Berner
Fachhochschule

Anforderungsdokumentation Projekt 1: easyGraph

Autoren: Kaspar Engel, Mathias Wenger, Lukas Weber

Betreuer: Dr. Peter Schwab

Inhaltsverzeichnis

1 Vorwort

Das vorliegende Dokument beschreibt die geplante Funktionsweise der zu entwickelnden Applikation „easyGraph“. Diese Software wird im Rahmen des Moduls „Projekt 1“ an der BFH zum Thema „Visualisieren von Graphen und Graphen-Algorithmen“ im Auftrag von Dr. Peter Schwab erstellt.

1.1 Änderungsverwaltung

Dieses Kapitel dient der Übersicht und Kontrolle über Änderungen am vorliegenden Dokument.

Datum	Version	Autor	Kommentar
29.10.2014	0.1	Mathias Wenger	Initiale Version erstellt
21.11.2014	0.2	Alle	Anpassen der Kapitel 1 - 3
12.12.2014	0.3	Alle	Kapitel 1-3 fertiggestellt
15.12.2014	0.4	Mathias Wenger	Kapitel zur Projekt Dokumentation hinzugefügt
16.01.2015	1.0	Alle	Alle Kapitel überarbeitet, ergänzt und fertiggestellt. Abschlusskontrolle durchgeführt.

2 Einleitung

2.1 Zweck des Dokuments

Das Dokument dient der Spezifikation und Auflistung der Anforderungen der Software easyGraph.

2.2 Ausgangslage

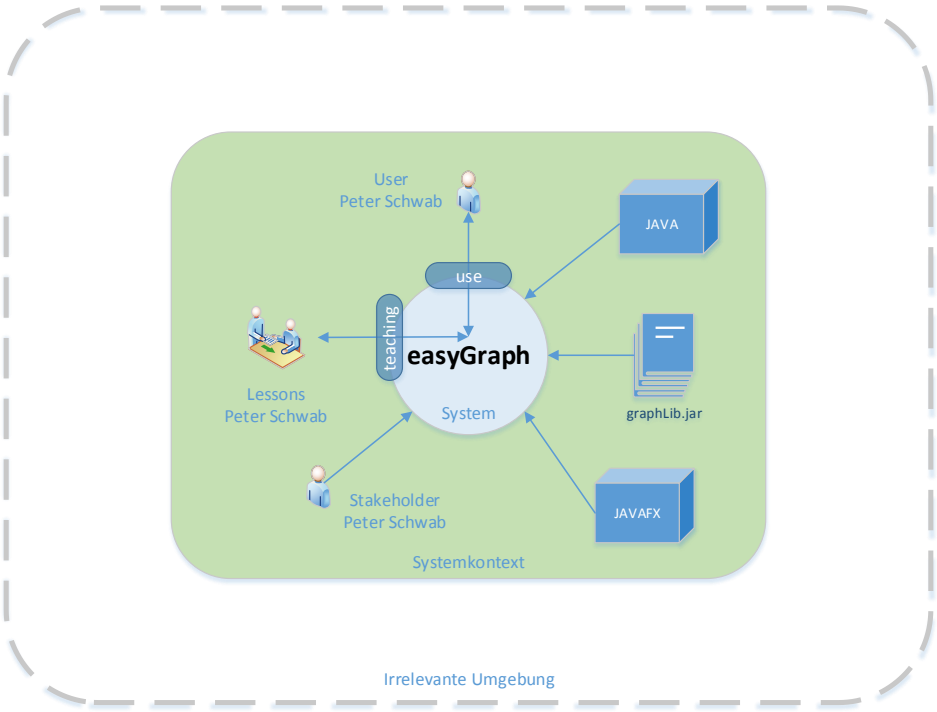
Dr. Peter Schwab benötigt eine Visualisierung seiner Graphen und Graphen-Algorithmen für den Unterricht im Modul BTI7062, Algorithmen und Datenstrukturen. Die Programmlogik wird in der Form einer Library von Dr. Peter Schwab bereitgestellt. Die Library graphLib.jar beinhaltet alle notwendigen Strukturen sowie Funktionen, um die im Unterricht behandelten Themengebiete zu erläutern. Jedoch fehlt eine ansprechende und korrekte Visualisierung der Graphen und Graphen-Algorithmen.

2.3 Anwendungsbereich

Das Ergebnis des Projekts soll eine einfach bedienbare, übersichtliche Applikation sein, welche die Benutzer (Dr. Peter Schwab oder Studenten, die das Modul besuchen) in ihrer Funktion unterstützt und das Erlernen der notwendigen Kenntnisse im Modul BTI7062 Algorithmen und Datenstrukturen vereinfacht.

2.4 System und Systemkontext

Die folgende Darstellung stellt das System und dessen Systemkontext dar.



3 Glossar

Begriff	Bedeutung
System	Das System ist die zu entwickelnde Applikation easyGraph.
Graphen	Ein Graph besteht aus einer Menge von Knoten und einer Menge von Kanten, welche immer zwei Knoten verbinden.
Algorithmus	Unter einem Algorithmus versteht man die Umschreibung der schrittweisen Lösung eines Problems mittels einer endlichen Anzahl von Regeln.
Inzidenz	Inzidenz bezeichnet eine Beziehung zwischen Knoten und Kanten in einem ungerichteten Graphen. Ein Knoten heißt in einem ungerichteten Graph inzident mit einer Kante, wenn er von dieser Kante berührt wird, das heißt, wenn diese ihn enthält.
Kante	Eine Kante ist ein Element der Kantenmenge eines Graphen.
Knoten	Als Knoten oder Ecke bezeichnet man ein Element der Knotenmenge eines Graphen.
Benutzer	Die Benutzer von easyGraph sind entweder Dr. Peter Schwab oder Studenten des Unterrichts Modul BTI7062 Algorithmen bei Dr. Peter Schwab.

4 Projekt Ziele

Folgende Projektziele sind definiert und bestimmen den gewünschten Endzustand des Projekts.

4.1 Hauptziel

Eine Visualisierungssoftware für Graphen und Graphen-Algorithmen.

4.2 Unterziele

1. easyGraph soll Graphen darstellen und editieren.
2. Bereitstellen von Funktionsschnittstellen, um bestimmte Einzelschritte im Ablauf von Graphen-Algorithmen in der Graphen-Visualisierung zu markieren.
3. easyGraph stellt für didaktische Zwecke vordefinierte Graphen als Beispiele bereit.
4. Die Studenten des Moduls Algorithmen und Datenstrukturen können easyGraph benutzen.

4.3 Randbedingungen

- Verwendung der bestehenden Klassen und Algorithmen aus der graphLib von Dr. Peter Schwab
- Für die Entwicklung von easyGraph soll der Source Code der graphLib möglichst wenig oder gar nicht verändert werden
- Die Anwendung von easyGraph bedingt keine Änderung am Source Code von easyGraph
- Die Programmiersprache der Applikation ist Java

5 Anforderungskatalog

Die anschließenden Anforderungen bestimmen die Eigenschaften und Funktionen von easyGraph.

