Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Studiengang Media Systems

Implementierung eines Grapheneditors für Lehrzwecke

Erstellung des Prototypen *WildGraphs*

Jennifer Wilde

2038329

Hamburg, 24.07.2014

Erstprüfer: Prof. Dr. Edmund Weitz

Zweitprüfer: Daniel Heid

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist das Ziel dieser Arbeit?

Diese Arbeit beschäftigt sich mit der Programmierung eines Programms, welches als Editor für Graphen verwendet wird. Es soll ein Editor erstellt werden, in dem Graphen aus der Graphentheorie schnell gezeichnet und verändert werden können. Das primäre Einsatzgebiet ist die Lehre der Graphentheorie. Für diesen Zweck dient das Programm als Hilfsmittel zur Veranschaulichung von Graphen und graphentheoretischen Algorithmen. Am Ende dieser Arbeit sollte ein funktionsfähiger Prototyp fertiggestellt worden sein.

1.2 Welche Programme gibt es schon?

Programme für programmierte Graphen

Es gibt ein paar Grapheneditoren, mit denen Graphen durch ihre mathematische Definition erzeugt werden können. In ***SciLab*** (5.4.1)[4] oder ***Graphviz*** (2.36)[18] werden Graphen durch Programmierung erstellt. Es steht keine Benutzeroberfläche zur Verfügung, um Graphen direkt visuell zu erzeugen. Neben diesen Programmen kann auch LaTeXmit Hilfe des Package ***tikz*** (3.0.0)[22] verwendet werden, um Graphen darzustellen. Dabei handelt es sich ebenfalls um reine Programmierung. Einen Graphen in LaTeXzu erzeugen, ist sehr zeitaufwendig. Dies trifft vor allem zu, wenn der Graph auf eine bestimmte Art und Weise angeordnet werden soll. Es können damit aber sehr ansprechende Bilder erstellt werden.

Programme für gezeichnete Graphen

Programme, mit denen Graphen direkt (mit der Maus) gezeichnet werden können, gibt es einige. Diese sind in der Regel nicht auf Graphen aus der Graphentheorie abgestimmt. Es ist möglich mit ihnen Graphen zu zeichnen, aber dies ist entweder sehr eingeschränkt möglich oder die Programme sind zu komplex für eine schnelle Zeichnung.

GeoGebra (4.4)[19] ist ein Programm, mit dem Funktionsgraphen gezeichnet werden können. Da Punkte auch einzeln platzierbar sind und Strecken zwischen diesen gezogen werden können, können einfache Graphen gezeichnet werden. Die Bedienung ist auch für jüngere Benutzer ausgelegt und deshalb sehr intuitiv.

Andere Programme wie ***uDraw(Graph)*** (3.1.1)[3] oder ***Tulip*** (4.4)[16] sind speziell für Graphen ausgelegt. Die Steuerung ist in beiden Fällen schwerfällig. *Tulip* ist zudem sehr komplex und besitzt Funktionen, um graphentheoretische Algorithmen durchlaufen zu lassen. Da dieses Programm durch die vielen Optionen unübersichtlich wird, ist es schwierig, es schnell zu beherrschen.

Einige Programme wie ***yEd*** (3.11.1)[34] werden benutzt, um viele verschiedene Arten von Diagrammen erstellen zu können. Sie besitzen eine große Bandbreite an Einstellmöglichkeiten und können für unterschiedliche Aufgaben genutzt werden. Einfache Graphen können in diesen Programmen gezeichnet werden, aber dies ist nicht der Schwerpunkt dieser Programme.

Programme für spezielle Graphen

Für spezielle Arten von Graphen gibt es unterschiedliche Programme, die jeweils ihre eigene Nische abdecken. Für Petri-Netze kann das Programm ***Snoopy*** (1.13)[17] verwendet werden. Damit lassen sich grafisch Netze erstellen und simulieren. Mit *Snoopy* ist es deshalb nur möglich, gerichtete Graphen zu erzeugen. Für das Lehren der Automatentheorie eignet sich das Programm ***JFLAP*** (7.0)[21]. Mit diesem können Automaten erzeugt und Funktionen auf diesen angewendet werden. Es hat eine einfache Steuerung, ist für einfache Graphen aber zu speziell.

1.3 Warum wird ein weiterer Grapheneditor benötigt?

Programme für einfache Graphen gibt es kaum. Die meisten sind zu speziell, andere sind zu breit gefächert und besitzen zu viel Funktionalität.

Ein Ziel dieses Editors ist es, eine einfache Steuerung zu besitzen, mit der schnell ein neuer Graph gezeichnet werden kann. Es muss für den Lehrenden möglich sein, spontan auf eine neue Situation reagieren zu können. Graphen sollten in ihrem Aussehen anpassbar sein, schnell automatisch erzeugt und angeordnet werden können. Für die Lehre sind zudem einige spezielle Graphen, z.B. Bäume, notwendig.

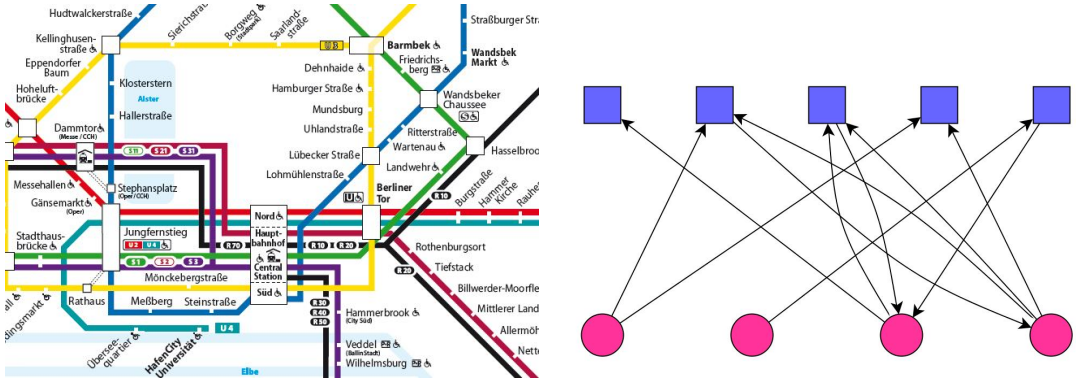
Neben den reinen Graphen gibt es zusätzlich die graphentheoretischen Algorithmen, die eine wichtige Rolle spielen. Sie werden oft an der Tafel mit verschieden farbigen Kreiden dargestellt, wodurch ein Bild schnell unübersichtlich wird. Bei größeren Graphen stellt zudem der benötigte Platz ein Problem dar. Graphentheoretische Algorithmen benutzen meist denselben Graphen, der nur farblich angepasst wird. Sollte eine Animation mit einem der oben genannten Programme erstellt werden, müsste jede Änderung eine eigene Datei sein. Dies ist innerhalb der Lehrveranstaltung wenig praktikabel. Deshalb muss das Programm dem Benutzer ermöglichen, eine Animation zu erstellen, die er abspielen, anhalten oder einzeln durchlaufen kann.

Die Erstellung eines Graphen in einem Editor fördert die Lesbarkeit und Reproduzierbarkeit. Animationen müssen nicht von der Tafel abgeschrieben werden, sondern können in einer Datei gespeichert und ausgedruckt werden. Somit liegt die Konzentration auf dem Verständnis des Algorithmus und nicht auf dem Zeichnen.

2 Ein Blick in die Graphentheorie

2.1 Wofür werden Graphen benötigt?

Ein Graph ist ein mathematisches Modell aus der Graphentheorie, mit welchem sich Beziehungen zwischen Objekten veranschaulichen lassen. Dieses Modell wird von sehr vielen unterschiedlichen Arbeitsgebieten verwendet, um Strukturen, Abläufe und Probleme darzustellen und zu analysieren.



(a) Ein kleiner Ausschnitt der U- und S-Bahn-Verbindungen in Hamburg

(b) Ein möglicher Beziehungsgraph

Abbildung 2.1: Zwei Beispiele für Graphen

Ein bekanntes Beispiel ist die Abbildung der U-Bahn-Verbindungen. Jede Haltestelle ist mit anderen Haltestellen über Linien verbunden, die die Schienenführung darstellen. Eine Haltestelle hat dann zu einer anderen Haltestelle die Beziehung *Ein Zug fährt von X nach Y*. Weitere Beispiele sind Stammbäume, Mindmaps oder ein Beziehungsgraph. Bei letzterem können Männer und Frauen in einer Reihe angeordnet werden und ein Pfeil symbolisiert die Beziehung *X mag Y*.

Mit Hilfe von Graphen lassen sich auch Problemstellungen abstrahieren und durch geeignete Verfahren lösen. Wenn beispielsweise ein Navigationsgerät den kürzesten Weg herausfinden soll, wird dieses Problem zunächst durch einen Graphen vereinfacht. Kreuzungen sind Punkte und Straßen die Linien zwischen ihnen. Jede Straße besitzt eine Anzeige, wie lang diese ist. Der *kürzeste Weg* kann in diesem Zusammenhang unter anderem die Anzahl der Kilometer oder auch die Anzahl der benötigen Minuten sein. Nach der Ermittlung zeigt das Navigationssystem entsprechend passende Vorschläge an und hebt sie farblich hervor.

2.2 Wie ist ein Graph definiert?

Ein ungerichteter **Graph G = (V, E)** besteht aus einer **Eckenmenge V** (engl. vertices) und einer **Kantenmenge E** (engl. edges)[14, 28, 29]. Eine Kante verläuft von einer Start-Ecke zu einer Ziel-Ecke, wodurch sie eine Beziehung zwischen diesen Ecken symbolisiert. Die beiden Ecken können auch identisch sein. Eine Ecke hat somit eine Beziehung zu sich selbst. Ecken sind **benachbart**, wenn sie über eine Kante miteinander verbunden sind.

Die gebräuchliche grafische Darstellung der Ecken und Kanten sind Punkte und Linien. Je nach Anwendungsgebiet können sich Graphen in ihrer Darstellung unterscheiden.

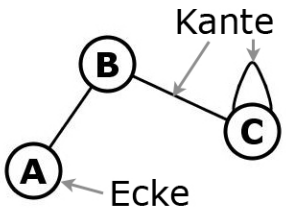


Abbildung 2.2: Begriffserläuterung anhand eines ungerichteten Graphens

Ein Graph ist stets durch seine mathematische Definition vollständig beschrieben.

Er wird durch seine Mengen definiert, die wie folgt notiert werden:

$$\begin{aligned} G &= (V, E) \\ V &= \{a, b, c, d\} \\ E &= \{ab, ac, bc, cd, bd, dd\} \end{aligned} \tag{2.1}$$

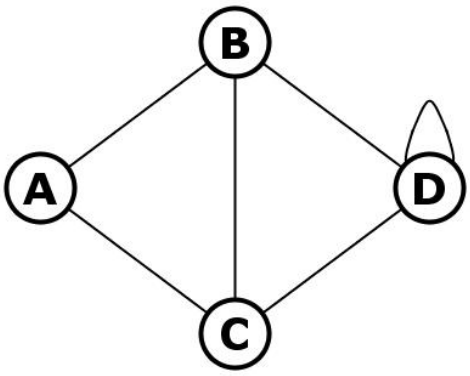
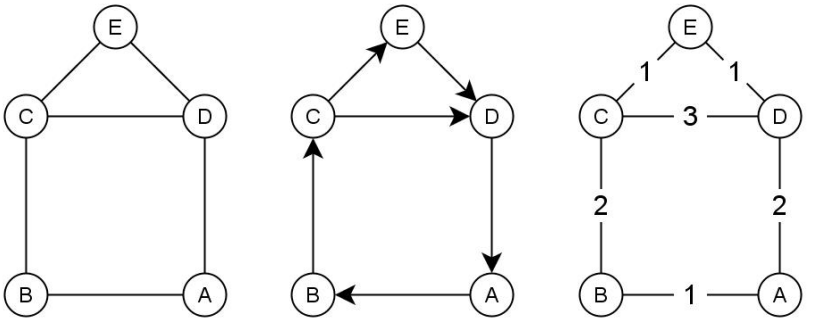


Abbildung 2.3: Darstellung des oben definierten Graphen

2.3 Welche Arten von Graphen gibt es?



(a) Ungerichtet **(b)** Gerichtet **(c)** Ungerichtet und gewichtet

Abbildung 2.4: Verschiedene Arten von Graphen

Graphen können gerichtet oder ungerichtet sein. Bei **ungerichteten Graphen** gilt die Beziehung in beide Richtungen (einfache Linie). In diesem Fall, ist die Definition der Kante (A, B) identisch mit (B, A). Bei einem **gerichteten Graphen** besteht die Beziehung nur in eine Richtung (Pfeil). Die Kanten (A, B) und (B, A) sind dann nicht identisch, können aber beide gleichzeitig im Graphen auftreten. Ein ungerichteter Graph kann als ein gerichteter aufgefasst werden, bei dem eine neue Kante in beide Richtungen eingefügt wird.

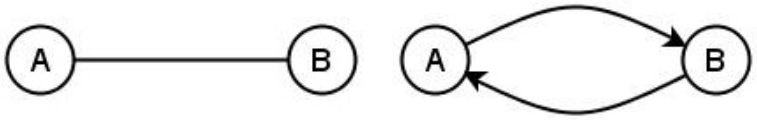


Abbildung 2.5: Vergleich ungerichteter und gerichteter Graph

Bei einem gerichteten Graphen werden die Nachbarn einer Ecke in Vorgänger und Nachfolger unterteilt. Wenn A die aktuelle Ecke ist und sie mit B über die Kante (A,B) verbunden ist, ist B ein **Nachfolger** von A. Ist A mit C über (C,A) verbunden, ist C ein **Vorgänger** von A.

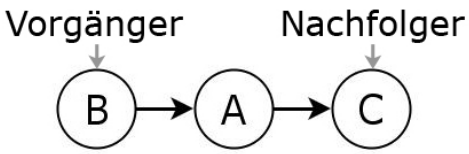
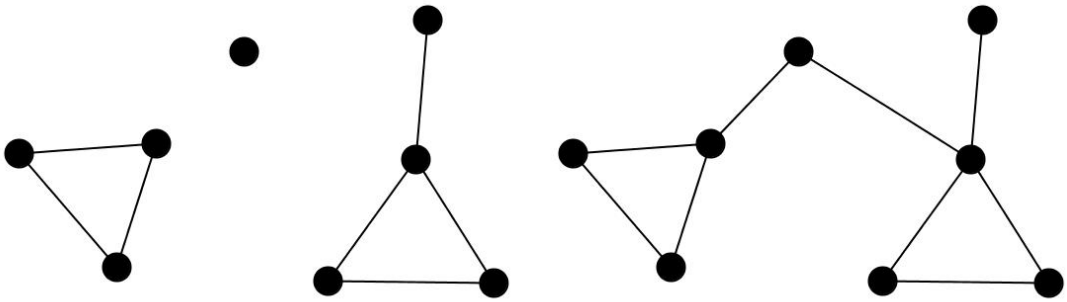


Abbildung 2.6: Erklärung der Begriffe Vorgänger und Nachfolger anhand eines Graphen

Zudem gibt es **gewichtete Graphen**. Dabei können sowohl Kanten als auch Ecken ein Gewicht in Form einer Zahl aufweisen, die in dem jeweiligen Einsatzgebiet eine Bedeutung hat, z.B. Anzahl der Kilometer. Enthält ein Graph keine Gewichte, wird er als **ungewichtet** bezeichnet.

Ein Graph kann auch Ecken enthalten, die mit keiner Kante verbunden sind (freie Ecken). Ferner können zwei Eckenmengen existieren, die untereinander nicht verbun-

den sind. Wenn in einem Graphen jede Ecke mit jeder anderen Ecke über Kanten verbunden ist, ist der Graph **zusammenhängend**.



(a) Nicht zusammenhängend **(b)** Zusammenhängend

Abbildung 2.7: Beispiele für den Zusammenhang von Graphen

Speziellere Formen von Graphen sind unter anderem bipartite Graphen, vollständige Graphen und Bäume. Ein **vollständiger Graph** enthält alle möglichen Kanten, die zwischen allen Ecken existieren können.

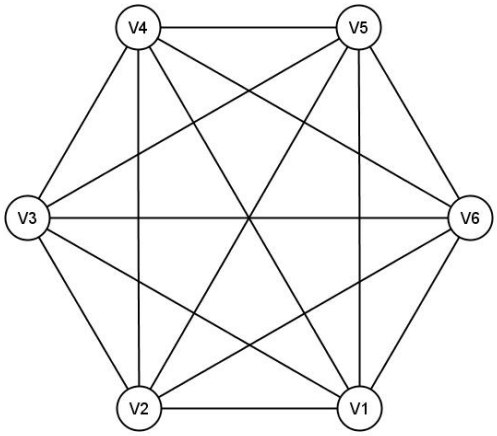
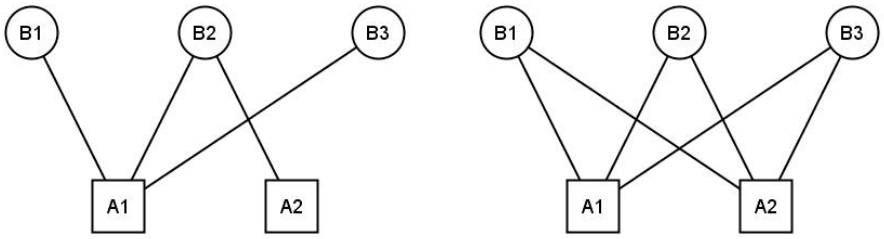


Abbildung 2.8: Beispiel eines vollständigen Graphen

Ein **bipartiter Graph** zeichnet sich dadurch aus, dass seine Eckenmenge V in zwei Eckenmengen A und B unterteilt werden können [9]. Kanten verbinden dann eine Ecke aus A mit einer Ecke aus B. Es existieren keine Kanten, die zwei Ecken aus

A miteinander verbinden. Das gleiche gilt für die Menge B. Bei einem vollständig bipartiten Graphen sind dies alle Kanten, die zwischen den beiden Eckenmengen erzeugt werden können.



