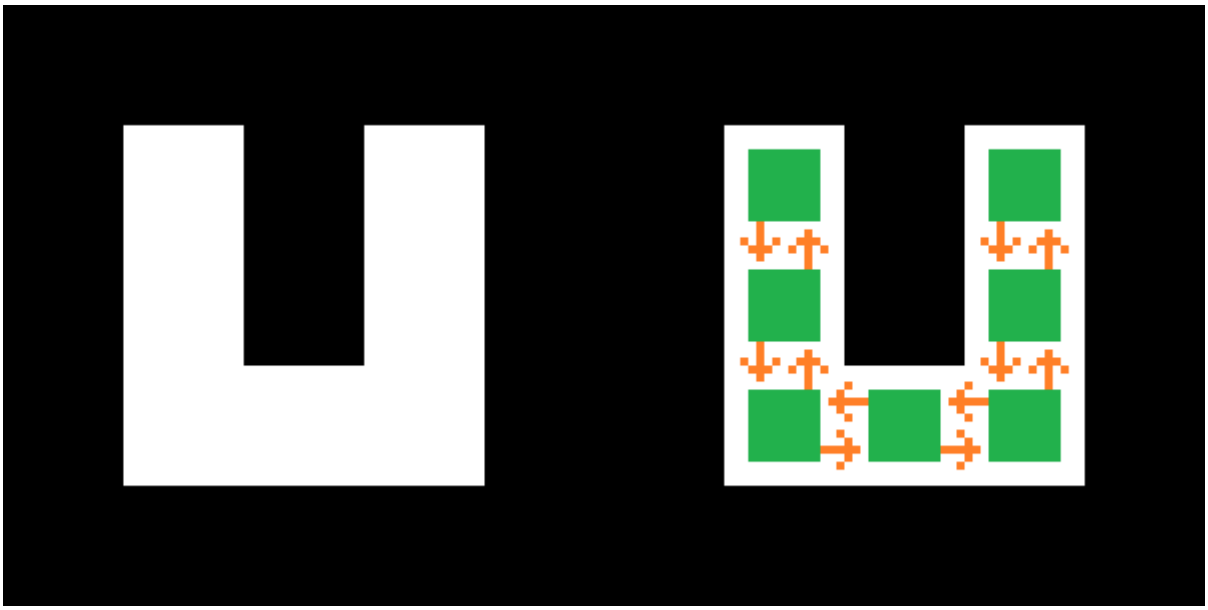
Wenn Sie die Unterteilung in Template und konkrete Klasse ebenfalls umsetzen, ist eine Erweiterung des Schemas durch Erzeugung neuer Unterklassen sehr einfach. Um das konkreter nachvollziehen zu können, wollen wir im folgenden einen weiteren Graphtypen im GXL-Format einlesen: Einen `Graph<String, Map<String, String>>`. Die Knoten dieses Graphen sind einfache Labels (Texte), jede Kante kann beliebig viele Eigenschaften als key-value-Paar in einer Map speichern.

Im Paket `de.jpp.model` finden Sie bereits die Klasse `LabelMapGraph`, diese können Sie verwenden. Entfernen Sie dazu den `abstract`-Tag und erweiteren Sie `GraphImpl<String, Map<String, String>>`. Das kann im folgendem viel Schreibarbeit bei den Typen sparen. Konsultieren Sie zur Implementierung auch wieder die Beispiele. Diese sehen teilweise sehr ähnlich zu den "normalen" `gxl`-Dateien aus, achten Sie also darauf, dass Sie nur die relevanten Dinge parsen. Parsen sie Attribute auch wenn sie nicht vom Typ `String` sind und fügen Sie sie als `String` ein. Beim Schreiben müssen Sie ebenfalls nicht auf die Typen achten und können alles als `String` in die Struktur einbringen.
Stellen Sie insgesamt anschließend mindestens die folgende Methoden in der Klasse `de.jpp.factory.IOFactory` zusätzlich zur Verfügung:

- `public GraphReader<String, Map<String, String>, LabelMapGraph> getLabelMapGraphGxlReader()`:
  Erstellt einen GraphReader, der einen LabelMapGraph aus einem String im gxl-Format rekonstruiert.
- `public GraphWriter<String, Map<String, String>, LabelMapGraph> getLabelMapGraphGxlWriter()`:
  Erstellt einen GraphWriter, der einen LabelMapGraph in einen String im gxl-Format umwandelt. Stellen Sie außerdem die folgende Methode in der Klasse `de.jpp.factory.GraphFactory` zur Verfügung:
- `public LabelMapGraph createNewLabelMapGraph()`:
  Erstellt einen neuen und leeren LabelMapGraph.