5.1 Anforderungsübersicht Funktionale Anforderungen

Legende: **P**riorität, **V**ariabilität, **K**omplexität, **R**isiko = ((1xP+2xV+3xK)/6) **Skala:** 1 = niedrig, 2 = mittel, 3 = hoch

Nr.	Kurzbezeichnung	Status	P	V	K	R	Quelle	Datum	Ziele
1	**Modifizieren von Knoten, Kanten und Inzidenz Graphen**								
1.1	Hinzufügen von Knoten	Erledigt	3	2	2	2.2	Schwab (KO)	10.09.14	1
1.2	Löschen von Knoten	Erledigt	3	1	1	1.3	Schwab (KO)	10.09.14	1
1.3	Verschieben von Knoten	Erledigt	3	1	2	1.8	Schwab (KO)	10.09.14	1
1.4	Verschieben von Knoten mit angehängten Kanten	Erledigt	3	2	3	2.7	Schwab (KO)	10.09.14	1
1.5	Hinzufügen von Kanten	Erledigt	3	2	2	2.2	Schwab (KO)	10.09.14	1
1.6	Löschen von Kanten	Erledigt	3	1	1	1.3	Schwab (KO)	10.09.14	1
1.7	Löschen von Inzidenz-Graphen	Erledigt	3	1	1	1.3	Schwab (KO)	10.09.14	1
1.8	Modifizieren von Knoten, Kanten oder Inzidenz-Graphen Parametern	Erledigt	2	2	1	1.5	Schwab (1.S)	13.09.14	1
2	**Speichern und Laden**								
2.1	Speichern von Inzidenz-Graphen	Erledigt	1	1	1	1.0	Schwab (2.S)	20.09.14	3
2.2	Laden von Inzidenz-Graphen	Erledigt	1	1	1	1.0	Schwab (2.S)	20.09.14	3
2.3	Bereitstellen von vordefinierten Inzidenz-Graphen Beispielen	Erledigt	1	2	1	1.3	Schwab (2.S)	20.09.14	3
3	**Visualisierung**								
3.1	Visualisierung von Graphen-Algorithmen	Erledigt	3	3	3	3.0	Schwab (KO)	10.09.14	2
3.2	Darstellung von Graphen-Algorithmen Ablaufschritten	Erledigt	3	3	3	3.0	Schwab (KO)	10.09.14	2
3.3	Automatisches Anordnen von Inzidenz-Graphen ohne Koordinaten	Erledigt	1	2	3	2.3	Schwab (2.S)	20.09.14	2
4	**Benutzeroberfläche**								
4.1	Anzeigen von Knoten, Kanten oder Inzidenz-Graphen Parametern	Erledigt	2	2	1	1.5	Schwab (KO)	10.09.14	4
4.2	Anzeigen der Graphen-Algorithmen aus der graphLib.jar	Erledigt	3	2	2	2.2	Schwab (KO)	10.09.14	4
4.3	Bereitstellen von Bedienelementen für den Benutzer	Erledigt	3	2	1	1.7	Schwab (1.S)	13.09.14	4

5.2 Anforderungsübersicht nicht-funktionale Anforderungen

Legende: **P**riorität, **V**ariabilität, **K**omplexität, **R**isiko = ((1xP+2xV+3xK)/6) **Skala:** 1 = niedrig, 2 = mittel, 3 = hoch

Nr.		Kurzbezeichnung	Status	P	V	K	R	Quelle	Datum	Ziele
5		**Performance**								
	5.1	Inzidenz-Graphen bis zu einer Grösse von maximal 20 Knoten und 50 Kanten darstellen	Erledigt	2	1	1	1.2	Schwab (2.S)	20.09.14	2
	5.2	Gleiche Berechnungsgenauigkeit wie die graphLib.jar	Erledigt	3	1	2	1.8	Schwab (KO)	10.09.14	2
6		**Ergonomie**								
	6.1	Die Benutzeroberfläche ist in Englisch verfasst	Erledigt	1	1	1	1.0	Schwab (2.S)	20.09.14	4
	6.2	Die minimale Auflösung der Benutzeroberflache ist mindestens 600x400 Pixel	Erledigt	1	1	1	1.0	Schwab (2.S)	20.09.14	4
	6.3	Bedienelemente besitzen Icons	Erledigt	1	1	1	1.0	Schwab (2.S)	20.09.14	4
	6.4	Bedienelemente verfügen über Tastenkombinationen	Erledigt	1	1	1	1.0	Schwab (2.S)	20.09.14	4

Legende: **P**riorität, **V**ariabilität, **K**omplexität, **R**isiko = ((1xP+2xV+3xK)/6) **Skala:** 1 = niedrig, 2 = mittel, 3 = hoch

5.3 Beschreibung der Benutzer

Die Benutzer von easyGraph sind entweder Dr. Peter Schwab oder Studenten des Unterrichts Modul BTI7062 Algorithmen bei Dr. Peter Schwab. Die Benutzer bringen somit entsprechende und gezielte Vorkenntnisse mit und kennen den Aufbau und die Funktionen der von easyGraph zu Grunde liegende graphLip.jar. Dies erleichtert uns, easyGraph benutzerfreundlich und einfach zu gestalten. Der Benutzer kann nach Bedarf die von easyGraph angebotenen Visualisierung Funktionen in einem Algorithmus verwenden oder auch nicht.

5.4 Detailbeschreibung der Anforderungen

1 Modifizieren von Knoten, Kanten und Inzidenz-Graphen

1.1 Hinzufügen von Knoten

Das System muss dem Benutzer die Möglichkeit bieten, zusätzliche Knoten zu einem bestehenden oder leeren Inzidenz-Graphen hinzuzufügen

1.2 Löschen von Knoten

Das System muss dem Benutzer die Möglichkeit bieten, einen ausgewählten Knoten zu löschen.

1.3 Verschieben von Knoten

Das System muss dem Benutzer die Möglichkeit bieten, einen ausgewählten Knoten zu verschieben.

1.4 Verschieben von Knoten mit angehängten Kanten

Das System muss fähig sein, Knoten mit angehängten Kanten zu verschieben.

1.5 Hinzufügen von Kanten

Das System muss dem Benutzer die Möglichkeit bieten, eine Kante zwischen 2 bereits bestehenden Knoten hinzufügen.

1.6 Löschen von Kanten

Das System muss dem Benutzer die Möglichkeit bieten, eine ausgewählte Kante zu löschen.

1.7 Löschen von Inzidenz Graphen

Das System muss dem Benutzer die Möglichkeit bieten, den angezeigten Inzidenz-Graphen zu löschen.

1.8 Modifizieren von Knoten-, Kanten- oder Inzidenz-Graphen-Eigenschaften

Das System muss dem Benutzer die Möglichkeit bieten, ausgewählte Knoten-, Kanten- oder Inzidenz-Graphen-Eigenschaften zu modifizieren.

2 Speichern und Laden

2.1 Speichern von Inzidenz-Graphen

Das System muss dem Benutzer die Möglichkeit bieten, Inzidenz-Graphen aus dem Programmspeicher in einem vordefinierten Format in ein Filesystem zu speichern.

2.2 Laden von Inzidenz-Graphen

Das System wird dem Benutzer die Möglichkeit bieten, Inzidenz-Graphen vom Filesystem, dass in einem vordefinierten Format vorliegt, in den Programmspeicher zu laden.

2.3 Bereitstellen von vordefinierten Inzidenz-Graphen-Beispielen

Das System wird vordefinierte Beispiele von Inzidenz-Graphen bereitstellen.

3 Visualisierung

3.1 Visualisierung von Graphen Algorithmen

Das System wird fähig sein, Graphen-Algorithmen anhand eines Inzidenz-Graphen zu visualisieren.

3.2 Darstellung von Graphen-Algorithmen-Ablaufschritten

Das System muss fähig sein, einzelne Ablaufschritte der Graphen-Algorithmen automatisch oder manuell darzustellen.

3.3 Automatisches anordnen von Inzidenz-Graphen ohne Koordinaten

Besitzt ein Inzidenz-Graph keine Koordinaten, wird das System fähig sein, den Graph für die Visualisierung automatisch anzuordnen.

4 Benutzeroberfläche

4.1 Anzeigen von Knoten, Kanten oder Inzidenz-Graphen-Eigenschaften

Das System wird Knoten-, Kanten- oder Inzidenz-Graphen-Eigenschaften anzeigen.

4.2 Anzeigen der Graphen-Algorithmen aus der graphLib.jar

Das System muss fähig sein, alle programmierten Graphen-Algorithmen aus der graphLib.jar anzuzeigen.

4.3 Bereitstellen von Bedienelementen für den Benutzer

Das System wird Bedienelemente für die folgenden Funktionen bereitstellen: Steuerung der Visualisierung, Speichern und Laden von Inzidenz-Graphen und Modifizieren von Knoten-, Kanten- und Inzidenz-Graphen-Eigenschaften.