(a) normal **(b)** vollständig

Abbildung 2.9: Beispiel bipartiter Graphen

Die sehr bekannte Form eines **Baumes**, z.B. als Stammbaum, ist eine spezielle Form eines Graphen [25]. Üblicherweise werden die Ecken eines Baumes in verschiedenen Reihen untereinander angeordnet. Dabei kann nur eine Kante zwischen benachbarten Reihen existieren. Eine Ecke aus einer unteren Reihe kann nicht mit mehr als einer Ecke der darüber liegenden Reihe verbunden sein. Anders formuliert muss ein Graph bestimmte mathematische Kriterien erfüllen, um ein Baum zu sein. Er ist stets zusammenhängend und besitzt somit keine freien Ecken. Zudem ist ein Baum immer ein bipartiter Graph, der eine Kante weniger besitzt als Ecken.

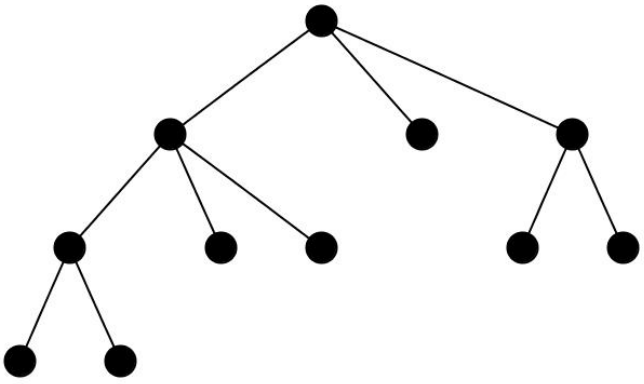


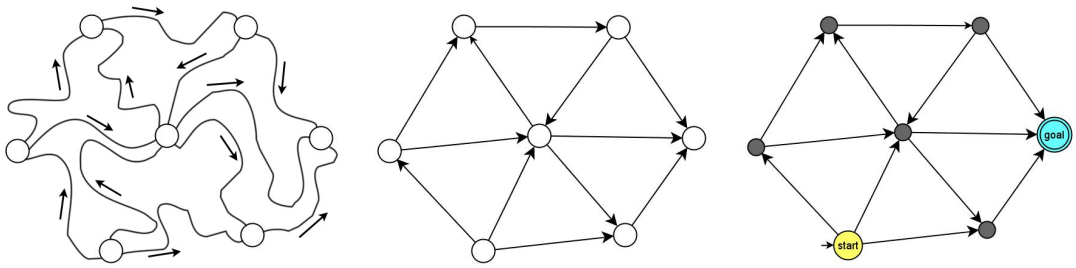
Abbildung 2.10: Beispiel eines Baumes

Es existieren noch viele weitere Arten und Spezialfälle von Graphen, die in dieser Arbeit keine Relevanz haben.

2.4 Wie wird ein Graph dargestellt?

Ein Graph ist vollständig durch seine mathematische Definition beschrieben. Eine grafische Darstellung ist nicht zwingend erforderlich, für den Menschen aber leichter zugänglich.

Das Aussehen der Ecken und Kanten ist nicht festgelegt. Je nach Anwendungsgebiet gibt es gängige Normen, um einen Graphen und die gewünschte Absicht abzubilden [31, 27]. Ecken können beispielsweise unterschiedliche Formen, Farben und Beschriftungen haben. Kanten können Geraden, Kurven, Pfeile oder andere Arten der Verbindung sein.

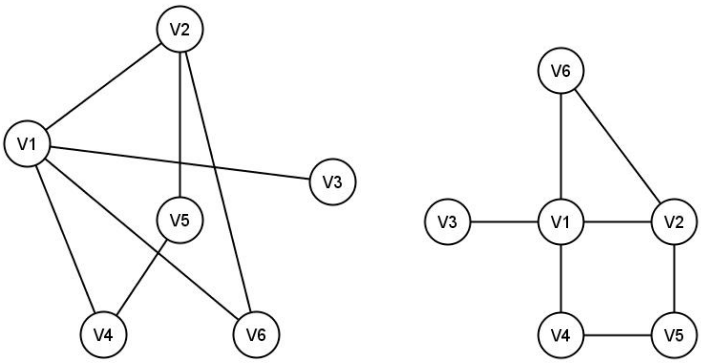


(a) Originale Wegführung **(b)** Abstrahiert **(c)** Farblich hervorgehoben

Abbildung 2.11: Verschiedene Darstellungen eines Graphen

Neben dem reinen Aussehen spielt die Anordnung (Layout) der Ecken und Kanten eine große Rolle. Der Mensch möchte eine klare, verständliche Anordnung und erzeugt diese intuitiv auf dem Papier. Eine ideale Anordnung gibt es für einen Graphen nicht, da dies je nach Thema und Geschmack variieren kann. Erstrebenswert ist stets, dass sich so wenig Kanten wie möglich kreuzen, da dies zur Klarheit beiträgt. Die meisten gebräuchlichen Layouts verwenden nur Geraden und keine Kurven als Kanten. Durch gerade Linien ist es schneller zu erkennen, welche Ecken zusammenhängen. Dieses Thema ist in der Graphentheorie ein gesonderter Forschungszweig.

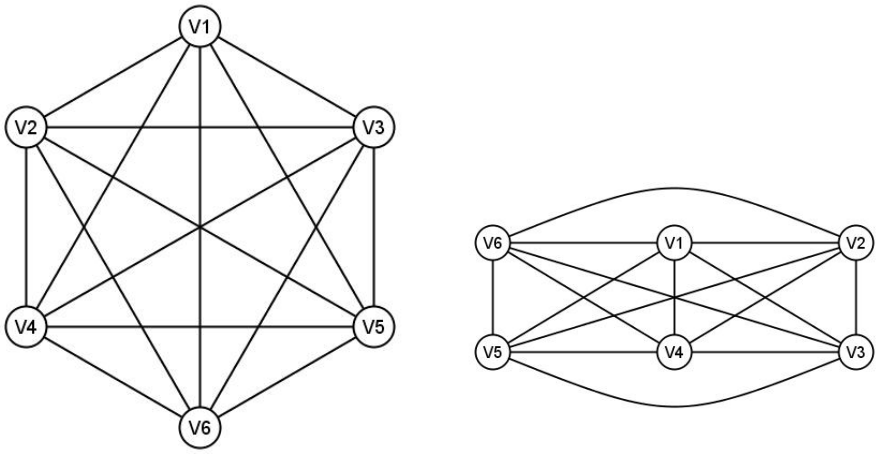
Im Kapitel *Wie funktioniert ein kräftebasiertes Layout?* wird eine der Vorgehensweisen erläutert.



(a) Zufällig **(b)** Geordnet

Abbildung 2.12: Zwei unterschiedliche Darstellungen desselben Graphen

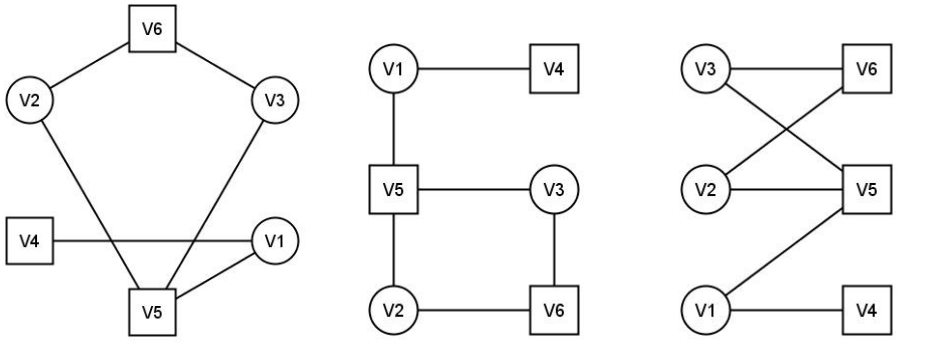
Für die weiter oben angesprochenen Arten von Graphen gibt es trotzdem Anordnungen, die sich bewährt haben. Ein vollständiger Graph besitzt viele Kanten und kann normalerweise nicht so angeordnet werden, dass sich keine Kanten überschneiden. Die Ecken auf einem Kreis anzuordnen, verschafft mehr Übersichtlichkeit und verdeutlicht die Absicht besser als andere Layouts.



(a) Kreis **(b)** Bipartit

Abbildung 2.13: Anordnungen eines vollständigen Graphen

Bipartite Graphen lassen sich gut in Reihen anordnen. Dies sortiert die Ecken nach ihren Eckenmengen, die somit leicht zu erkennen sind. Kantenüberschneidungen sind in diesem Layout meist nicht zu vermeiden. Je nach Absicht kann ein bipartiter Graph auch in einem Raster angeordnet werden, sodass sich keine Kanten kreuzen. Dabei geht die Sortierung der Eckenmengen verloren. Das geeignete Layout hängt somit stark von dem gewünschten Schwerpunkt ab.



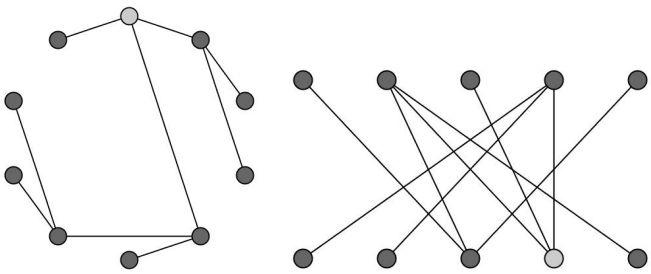
(a) Auf einem Kreis

(b) Ohne Überschneidungen der Kanten (Raster)

(c) Nach den Eckenmengen sortiert (bipartit)

Abbildung 2.14: Anordnungen eines bipartiten Graphen

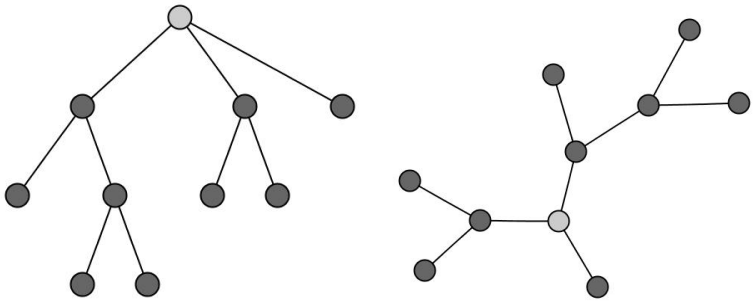
Für Bäume existieren zwei gängige Layouts, die das Wesen eines Baumes unterstreichen. Beide Layouts sind hierarchisch aufgebaut. Bei dem einen werden die Ecken in Reihen aufgeteilt, bei dem anderen auf Kreisen angeordnet, deren Radien immer größer werden.



(a) Kreis

(b) Bipartit

Abbildung 2.15: Ungeeignete Anordnungen eines Baumes



(a) Hierarchisch Reihen **(b)** Hierarchisch Kreise

Abbildung 2.16: Geeignete Anordnungen eines Baumes

2.5 Warum wird mehr als eine Graphendarstellung benötigt?

Die Darstellung eines Graphen ist eine Momentaufnahme und kann seinen aktuellen **Status** widerspiegeln. Wie genau dieser Status des aktuellen Graphen definiert ist, hängt vom jeweiligen Kontext hab. Wenn sich der Status ändert, ändert sich auch die Darstellung. Soll eine fortlaufende Änderung des Status nachvollziehbar sein, muss jede Änderung festgehalten werden. Die Statusänderung auf dem Papier logisch darzustellen, stellt sich als schwierig heraus. Üblicherweise wird der Graph komplett neu gezeichnet und (farblich) angepasst.

Als Beispiel soll *der schnellste Weg von A nach B* dienen (Abb. 2.17). Die Kanten haben ein Gewicht, welches die Zeit in Minuten repräsentiert. Mit Hilfe der Kanten kann, über mehrere Ecken hinweg, ein Weg gefunden werden, der am schnellsten ist.

Die ursprüngliche Definition des Graphen im mathematischen Sinne hat sich in keiner der Bilder geändert. Es wurden nur farbliche Anpassungen vorgenommen.

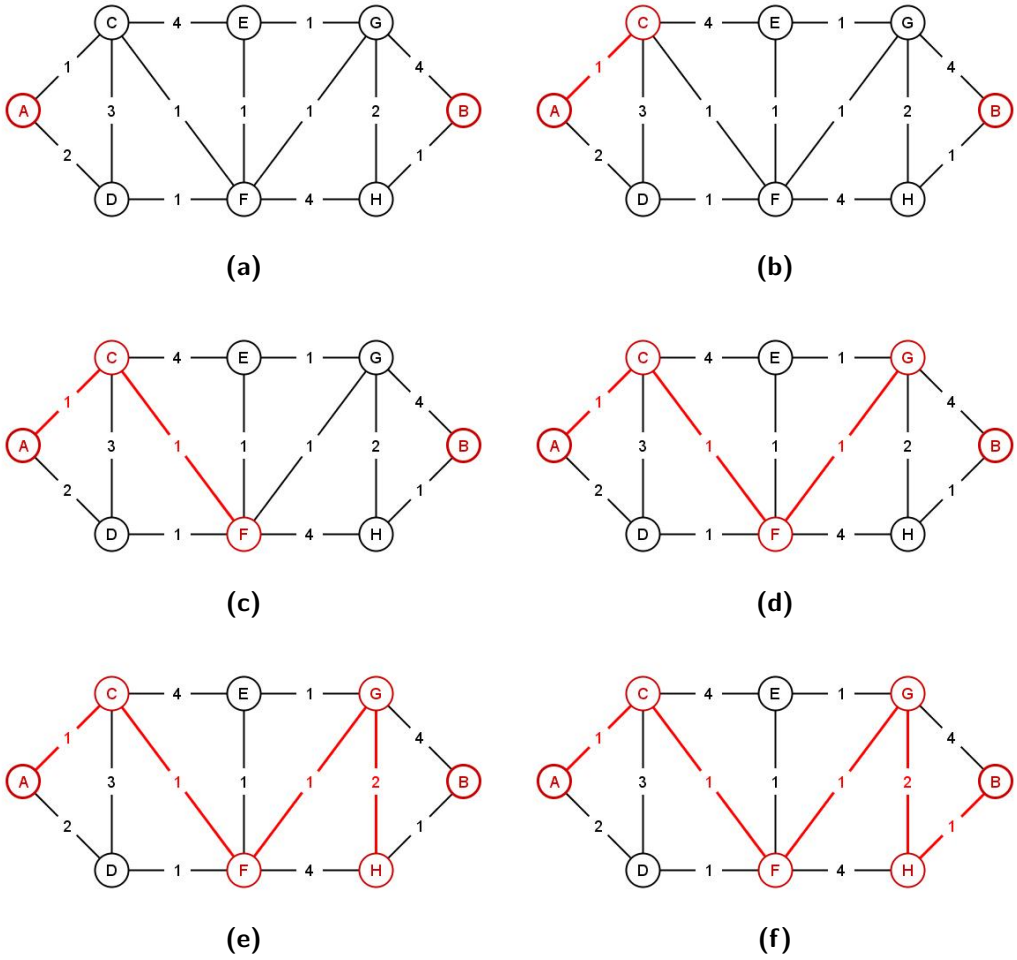


Abbildung 2.17: Animation des schnellsten Weges

3 Der Editor *WildGraphs*

3.1 Vorbereitung

3.1.1 In welchem Umfeld soll der Editor zum Einsatz kommen?

Der Editor ist darauf ausgelegt, als Hilfsmittel in der Lehre zu dienen. Er kann während der Vorlesung oder einem Vortrag genutzt werden, um Algorithmen oder die Erzeugung eines Graphen zu erläutern. Interessierte der Graphentheorie können den Editor zur Erstellung eigener Graphen verwenden und experimentieren. Das primäre Ziel ist hierbei, Lesbarkeit und Klarheit der so erstellten Graphen im Gegensatz zu handgezeichneten Graphen zu erhöhen. Sowohl Lehrende als auch Lernende müssen schnell einen Graphen erstellen, erkennen und interpretieren können. Dem Lehrenden sollte es möglich sein, bestimmte wichtige Stellen eines Graphen durch Hervorhebung zu markieren. Damit ergeben sich die beiden wichtigsten Anforderungen an den Editor: **Klarheit der Darstellung** und **Benutzerfreundlichkeit**.

3.1.2 Welche Personen benutzen den Editor?

Die Zielgruppe besteht größtenteils aus Lehrenden (Lehrern, Professoren usw.) und Lernenden (Schüler, Studenten usw.). Der Editor kann auch von jedem anderen Personenkreis verwendet werden. Voraussetzung ist, dass die Person weiß, was ein Graph ist, damit sie den Editor sinnvoll nutzen kann. Damit dieser auch von Personen benutzt werden kann, die nicht Deutsch sprechen, wurde der Editor in englischer Sprache gehalten.

Rahmenbedingungen

Die Hardware kann je nach Einsatzumgebung sehr unterschiedlich ausfallen, weil verschiedene Räume und Geräte zum Einsatz kommen. Darunter fallen beispielsweise Laptops und Standrechner in Computerräumen. Der Editor sollte die meisten Betriebssysteme unterstützen und auch auf schwächeren Rechnern flüssig laufen. Um diese Bedingungen zu erfüllen, wurde das Programm in Java 1.7 geschrieben. Es muss zudem darauf geachtet werden, dass einige Eingabemethoden ggf. nicht zur Verfügung stehen, z.B. die mittlere Maustaste.

Weiterentwicklung

Lehrpläne und -inhalte können sich mit der Zeit und der Weiterentwicklung der Technologie ändern oder erweitert werden. Deshalb muss das Programm in der Zukunft schnell und einfach erweitert werden können. Um dies zu erreichen, muss die interne **Programmstruktur einfach gehalten** sein. Außerdem wurde der Programmcode auf Englisch gehalten, damit dieser auch von Personen verstanden und verändert werden kann, die der deutschen Sprache nicht mächtig sind.

3.1.3 Wie sieht der Funktionsumfang des Editors aus?

Einschränkungen

Der Editor ist beschränkt auf sein Einsatzgebiet. Deshalb soll überflüssige Funktionalität vermieden werden, damit der Editor nicht zu kompliziert wird. Er dient weder als Simulationssoftware noch als Lernsoftware und vermittelt kein Verständnis der Graphentheorie. Dennoch muss er schnell genug arbeiten, damit Verzögerungen vermieden oder reduziert werden. In der Lehre der Graphentheorie werden normalerweise keine speziellen Graphen behandelt. Die meisten Graphen dienen der Veranschaulichung und sind deshalb nicht sehr groß. Sie besitzen meistens nicht mehr als 15 Elemente. Ein **Element** kann eine Kante oder eine Ecke sein. Der Editor kann aus diesem Grund nur einfache Arten von Graphen darstellen. Die Funktionen sollten mit Graphen bis zu 20 Elementen ohne spürbare Verzögerungen umgehen

können.