Teil 2 Aufgabe 3: Einlesen von Bildern

Neben den etablierten Graph-Formaten wollen wir auch Bilder von Labyrinthen als Graphen interpretieren können. In einem Bild interpretieren wir alle dunklen Pixel als Mauer und alle Hellen als Weg. Zwei Wegzellen sind mit Distanz 1 verbunden, wenn sie benachbart sind. Aus dem linken 5x5 Pixelbild entsteht also folgender Graph:



Dabei sind Knoten im rechten Bild grün gezeichnet, Kanten mit Gewicht 1 in Orange. Die Knoten haben die 2D-Position (1|1), (1|2), (1|3), (2|3), (3|3), (3|2), und (3|1). Ein Knoten zählt dabei als "dunkel" bzw. Wand, wenn der HSB-Helligkeitswert eines Pixel (echt) kleiner als

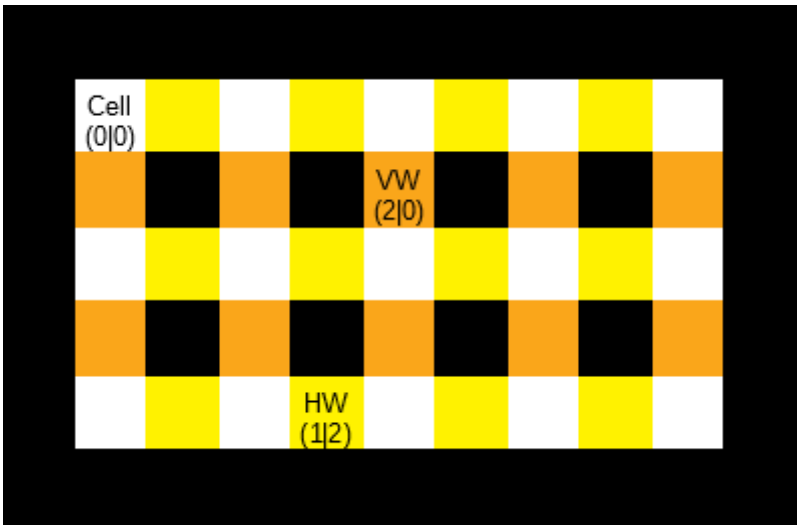
0,5 ist. Bei guter API-Arbeit ist nicht notwendig, den Code zur Umwandlung von Farbformaten selbst zu schreiben!

Stellen sie eine weitere Factory-Methode in `de.jpp.factory.IOFactory` zur Verfügung, die ein Bild (gegeben als `BufferedImage`) als TwoDimGraph einliest:

- `public GraphReader<XYNode, Double, TwoDimGraph, BufferedImage> getTwoDimImgReader()`:
  Erstellt einen GraphReader, der einen TwoDimGraph aus einem pixelbasiertem BufferedImage rekonstruiert.

Teil 2 Aufgabe 4: Generieren von Labyrinthen

In diesem Abschnitt wollen wir verschiedene Labyrinthe zufällig erstellen lassen, an denen wir später unsere Suche testen können. Wir verwenden dabei das Interface `de.jpp.de.jpp.maze.Maze` als Grundlage. Diese soll Funktionen bündeln, um ein Labyrinth zu erstellen.
Dabei werden die Zellen (`Cell`), die horizontalen (`HWall` oder `HW`) und die vertikalen Wände (`VWall` oder `VW`) jeweils mit einem eigenen Koordinatensystem angesprochen. Das kann man sich graphisch folgendermaßen vorstellen:



Das Labyrinth hat also die Größe 5x3 und 4x3 horizontale sowie 5x2 vertikale Wände. Wegen dem immer gegebenen Randes des Labyrinths liegt die Zelle (0|0) auf Pixelposition (1|1).