5 Performance

5.1 Inzidenz-Graphen bis zu einer Grösse von maximal 20 Knoten und 50 Kanten darstellen

Das System muss fähig sein, Inzidenz-Graphen bis zu einer Grösse von 20 Knoten und 50 Kanten darzustellen.

5.2 Gleiche Berechnungsgenauigkeit wie die graphLib.jar

Zu jedem Zeitpunkt muss das System die Graphen-Algorithmen-Berechnungen aus der graphLib.jar in der gleichen Reihenfolge und Genauigkeit visualisieren.

6 Ergonomie

6.1 Die Benutzeroberfläche ist in Englisch verfasst

Das System wird die Benutzeroberfläche und alle darin enthaltenen Textelemente in Englisch darstellen.

6.2 Die minimale Auflösung der Benutzeroberflache ist mindestens 600x400 Pixel

Das System muss fähig sein, eine Auflösung der Benutzeroberfläche von mindesten 600 x 400 Pixel (Breite x Höhe) anzubieten.

6.3 Bedienelemente besitzen Icons

Das System wird zur einfacheren Illustration der Funktionen passende Icons der Bedienelemente in der Benutzeroberfläche enthalten.

6.4 Bedienelemente verfügen über Tastenkombination

Das System wird Tastenkombination zu jedem der Bedienelemente in der Benutzeroberfläche bereitstellen.

6 easyGraph

6.1 Systemübersicht

Bei der Realisierung von easyGraph wurde darauf geachtet, dass die als „best practice“ bezeichneten Programmierregeln umgesetzt werden. So wurden beispielsweise die Klassen sinngemäss in Pakete unterteilt. Besonderes Augenmerk erhielt die Trennung des GUI-Layers vom Logik-Layer. JavaFX war dabei sehr hilfreich, da das Layout in XML-Dateien ausgelagert wird. Das Model war bereits durch die bestehende GraphLib vorgegeben und konnte dank des bestehenden Decorator-Patterns einfach erweitert werden. Weiter haben wird alle GUI-Events gebündelt und lassen sie durch eine State-Machine abarbeiten. Dabei werden nur diejenigen Events verarbeitet, welche im betroffenen Status auch einen Effekt haben sollen. Mit diesen Massnahmen ist es uns gelungen, die Komplexität zu verringern und den Code systematisch zu ordnen.

Die Schnittstelle zur GraphLib wurde so klein und einfach wie möglich gehalten. Einerseits sollten keine Anpassungen an der GraphLib nötig werden (was aufgrund der starken Kapselung von graphLib nicht ganz geklappt hat) und andererseits soll sich die Verwendung von easyGraph so einfach wie möglich gestalten.

Eine detaillierte Übersicht über unser System bieten die Klassendiagramme unter Abschnitt 10.2 und 10.3.

6.2 Technologien

Folgende Produkte, Software oder Technologien wurden für die Realisation von easyGraph verwendet:

- Java 8 Update 25
- JavaFX 8 mit SceneBuilder
- Eclipse Luna Service Release 1 (4.4.1)
- JavaFX Plugin “e(fx)eclipse“
- Github als Projekt Repository

6.3 Projekt Meilensteine

Die nachfolgenden Entscheidungen oder Meilensteine wurden im Laufe der Realisation getroffen oder erreicht:

1. Bestimmen der zu verwenden Technologien
    - JavaFX 8 hat unser Interesse geweckt, da es viele neue interessante Funktionen und Erweiterungen bietet. Da wir bereits Projekte mit Swing oder AWT durchgeführt haben, möchten wir mit JavaFX 8 neue Erfahrungen sammeln.
    - Für die Entwicklungsumgebung wurde Eclipse Luna (4.4.1) verwendet. Zusätzlich diente das JavaFX Plugin “e(fx)eclipse“ und der SceneBuilder als hilfreiche Erweiterung für JavaFX 8.

2. Analyse der graphLib.jar
    - Zum besseren Verständnis schauten wir uns die graphLib genau an und erstellten zur einfacheren Darstellung ein Klassendiagramm (-> Anhang 10.1).

3. Aufruf von easyGraph
    - Um eine möglichst einfache Anbindung an graphLib anbieten zu können, beschlossen wir eine statische Klasse als Interface zu erstellen. Diese Klasse nannten wir EasyGraph. Wir erweiterten das Klassendiagramm von graphLib um unsere Vorstellungen, wie EasyGraph mit graphLib zusammenarbeiten sollte (-> Anhang 10.1).

4. Einarbeiten in JavaFX 8
    - Um einen ersten Eindruck von JavaFX 8 zu erhalten, haben wir folgendes Tutorial durchgearbeitet: http://code.makery.ch/java/javafx-8-tutorial-intro/

5. Definition des Layouts und Funktionen des Benutzerinterfaces
    - Grundsätzlich ist die Benutzeroberfläche von easyGraph in 3 Teile unterteilt: Menuleiste, Toolbox sowie Arbeitsfläche
        - Die Menuleiste bietet unter anderem Funktionen zum Speichern, Laden, Erstellen von Graphen und eine Auswahl von vordefinierten Graphen Beispielen an.
        - Die Toolbox bietet verschiedene Buttons für die Bedienung von easyGraph: Verschieben und Hinzufügen von Kanten und Knoten, Auswahl des gewünschten Algorithmus, sowie Bedienelemente zur Steuerung während der Algorithmus-Visualisierung (Debug-Modus).
        - Auf der Arbeitsfläche wird der Graph dargestellt und kann mit Hilfe der Toolbox Elemente bearbeitet werden.

6. Schnittstellen zu graphLib.jar definieren
    - Damit wir das Problem lösen, Algorithmen als solche zu erkennen, lösen konnten, entschieden wir uns zur Verwendung von Annotations. Diese bieten einerseits die Funktionalität, dass mittels Java Reflection die annotierten Methoden erkannt werden können. Andererseits können Default Werte innerhalb der Annotation-Deklaration hinterlegt werden. Dadurch muss nicht zwingend jede mit einer Annotation versehene Methodendeklaration konfiguriert werden, sondern nur dann, wenn sie die Default-Werte explizit überschreiben will.

7. Integration des Datenmodells von graphLib
    - Damit Daten nicht mehrfach redundant gespeichert werden, was nur die Komplexität unnötig aufgeblasen hätte, haben wir uns entschieden, das Datenmodell von graphLib vollständig zu übernehmen und keine eigenen Implementationen von Graphen, Vertices und Edges zu erstellen.
    - Dafür mussten wir kleinere Anpassungen am graphLib Code vornehmen und haben die Eigenschaften, welche auf ein Objekt via dem „Decorable“ Interface gespeichert werden können, aus der privaten Klasse „GraphExamples“ in eine öffentliche Klasse „Property“ ausgelagert.

8. Umsetzung von MVC
    - Zur besseren Trennung von Logik- und UI-Elementen haben wir uns entschieden, das MVC-Prinzip so gut wie möglich umzusetzen:
        - Wie beschrieben, haben wir das Datenmodell von graphLib adaptiert, insbesondere die folgenden Modelklassen „Graph“, „Vertex“ und „Edge“.
        - Zur Erstellung der UI’s benutzten wir den SceneBuilder, welcher die UI-Definitionen in sogenannten FXML-Files persistiert. Diese sind sehr schlank, einfach zu lesen und machen die Verwendung von Java-Layout-Managern überflüssig.
        - Die Controller implementieren die Business-Logik und sind mit der JavaFX-spezifischen Annotation „@FXML“ versehen. Dadurch werden sie von JavaFX beim Starten der Applikation als Controller zu den zugehörigen Views initialisiert.