Mindestanforderungen

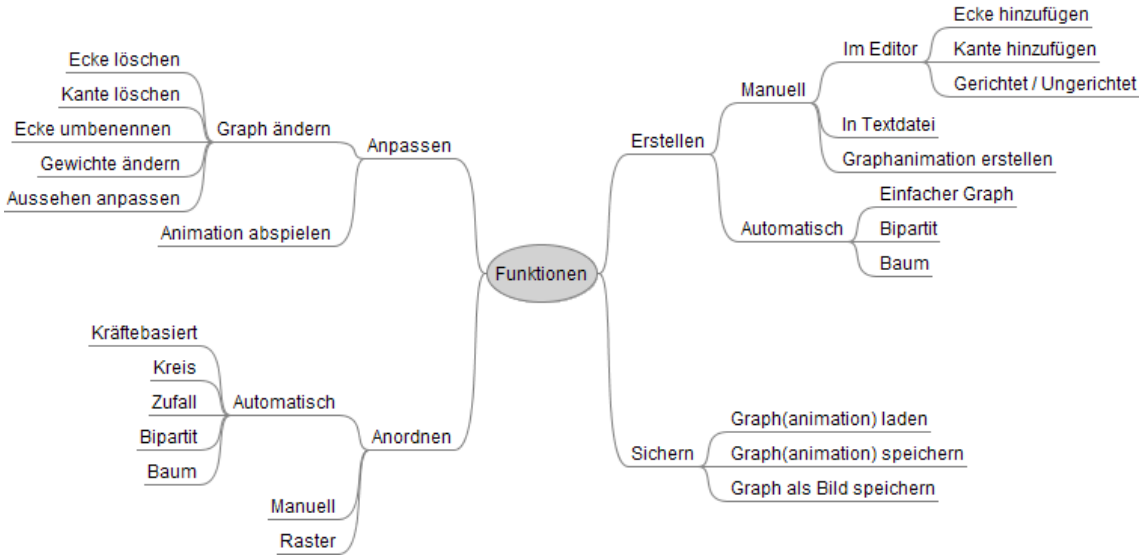


Abbildung 3.1: Funktionsumfang aus Benutzersicht

Die obige Darstellung deckt die Grundfunktionen ab, die der Editor im endgültigen Prototypen aufweisen sollte. Der Benutzer sollte in der Lage sein, einen Graphen zu erstellen, ihn anzuordnen und seine Darstellung anzupassen. Mehrere Graphen sollten zu einer Animation zusammengefasst und gespeichert werden können. Neben dem manuellen Erstellen und Anordnen sollten Möglichkeiten integriert sein, um dies automatisch vom Editor durchführen zu lassen. Dabei werden die gängigsten Graphenarten und -anordnungen unterstützt. Häufig verwendete Graphen sollten mittels Sicherung gespeichert und wiederverwendet werden können.

3.2 Aufbau

3.2.1 Wie sieht der Editor aus?

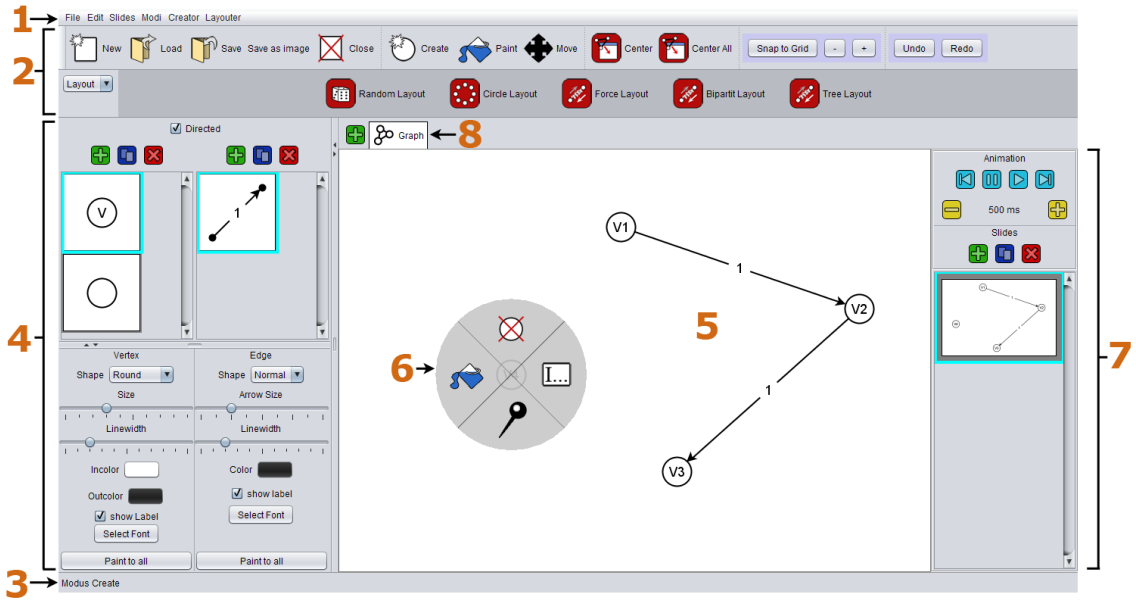


Abbildung 3.2: Screenshot des Editors

Der Editor besteht aus fixen und variablen Teilen. Menüliste (1), Werkzeugleiste (2) und die Statusleiste (3) sind die fixen Elemente. Auch wenn die Buttons der Werkzeugleiste geändert werden können, können keine Buttons dynamisch während des Betriebs hinzugefügt werden. Auf der linken Seite befindet sich die Optionsbox (4), in der verschiedene Werte eingestellt werden können. Je nachdem, in welchem Modus sich der Editor befindet, ändert sich diese Anzeige und der Benutzer kann verschiedene Einstellungen vornehmen. In der Mitte sitzt die Zeichenfläche (5) bzw. Tafel. Hier kann der Benutzer mit dem Editor über Mausbewegungen und -klicks interagieren. Der ausgewählte Graph wird auf der Tafel gezeichnet und kann verändert werden. Zusätzliche Markierungen, die ebenfalls dort gezeichnet werden, hängen vom aktuellen Modus ab. Das Kontextmenü (6) für Ecken und Kanten lässt sich nur innerhalb der Zeichenfläche öffnen. Auf der rechten Seite befindet sich die Anzeige der Folien (7) zu der aktuellen Datei. Diese können als Animation abgespielt oder

angepasst werden. Die aktuelle Folie wird gekennzeichnet und ändert ihr Aussehen, sobald etwas auf der Tafel geändert wird. Über der Tafel befindet sich die Auswahlleiste (8) für die Dateien. Ein Klick auf einen anderen Tab, wählt die entsprechende Datei aus.

3.2.2 Aus welchen Bereichen besteht der Editor?

Nachdem der äußerliche Aufbau erläutert wurde, wird nun auf die interne Struktur und die Programmierung eingegangen.

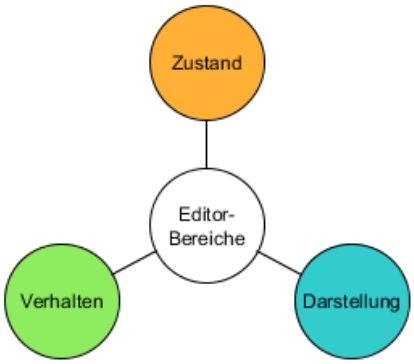


Abbildung 3.3: Bereiche des Editors

Der Editor kann intern in drei Bereiche unterteilt werden: Zustand, Verhalten und Darstellung. Klassen, die in den Bereich **Zustand** fallen, beinhalten wichtige Daten. Sie speichern beispielsweise einen Graphen oder die Reihenfolge der Folien einer Animation. Von Außen kann der Zustand durch geeignete Funktionen gesetzt werden. In der Regel kann *jeder* Wert verändert werden. Die meisten Klassen fallen in den Bereich **Verhalten**. Sie dienen dazu, den Zustand zu verändern oder bieten andere Funktionalitäten an, z.B. die Prüfung, ob ein Graph bipartit ist. Unter **Darstellung** werden Klassen eingeordnet, die Daten nach außen sichtbar machen. Das gesamte Interface fällt darunter, aber auch die Zeichnung eines Graphen oder das Kontextmenü.

Es ist nur sehr schwer möglich, die drei Bereiche strikt voneinander zu trennen. Oft entstehen Hybrid-Klassen, die einen Bereich vertreten, aber zusätzlich ein paar

Aufgaben aus einem anderen Bereich übernehmen. Um dennoch die Bereiche in so einer Klasse voneinander zu trennen, werden interne Klassen verwendet. Dieses Prinzip ist sehr bekannt unter Java Swing-Klassen, bei denen die Listener interne Klassen sind. Listener sind Klassen, die auf eine Benutzereingabe reagieren, z.B. einen Mausklick.

3.2.3 Wie arbeiten die Bereiche zusammen?

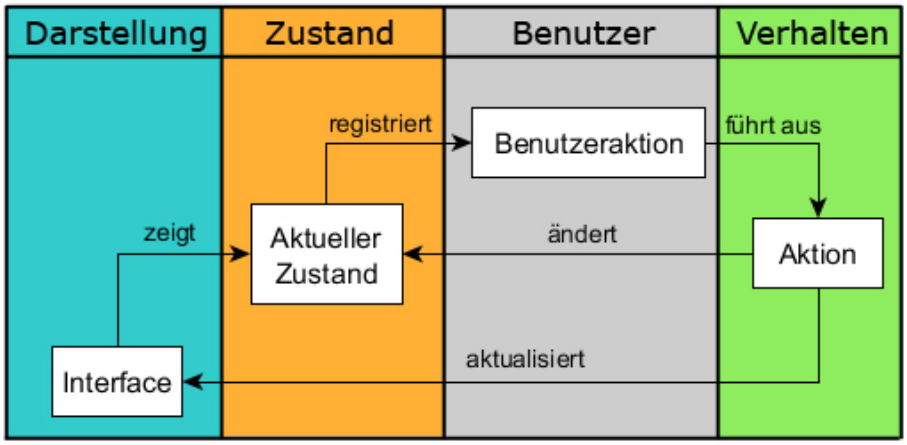


Abbildung 3.4: Zusammenarbeit der Bereiche

Damit sich der Zustand des Editors ändern kann, muss der Benutzer von außen einen Impuls geben. Soll der Zustand geändert werden, greift eine Zustandsklasse nicht direkt auf eine andere zu, um dies zu erreichen. Stattdessen leitet sie die Ausführung an eine Verhaltensklasse weiter. Diese tragen keine Werte, die für den Zustand von Bedeutung sind. Sie führen ihre Funktion aus, indem sie über eine Ansprechstelle der Zustandsklassen auf die gewünschte Klasse zugreifen, die Werte ändern und schließlich die Anzeigen aktualisieren.

Im Folgenden werden einige Entscheidungen und Probleme dargelegt. Es wird mit dem Bereich Zustand begonnen. Anschließend folgt das Verhalten und zum Schluss die Darstellung.

3.3 Zustand und Definitionen

3.3.1 Wo wird der Zustand festgehalten?

Der aktuelle Zustand in dem Editor wird durch viele Werte widergespiegelt. Diese Werte sind auf mehrere Klassen verteilt, die diese nach Zuständigkeitsbereichen gliedern. Um den Zustand von außen ändern zu können, existiert die Klasse **Collector**. Diese dient als Container-Klasse und gibt über mehrere Schritte eine Referenz auf die gewünschte Klasse zurück. Wichtige Werte, wie die Elemente eines Graphen, dürfen nur mittels dieser Klasse verändert werden. Eine Zustandsklasse sollte ihre untergeordneten Elemente kennen, aber nicht die übergeordneten.

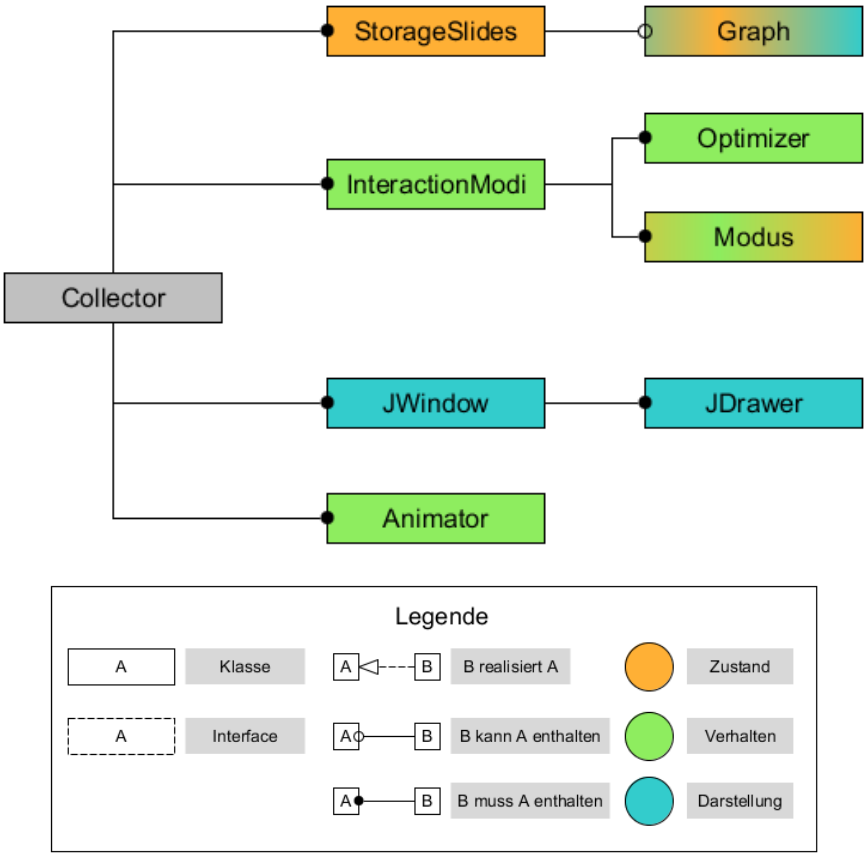


Abbildung 3.5: Ein verkürzter Ausschnitt der erreichbaren Klassen

3.3.2 Wie sind Graphen, Ecken und Kanten definiert?

Bei der Definition eines Graphen stellte sich die Frage, ob es möglich sein soll, ihn zu jeder Zeit auch zeichnerisch darzustellen. Die Implementierung hängt nicht alleine von der Aufgabenverteilung sondern auch von den Klassen ab, die diese Klassen benutzen und entsprechende Werte voraussetzen. Im folgenden werden zwei Ansätze für die Definition einer Ecke vorgestellt. Da Graphen, Ecken und Kanten eine Einheit bilden, können die Ansätze auf die anderen Elemente übertragen werden.

Reine mathematische Definition

Wenn eine Ecke nicht direkt gezeichnet werden kann und die entsprechenden Werte nicht in dem Objekt der Ecke festgehalten werden, ist das Aussehen einer bestimmten Ecke undefiniert. Ein Objekt einer Ecke wäre dann nur durch seinen Namen und die Verbindungen zu anderen Elementen bestimmt. Damit wüsste es weder seine Position noch seine Farbe oder Form. Es hätte Funktionen, mit denen es neue Nachbarn und die dazugehörigen Kanten hinzufügen oder entfernen könnte. Diese Definition erschwert das Zuweisen eines individuellen Aussehens, trennt aber die Zuständigkeiten *Ecke definieren* und *eine Ecke malen*. Um Aussehen und die dazugehörige Ecke wieder zu vereinen, müsste eine neue Klasse geschaffen werden, welche beide miteinander verbindet. Dabei ist die Herausforderung, welche Werte in welcher Klasse gespeichert werden sollten. Es gibt Klassen in diesem Programm, beispielsweise der Layouter, die das Aussehen einer Ecke nicht kennen sollen. Trotzdem benötigen sie bestimmte Werte (hier die Position der Ecke), die der Zeichner-Klasse zugeordnet wären. Weil die Klassen die Zeichner-Klasse bewusst nicht verwenden sollen, müssten diese Werte in der Eckendefinition gespeichert werden, wo sie sinngemäß nicht hingehören. Dieser Wert könnte somit nicht eindeutig zugeordnet werden.

Elemente als Zeichenobjekt

Sollte umgekehrt eine Ecke eine Funktion besitzen, womit diese gezeichnet werden kann, muss sie viele zusätzliche Werte und Algorithmen enthalten. Dies vermischt die Zuständigkeiten und lässt die Klasse sehr groß und unübersichtlich werden. Es

wäre bei dieser Definition ohne weitere Klassen nicht möglich, die Form einer Ecke zu ändern. Der Quellcode zum Zeichnen wäre für jedes Objekt der Klasse identisch.

Implementierung

Weil die Hauptaufgabe des Editors das Darstellen von Graphen ist, muss jeder Graph und somit jede Ecke und Kante jederzeit gezeichnet werden können. Um trotzdem Zeichnung und mathematische Definition voneinander zu trennen, wird das Zeichnen mehreren zusätzlichen Klassen überlassen. Im Falle der Ecken heißt die Klasse, die die Werte für die Zeichnung speichert, VertexDrawOption (VDO). Damit verschiedene Formen möglich sind, enthält sie wiederum ein Objekt des Interfaces VertexShape (VS). Gemäß dem *Strategy-Entwurfsmuster* [6, 26] kann die Form so auch während des laufenden Betriebs ausgetauscht werden. Äquivalent entstanden die Klassen EdgeDrawOption (EDO) und EdgeShape (ES).