Stellen Sie in der Klasse `de.jpp.factory.MazeFactory` folgende Funktionen bereit. Achten Sie dabei darauf, dass zwei `Maze`s per `equals()` und `hashCode()` miteinander vergleichbar sind, wenn sie das gleiche `Maze` repräsentieren:

- `public Maze getEmptyMaze(int width, int height)`:
  Erstellt ein neues, leeres Labyrinth mit der angegebenen Breite & Höhe
- `public GraphReader<XYNode, Double, TwoDimGraph, Maze> getMazeReader()`:
  Erstellt einen GraphReader, der einen TwoDimGraph aus einem Maze-Objekt erstellt.
- `public BufferedImage getMazeAsImage(Maze maze)`:
  Gibt das übergebene Labyrinth als Bild zurück. Wände sollen dabei als schwarz (RGB #000), Wege als weiß (RGB #FFF) dargestellt werden.
  Sie können sich an dem obigen Bild orientieren. Erstellen Sie ein Bild der Breite

`width`*2+1 und der Höhe `height`*2+1. Nun können Sie eine aktive Mauer als schwarzen Pixel und eine nicht gesetzte Mauer als weißen Pixel an die Stellen der orangenen/gelben Felder setzen.

- `public Maze getRandomMaze(Random ran, int width, int height)`: Gibt ein neues, zufällig erstelltes Labyrinth mit der angegebenen Breite & Höhe zurück. Benutzen Sie dafür den unten angegebenen Algorithmus:

```
UnterteileLabyrinth(Labyrinth):

  Wenn das Labyrinth nur aus einen Gang besteht:
    Breche den Algorithmus ab

  Wenn das Labyrinth breiter als hoch ist:
    Unterteile das Labyrinth horizontal

  Ansonsten, wenn das Labyrinth höher als breit ist:
    Unterteile das Labyrinth vertikal

  Ansonsten:
    Wähle einen zufälligen Wahrheitswert
    Ist dieser wahr, unterteile das Labyrinth horizontal, sonst vertikal


UnterteileLabyrinthHorizontal(Labyrinth):

  rnd1 = Zufallszahl zwischen 0 und width-1, wobei width die aktuelle
Breite des Labyrinthes ist.
  rnd2 = Zufallszahl zwischen 0 und height-1, wobei height die aktuelle
Höhe des Labyrinthes ist.
  Setze alle horizontalen Wände an den Positionen (rnd1,_) außer
(rnd1,rnd2) auf aktiv.
  Unterteile das Labyrinth erneut (erst den linken Teil (bezogen auf die
Mauer), dann den rechten)

UnterteileLabyrinthVertikal(Labyrinth):

  rnd1 = Zufallszahl zwischen 0 und height-1, wobei height die aktuelle
Höhe des Labyrinthes ist
  rnd2 = Zufallszahl zwischen 0 und width-1, wobei width die aktuelle
Breite des Labyrinthes ist.
  Setze alle vertikalen Wände an den Positionen (_, rnd1) außer (rnd2,
rnd1) auf aktiv.
  Unterteile das Labyrinth erneut (erst den oberen Teil (bezogen auf die
Mauer), dann den unteren)
```

Hinweise:

- Die Aufgabenstellung ist direkt in Programmcode umwandelbar. Das _ ist eine Wildcard, die für "alle Werte im möglichen Bereich" steht. Diese Pseudocode erzeugt rekursiv ein Labyrinth. Die -1 sind notwendig, damit bei der Labyrintherstellung kein Bereich abgetrennt wird.
- Eine so definierte horizontale Unterteilung setzt viele horizontale Wände, sorgt also für eine **vertikale** Linie im Bild.
- Die Zufallszahlen sind ausschließlich aus dem übergebenen Zufallszahlengenerator und **exakt** in oben beschriebener Reihenfolge und Anzahl zu übernehmen!

- Bitte verwenden Sie für die Generierung von zufälligen Ganzzahlen **ausschließlich** die Methode `nextInt(int)` mit einem Parameter! Andernfalls können die Tests nicht bestanden werden.

Teil 3: Graphensuche

In diesem Abschnitt sollen 4 Suchalgorithmen (Tiefensuche, Breitensuche, Dijkstra und A*-Suche) auf dem eben geschriebenen Graphen umgesetzt werden. Zur genaueren Erklärung der Algorithmen können sie sich auch an der Vorlesung *(Grundlagen der) Algorithmen und Datenstrukturen* orientieren, falls Sie diese besucht haben. Da alle Algorithmen das gleiche Problem lösen (eine Wegsuche, ausgehend von einem Startknoten) kann das Strategy-Pattern verwendet werden. Die Schnittstelle zur konkreten Strategie wird im Interface `SearchAlgorithm` definiert.