9. Eventhandling und State Machine
    - Je mehr Events wir auslösten, desto mehr Eventhandler mussten registriert werden. Dies führte einerseits zu ziemlich unübersichtlichem Code. Andererseits entstanden Probleme, wenn zu einem Event, z.B. einem MouseClickEvent, mehrere Aktionen möglich waren.
    - Wir entschieden uns, dass ab diesem Zeitpunkt die Funktionalität der Applikation kaum oder nur noch unter mühsamen Umständen weiterentwickelt werden kann, was Events angeht. Deshalb entschieden wir uns dazu, eine State Machine zu implementieren, um die verschiedenen Zustände zu handeln, welche die Applikation einnehmen kann.
    - Ausserdem implementierten wir eigene Events, so dass eine daraus resultierende Aktion eindeutig ist. Beispielsweise unterteilten wir einen ClickEvent in Left- und RightClickEvents, diese wiederum in Events, abhängig vom Ursprung (z.B. PaneLeftClickEvent, VertexRightClickEvent, ForwardEvent, etc).
    - Durch die Kombination der State Machine und der spezifischen Events konnten wir nun Statusübergänge verwenden, welche nur noch diejenigen Events abarbeiteten, welche sie betrafen.

10. Timer-Problem mit Threads
    - Nachdem wir einen Algorithmus dank den Annotations und Reflection erfolgreich aufrufen konnten, traten Probleme mit den verschiedenen Threads auf, und zwar konnte das schrittweise Darstellen des Algorithmus nicht mehr unterbrochen werden, weil dessen Thread das UI fortlaufend aktualisierte. Mehrfaches Klicken auf den Bedienelementen liess die Applikation sogar abstürzen.
    - Wir fanden die Lösung schliesslich in der Verwendung der Java Klasse „TimerTask“ respektive „Timer“: Diese ermöglichten es uns, einen Timer so laufen zu lassen, dass er zu einem definierten Intervall jeweils einen neuen TimerTask laufen lässt, welcher dann die Graphendarstellung gemäss Algorithmen-Abarbeitung aktualisiert, sofern kein Abbruch oder sonstige Handlung via UI gemacht wurde. Dadurch konnten wir die Steuerung via UI wieder weiterentwickeln.

11. Manipulation der Kanten und Knoten mit Hilfe eines Kontextmenus
    - Damit Manipulationen an einem Objekt vorgenommen werden können (Setzen eines Vertex als Startvertex, Löschen eines Vertex / Edge, oder Editieren von Objekteigenschaften), mussten wir ein Kontextmenu implementieren, damit wir den Kontext des zu bearbeitenden Objektes kannten. Wir orientierten uns dazu am wohlbekannten Kontextmenu von Windows, welches per Rechtsklick auf ein Objekt geöffnet wird.

12. Speicherung von Graphen
    - Zur angenehmeren Anwendung wollten wir Graphen persistieren können und entschieden uns der Einfachheit halber, dies durch Serialisierbarkeit von Objekten umzusetzen. Dabei trat das Problem auf, dass nicht alle verwendeten Klassen von graphLib persistierbar waren, weil sie das „Serializable“-Interface nicht implementierten. Herr Schwab half uns, den Code von graphLib so anzupassen, dass die Objekte von graphLib vollständig serialisierbar waren. Dadurch konnten wir die Funktionen zum Speichern und Laden von Graphen fertig implementieren.

7 Weiterentwicklung des Systems

Aufgrund der begrenzten Zeit konnten leider nicht alle Ideen im Verlaufe des Projekts umgesetzt werden. Mögliche Erweiterungen des Systems könnten wie folgt sein:

- Speichern von Graphen in anderem Format (XML, JSON, …)
- Verbesserter Mechanismus für die automatische Graphen-Anordnung anbieten
- Die Konfiguration der Applikation in ein Konfigurations-File auslagern, anstatt in einer Java-Klasse abzulegen
- Zoom-Funktion für die Arbeitsfläche zum angenehmeren Arbeiten mit grösseren Graphen
- Integration der Applikation in den Browser
- Erstellen einer ersten finalen Version und zur angenehmeren Verwendung in ein jar-File exportieren

8 Installation von easyGraph

8.1 Vorbedingungen:

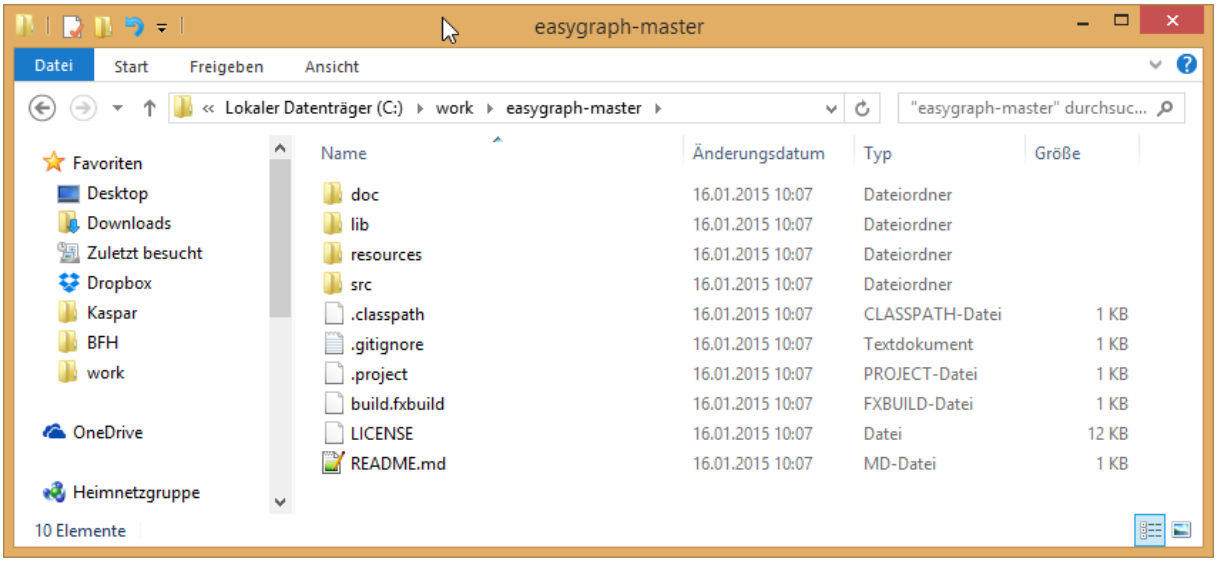
1. Java Development Kit 8 (liefert JavaFX 8 mit)
2. Eclipse in einer Version 4.3+
3. Eclipse Plugin „e(fx)clipse“ zur Erstellung von JavaFX Projekten

8.2 Projekt Importieren:

1. Ein Workspace Verzeichnis für Eclipse anlegen, z.B. C:\work\workspaces\EasyGraphWorkspace\

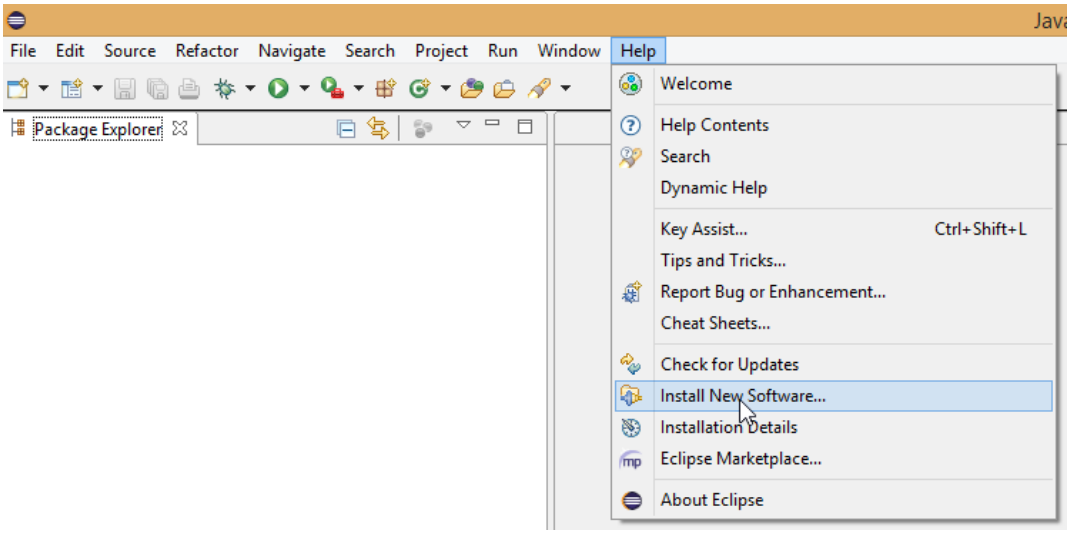
2. Source Code von easyGraph von Github herunterladen (als ZIP) https://github.com/engek1/easygraph