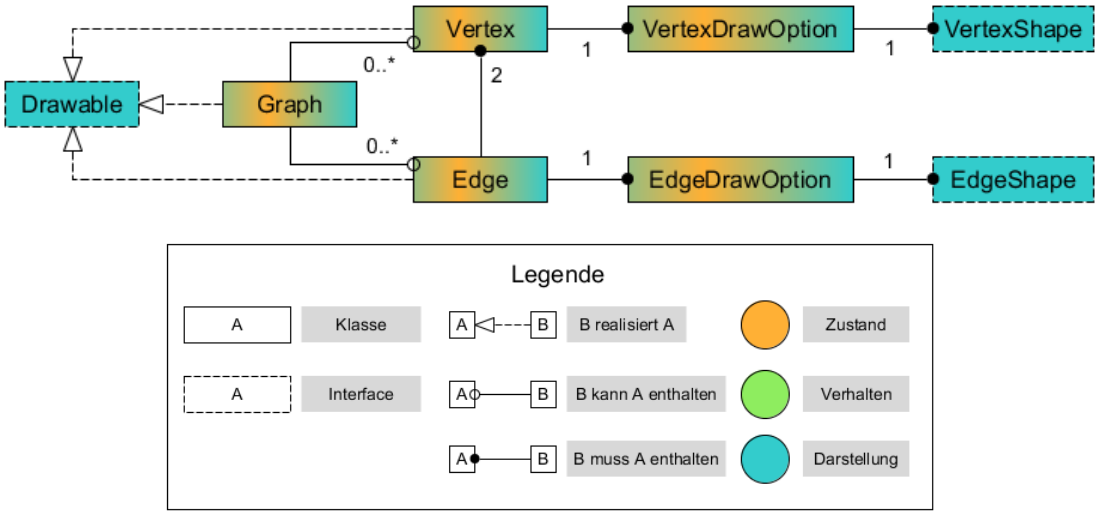


Abbildung 3.6: UML-Diagramm der Klassen und ihren Beziehungen zueinander

Die übergeordnete Klasse nennt sich Graph und speichert alle Elemente, die mathematisch zu einem Graphen gehören. Dabei werden Ecken durch die Klasse Vertex repräsentiert, Kanten durch die Klasse Edge. Darüber hinaus muss ein Graph wissen, ob er gerichtet oder ungerichtet ist. Dies ist wichtig, weil Graphen stets nur

eines von beiden sein können. Deshalb wird verhindert, dass gemischte Graphen mit Linien und Pfeilen entstehen. Die Klasse Graph speichert diese Information. Ausschlaggebend ist die erste Kante, die hinzugefügt wird. Werden alle Kanten entfernt und eine neue Kante hinzugefügt, gilt diese wiederum als erste Kante.

3.3.3 Wie wird ein Graph intern gespeichert?

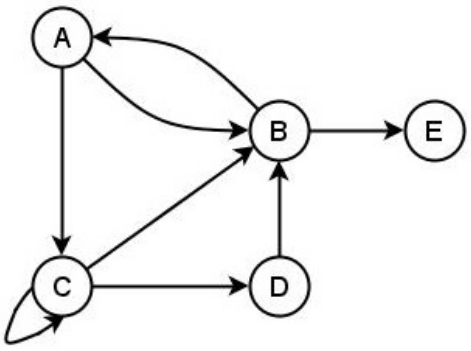
Nach der Definition eines Graphen muss festgelegt werden, wie die Zusammenhänge der Elemente gespeichert werden. Die Art der Speicherung muss es unter anderem erlauben Nachbarschaftsbeziehungen zwischen Ecken festzuhalten. Hierfür gibt es zwei bekannte Methoden.

Adjazenzmatrix

Weil ein Graph allein durch die Angaben der Kanten vollständig beschrieben werden kann, reicht eine Matrix aus, die die Verbindungen der Kanten symbolisiert. Eine solche Matrix wird Adjazenzmatrix genannt [24]. Sie wird in Form einer quadratischen Tabelle dargestellt.

x\y	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	0	1
C	0	1	1	1	0
D	0	1	0	0	0
E	0	0	0	0	0

(a) Adjazenzmatrix



(b) Beispielgraph

Abbildung 3.7: Adjazenzmatrix und eine grafische Darstellung des Graphen

Eine Eins stellt die Kante (x, y) von der Startecke x (links) zu der Endecke y (oben)

dar. Ist keine Kante vorhanden zwischen zwei Ecken, wird eine Null eingetragen. Gerichtete und ungerichtete Graphen belegen dabei den gleichen Speicherplatz.

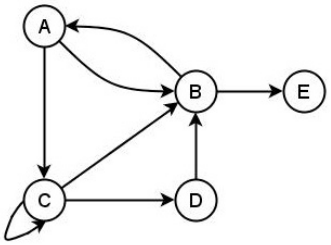
Der Speicherbedarf bei dieser Methode steigt quadratisch mit der Anzahl der Ecken an, unabhängig davon, wie viele Kanten es in dem Graphen gibt. Diese Matrix muss in einem übergeordneten Objekt gespeichert werden. Sowohl Ecken als auch Kanten kennen ihre Verbindungen zu anderen Elementen nicht. Wird ein Nachbar von einer Ecke gesucht, geschieht dies stets über diese Matrix. Soll eine neue Kante hinzugefügt werden, wird an der entsprechenden Stelle eine Eins eingetragen. Wird eine neue Ecke hinzugefügt, muss die Tabelle um eine neue Spalte und Zeile erweitert werden. Beim Entfernen einer Ecke und allen Kanten, die sie besitzt, muss die korrekte Zeile und Spalte gelöscht werden. Dadurch verschieben sich die übrigen Eintragungen. Dies kann je nach Implementierung kompliziert sein. In der Matrix alleine werden nur die Kanten gespeichert. Sie enthält keine Ecken, weshalb diese in einer gesonderten Liste aufgeführt werden müssen. Jede Ecke hat einen eindeutigen Index, welcher die entsprechende Zeile/Spalte in der Matrix repräsentiert. Eine Adjazenzmatrix besitzt immer eine gewisse Unsicherheit, weil die Zeilen und Spalten exakt zur Referenzliste passen müssen.

Ecke	Index
A	E1
B	E2
C	E3
D	E4
E	E5

(a) Referenzliste

x\y	E1	E2	E3	E4	E5
E1	0	1	1	0	0
E2	1	0	0	0	1
E3	0	1	1	1	0
E4	0	1	0	0	0
E5	0	0	0	0	0

(b) Adjazenzmatrix



(c) Beispielgraph

Abbildung 3.8: Referenzliste, Adjazenzmatrix und eine grafische Darstellung des Graphen

Adjazenzlisten

Eine Alternative zu einer Matrix sind verkettete Listen. Diese nennen sich Adjazenzlisten [23]. Dabei enthält jede Ecke eine Liste ihrer Nachbarn. Für gerichtete Graphen ist es notwendig getrennte Listen für Vorgänger und Nachfolger zu speichern. Eine ungerichtete Kante wird in beide Listen der Ecke eingetragen. Die Adjazenzlisten lassen sich aus einer Matrix ableiten.

x\y	A	B	C	D	E
A	0	1	1	0	0
B	1	0	0	0	1
C	0	1	1	1	0
D	0	1	0	0	0
E	0	0	0	0	0

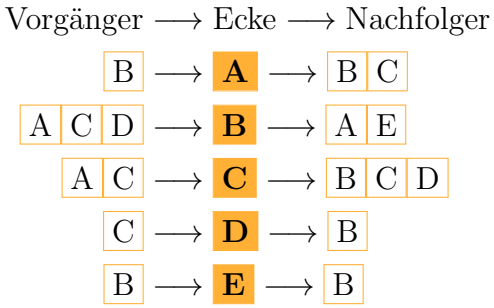
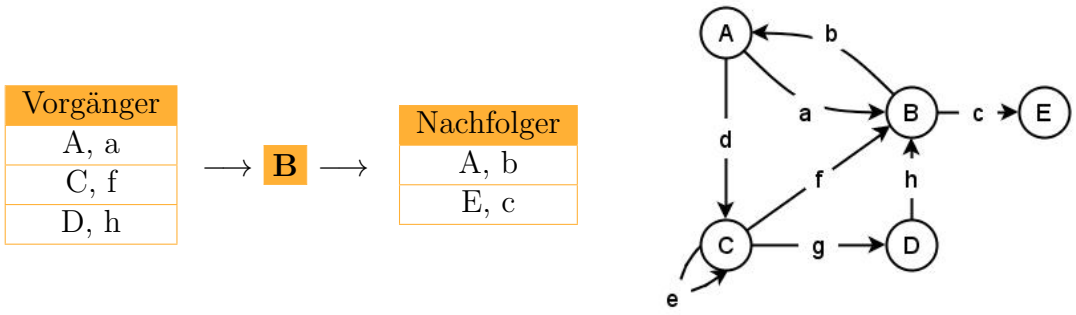


Abbildung 3.9: Herleitung der Adjazenzlisten

Durch diese Art der Speicherung ist es Ecken möglich, ihre Nachbarn direkt zu kennen. Somit können die meisten Algorithmen sehr nahe an ihrem Pseudocode implementiert werden, da oft mit benachbarten Ecken gearbeitet wird. Für diese Methode wird weniger Speicherplatz reserviert und es gibt keine überflüssigen Objekte (Null-Eintrag in der Matrix). Auch bei dieser Methode muss es ein übergeordnetes Objekt geben, welches alle Ecken und Kanten in Listen beinhaltet. Grund hierfür sind in erster Linie freie Ecken, die keine Kanten besitzen. Sie würden an keiner Stelle als Referenz eingetragen sein und somit nicht zum Graphen gehören. Im Gegensatz zu einer Matrix kennt das übergeordnete Objekt keine Zusammenhänge zwischen den Ecken und Kanten. Als Nachteil können die vielen Referenzen eines Elements genannt werden, die in vielen Objekten stets konsistent gehalten werden müssen. Wenn eine Ecke gelöscht werden soll, reicht es nicht aus, nur eine Referenz dieses Objektes zu löschen, auch alle anderen Elemente müssen ihre Referenzen darauf entfernen.

Implementierung

Die Wahl fiel auf die Adjazenzlisten, weil sie besser mit möglichem Pseudocode kombinierbar sind. Jede Ecke hat zwei Listen, die Paare von Ecken und Kanten besitzen. Die eine Liste zeigt auf alle Vorgänger, während die anderen auf alle Nachfolger zeigt. Ein Paar (v,e) besteht aus der Nachbarecke v und der Kante e, durch die sie mit der aktuellen Ecke verbunden ist. Es ist somit jederzeit möglich, sowohl auf alle benachbarten Ecken wie auch auf alle aus- bzw. eingehenden Kanten von einer Ecke zuzugreifen. Alle Ecken und Kanten werden zusätzlich separat in entsprechenden Listen in der Klasse Graph gespeichert.



(a) Nachbarschaftslisten **(b)** Erweiterter Beispielgraph

Abbildung 3.10: Adjazenzlisten der Ecke B

3.3.4 Wie wird eine Animation intern gespeichert?

Eine Animation ist die Aneinanderreihung von mehreren Graphen. Realisiert wird dies durch eine Liste, in der sich alle Graphen befinden. Die Reihenfolge der Graphen ist dabei wichtig. Die Folien auf der rechten Seite des Editors stellen die verschiedenen Graphen in einer Animation dar.

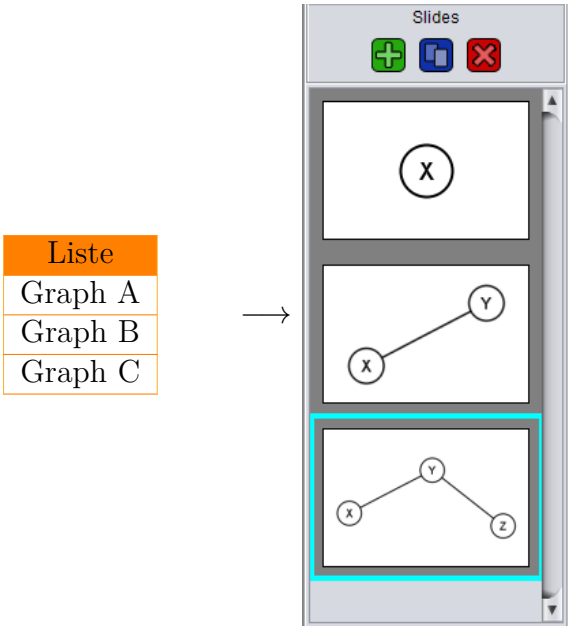


Abbildung 3.11: Die Liste der Graphen und ihre Folien

Über die Folien kann ein Graph ausgewählt werden. Es ist nur möglich, einen Graphen zur selben Zeit auf der Zeichenfläche abzubilden. Die Liste muss also so angelegt sein, dass immer nur ein Graph ausgewählt ist. Es muss möglich sein, an jeder Stelle eine neue Folie und somit einen neuen Graphen einzufügen. Das gleiche gilt für die Löschung eines Graphen. Gerade für die Veranschaulichung eines Algorithmus, indem nur die Farben angepasst werden, ist eine Klon-Funktion für eine Folie sinnvoll. Die geklonte Folie wird dann hinter der ausgewählten eingefügt.

Realisiert werden diese Funktionalitäten in der allgemein gehaltenen Klasse **Slides<E>**. Dies ist eine generische Klasse, weshalb E jede beliebige Klasse sein kann. Sie verwaltet eine Liste von Objekten der gewünschten Klasse. Innerhalb der Klasse existiert ein Marker, der auf das aktuelle ausgewählte Objekt in der Liste zeigt. Mit Hilfe dieses Markers und zusätzlichen Funktionen kann durch die Liste navigiert werden. Das Einfügen neuer Objekte und das Entfernen vorhandener wird ebenfalls über die Position des Markers gesteuert.

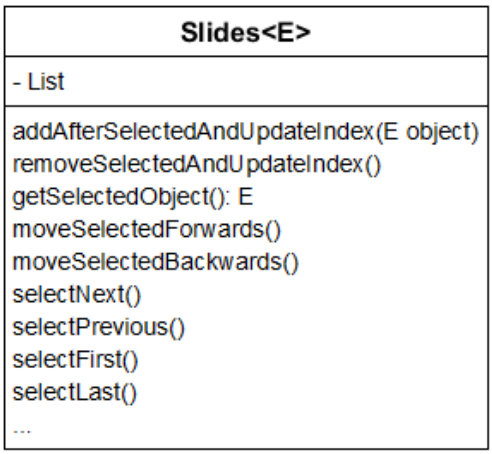


Abbildung 3.12: Auszug aus der Klasse Slides<E>

Eine Animation wird dann als Slides<Graph> gespeichert. Dies ist eine vereinfachte Beschreibung.[1] Weil die Klasse Slides an keine bestimmte andere Klasse gebunden ist, kann sie für andere Bereiche ebenfalls genutzt werden.

3.3.5 Wie wird ein Graph extern gespeichert?

Um einen gezeichneten Graphen oder eine Animation wiederverwenden zu können, muss es möglich sich, diese in einer Datei zu speichern. Innerhalb dieser Datei muss ein Format eingehalten werden, welches der Editor versteht, damit er den Graphen wiederherstellen kann. Es gibt dafür viele verschiedene Möglichkeiten.

Ein vorhandenes Format verwenden

Die am meisten verwendeten Formate wie *XML* benutzen eine eigene Programmiersprache. Da diese Sprachen weit verbreitet sind, könnten entsprechend aufgebaute Dateien später auch von andere Anwendungen verstanden werden. So könnte ein Graph eventuell auch in einem anderen Programm geladen werden. Durch diese

[1] Die genaue Darstellung entnehmen Sie dem Kapitel *Wie können Aktionen rückgängig gemacht werden?*

Vielseitigkeit kommt es zu sehr viel Quellcode, der in den Dateien benötigt wird, welcher im Verhältnis sehr wenig repräsentiert. Als Beispiel soll das Dateiformat *GraphML*[30] dienen, welches der Nachfolger von *GML* (Graph Modelling Language) ist. *GML* ist eine auf *XML*-basierende Sprache zur Beschreibung von Graphen.

