Bachelor - Praktikum (Graphenlayout)

Designdokumentation Rev. 1

Gruppe: $G^{2^{2^2}}$

postfuse

one step beyond

Stand: 12. Januar 2007

http://bp.macrolab.de

Tutor: Thorsten Volland
Aufraggeber: Michael Eichberg
Gruppenmitglieder: bp@macrolab.de

- Andreas Franke franke.andreas@tiscali.de
- Peter Schauss peter.schauss@web.de
- Martin Konrad mkon@gmx.de
- Marco Möller marco.moeller@macrolab.de

Inhaltsverzeichnis

1 Einleitung **3**

2 Systemüberblick **3**

3 Design Überlegungen **4**
3.1 Annahmen und Abhängikeiten 4
3.2 Allgemeine Vorgaben 4
3.2.1 Verfügbarkeit und Beständigkeit von Ressourcen 4
3.2.2 Standards 4
3.2.3 Geforderte Qualitätsmerkmale 4
3.2.4 Nichtfunktionale Anforderungen an die Architektur 5
3.3 Designziele und Richtlinien 5
3.4 Entwicklungsmethode 5
3.5 Begründung für Entscheidungen 6
3.6 Kurzbeschreibung des Einsatzes von Design Patterns 6

4 System Architektur **6**
4.1 Subsysteme und ihre Aufgaben 6
4.2 Kooperation der Teilsysteme 7
4.2.1 Klassenübersicht 7
4.2.2 Use Case Realizations 9
4.3 Beschreibung der wichtigsten Klassen 12
4.3.1 Spezifikation von GGraph: 13
4.3.2 Spezifikation von GScriptable: 13

5 Klassenübersicht **13**
5.1 Packages und ihre Responsibility 13
5.2 Wichtigsten Designklassen und ihre Responsibility 14

6 Änderungshistorie **14**

Glossar **15**

Literatur **15**

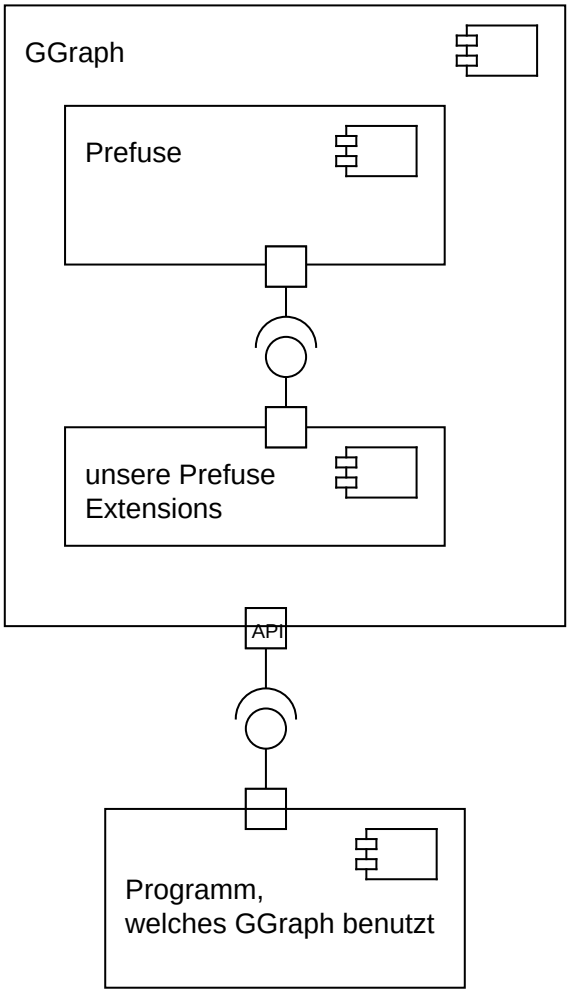

Abbildung 1: Die grobe Struktur unseres Programms

1 Einleitung

Das Designdokument ist Teil unserer Dokumentation für das Bachelorpraktikum 'Graphenlayout'. Es soll einen Überblick über die Architektur unseres Produktes geben. Zudem sind unsere Designentscheidungen über die verwendeten Programme samt deren innerer Struktur und Zusammenspiel dokumentiert und deren Verwendung begründet.