3. Sourcen ins Workspace Verzeichnis extrahieren. Wichtig: Sourcen sollen nicht im Workspace liegen, sondern in einem separaten Projektordner! C:\work\easygraph-master

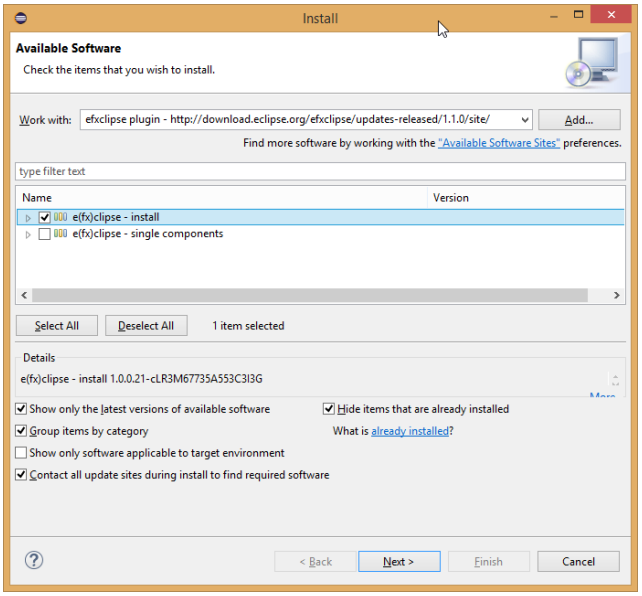


4. Eclipse starten mit den oben angelegten Workspace.

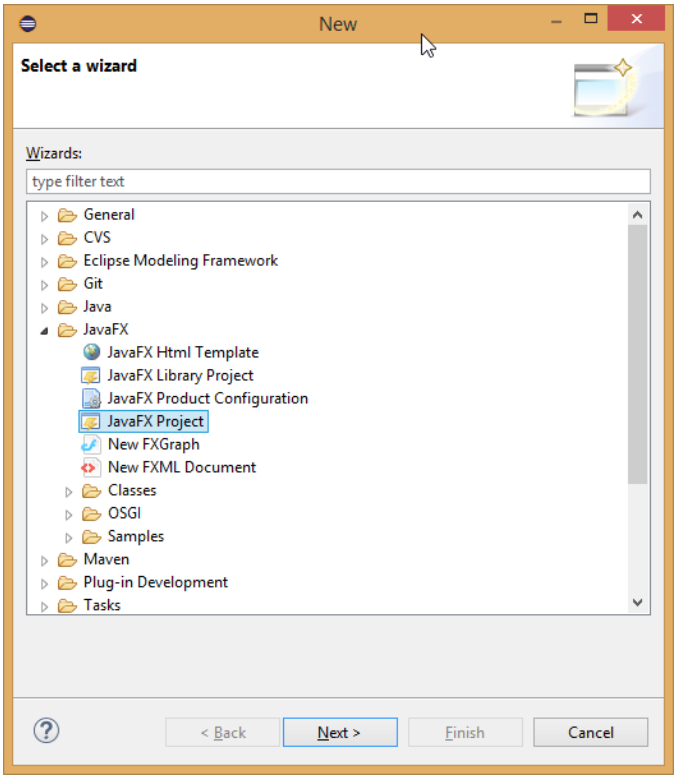
5. Neue Software installieren:

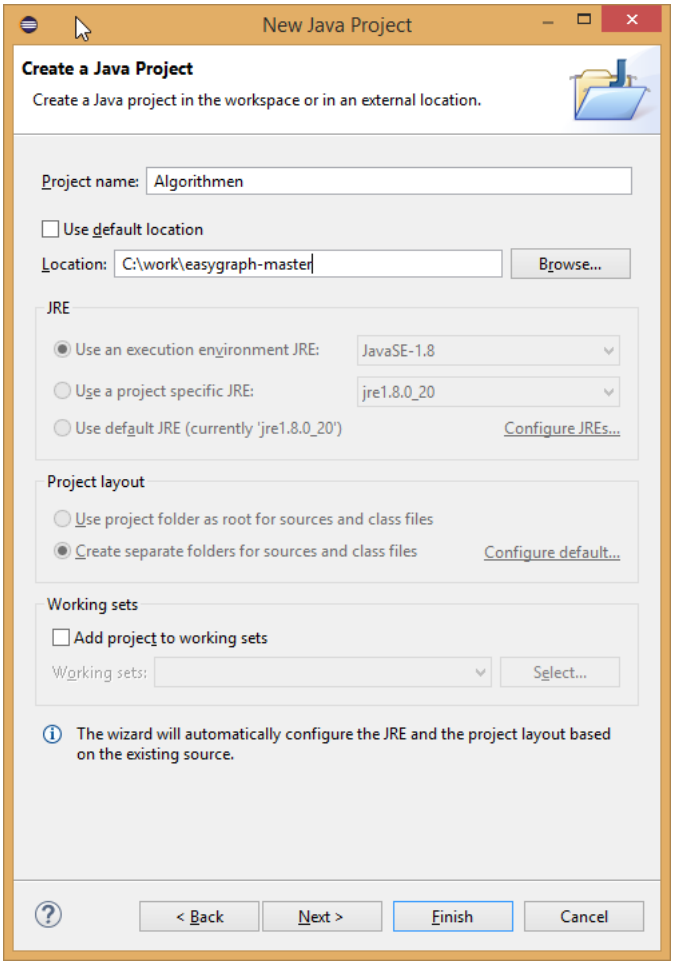


6. Update Site hinzufügen: http://download.eclipse.org/efxclipse/updates-released/1.1.0/site/



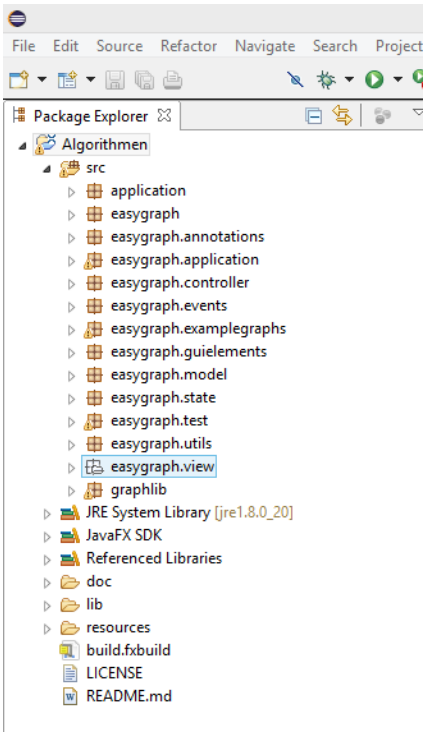
7. Java FX Plugin installieren: Neues Java FX Projekt erstellen: File -> new -> other





Wichtig: unter Location muss der Ordner mit den Sourcen von easyGraph ausgewählt werden. C:\work\easygraph-master

8. Jetzt erscheint das Projekt im Package-Explorer.



8.3 Algorithmen hinzufügen:

1. Algorithmen hinzufügen: Fügen sie die Algorithmen der Klasse graphlib.GraphExamples hinzu. Sie können auch eine andere Klasse verwenden, dafür muss aber die Konfiguration ALGORITHM_CLASS_NAME in der Klasse easygraph.utils.Config angepasst werden.

2. Algorithmen markieren: Markieren sie die Klasse mit den Algorithmen mit der Annotation @AlgorithmClazz und die Methoden, welche die Algorithmen enthalten, mit @AlgorithmMethod.