Code-Ausschnitt 3.1: Programmierung eines Graphen in GraphML[30]

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://..."
    xmlns:xsi="http://..."
    xsi:schemaLocation="...">
  <graph id="G" edgedefault="undirected">
    <node id="A"/>
    <node id="B"/>
    <node id="C"/>
    <node id="D"/>
    <edge id="ab" source="A" target="B"/>
    <edge id="bc" source="B" target="C"/>
    <edge id="cd" source="C" target="D"/>
    <edge id="da" source="D" target="A"/>
  </graph>
</graphml>
```

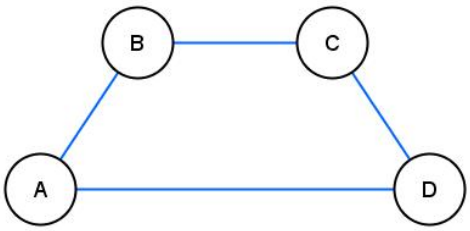


Abbildung 3.13: Resultierender Graph

Ein eigenes Format erstellen

Ein eigenes Format zu verwenden, benötigt mehr Zeit und Programmieraufwand. Dies lohnt sich, wenn bestimmte Kriterien von dem Editor erfüllt werden sollen, was durch ein vorgeschriebenes Format nicht erreicht werden könnte. Solche speziellen Formate, die für eine bestimmte Anwendung geschrieben wurden, können nur mit Aufwand in andere Systeme integriert werden. Sie bieten die Möglichkeit, die Repräsentation der Daten nach eigenen Vorstellungen anzupassen, sodass diese auf die Anwender bzw. Programmierer abgestimmt werden kann.

Implementierung

Dieser Editor verwendet ein eigenständiges Format. Es ist einfach gehalten, kann aber erweitert werden. Der Wunsch, einen Graphen nur durch seine rein mathematische Definition zeichnen zu lassen, wurde in der Form des Formats umgesetzt. Dabei hält sich die Beschreibung des Graphen sehr dicht an den Ecken- und Kantenmengen V und E. Das Aussehen und die Positionen der Ecken können zusätzlich gespeichert werden. Der Editor wird diese Werte immer automatisch mit speichern. Es folgt die kürzeste Definition des oben bereits beschriebenen Graphen.

Code-Ausschnitt 3.2: Einfachste Definition

```
edges
D, A
B, C
A, B
C, D
```

Diese einfache Auflistung der Kanten erzeugt automatisch die notwendigen Ecken. Alle Ecken werden dabei jedoch an dieselbe Position platziert, sodass alle übereinander liegen. Mit den integrierten Layouts des Editors können diese schnell neu angeordnet werden. Sowohl Kanten als auch Ecken werden hier mit einem Standard-Aussehen erzeugt, weil keine nähere Definition vorliegt. Möchte der Benutzer die Kanten und Ecken umgestalten, können zusätzliche Werte hinzugefügt werden. Beispielsweise werden die Kanten blau gefärbt, indem eine neue EdgeDrawOption (im

Code: edgetype) erzeugt wird, welche die Farbe (color: 0, 102, 255) setzt.

Code-Ausschnitt 3.3: Definition inklusive Aussehen

```
edges
D, A -- type: edgetype_1
B, C -- type: edgetype_1
A, B -- type: edgetype_1
C, D -- type: edgetype_1

vertices
A -- pos: 360, 280 -- type: vertextype_1
B -- pos: 440, 160 -- type: vertextype_1
D -- pos: 680, 280 -- type: vertextype_1
C -- pos: 600, 160 -- type: vertextype_1

edgetype
edgetype_1 -- color: 0, 102, 255 -- showText: false --
   shape: Normal -- stroke: 2.0 -- arrowsize: 20

vertextype
vertextype_1 -- size: 60 -- incolor: 255, 255, 255 --
   outcolor: 0, 0, 0 -- showText: true -- stroke: 2.0 --
   shape: Round
```

Ecken und Kanten werden bei diesem Format, genau wie bei der mathematischen Definition, in einzelne Mengen unterteilt. Ein Schlüsselwort leitet die darauffolgenden Zeilen ein. Eine Leerzeile signalisiert, dass die Menge dort aufhört. Die Reihenfolge der Mengen spielt keine Rolle, da stets nach dem Schlüsselwort gesucht wird. Der Inhalt der obigen Datei könnte auch folgendermaßen aussehen:

Code-Ausschnitt 3.4: Beliebige Reihenfolge der Mengen

```
vertextype
...
```

```
vertices
...

edgetype
...