Mithilfe der `SearchStopStrategy` ist es möglich, den Abbruch der Suche zu steuern. Ausprägungen dieses Interfaces werden informiert, sobald der Weg zu einem Knoten abschließend gefunden wurde (= dieser geschlossen wurde). Gibt die Funktion `boolean stopSearch(Node)` true zurück, soll der Algorithmus die Suche beenden. Das soll vermeiden, dass Rechenzeit für unnötige Informationen aufgewendet wird. `SearchStopStrategy` ist dabei ein funktionales Interface, dessen Ausprägungen auf drei unterschiedliche Art und Weisen (Lambda, Anonyme Klasse, normale Klasse) implementiert werden können. Die folgenden Funktionen der Klasse `de.jpp.factory.SearchStopFactory` sollen folgende Ausprägungen erstellen:

- `public <N> SearchStopStrategy<N> maxNodeCount(int maxCount)`:
  Diese Ausprägung soll den Algorithmus solange ausführen, bis die übergebene Anzahl an Knoten geschlossen wurde.
  Implementieren Sie diese Ausprägung mithilfe einer anonymen Klasse!
- `public <N> SearchStopStrategy<N> expandAllNodes()`:
  Diese Ausprägung soll den Algorithmus nie stoppen.
  Implementieren Sie diese Ausprägung mithilfe eines Lambdas!
- `public <N> StartToDestStrategy<N> startToDest(N dest)`:
  Diese Ausprägung stoppt den Algorithmus, sobald der Weg zum Knoten `dest` gefunden wurde.
  Implementieren Sie diese Ausprägung mithilfe einer Klasse `de.jpp.algorithm.StartToDestStrategy`.

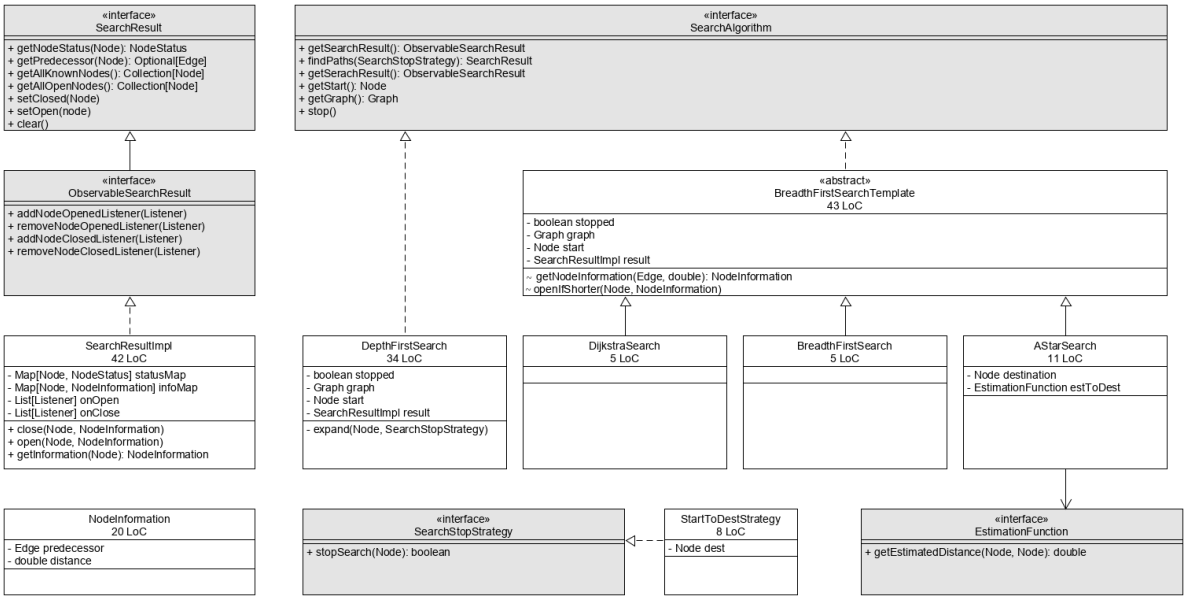
Die von Ihnen implementierten konkreten Ausprägungen der Suche sollen durch die Klasse `de.jpp.model.factory.SearchFactory<N, A>` mit folgenden Methoden erstellt werden können:

- `public <G extends Graph<N, A>> SearchAlgorithm<N, A, G> getDepthFirstSearch(G graph, N start)`:
  Erstellt einen Suchalgorithmus, der eine Tiefensuche auf einem beliebigen Graph durchführt.
  Hinweis: Ein Knoten gilt hier als geschlossen, sobald er geöffnet und alle seine Nachbarn geöffnet wurden. Um dieses Verhalten korrekt implementieren zu können, **müssen** Sie die Tiefensuche iterativ implementieren.
- `public <G extends Graph<N, A>> SearchAlgorithm<N, A, G> getBreadthFirstSearch(G graph, N start)`:

Erstellt einen Suchalgorithmus, der eine Breitensuche auf einem beliebigen Graph durchführt.