GraphExamples.java | Config.java

```
1  package graphlib;
2
3  import java.util.ArrayList;
10
11 @AlgorithmClazz
12 public class GraphExamples<V, E> {
13
14     @AlgorithmMethod
15     public void kruskal(Graph<V, E> g) {
16         // gives the Object MSF to each
17         // edge belonging to an minimal spanning forest
18
19         // create clusters, put the vertex in it
20         // and assign them to the vertice
21         Iterator<Vertex<V>> it = g.vertices();
22         while (it.hasNext()) {
23             Vertex<V> v = it.next();
24             ArrayList<Vertex<V>> cluster = new ArrayList<>();
25             cluster.add(v);
26             v.set(Property.CLUSTER, cluster);
27         }
28         PriorityQueue<Double, Edge<E>> pq = new HeapPriorityQueue<>();
29         Iterator<Edge<E>> eIt = g.edges();
30         while (eIt.hasNext()) {
31             Edge<E> e = eIt.next();
32             double weight = 1.0;
33             if (e.has(Property.WEIGHT))
34                 weight = (Double) e.get(Property.WEIGHT);
35             pq.insert(weight, e);
36         }
```

8.4 easyGraph API verwenden

Sie können alle Methoden, welche von easyGraph angeboten werden, statisch aufrufen.

```
// neighbours:
Iterator<Edge<E>> eIt;
EasyGraph.setSelected(u);
if (u.has(Property.VISITED)) EasyGraph.setSelected((Edge)u.get(Property.VISITED));
if (g.isDirected())
    eIt = g.incidentInEdges(u); // backwards!
```

Funktionen welche angeboten werden:

- setDiscovered(Edge<?>) : void
- setDiscovered(Edge<?>, Color) : void
- setDiscovered(Vertex<?>) : void
- setDiscovered(Vertex<?>, Color) : void
- setSelected(Edge<?>) : void
- setSelected(Edge<?>, Color) : void
- setSelected(Vertex<?>) : void
- setSelected(Vertex<?>, Color) : void
- setDisabled(Edge<?>) : void
- setDisabled(Edge<?>, Color) : void
- setDisabled(Vertex<?>) : void
- setDisabled(Vertex<?>, Color) : void
- launchGui() : void
- launchGui(Graph<?, ?>) : void

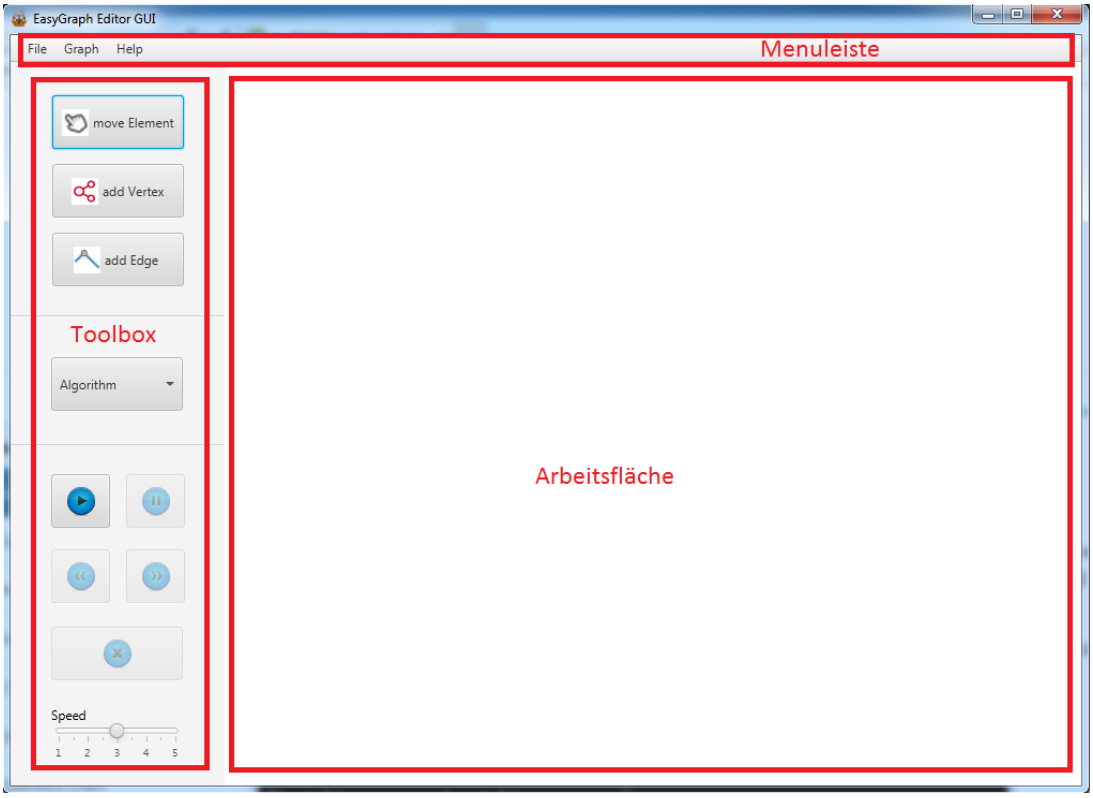
8.5 Programm starten

- Die Klasse easygraph.test.TestLauncher startet das Programm. (Rechtsklick -> run as)
- Oder eine eigene Launcher-Klasse verwenden:
    - EasyGraph.launchGui() aufrufen.
    - EasyGraph.launchGui(Graph<?, ?>) aufrufen, wenn ein eigener Graph verwendet werden soll.

9 Handbuch EasyGraph Editor GUI

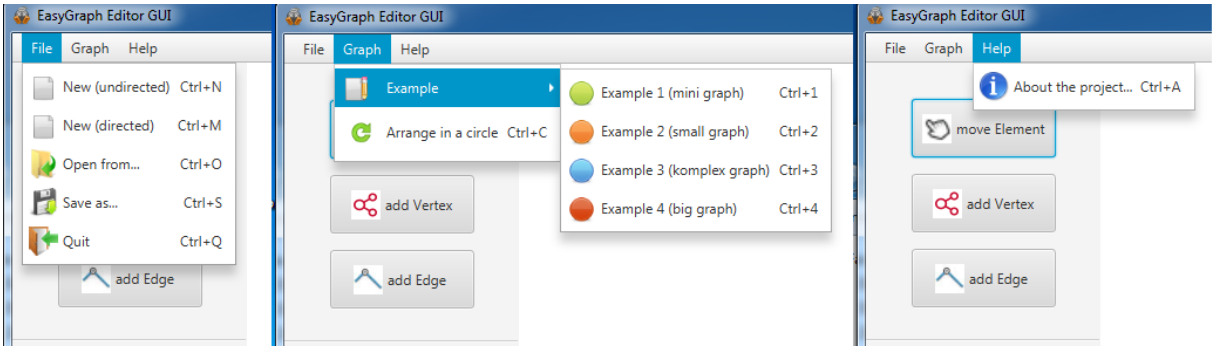
9.1 Benutzeroberfläche

Die Benutzeroberfläche ist in 3 Teile unterteilt: Menuleiste, Toolbox und Arbeitsfläche



9.2 Menuleiste

Die Menuleiste bietet dem Benutzer erweiterte Funktionen für die Bedienung von easyGraph. Im Menu „File“ können neue Graphen erstellt, gespeichert oder geöffnet werden. Das Menu „Graph“ bietet 4 unterschiedliche Beispiel-Graphen und eine automatische Anordnungsfunktion von Graphen. Weiterführende Informationen zum Projekt findet man im Menu „Help“. Hier wird das Projekt Repository verlinkt. Somit sind der Code und alle Projekt Dokumentationen für alle verfügbar.