edges
...
```

Alle Beispiele beziehen sich auf einen einzigen Graphen. Wenn eine Animation gespeichert werden soll, werden alle Graphen nacheinander in die Datei geschrieben mit jeweils einem Trennzeichen dazwischen. Es werden auch hierbei komplette Graphen und nicht nur die jeweilige Änderung zwischen diesen gespeichert.

3.3.6 Welche grafischen Anpassungen kann der Benutzer vornehmen?

Ein Graph kann auf sehr unterschiedliche Arten dargestellt werden. Sowohl die Ecken als auch die Kanten können verschiedene Formen annehmen. Je spezieller der Graph wird, desto spezieller werden auch die Darstellungen. Weil nicht jede denkbare Möglichkeit in diesem Editor zur Verfügung stehen kann, müssen entsprechende Einschränkungen erfolgen. Eine Ecke kann unendlich viele verschiedene Gestalten annehmen. Darunter fallen Form, Farbe, Beschriftung, Größe und Strichart. Neben diesen einfachen Eigenschaften kommen weitere hinzu, wenn zusätzliche Dinge angezeigt werden sollen, z.B. das Gewicht einer Ecke.

Abbildung 3.14: Beispiele für darstellbare Arten von Ecken

Bei den Kanten ist es ähnlich. Dort sind es besonders Farbe, Pfeilspitze, Strichstärke und der Linienzug, die die Darstellung ausmachen. Weil der Linienzug beliebig verlaufen kann, dies aber bei verschlungenen Linien sehr unübersichtlich wird, kann in diesem Editor nur eine gerade Linie oder eine einfache Kurve als Kante benutzt werden.

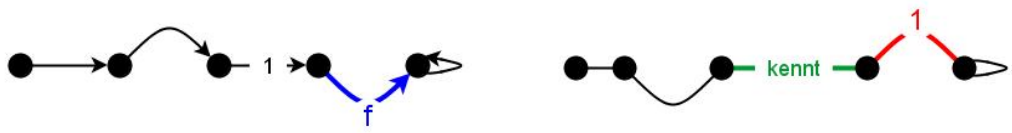


Abbildung 3.15: Beispiele für darstellbare Arten von Kanten

Aus diesen Überlegungen können zwei Listen erstellt werden, die wiedergeben, welche Einstellungsmöglichkeiten zur Verfügung stehen. Sie spiegeln die beiden Klassen VertexDrawOption und EdgeDrawOption wider. Nicht aufgelistete Optionen werden nicht unterstützt.

Ecken (VDO)

- Form
- Größe
- Innenfarbe
- Außenfarbe (Form und Schrift)
- Strichstärke
- Schrift (Art und Größe)
- anzeigen/ausblenden der Schrift

Kanten (EDO)

- Strichart
- Linienstärke
- Pfeilgröße
- Farbe
- Schrift (Art und Größe)
- anzeigen/ausblenden der Schrift

3.4 Verhalten und Funktionalität

3.4.1 Was ist ein automatisches Layout?

Nachdem ein Graph erstellt wurde, ist es möglich, ihn manuell neu anzuordnen. Dies gilt sowohl für die Ecken als auch für die Kanten. Um dem Benutzer diese Arbeit zu erleichtern oder sogar vollständig abzunehmen, muss der Editor in der Lage sein, vorgegebene Graphen passend anzuordnen. Passend ist relativ, da, wie bereits erwähnt, keine perfekte Anordnung für einen bestimmten Graphen existiert.

Bei automatischen Layouts wird zwischen zwei Gruppen unterschieden. Die eine Gruppe beinhaltet alle bekannten Layouts (Standardlayouts), die sich aus bestimmten Graphentypen ergeben. Darunter fallen unter anderem vollständige und bipartite Graphen. Diese Layouts sind relativ einfach zu implementieren, wenn der Algorithmus davon ausgehen kann, dass er einen entsprechenden Graphen als Grundlage übergeben bekommt. Deshalb werden diese Layouts nicht weiter behandelt.

In der anderen Gruppe finden sich eigenständige Verfahren, die darauf ausgelegt sind, jede Art von Graph anzuordnen. Die primären Ziele dabei sind, dass sich möglichst keine Kanten überschneiden und der Graph ausgewogen ist. Ausgewogen ist ein Graph, wenn die Abstände zwischen den Ecken überall gleich groß sind. Es gibt keine allgemein anerkannte Formel, um diese komplexen Probleme zu lösen. Eine bekannte Variante verwendet das sogenannte Kräftemodell, welches mit physikalischen Kräften arbeitet.

Alle Klassen, die ein solches Layout repräsentieren, sind Realisierungen des Interfaces **Layouter**. Sie funktionieren nach dem Prinzip des *Command-Entwurfsmuster* [6, 26]. Benötigte Werte werden im Konstruktor übergeben. Anschließend kann zu einem beliebigen Zeitpunkt das Layout mit einem ausgewählten Graphen ausgeführt werden.

3.4.2 Wie funktioniert ein kräftebasiertes Layout?

Grundprinzip

Das kräftebasierte Layout arbeitet mit dem Kräftemodell. Die Formeln sind der Physik entlehnt. Dabei werden Kräfte gegeneinander verrechnet und somit die Ecken verschoben [8, 12]. Dies geschieht in mehreren Durchläufen, so dass sich die Ecken schrittweise der endgültigen Position nähern. Ändern sich die Positionen der Ecken nach einem Durchlauf nicht mehr, wird der Algorithmus beendet. Eine Ecke stößt alle anderen Ecken in ihrer Nähe ab, während sie Ecken zu sich heranzieht, mit denen sie über eine Kante verbunden ist. Sind anziehende und abstoßende Kraft auf einer Ecke identisch, ändert sich die Position der Ecke nicht mehr. Die anziehende Kraft $(F(v,n))$ auf eine Ecke v berechnet sich aus der Summe aller anziehenden Kräfte durch benachbarte Ecken. Die abstoßende Kraft $(H(v,w))$ berechnet sich aus der Summe aller abstoßenden Kräfte von jeder anderen Ecke.

$$K(v) = \sum_{n \in N} F(v,n) + \sum_{w \in W} H(v,w)$$

$$N := \{n | n \in V \text{ und } n \text{ ist Nachbar von } v \text{ und } n \neq v\}$$

$$W := \{w | w \in V \text{ und } w \neq v\}$$

Weil eine exakte Übereinstimmung der anziehenden und abstoßenden Kräfte sehr unwahrscheinlich ist, kann es unter bestimmen Umständen zu einer Endlosschleife bei den Durchläufen kommen. Dabei entsteht ein *Wabern*, bei dem sich Ecken zwischen zwei Punkten hin- und her bewegen. Grund hierfür ist eine minimale Differenz zwischen Anziehung und Abstoßung einer Ecke. Dadurch wird die Ecke ein wenig zu weit herangezogen oder abgestoßen. Im nächsten Durchlauf wird die entsprechend entgegengesetzte Kraft, die geringer war, nun um den gleichen oder einen ähnlichen Betrag größer sein. Die Ecke erreicht deshalb nie einen ausgeglichenen Ruhezustand, an dem alle Kräfte annähernd null sind. Damit dies nicht geschieht, werden die Durchläufe begrenzt. Es kann je nach Implementierung möglich sein, Ecken zu markieren, die nicht verschoben werden sollen.

Federn und Gravitation

Die Kraft, die zwei Ecken zueinander zieht, kann beispielsweise durch die physikalische Formel für die Kraft einer Feder nach dem *Hooke'schen Gesetz* beschrieben werden [20]. Dabei stellen die Kanten zwischen den Ecken die Federn dar.

$$F = D \cdot (L_2 - L_1)$$

$$\text{Federkraft} = \text{Federkonstante} \cdot (\text{Neue Federlaenge} - \text{Originallaenge})$$

Für die abstoßende Kraft kann das Gravitationsgesetz herangezogen werden [20]:

$$F = G \cdot \frac{m_1 \cdot m_2}{r^2}$$

$$\text{Kraft} = \text{Gravitationskonstante} \cdot \frac{\text{Masse}_1 \cdot \text{Masse}_2}{\text{Entfernung der Punkte}^2}$$

Das Gesetz berechnet die anziehende Kraft zwischen zwei Körpern. Wenn diese Formel für die abstoßende Kraft verwendet werden soll, muss deshalb die Kraft umgekehrt werden.

Die beiden zu berechnenden Kräfte werden als Vektoren gespeichert. $\overrightarrow{F}(v,n)$ stellt die anziehende Kraft zwischen der aktuellen Ecke v und einer benachbarten Ecke n dar.

$$\overrightarrow{F}(v,n) = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} F_x(v,n) \\ F_y(v,n) \end{pmatrix}$$

$$F_x(v,n) = D \cdot (\triangle d - L) \cdot \frac{\triangle x}{\triangle d} \cdot Z$$

$$F_y(v,n) = D \cdot (\triangle d - L) \cdot \frac{\triangle y}{\triangle d} \cdot Z$$

L = Federlänge
D = Federstärke
$Z = 0{,}5$ = Dämpfung
$\triangle d$ = (Euklidischer) Abstand zwischen v und n
$\triangle x = x_n - x_v$
$\triangle y = y_n - y_v$

Die Originalformel wurde durch eine Dämpfung Z ergänzt und die Berechnung der Vektorstärke in die x bzw. der y-Komponente aufgeteilt. Die Dämpfung dient dazu,

dass die Ecken nicht in zu großen Schritten verschoben werden. $\triangle d$ entspricht in dieser Formel der neuen Federlänge L_2. Die Federkonstante D spiegelt die Stärke der Feder wider. Die Kraft $\overrightarrow{F}(v,n)$ wird nur berechnet, wenn v und n über eine Kante miteinander verbunden sind, andernfalls beträgt ihr Wert 0.

Bei der Gravitationsformel wird angenommen, dass die Massen m_1 und m_2 keine Rolle spielen (alle Ecken sind gleich schwer) und deshalb beide den Wert 1 haben und somit aus der Formel verschwinden. Die Kraft $\overrightarrow{H}(v,w)$ wird zwischen jedem Eckenpaar (v,w) berechnet, unabhängig davon, ob die Ecken miteinander verbunden sind oder nicht.

$$\overrightarrow{H}(v,w) = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} H_x(v,w) \\ H_y(v,w) \end{pmatrix}$$

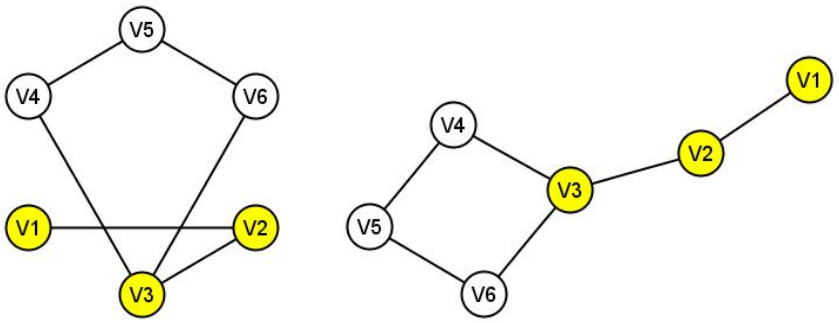
$$H_x(v,w) = -\frac{G}{\triangle d^2} \cdot \frac{\triangle x}{\triangle d} \cdot Z$$

$$H_y(v,w) = -\frac{G}{\triangle d^2} \cdot \frac{\triangle y}{\triangle d} \cdot Z$$

G = Stärke der Abstoßung
$Z = 0{,}5$ = Dämpfung
$\triangle d$ = (Euklidischer) Abstand zwischen v und w
$\triangle x = x_w - x_v$
$\triangle y = y_w - y_v$

Vorteile

In der Regel ist $\overrightarrow{H}(v,w)$ größer als $\overrightarrow{F}(v,n)$. Dies hat den positiven Effekt, dass Teile eines Graphen, die eine Reihe von zusammenhängenden Ecken besitzen, als Linie angeordnet werden. In diesem Beispiel betrifft dies die Ecken *V1*, *V2* und *V3*.



(a) Original **(b)** Nach dem Layout

Abbildung 3.16: Ein Graph vor und nach dem Layout

Nachteile

Die anziehende Kraft $\overrightarrow{F}(v,n)$ kann negativ werden, wenn die Ecken näher beieinander sind, als die Federlänge lang ist ($\triangle d - L$). Dies gilt es zu verhindern, da sonst eine zusätzliche abstoßende Kraft erzeugt wird.

Bei der abstoßenden Kraft tritt das Problem auf, dass sie niemals 0 werden kann. Es findet somit immer eine minimale Abstoßung statt, auch wenn zwei Ecken sehr weit voneinander entfernt sind. Durch Abrunden des Wertes oder Begrenzung der Reichweite kann so ein Verhalten größtenteils unterbunden werden.

Die Formeln erfordern viele Variablen, die eingestellt werden können (L, D, G, Z). Für den Benutzer sollte es nicht nötig sein, die Implementierung zu kennen, um zu verstehen, welche Variable welche Auswirkung hat. Eine Reduzierung der Variablen wäre deshalb sinnvoll. Erstrebenswert wären zwei Werte: einer für die Stärke der Abstoßung und einer für die Stärke der Anziehung. Die Dämpfung Z kann fest vorgegeben werden, die anderen drei Werte werden aber für die Formeln benötigt, um das Verhalten zu steuern. Zudem sind die Wertebereich von D und G sehr unterschiedlich, da $D \cdot (\triangle d - L)$ proportional steigt und $-G/\triangle d^2$ exponentiell fällt. Für D werden sehr kleine Werte ($\sim 0,8$) benötigt, während G mit Werten im Bereich (~ 40.000) arbeitet. Die Formel für $H_x(v,w)$ könnte so anpasst werden, sodass G quadriert wird und der Wertebereich bei ~ 200 liegt. Entsprechend sieht die Formel für $H_y(v,w)$ aus.

$$H_x(v,w) = -\frac{G^2}{\triangle d^2} \cdot \frac{\triangle x}{\triangle d} \cdot Z$$

Die Annäherung der Werte hat zur Folge, dass sehr viele Zwischenwerte nicht mehr einstellbar sind. Für den Benutzer ist es auch bei dieser Wertespanne nicht zu erkennen, warum der eine Wert sehr klein und der andere sehr groß sein muss, um ausgeglichene Kräfte zu erhalten.

Eigene Formel

Um einige der Nachteile der oben beschriebenen Formeln auszugleichen, wurde eine eigene abgewandelte Formel entwickelt. Die zwei Kräfte werden dabei nicht auf Grundlage des Abstandes zwischen den Ecken zu berechnet. Stattdessen gibt es

einen einstellbaren Idealabstand zwischen den Ecken, der eingehalten werden muss. Es werden beide Kräfte berechnet, aber wenn beide auf eine Ecke wirken, wird eine von ihnen 0, wodurch nur eine Kraft ihre Wirkung zeigt.

(a) A und B sind verbunden

(b) A und B sind nicht verbunden

Abbildung 3.17: Berechnung der Kräfte, Abstoßung (rot) und Anziehung (blau), je nach Situation. Das schwarze Quadrat markiert den Idealabstand zu A.

Die Formeln basieren ebenfalls auf dem Gesetz der Federkraft. $\overrightarrow{F}(v,n)$ und $\overrightarrow{H}(v,w)$ sind wieder als Vektoren definiert und in ihre Komponenten unterteilt.

$$F_x(v,n) = (\triangle d - L) \cdot \frac{\triangle x}{\triangle d} \cdot Z$$

$$F_y(v,n) = (\triangle d - L) \cdot \frac{\triangle y}{\triangle d} \cdot Z$$

$$H_x(v,w) = -(L - \triangle d) \cdot \frac{\triangle x}{\triangle d} \cdot Z$$

$$H_y(v,w) = -(L - \triangle d) \cdot \frac{\triangle y}{\triangle d} \cdot Z$$

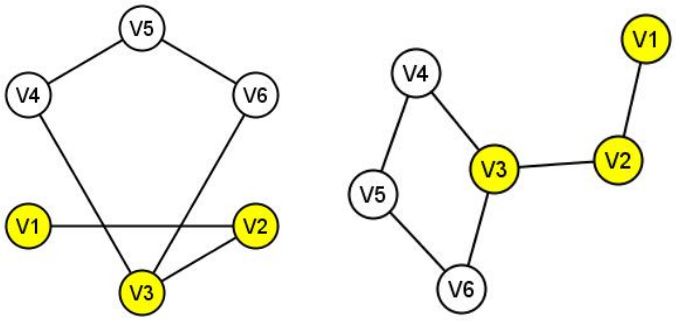
L = Federlänge
$Z = 0{,}5$ = Dämpfung
$\triangle d$ = (Euklidischer) Abstand zwischen v und n bzw. w
$\triangle x = x_{n/w} - x_v$
$\triangle y = y_{n/w} - y_v$

$\overrightarrow{F}(v,n)$ wird nur berechnet, wenn die Ecken miteinander verbunden sind und $\triangle d$ größer ist als L, somit wird es niemals negativ. Umgekehrt wird $\overrightarrow{H}(v,w)$ nur berechnet, wenn L größer ist als $\triangle d$. Dadurch berechnen beide Formeln denselben Wert. Es würde demzufolge ausreichen, nur die Formeln von $\overrightarrow{F}(v,n)$ zu verwenden. Würde das Ergebnis negativ ausfallen, würde die Ecke abgestoßen, andernfalls angezogen

werden.

Nachteile

Diese Formeln ordnen Ecken, die wie eine Kette miteinander verbunden sind, nicht zwingend in eine Linie an. Dafür werden die Ecken gleichmäßiger platziert.

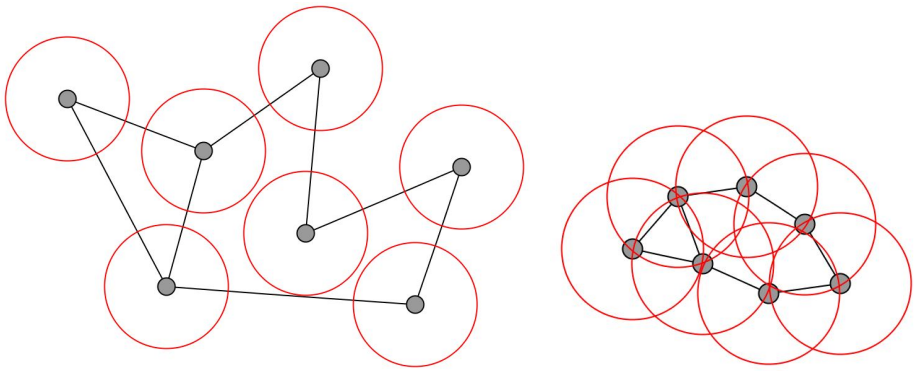


(a) Original **(b)** Nach dem Layout

Abbildung 3.18: Ein Graph vor und nach dem Layout

Vorteile

Bei der abgewandelten Formel muss der Benutzer nur die Federlänge L verändern. Sie repräsentiert den Idealabstand. Dieser kann zudem gut grafisch dargestellt werden, um dem Benutzer zu zeigen, welchen Wert er eingestellt hat.



(a) Originalgraph **(b)** Nach dem Layout

Abbildung 3.19: Idealabstand grafisch veranschaulicht

Beide Kräfte wachsen proportional, wodurch keine zusätzliche Werteanpassung erfolgen muss. Weil eine Kraft 0 wird, wenn beide wirken, wird schlussendlich nur eine einzige Kraft berechnet. Die abstoßende Kraft wirkt nur dann, wenn sich eine Ecke zu nahe an einer anderen befindet (näher als Idealabstand). Die entsprechende Ecke wird nur so weit abgestoßen, bis sie diesen Abstand erreicht. Somit kommt es nicht vor, dass Ecken über diesen Abstand hinaus verschoben werden.

Ein zusätzlicher Nebeneffekt ist, dass diese Berechnung schneller ist. Sie benötigt bei den meisten Graphen nur die Hälfte der Durchläufe und damit die Hälfte der Zeit. Je nach Anordnung und Kantenanzahl eines Graphen, kann die eine Variante oder die andere das Ergebnis schneller liefern. Sehr große Graphen lassen den Algorithmus meistens mit Ablauf der maximalen Durchgänge abbrechen.

3.4.3 Wie funktioniert die Interaktion?

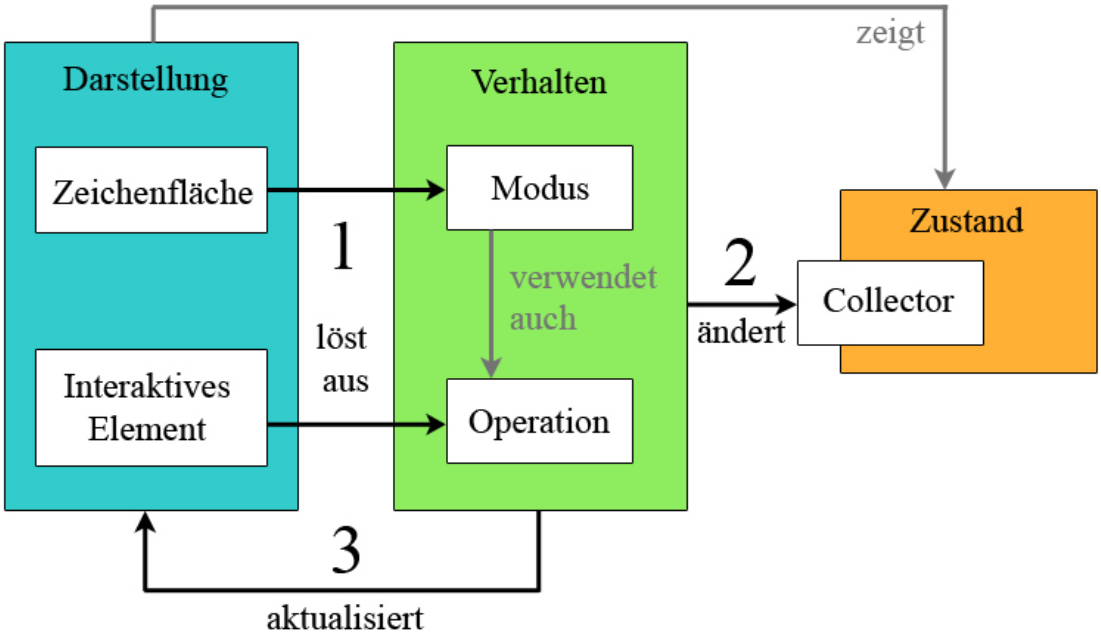


Abbildung 3.20: Diagramm zum Ablauf einer Interaktion

Es wurde versucht, das Verhalten in gesonderten Klassen zu kapseln, sodass nur diese Klassen den internen Zustand verändern. Es gibt nur zwei Klassen, die befugt sind, direkt auf die Container-Klasse Collector zuzugreifen, welcher die Schnittstelle zu allen anderen Zustands- und Darstellungsklassen repräsentiert. Die Interaktion ist ein Zusammenspiel von den Bereichen der Darstellung und des Verhaltens (Abb. 3.20). Der Benutzer kann nur mit Elementen interagieren, die er sehen kann. Es gibt zwei interaktive Gruppen. Die eine enthält ausschließlich die Zeichenfläche, die andere alle anderen interaktiven Elemente wie Buttons oder Regler. Je nach Gruppe werden andere Klassen für die Ausführung von Aktionen aufgerufen. Alle Aktionen, die auf der Zeichenfläche stattfinden, werden von verschiedenen Subklassen der Klasse Modus gesteuert (Schritt 1). Ein Modus dient nur diesem Zweck. Die meisten Modi besitzen zusätzliche Einstellungsmöglichkeiten und bilden ein eigenes Subsystem von mehreren Klassen. Die andere Gruppe mit den übrigen interaktiven Elementen benutzt Realisierungen des Interfaces Operation. In der Regel wird ein in-

teraktives Element mit einer Operation in Verbindung gesetzt, sodass, beispielsweise bei einem Klick auf einen Button, die entsprechende zugewiesene Operation ausgeführt wird (Schritt 1). Modi und Operations enthalten kleine Code-Abschnitte, die zur Änderung des Zustands dienen (Schritt 2). Nach erfolgreicher Änderung werden die Darstellungen aktualisiert (Schritt 3).