- `public <G extends WeightedGraph<N, A>> SearchAlgorithm<N, A, G> getDijkstra(G graph, N start)`:
  Erstellt einen Suchalgorithmus, der eine Dijkstra-Suche auf einem gewichteten Graphen durchführt.
  Hinweis: Damit das Interface `SearchStopStrategy` sinnvoll genutzt werden und Suchen rechtzeitig abgebrochen werden können, **müssen** Sie Dijkstra iterativ implementieren.
- `public <G extends WeightedGraph<N, A>> SearchAlgorithm<N, A, G> getAStar(G graph, N start, N dest, EstimationFunction<N> estToDest)`:
  Erstellt einen Suchalgorithmus, der eine A*-Suche auf einem gewichteten Graphen durchführt.

Man beachte, dass die A*-Suche nur für eine Wegfindung zu einem Ziel definiert ist! Die dort auftauchende Funktion `estToDest` wird dabei als Schätzfunktion zum Ziel verwendet und in der Literatur (oder auf Wikipedia) häufig mit `h` für "Heuristik" bezeichnet. Sie können sich bei ihrer Implementierung von folgendem Klassendiagramm inspirieren lassen. Aus Gründen der Übersichtlichkeit sind nicht alle Nutzungspfeile eingezeichnet:



Es folgen einige Erklärungen zu Methoden und Klassen im Klassendiagramm:

- Das Enum `NodeStatus` speichert Literale für den Status eines Knotens.
- Die Klasse `NodeInformation` ist eine reine Datenklasse. Sie soll in `SearchResult` einem Knoten zugeordnet werden und für diesen immer den aktuellen Vorgänger auf einem kürzesten Weg vom Startknoten und das Gewicht des gesamten, kürzesten Weges zu ihm. Beispiel: Der kürzeste Weg ist A -> B -> C -> D wobei jede Kante das Gewicht 3 hat. Dann enthält das zu D passende `NodeInformation`-Objekt die Kante C -> D als Vorgängerkante und 9 als Gewicht.
  Implementieren Sie zusätzlich zum Klassendiagramm entsprechende Getter und Setter.
- `public void open(N node, NodeInformation<N, A> information)` in der Klasse `de.jpp.algorithm.SearchResultImpl`:

Diese Methode setzt den Status von `node` auf `NodeStatus.OPEN` und assoziiert `information` mit dem Knoten `node`.

- `public void close(N node, NodeInformation<N, A> information)` in der Klasse `de.jpp.algorithm.SearchResultImpl`:
  Diese Methode setzt den Status von `node` auf `NodeStatus.CLOSED` und assoziiert `information` mit dem Knoten `node`.
- `public NodeInformation<N, A> getInformation(N node)` in der Klasse `de.jpp.algorithm.SearchResultImpl`:
  Gibt die aktuell zu diesem Knoten gespeicherten Informationen zurück (falls vorhanden).
- `public void stop()` in der Klasse `de.jpp.algorithm.interfaces.SearchAlgorithm`:
  Beendet den aktuellen Suchvorgang. Alle Knoten, die noch nicht abgeschlossen (`NodeStatus.CLOSED`) wurden werden wieder auf unbekannt (`NodeStatus.UNKNOWN`) gesetzt.
- `NodeInformation<N, A> getNodeInformation(Edge<N, A> edge, double weight)` in der Klasse `de.jpp.algorithm.BreadthFirstSearchTemplate`:
  Erzeugt ein neues `NodeInformation`-Objekt mit den gegebenen Parametern.
- `void openIfShorter(N node, NodeInformation<N, A> info)` in der Klasse `de.jpp.algorithm.BreadthFirstSearchTemplate`:
  Diese Methode überprüft, ob die übergebene `info` für den Knoten `node` besser ist als die aktuell gespeicherte. Besser heißt, der Pfad muss kürzer sein. Falls dies der Fall ist, wird das übergebene `info`-Objekt anstelle des alten mit dem Knoten assoziiert. Weiterhin muss der Knoten danach den `NodeStatus.OPEN` haben. Dieser muss auch **erneut** gesetzt werden, falls der Knoten diesen bereits vorher hatte, um eventuelle Updates weiterzupropagieren (z.B. an eine GUI).

Viel Erfolg!