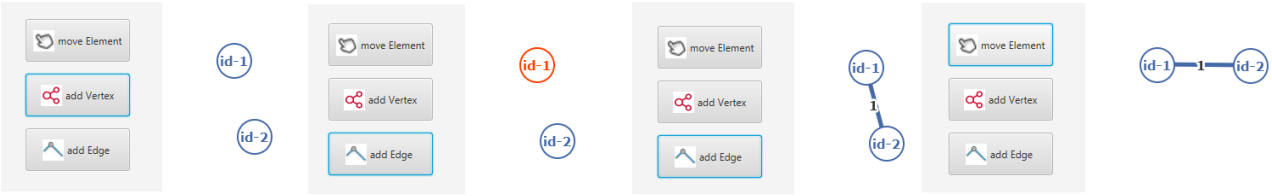


9.3 Toolbox

Die Toolbox stellt dem Benutzer verschiedene Bedienelemente für die Manipulation und Visualisierung der Graphen Algorithmen bereit.

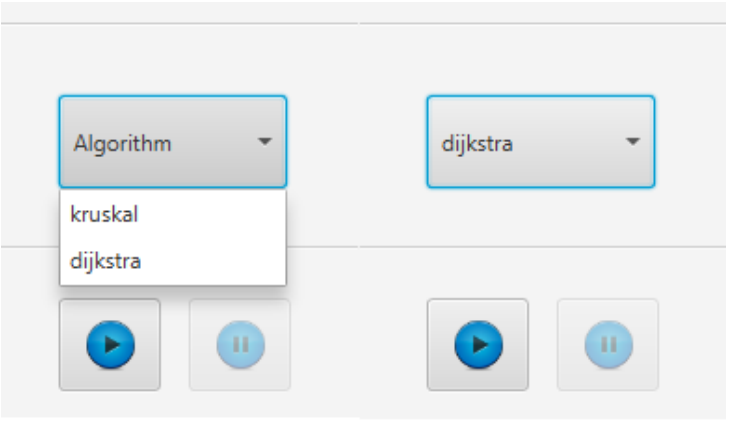
Bedienelemente: Move Element, add Vertex und add Edge

Mit Hilfe dieser 3 Bedienelemente können neue Kanten und Knoten auf die Arbeitsfläche hinzugefügt werden. Falls gewünscht, kann man Knoten verschieben.



Bedienelement: Algorithm

Dient zur Auswahl des gewünschten Anzeige Algorithmus. Die Auflistung und die Anzahl der Algorithmen können unterschiedlich sein und sind abhängig von der Anzahl annotierten Algorithmen in der vorkonfigurierten Klasse.



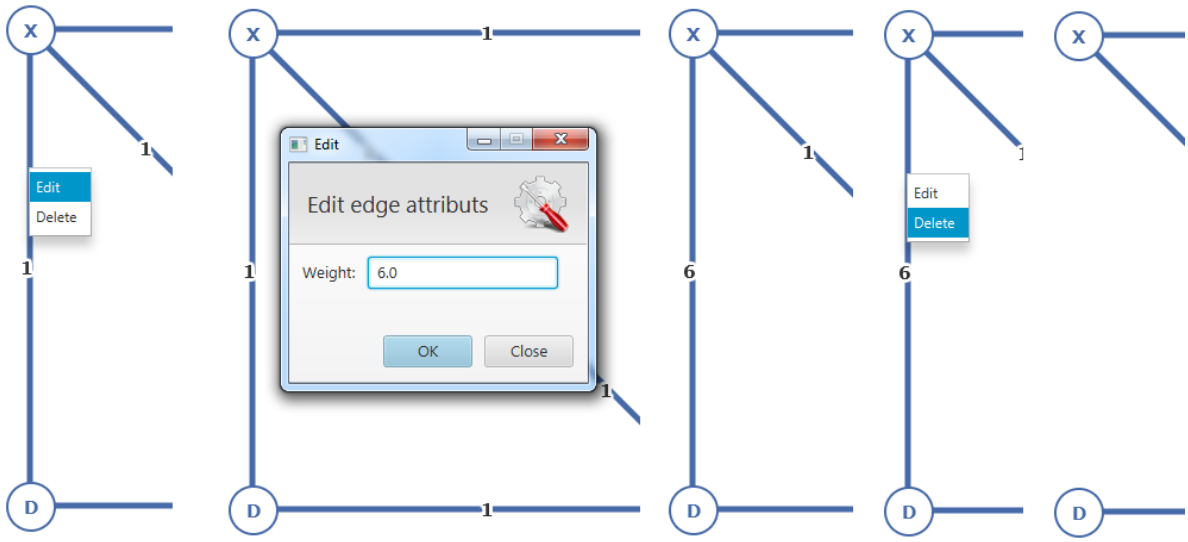
Bedienelemente: Play, Stop, Forward, Backward , Reset und Speed

Diese Bedienelemente steuern die Visualisierung der Graphen Algorithmen auf der Arbeitsfläche.

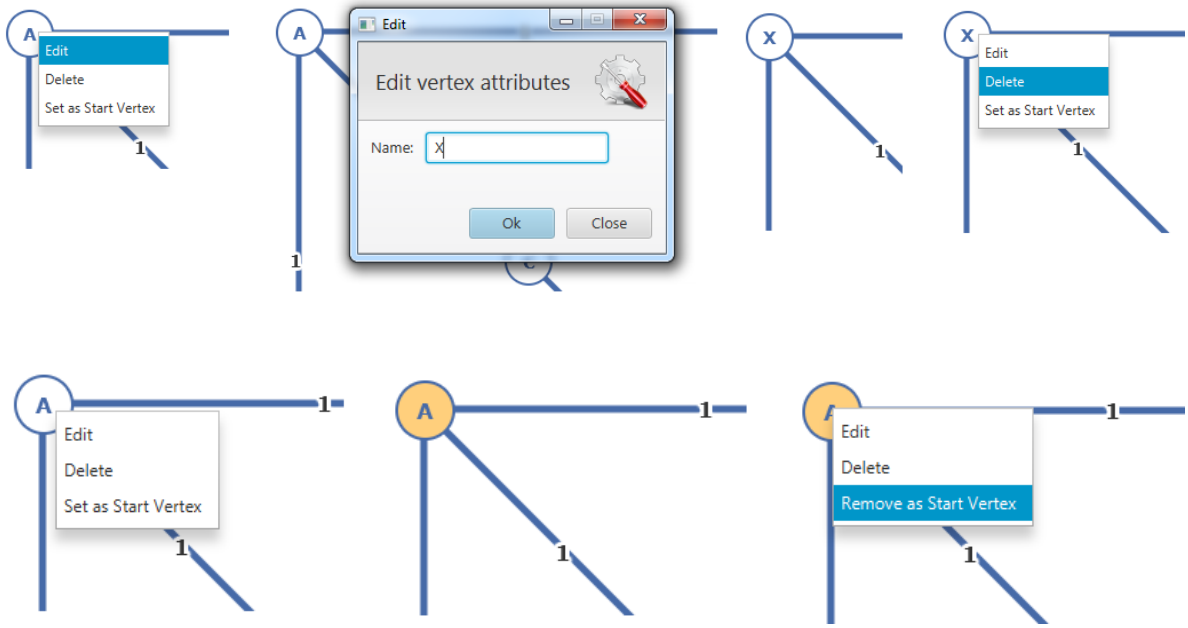
9.4 Arbeitsfläche

Die Arbeitsfläche dient der Anzeige der Graphen und Graphen Algorithmen. Des Weiteren kann mit Hilfe eines Rechtsklicks ein auf das Objekt bezogenes Kontextmenu angezeigt werden.