3.4.4 Was ist eine Operation?

Eine **Operation** kapselt Verhalten, welches einzelne Aktionen ausführt oder den Zustand des Editors verändert. Sie bilden an vielen Stellen im Editor die Schnittstelle, um interne Daten anzupassen. Auf diese Weise wird verhindert, dass die zustandsändernden Klassen eine direkte Referenz auf die Originalklassen erhalten und somit mögliches nicht nachvollziehbares Verhalten erzeugt wird. Eine Operation hat eine einzige Funktion, die durch ihren Namen beschrieben ist. Für weitere Funktionen gibt es je eine eigene Operation. Dadurch sind die Klassen sehr klein und übersichtlich, was aber zu einer großen Anzahl von Klassen führt. Die Realisierungen dieser Klasse funktionieren nach dem Prinzip des *Command-Entwurfsmusters* [6, 26]. So können Operations ausgetauscht und zu einem späteren Zeitpunkt nach der Erstellung ausgeführt werden. Alle benötigten Werte und Objekte werden einer Operation bei ihrer Erstellung im Konstruktor übergeben und können später nicht mehr geändert werden. Dadurch wird ein konstantes Verhalten erzeugt, wenn die Operation mithilfe der Funktion *run()* ausgeführt wird. Sie sind so programmiert, dass sie in den meisten Fällen von keiner anderen Operation abhängen. Weil das Interface nur die Funktion *run()* vorgibt, können neue Operations schnell und einfach hinzugefügt werden.

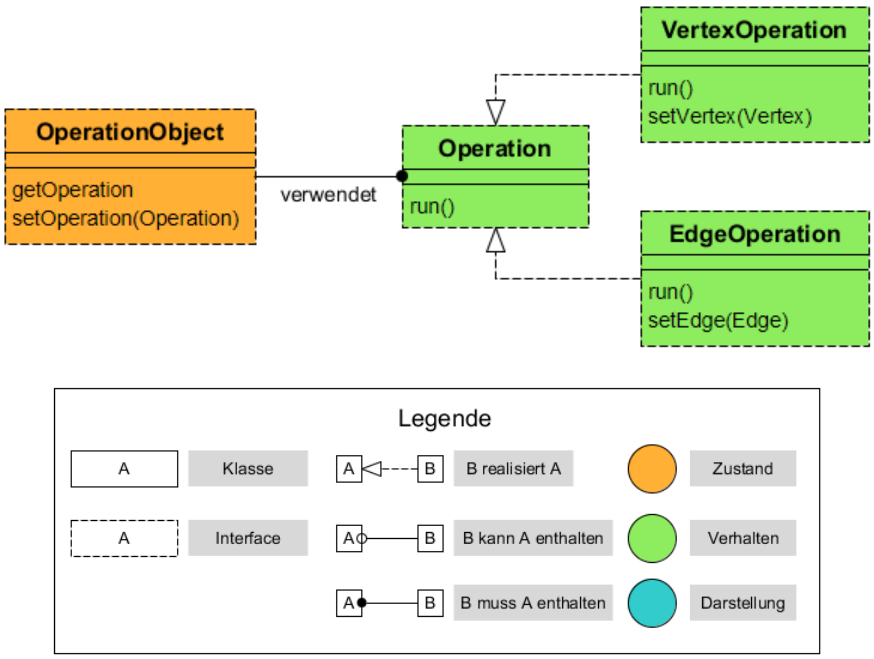


Abbildung 3.21: UML-Diagramm aller Operation-Interfaces.

Durch ihre Kompaktheit sind Operations vielfältig einsetzbar. Klassen, die eine Operation speichern, implementieren das Interface **OperationObject**. Auf diese Weise kann eine Operation gesetzt oder auch zurückgegeben werden. Beispielsweise besitzen die meisten Buttons eine Operation, die ausgeführt wird, wenn der Benutzer auf diesen Button klickt. Sie können aber auch für einzelne Ausführungen neu erstellt und direkt ausgeführt werden.

Das Kontextmenü arbeitet ebenfalls mit Operations. Für dieses wurde eine Ausnahme gemacht. Weil alle Operations des Kontextmenüs direkt einen Bezug zu einer bestimmten Ecke oder Kante herstellen, kann das jeweilige Objekt auch nach der Erstellung (Konstruktor) der Operation gesetzt werden. Dafür stehen die Interfaces **EdgeOperation** und **VertexOperation** zur Verfügung, die das Interface Operation erweitern.

3.4.5 Was ist ein Modus?

Ein **Modus** repräsentiert das Verhalten des Editors, wenn der Benutzer auf der Zeichenfläche eine Mausaktion durchführt. Darunter zählen Mausbewegungen sowie Mausklicks. Jeder Maustaste kann ein eigener Modus zugeordnet werden. Dabei wird nur die linke und rechte Maustaste unterstützt, da besonders bei Laptops die mittlere Maustaste nicht verwendet werden kann. Die Erkennung der jeweiligen Maustaste ist nicht Bestandteil der Klasse Modus. Ein Modus kann im Sinne des *Strategy-Entwurfsmusters* [6, 26] zur Laufzeit ausgetauscht werden.

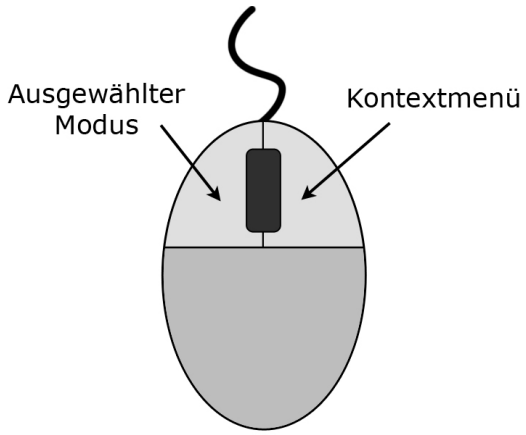


Abbildung 3.22: Belegung der Maustasten

Die abstrakte Superklasse Modus dient als Vorlage für konkrete Implementierungen. Sie ist kein Interface, weil bei einer abstrakten Klasse leeres Verhalten implementiert werden kann und dies in einem Interface nicht möglich ist. Eine abstrakte Superklasse erleichtert das Erstellen neuer Modi, weil nur die Mausaktionen überschrieben werden müssen, die benötigt werden. So ist es ebenfalls möglich weitere Funktionen in die Klasse Modus einzubauen, ohne die Implementierung der bereits vorhandenen Subklassen anpassen zu müssen. Alle konkreten Implementierungen von Modus sind *Singletons*[6, 26]. Da ein Modus über viele verschiedene Orte innerhalb des Editors gewechselt werden kann (Menüleiste, Toolbar, Kontextmenü), das Verhalten und die getroffenen Einstellungen aber gleich bleiben sollen, muss jede Referenz immer auf dasselbe Objekt zeigen.

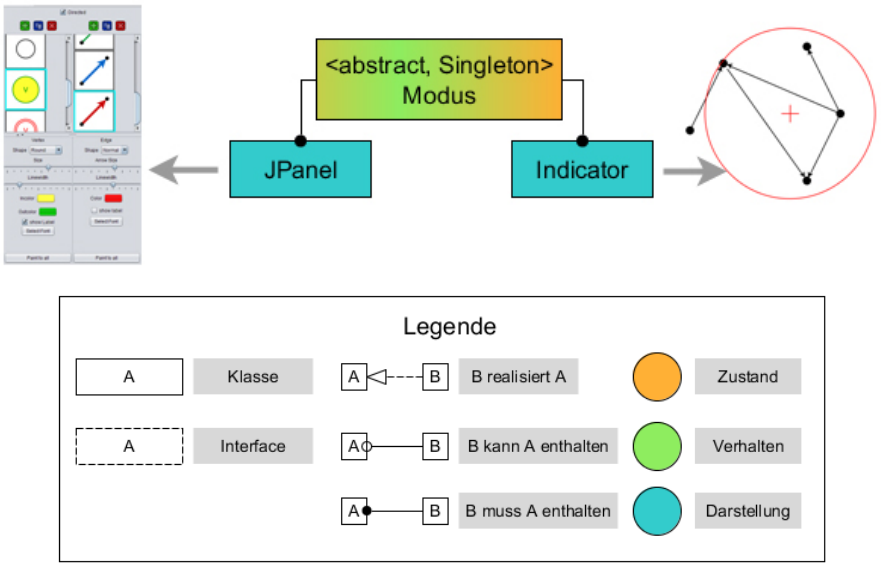


Abbildung 3.23: Subsystem der Klasse Modus

Ein Modus unterteilt sich in die drei Bereiche Verhalten, Zustand und Darstellung. Innerhalb einer Subklasse von Modus ist das Verhalten definiert, was genau passieren soll, wenn der Benutzer eine Mausaktion ausführt. Abhängig vom jeweiligen Modus besitzt dieser unterschiedlich viele Einstellungsmöglichkeiten, die gespeichert werden müssen. Um den aktuellen Zustand anzuzeigen, besitzt ein Modus weitere Klassen vom Typ JPanel und Indicator. Das JPanel ist die Optionsbox auf der linken Seite des Editors. Dort können Einstellungen vorgenommen werden, sodass das Verhalten des Modus angepasst wird. Durch die internen Listener interaktiver Objekte im JPanel wird hier ebenfalls Verhalten beschrieben. Die Ausführung des Verhaltens wird aber in der Regel in einem Objekt der Klasse Operation ausgelagert. Der **Indicator** ist eine Klasse, welche gezeichnet werden kann. Er wird auf der Zeichenfläche dargestellt und spiegelt die Mausaktionen des Benutzers wider. Durch ihn wird dem Benutzer gezeigt, in welchem Umfang der aktuelle Modus Einfluss nimmt. Der Indicator ist üblicherweise rot und verwendet einfache geometrische Formen zur Veranschaulichung. Die Werte, die für einen Modus eingestellt werden können, werden in der Subklasse des Modus gespeichert. Es existiert sowohl ein *Null-Object*[6, 26] für das JPanel (ein leeres) als auch für den Indicator (zeichnet nichts). Somit wird die Problematik um-

gangen, an verschiedenen Stellen im Code die Referenzen darauf zu überprüfen, ob sie *null* sind.

Dem Modus wird vom Collector ein Objekt der Klasse **Optimizer** angeboten, mit dessen Hilfe es möglich ist, die aktuelle Mausposition auf bestimmte vorgeschriebene Positionen abzubilden. Dies geschieht, wenn der Benutzer das Raster (Grid) aktiviert. Dabei wird die Mausposition des Benutzers auf die Schnittpunkte der Rasterlinien beschränkt. Erst nachdem die neue angepasste Mausposition bestimmt wurde, werden die Aktionen vom Modus durchgeführt. Der Optimizer kann ebenfalls jederzeit ausgetauscht werden. Intern ist er ähnlich wie der Modus aufgebaut. Er besitzt einen Indicator (Raster auf der Zeichenfläche) und ein JPanel (Buttons), welches beim Laden des Editors der Toolbar hinzugefügt wird. Die Nutzung dieser vorgeschalteten Klassen ist nicht in jeder Funktion nötig oder sinnvoll. Beispielsweise soll es trotz des Rasters möglich sein, zwischen zwei Ecken, die nicht auf Schnittpunkten liegen, eine Kante zu erstellen.

Ein Modus kann verschiedenste Ausprägungen haben. Innerhalb des Editors ist nicht nur die Erstellung eines Graphen, das Verschieben oder Anmalen, sondern auch ein Layouter ein Modus. Auch das Kontextmenü ist ein Modus, der mit der rechten Maustaste verknüpft ist. Weil dieser Modus nicht mit der linken Taste verknüpft ist, wird das interne JPanel nicht als Optionsbox angezeigt. Aus diesem Grund besitzt das Kontextmenü kein JPanel, dafür jedoch mehrere Indikatoren; einen für das Menü an einer leeren Stelle, einen für eine Ecke und einen für eine Kante. Je nachdem, was bei einem Klick getroffen wurde, wird der entsprechende Indicator angezeigt. Ein Indicator des Kontextmenüs besitzt mehr Funktionalität als die der Modi der linken Maustaste. Er beinhaltet mehrere Bilder, die mit je einer Operation verknüpft sind. Wenn der Benutzer über einem Bild die Maustaste loslässt, wird die entsprechende Operation auf das gewählte Objekt (Ecke, Kante) ausgeführt. Weil die Auswahl des gewünschten Objektes auf der Zeichenfläche per Rechtsklick geschieht, ist es nicht möglich, diese Aktionen über andere Bereiche (Menüleiste, Toolbar) des Editors aufzurufen, da nicht klar wäre, auf welchem Objekt die Operation angewendet werden soll.

3.4.6 Wie kann eine Aktion rückgängig gemacht werden?

In den meisten Programmen ist es möglich, eine Aktion rückgängig zu machen. Dies sollte auch in *WildGraphs* möglich sein. Der Benutzer kann über entsprechende Buttons eine Aktion rückgängig machen oder wiederholen.

Entgegengesetzte Operations

Die Umsetzung einer entsprechenden Funktionalität wird üblicherweise immer auf die gleiche Weise implementiert. Ausgehend von dem aktuellen Status werden zwei Listen als Warteschlangen festgelegt. Eine beinhaltet alle vergangene Schritte bzw. Aktionen (Undo-Liste). Die andere enthält alle Schritte, die wiederholt werden können, sollte der Benutzer einige Schritte rückgängig gemacht haben (Redo-Liste).

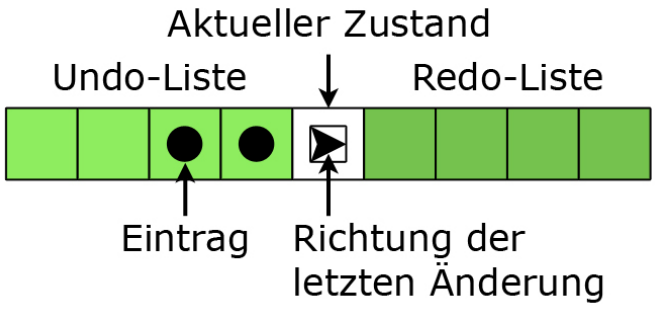


Abbildung 3.24: Erläuterung der Listen und deren Elemente

Wird eine Aktion ausgeführt, wird die entgegengesetzte Aktion in die Undo-Liste eingetragen und der aktuelle Status auf den Status nach der Ausführung der Aktion gesetzt (Abb. 3.25). Soll ein Schritt rückgängig gemacht werden, wird die entgegengesetzte Aktion in die Redo-Liste eingetragen und der Schritt ausgeführt (Abb. 3.26). Wenn der Benutzer einige Aktionen rückgängig gemacht hat und mit diesem Zustand weiterarbeitet, indem er eine Aktion ausführt, ist es nicht mehr möglich, die zuvor ausgeführten Schritte zu wiederholen (Abb. 3.27).

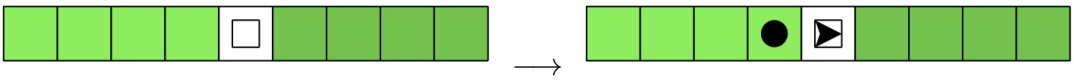


Abbildung 3.25: Die Listen sind leer und eine neue Aktion wird ausgeführt.

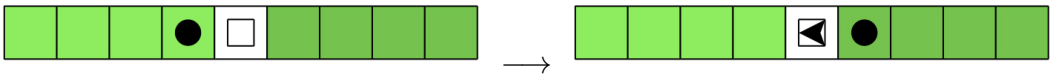


Abbildung 3.26: Die letzte Aktion wird rückgängig gemacht.

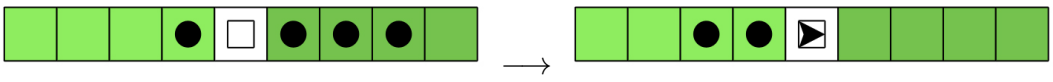


Abbildung 3.27: Der Benutzer hat mehrere Schritte rückgängig gemacht und führt dann eine neue Aktion aus.

Das oben beschriebene Interface Operation kapselt bereits einzelne Aktionen. Deshalb liegt es nahe, entsprechend entgegengesetzte Operations (Undo-Operations) zu erstellen, die das Gegenteil verursachen. Sie würden sowohl in die Undo- als auch in die Redo-Listen eingetragen. Die Struktur des *Command-Entwurfsmusters* [6, 26] eignet sich für diese Implementierung sehr gut. Eine Operation würde bei ihrer Erstellung gleich die entsprechende entgegengesetzte Operation mit erstellen und eine Referenz darauf speichern. Von Außen kann diese Referenz abgefragt werden, damit sie ausgeführt werden kann.

Auch wenn die entsprechenden Klassen dafür geeignet sind, gibt es viele Aspekte, die gegen diese Implementierung sprechen. Eine Operation sollte möglichst nicht von anderen Operations abhängen, damit dies bei Änderungen nicht zu Komplikationen führt. Die Alternative, jeder Operation eine zusätzliche Funktion für das Rückgängigmachen zu implementieren, widerspricht dem Prinzip, dass der Name einer Operation gleichzeitig ihre Implementierung widerspiegelt. Es würden auf diese Weise zwei Funktionalitäten in einer Operation beschrieben werden. Deshalb sollte eine Operation nicht wissen, wie sie rückgängig gemacht werden kann. Bei vielen Operations ist eine Undo-Funktion nicht nötig, weil es keine Möglichkeit gibt, sie rückgängig zu machen. Beispiele sind das Speichern des Graphen oder das Erstellen eines zufälligen

Graphen. Nur Operations, die einen Graphen direkt beeinflussen, können rückgängig gemacht werden, beispielsweise das Hinzufügen einer Kante. Würde das Interface Operation mit einer Funktion versehen werden, die die Undo-Operation zurückgibt, würden viele Operation eine leere Operation (*Null-Object* [6, 26]) oder einfach *null* zurückgeben. Dies würde zu vielen Zeilen überflüssigem und toten Quellcode führen. Für künftige Erweiterungen wäre dieses Vorgehen zudem sehr undurchsichtig. Weil die Operations durch ihre kurzen Funktionen sehr vielzählig sind, würden ihre Anzahl durch zusätzliche weitere Klassen, die ausschließlich dafür programmiert wurden, um andere Operations rückgängig zu machen, noch weiter wachsen. Das größte Problem stellen aber die Modi dar, denn diese sind hauptverantwortlich für die Veränderung eines Graphen. Allerdings verwenden sie keine Operations, sondern implementieren direkt ihr Verhalten. Deshalb wäre es nicht möglich, an dieser Stelle eine Undo-Operation zu erstellen ohne von der festgelegten Verwendungsart abzuweichen und eine Ausnahme zu machen.

Komplette Graphen zwischenspeichern

Statt eine Undo-Funktion in den Warteschlangen zu speichern, kann auch der komplette Graph vor der entsprechenden Änderung in die Listen eingetragen werden. Somit würde jeder Eintrag nur vollständig beschriebene Graphen enthalten. Auf diese Weise müsste eine Operation nicht wissen, wie sie rückgängig gemacht werden kann oder welche entsprechende andere Operation das Gegenteil erzeugt. Die Zuständigkeiten *Aktion ausführen* und *Aktion rückgängig machen* wären voneinander entkoppelt. Auch zukünftige komplexe Aktionen müssten sich nicht darum kümmern, wie sie umgekehrt werden können. Als Beispiel kann die Aktion *alle Ecken einfärben* dienen. Sie führt mehrere Änderungen am Graphen durch, die alle gleichzeitig rückgängig gemacht werden müssten. Durch das Speichern des kompletten Graphen vor diesen Änderungen kann der gewünschte vorherige Zustand exakt wieder rekonstruiert werden. Im Gegensatz zu einzelnen Operations würde bei dieser Methode mehr Speicherplatz benötigt werden, da ein Objekt der Klasse Graph größer ist als eine Operation.

Implementierung

Wegen der vielen Nachteile und den Problemen, die durch Undo-Operations entstehen würden, wurde die zweite Variante bevorzugt und umgesetzt.

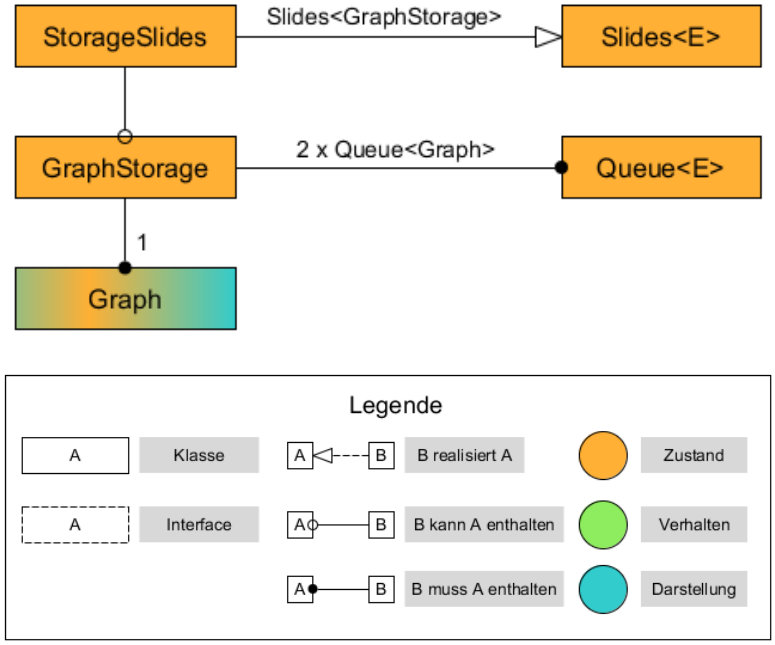


Abbildung 3.28: UML Diagramm der Klassen zur Speicherung einer Animation inklusive der Möglichkeit, Aktionen rückgängig zu machen

Die Implementierung erweitert die bereits erwähnte Klasse Slides<E>. Zunächst wurde die Klasse **Queue<E>** entworfen, die eine Warteschlange mit Elementen vom Typ E repräsentiert. Sie ist allgemein gehalten und funktioniert mit einer beliebigen Klasse. Deshalb kann sie in anderen Bereichen oder Programmen wiederverwendet werden. Anschließend wurde die Containerklasse **GraphStorage** erstellt, die ein Objekt vom Typ Graph und zwei Warteschlangen vom Typ Queue<Graph> beinhaltet. Die eine Warteschlange ist für vergangene Status (Undo) und die andere für zukünftige (Redo) vorgesehen. Um die Slides<E> mit der Klasse GraphStorage zu vereinen, wurde die Klasse **StorageSlides** erstellt, die von Slides<GraphStorage> erbt. StorageSlides enthält eine Liste von Graphen, die wiederum durch je zwei Warteschlangen ergänzt werden. Auf diese Weise ist es möglich, vorgenommene Ände-

rungen ausschließlich dem aktuell ausgewählten Graphen zuzuordnen. Umgekehrt wird nur eine vergangene Version des aktuellen Graphen durch den Undo-Button wiederhergestellt. Andere Graphen in der Liste sind davon nicht betroffen und führen ihre eigenen Warteschlangen.

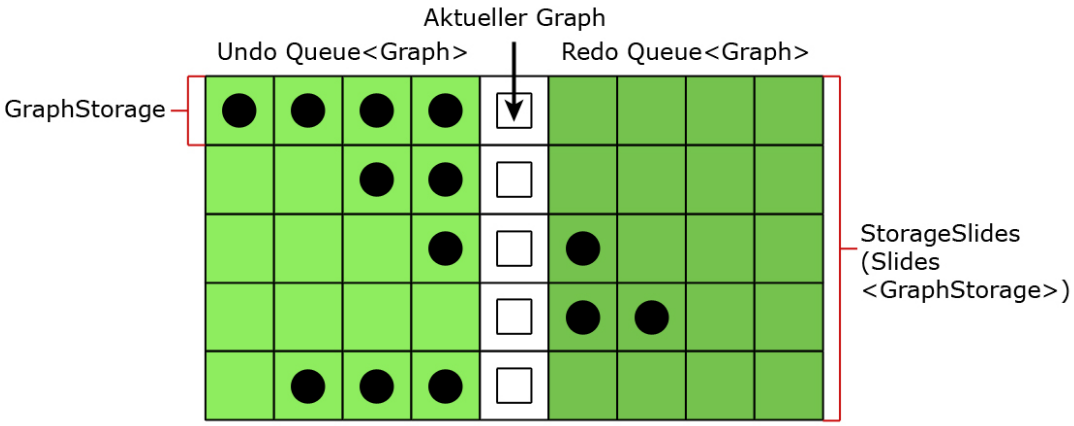


Abbildung 3.29: Grafische Darstellung der Klassen

3.5 Darstellung und Interface

3.5.1 Wie sieht das Kontextmenü aus?

In diesem Editor ist ein Kontextmenü für Ecken und Kanten essenziell notwendig, weil diese nicht als Objekte ausgewählt werden können. Trotzdem gibt es Aktionen, die gezielt auf eine ganz bestimmte Ecke oder Kante ausgeführt werden sollen. Wenn der Benutzer beispielsweise eine Kante entfernen möchte, sollte er bestimmen können, welche Kante dies sein soll. Es gibt weitere Aktionen wie das Umbenennen von Ecken, die ebenfalls auf ein bestimmtes Element angewendet werden sollen. Für diesen Zweck benötigt der Editor mehrere Kontextmenüs. Alle Menüs müssen möglichst klein und übersichtlich gehalten sein.

Das klassische Kontextmenü

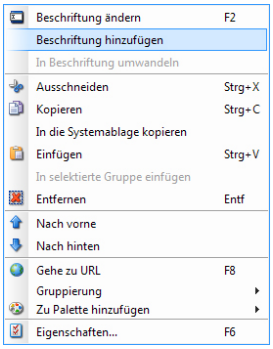


Abbildung 3.30: Ein klassisches Kontextmenü auf dem Editor yEd

Das klassische normale Kontextmenü funktioniert mit zwei Mausklicks. Der Benutzer klickt auf das gewünschte Element mit der rechten Maustaste, danach erscheint das Menü. Anschließend sucht er seine Aktion heraus und klickt mit der linken Maustaste darauf. Dann verschwindet das Menü wieder. Die verschiedenen Einträge können mit einem Bild versehen sein, enthalten aber oft nur Text. Dieses Modell ist bekannt und bedarf deshalb keiner weiteren Erklärung. In Java Swing gibt es bereits Klassen, die für diese Verwendung entworfen wurden. Das Entfernen und Hinzufügen dieser unterschiedlichen Menüs kann zu Performance-Problemen führen, weil