Kante bearbeiten:



Knoten bearbeiten:



10 Anhang

10.1 Klassendiagramm V1

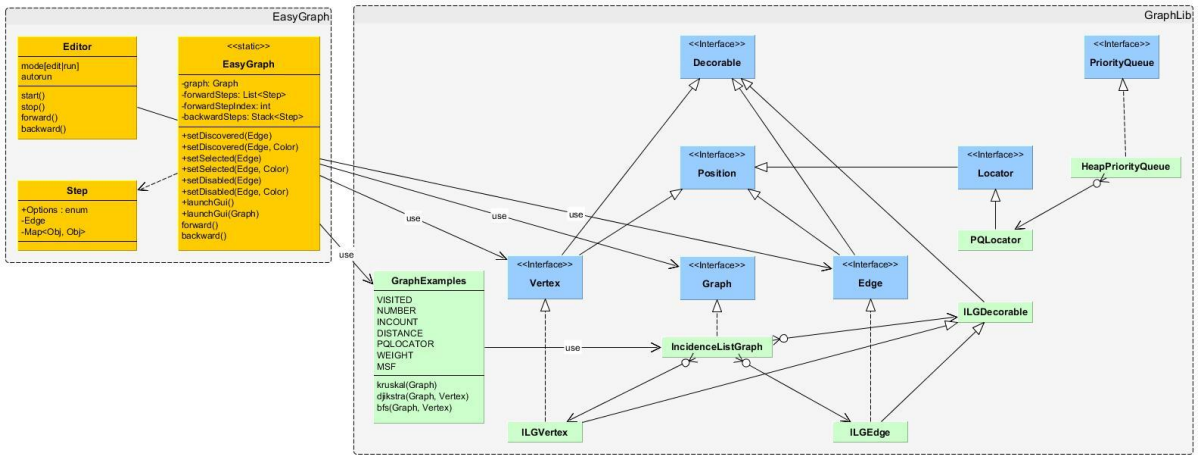


Abbildung 1: Klassendiagramm der Packages „graphLib“ und angedachtem „easyGraph“

10.2 Klassendiagramm V2

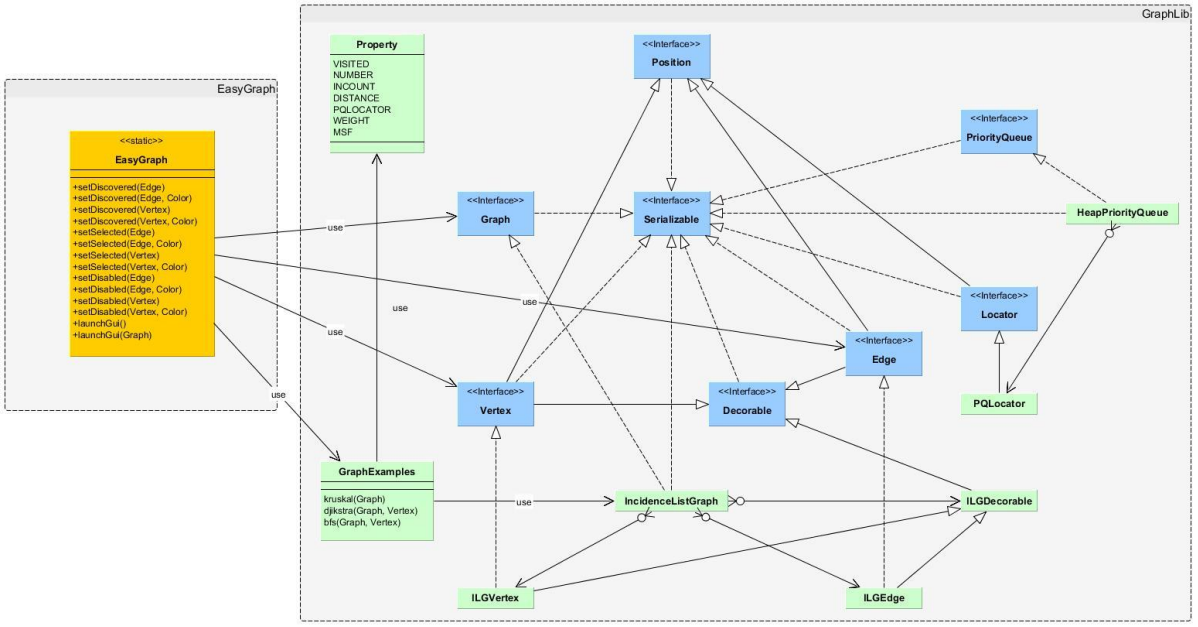


Abbildung 2: Klassendiagramm mit angepasstem Package „graphLib“ und umgesetztem „easyGraph“

10.3 Klassendiagramm von easyGraph

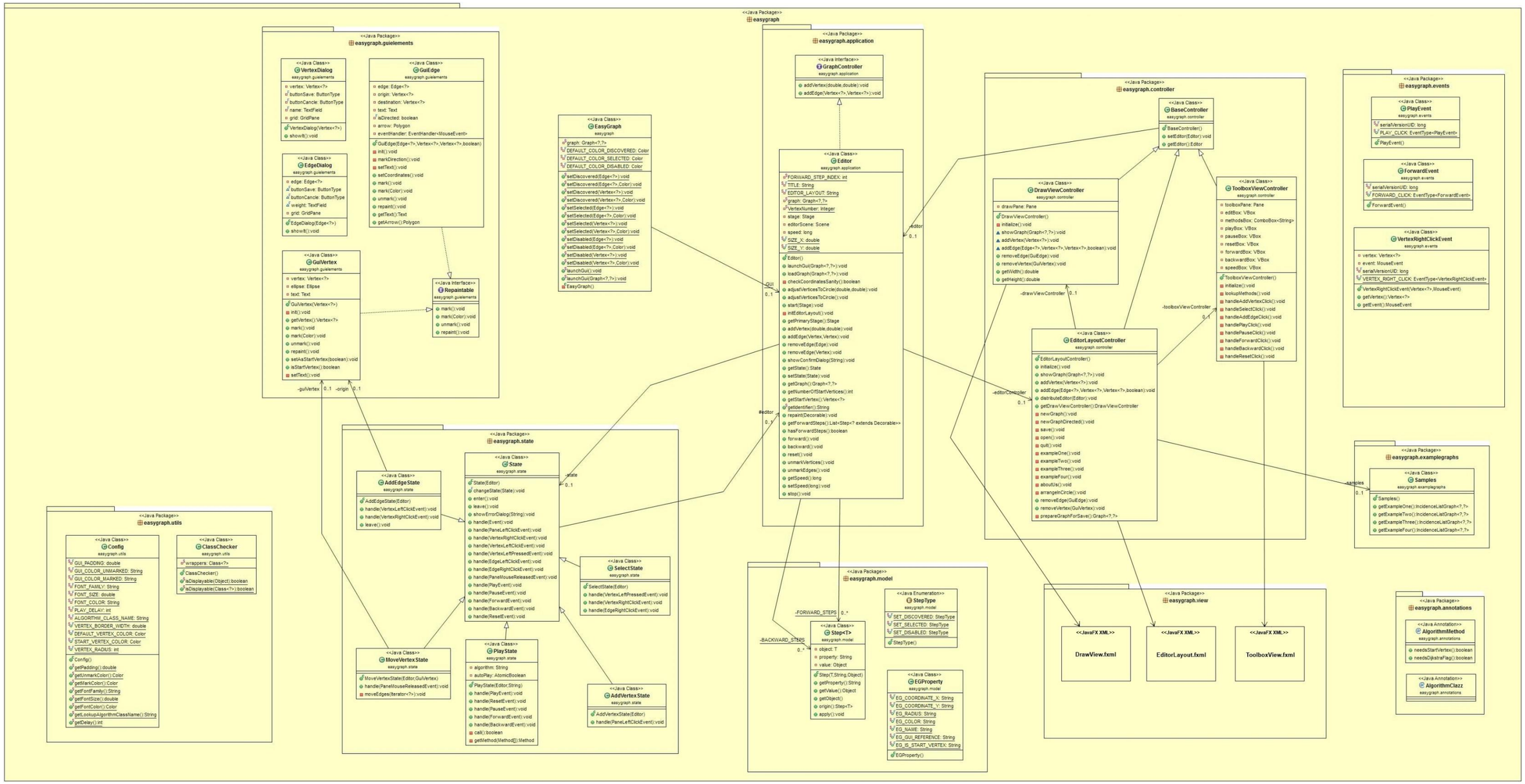


Abbildung 3: Klassendiagramm von "easyGraph" mit eingezeichneten Abhängigkeiten untereinander