dies viel Zeit kostet.

Ein Radialmenü

(a) Darksiders II **(b)** Prototype

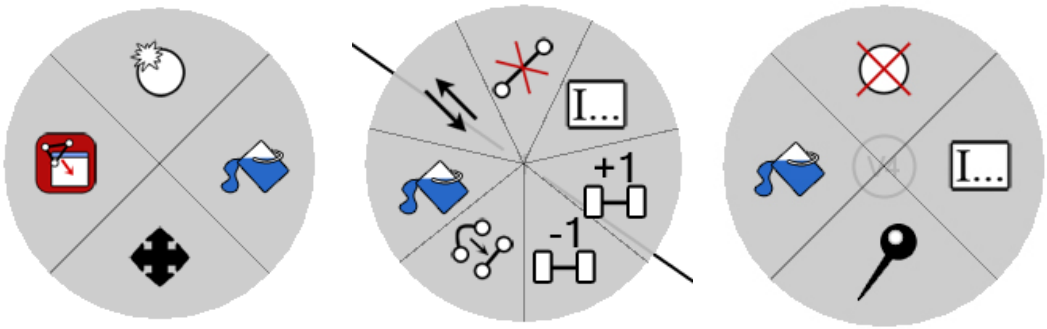
(c) League of Legends **(d)** wacom Stiftmenü

Abbildung 3.31: Bilder von Radialmenüs aus Spielen und Anwendungen

Eine modernere Variante eines Kontextmenü ist ein Menü, dass im Kreis angeordnet ist. Die einzelnen Einträge werden als Bilder symbolisiert und durch Texteinblendungen erklärt. Das Kontextmenü wird über einen Rechtsklick geöffnet. Bei gedrückter rechter Maustaste wird ein Bild ausgewählt, indem der Benutzer mit der Maus dar-

überfährt und die Maustaste loslässt. Dieses Konzept findet sich immer häufiger in modernen Spielen und Anwendungen. Ein solches Kontextmenü lässt sich schneller und intuitiver Bedienen als die klassische Variante. Bilder können schneller erkannt werden als ein Text und es ist einfacher, ein Bild auf einem Kreis auszuwählen als ein Texteintrag aus einer Liste. Weil der Benutzer sich nach häufiger Benutzung der Menüs die Position merken kann, ist er auf lange Sicht mit dieser Menüart schneller. Dem entgegen steht jedoch die Eingewöhnungszeit zu Beginn. Ein solches Radialmenü hat nur begrenzt Platz zur Verfügung, weshalb die Anzahl der Einträge gering gehalten werden muss. Gegenüber dem klassischen Kontextmenü benötigt ein Radialmenü weniger Platz nach unten. Während ein Listenmenü immer größer wird, werden in einem Radialmenü nur die Abstände der Bilder zueinander kleiner, wodurch die Einträge schlechter zu treffen sind. Ein Listenmenü könnte solche Ausmaße annehmen, dass einzelne Einträge nicht mehr erreichbar wären, weil diese über die grenzen des Desktops hinausgehen.

Implementierung



(a) Generelles Menü **(b)** Menü für Kanten **(c)** Menü für Ecken

Abbildung 3.32: Screenshots der Kontextmenüs aus WildGraphs

Das Kontextmenü ist ein Radialmenü, weil es schneller bedient werden kann. Es gibt drei verschiedene Menüs. Eines für die Ecken, eines für die Kanten und eines, welches erscheint, wenn der Benutzer eine freie Stelle auf der Zeichenfläche anwählt (globales Menü). Die Menüs besitzen keine Submenüs. Während die Menüs für Ecken

und Kanten dafür da sind, deren Werte zu verändern, ist das globale Menü für den schnellen Wechsel zwischen den Modi gedacht. Zwischen den wichtigsten drei Modi *Create*, *Paint* und *Move* kann auf diese Weise schnell umgeschaltet werden. Außerdem findet sich die hilfreiche Aktion zum Zentrieren des Graphen in diesem Menü.

3.5.2 Wie kann das Aussehen des Graphen geändert werden?

Am Anfang wurde definiert, welche grafischen Änderungen in einem Graphen möglich sind. Dafür wurden Einschränkungen bei den Darstellungsmöglichkeiten von Ecken und Kanten festgelegt. Nun muss überlegt werden, auf welche Weise der Benutzer auf das Aussehen Einfluss nehmen kann.

Über das Kontextmenü

Das Kontextmenü bietet einen einfachen Zugriff auf einzelne Elemente eines Graphen. Es eignet sich gut, um schnell gezielte Einstellungen vorzunehmen. Dabei nimmt es keinen Platz auf dem Bildschirm in Anspruch, sondern erscheint nur, wenn der Benutzer mit der rechten Maustaste in die Zeichenfläche klickt. Weil die Anpassungsmöglichkeiten sehr groß sind, wird ein entsprechendes Kontextmenü schnell überladen. Es ist nicht möglich, alle Einstellungen in ein solches Menü zu packen. Deshalb würde nur eine sehr begrenzte Auswahl an Möglichkeiten zur Verfügung stehen. Diese könnten sich in bestimmten Situationen als zu gering herausstellen, z.B. bei mehr als drei Farben oder Formen. Möchte der Benutzer mehrere Ecken mit demselben Aussehen versehen, müsste er jede Ecke einzeln entsprechend anpassen, was viel Zeit in Anspruch nehmen kann.

Auswählbare Elemente mit Eigenschaftsbox

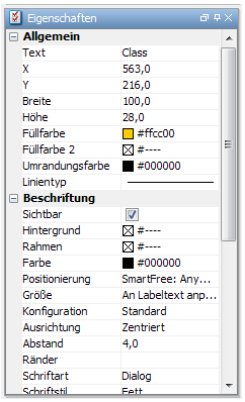


Abbildung 3.33: Eigenschaftsbox aus yEd

Diese Variante wird sehr häufig verwendet. Ecken und Kanten können dabei wie Spielfiguren ausgewählt werden. Ihre Eigenschaften finden sich in einer Eigenschaftsbox wieder, die jederzeit verändert werden können. Dadurch kann jedes Element sehr individuell angepasst werden. Dies bietet den größtmöglichen Freiraum bei der Gestaltung. Auch hierbei muss jedes Element einzeln angepasst werden, um ein einheitliches Bild zu erzeugen. Der Platz auf der Zeichenfläche wird durch solch eine Eigenschaftsbox zudem dauerhaft eingeschränkt.

Das Aussehen als Vorgabe speichern

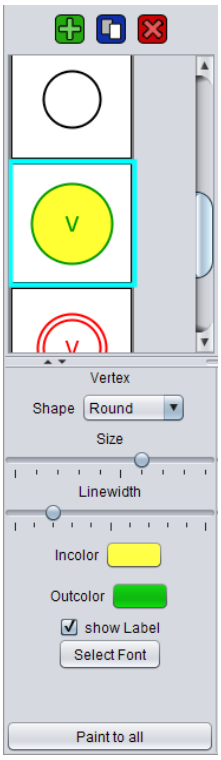


Abbildung 3.34: Screenshot der Optionsbox

Eine weitere Variante ist das Speichern des Aussehens der Elemente in Listen. Hierbei wird die jeweilige Darstellung eines Elements unabhängig von diesem gespeichert. Ein Eintrag in dieser Liste speichert nur Informationen zur Darstellung und wird durch eine Vorschau veranschaulicht. Die so erstellten Darstellungen können dann per Mausklick auf Elemente des aktuellen Graphen übertragen werden. Andersherum ist es möglich, das Aussehen eines bestimmten Elements in dieser Liste zu speichern. Über entsprechende Buttons können alle Elemente auf einmal umgestaltet werden, was bei großen Graphen Zeit spart. Bei der Erstellung eines neuen Graphen mit neuen Ecken und Kanten kann zuvor die entsprechende Darstellung gewählt werden, wodurch alle Elemente direkt mit diesem Aussehen erzeugt werden. Das spätere Anpassen fällt somit weg. Der Einstellungsumfang lässt sich beliebig erweitern und kann dem einer Eigenschaftsbox entsprechen.

Implementierung

Die endgültige Entscheidung für eine dieser Varianten wurde im Verlauf der Überarbeitungen gewechselt. In der ersten Version des Editors war es nur über das Kontextmenü möglich, einige Einstellungen vorzunehmen. Darunter fielen die Innen- und Außenfarbe der Ecken und die Strichfarbe der Kanten. Die Außenfarbe konnte in vier und die Innenfarbe in drei verschiedene Farben geändert werden.

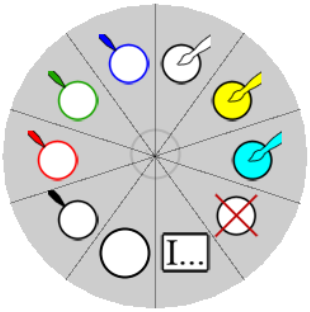


Abbildung 3.35: Das Kontextmenü für Ecken aus einer früheren Version

So waren Einstellungen sehr schnell möglich, aber das Kontextmenü hatte zu viele Einträge und wurde zu groß. Außerdem stellten sich die Möglichkeiten als zu gering heraus, um genug verschiedene Darstellungen zu ermöglichen. Eine zu beschränkte Farbauswahl stellt sich unter Personen mit Farbschwäche als Problem dar und die sehr kleine Farbpalette war bei der Visualisierung und Hervorhebung bestimmter Algorithmen ungenügend. Deshalb wurde dieser Ansatz später wieder entfernt und nicht weiter verfolgt.

Einzelne Elemente auszuwählen und sie über eine Eigenschaftsbox zu verändern, war nicht möglich. Grund hierfür war die Funktionsweise der Modi, die mit Mausklicks arbeiteten. Es wäre für den Editor nicht möglich gewesen, zu unterscheiden, ob ein Element ausgewählt oder ob die Funktionalität des aktuellen Modus ausgeführt werden soll. Zudem kostet diese Variante zu viel Zeit und nimmt dauerhaft Platz von der Zeichenfläche in Anspruch.

Weil das Kontextmenü schnell überladen war und bereits passende Klassen (VDO, EDO) implementiert wurden, die allein für die Darstellung der Elemente dienen,

wurde der letzte Ansatz umgesetzt. Dafür werden zwei getrennte Listen mit Darstellungen geführt, eine für die VDOs und eine für die EDOs. Eine VDO wird in der Optionsbox durch eine Folie mit einer Vorschau repräsentiert. Über diese Folie lässt sich die VDO auswählen und mit Hilfe der Optionseinstellungen darunter verändern. Möchte der Benutzer die Einstellung für eine Ecke übernehmen, wählt er den Modus *Paint*, klickt die entsprechende Folie an und dann auf die gewünschte Ecke. Diese wird dann entsprechend geändert. Soll eine Einstellung für alle Ecken des ausgewählten Graphen übernommen werden, kann dies über einen entsprechenden Button getan werden. Das spätere Anpassen einer VDO beeinflusst nicht nachträglich die Ecken, die mit ihr bereits versehen worden sind. Entsprechend verhält es sich mit den EDOs.

Diese Art der gestalterischen Anpassung bietet genug Freiraum, um Farben beliebig festzulegen und die Lesbarkeit und Klarheit zu erhöhen. Sollte sich später zeigen, dass die Einstellungsmöglichkeiten nicht ausreichen, könnten diese erweitert werden, indem zusätzliche Optionen eingefügt werden. Diese Vorgehensweise bringt auch Nachteile mit sich. Die Optionsbox nimmt im Gegensatz zum Kontextmenü viel Platz ein und beschränkt somit die Zeichenfläche. Bei einer komplizierten Optionsbox muss der Benutzer außerdem erst lernen, wie diese funktioniert.

4 Schlussergebnis

4.1 Mit welchem Ergebnis schließt diese Arbeit ab?

Der komplette geforderte Funktionsumfang konnte in dem Prototypen integriert werden. Es konnte über dies hinaus noch zusätzliche Funktionalität eingebaut werden. Die meisten dieser Funktionen sind klein und hilfreich und sind im Einzelnen:

- Eine ausgewählte Darstellung kann auf alle Ecken oder Kanten auf einmal übertragen werden.
- Ein Graph oder die gesamte Animation kann auf der Zeichenfläche zentriert werden.
- Ecken können fixiert werden, damit sie nicht von Layouts beeinflusst werden.
- Die Richtung einer (gerichteten) Kante kann über das Kontextmenü umgekehrt werden.
- Die Hintergrundfarbe der Zeichenfläche kann angepasst werden, damit diese bei stark reflektierenden Leinwänden in dunklen Räumen nicht unangenehm oder schmerzhaft für die Augen ist.
- Aktionen können rückgängig gemacht werden.

4.2 Wie kann der Editor verbessert werden?

An einigen Stellen im Editor und der internen Struktur dahinter können Verbesserungen vorgenommen werden.

- Um mehr Platz auf der Zeichenfläche zu schaffen, sollten die Icons in der

Toolbar verkleinert und einige Texte auf Buttons durch Bilder ersetzt werden

- Die Kriterien für die Auswahl einer Kante könnten verbessert werden, damit der Benutzer diese überall an ihrem Linienzug auswählen kann
- Besonders bei der Trennung der Definition und der Darstellung von Ecken und Kanten könnte eine alternative Klassenstruktur für mehr Flexibilität sorgen
- Kanten, die als Start und Ziel dieselbe Ecke besitzen, sollten formschöner gezeichnet werden
- Wenn eine gerichtete Kante auf eine Ecke zeigt (Pfeil), wird ein Außenpunkt berechnet, damit der Pfeil an der richtigen Stelle gezeichnet wird. Die Berechnung dieses Außenpunktes ist bei bestimmten Formen sehr schwierig. Hier gäbe es ebenfalls Verbesserungspotential.
- Neben einzelnen Punkten ist auch die Gesamtleistung des Editors wichtig. Mit effizienteren Methoden könnte die Performance des Editors möglicherweise gesteigert werden, damit auch große Graphen ohne Verzögerungen funktionieren.

4.3 Wie kann der Editor erweitert werden?

Durch den Gebrauch vieler Interfaces ist es möglich, an verschiedenen Stellen zusätzliche Inhalte einzufügen.

- Es könnten zusätzliche Eckenformen und neue Linienarten für Kanten eingebaut werden.
- Einfache graphentheoretische Algorithmen könnten automatisch durchlaufen und als Animation abgespielt werden.
- Es könnte ein interner Editor ergänzt werden, mit dem die Speicherdateien direkt bearbeitet werden können.
- Um zusätzliche Informationen zu einem Graphen zu erstellen, könnten auf der Zeichenfläche Notizen hinzugefügt werden.

Angaben zu verwendeten Quellen

Teile der Literatur wurden für die Programmierung des Programms gebraucht und tauchen deshalb nicht als Verweis auf. Nicht selbst erstellter Quellcode ist entsprechend innerhalb der Codedatei gekennzeichnet. Screenshots aus Programmen oder Spielen wurde alleinig für diese Arbeit geschossen. Alle anderen Bilder wurden eigenständig erstellt. Die meisten Abbildungen von Graphen wurden mit Hilfe von *WildGraphs* generiert.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Joshua Bloch. *Effektiv Java programmieren.* Addison-Wesley, 2002.

[2] Joshua Bloch. *Effective Java: Second Edition.* Addison-Wesley, zweite edition, 2008.

[3] Universität Bremen. udraw(graph). http://www.informatik.uni-bremen.de/, 03 2005. Version 3.1.1.

[4] Scilab Enterprises. Scilab (programm). http://www.scilab.org/, 04 2013. Version 5.4.1.

[5] Michel Goossens Frank Mittelbach. *Der LaTeX-Begleiter,.* Pearson Studium, zweite edition, 2005.

[6] Br. Gernot Starke Karl Eilebrecht. *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung.* Spektrum Akademischer Verlag Heidelberg, dritte edition, 2010.

[7] Thomas Künneth. *Einstieg in Eclipse 3.5.* Galileo Press Bonn, dritte edition, 2009.

[8] Stephen G. Kobourov. Force-directed drawing algorithms. http://cs.brown.edu/~rt/gdhandbook/chapters/force-directed.pdf. aufgerufen am 11.02.2014.

[9] Prof. Steffen Lange. Fallstudie bipartite graphen. https://www.fbi.h-da.de/fileadmin/personal/s.lange/Lehre_Sommer11/Graphen_und_Optimierung/go_teil01_2.pdf, 2011. aufgerufen am 24.05.2014.

[10] Oracle. Java™ platform, standard edition 7 api specification. http://docs.oracle.com/javase/7/docs/api/.

[11] Oracle. The java™ tutorials. http://docs.oracle.com/javase/tutorial/index.html.

[12] Dr. Aaron Quigley. Force directed algorithms by decomposed estimation. http://www.csi.ucd.ie/staff/hcarr/home/department/infovisforce.pdf. aufgerufen am 13.02.2014.

[13] Masahiko SAWAI. Java swing font chooser. http://stackoverflow.com/questions/2120228/java-swing-font-chooser. aufgerufen am 10.05.2014.

[14] Peter Tittmann. *Graphentheorie: Eine anwendungsorientierte Einführung,*. Carl Hanser Verlag München, zweite aktualisierte auflage edition, 2011.

[15] Michael Gerdom Torsten Posch, Klaus Birken. *Basiswissen Softwarearchitektur: Verstehen, entwerfen, wiederverwenden.* dpunkt.verlag, dritte edition, 2011.

[16] David Auber und andere. Tulip. http://tulip.labri.fr/TulipDrupal/, 11 2013. Version 4.4.

[17] Denny Bayer und andere. Snoopy. http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Snoopy, 07 2013. Version 1.13.

[18] John Ellson und andere. Graphviz - graph visualization software. http://www.graphviz.org/, 01 2014. Version 2.36.

[19] Markus Hohenwarter und andere. Geogebra. http://www.geogebra.org/, 12 2013. Version 4.4.

[20] Prof. Dr. Rüdiger Erbrecht und andere. *Das große Tafelwerk: Formelsammlung für die Sekundarstufen I und II.* Cornelsen Verlag Berlin, 2011.

[21] Susan H. Rodger und andere. Jflap. http://www.jflap.org/, 05 2011. Version 7.0.

[22] Till Tantau und andere. tikz. LaTeXPackage http://sourceforge.net/projects/pgf/, http://pgf.sourceforge.net/, 12 2013. Version 3.0.0.

[23] Wikipedia. Adjazenzliste. http://de.wikipedia.org/wiki/Adjazenzliste. aufgerufen 29.01.2014.

[24] Wikipedia. Adjazenzmatrix. http://de.wikipedia.org/wiki/

Adjazenzmatrix. aufgerufen am 29.01.2014.

[25] Wikipedia. Baum (graphentheorie). http://de.wikipedia.org/wiki/Baum_%28Graphentheorie%29. aufgerufen 29.01.2014 und 25.05.2014.

[26] Wikipedia. Entwurfsmuster. http://de.wikipedia.org/wiki/Entwurfsmuster. aufgerufen am 10.05.2014.

[27] Wikipedia. Graph drawing. http://en.wikipedia.org/wiki/Graph_drawing. aufgerufen 02.02.2014.

[28] Wikipedia. Graph (graphentheorie). http://de.wikipedia.org/wiki/Graph_%28Graphentheorie%29. aufgerufen am 29.01.2014.

[29] Wikipedia. Graphentheorie. http://de.wikipedia.org/wiki/Graphentheorie. aufgerufen am 29.01.2014.

[30] Wikipedia. Graphml. http://de.wikipedia.org/wiki/GraphML. aufgerufen am 16.06.2014.

[31] Wikipedia. Graphzeichnen. http://de.wikipedia.org/wiki/Graphzeichnen. aufgerufen 29.01.2014.

[32] Wikipedia. Hierarchisches layout. http://de.wikipedia.org/wiki/Hierarchisches_Layout. aufgerufen 26.05.2014.

[33] Wikipedia. Petri-netze. http://de.wikipedia.org/wiki/Petri-Netz. aufgerufen am 29.01.2014.

[34] yWorks GmbH. yed. http://www.yworks.com/, 09 2013. Version 3.11.1.

[35] Stefan Zörner. *Softwarearchitekturen dokumentieren und kommunizieren: Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten.* Carl Hanser Verlag München, 2012.

[36] John Zukowski. *The Definitive Guide to Java Swing.* Apress, dritte edition, 2005.

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Jennifer Wilde