Das Designdokument wird die Struktur des fertigen Produktes erklären. Dabei wird auf die Zusammenhänge der einzelnen Komponenten untereinander eingegangen, sowie die Interaktion der verschiedenen Klassen und Methoden betrachtet. Es wird dabei Bezug auf die Use-Cases aus unserem Pflichtenheft genommen.

2 Systemüberblick

Unser Produkt besteht aus drei Eclipse-Projekten. Das größte Projekt ist die Graph-API inkl. einer View-Komponente. Darauf aufbauend werden wir eine Standalone-Version und eine Eclipse-Plugin-Version unseres Produktes erstellen. Diese letzten beiden Projekte dienen nur als GUI-Wrapper für das Graph-API Projekt.
Dies hat den Vorteil, dass man eine klare Trennung von API und Programm erhält. Hinter der Trennung steht auch unser Konzept, dass wir Teile der API mit JUnit zu testen und das Testen der grafischen Oberfläche zu erleichtern.

In Abbildung 1 auf der vorherigen Seite kann man weitere wichtige Designentscheidungen erkennen. Unser Projekt wird zu großen Teilen auf Prefuse aufbauen. Prefuse ist ein weit fortgeschrittenes Framework zum Visualisieren und Verarbeiten von Graphen. Die Klassen von Prefuse werden erweitert, um die gewünschten Features zu implementieren. Um die gesamte Struktur für den Nutzer übersichtlich zu halten wollen wir keines der API Elemente von den Prefuse Klassen erben lassen.

Diese API Klassen werden hinterher die gesamten im Pflichtenheft beschriebenen API Use-Cases erfüllen müssen. Die GUI Use-Cases greifen wie bereits im Pflichtenheft beschrieben auf die API Use-Cases zu und stellen werden somit großteils ebenfalls durch das API Design festgelegt.

3 Design Überlegungen

3.1 Annahmen und Abhängikeiten

Zu berücksichtigende Systemumgebung Das Plug-In läuft unter Eclipse ab Version 3.2. Die API selbst kann man mit dem Java Compiler ab Version 5 verwenden. Das Standalone-Programm läuft mit der Java Runtime Environment Version 5.

Benutzercharakteristik Der Benutzer weiß, was ein Graph ist. Weiterhin verfügt über Fähigkeiten in Eclipse Plugins zu installieren und zu starten. Für die Benutzer der API gehen wir davon aus, dass sie in der Lage sind in Java Programme mit Interaktionen zu externen APIs zu schreiben.

Weitere Iterationen In weiteren Iterationen sollen auch die Punkte der Aufgabenstellung implementiert sein, die eine niedrige Priorität haben.

3.2 Allgemeine Vorgaben

3.2.1 Verfügbarkeit und Beständigkeit von Ressourcen

Unser Projekt baut fundamental auf der Verfügbarkeit und Beständigkeit von Java und Eclipse auf. Von einer Beständigkeit kann man hierbei leider nur bedingt ausgehen. Da es sich hierbei allerdings um professionell und im großen Umfang benutzte Software handelt kann man von einer gewissen Abwärtskompatibilität ausgehen. Falls diese Programme einmal nichtmehr verfügbar sein sollten wird damit auch unser Projekt nicht mehr benötigt. Also ist auch dieser Fall nicht weiter zu beachten.

Zudem bauen wir auf Projekten wie BeanShell und Prefuse auf. Bei diesen kann man nicht von einer dauerhaften Verfügbarkeit und Beständigkeit ausgehen, was allerdings nicht kritisch für uns ist da wir nur einen Snapshot des Codes benötigen.

3.2.2 Standards

An Standards werden wir auf XML aufbauende Dateiformate benutzen. Dies ist zum einen GraphML zum Abspeichern der Graphen und deren Metadaten. Um die zusätzlich benötigten Felder einzubinden werden wir GraphML mittels XML Schemata erweitern.
Als Vektorgrafikformat haben wir uns für SVG entschieden da dieses ebenfalls auf XML aufbaut. Der Standard hinter SVG scheint aber noch nicht sehr gefestigt zu sein, zumal es kaum zueinander kompatible Tools zu geben scheint. Vor allem bei komplizierteren SVG-Dateien treten bei gängigen SVG-Viewern teils gravierende Fehler bei der Anzeige auf. Allerdings scheint sich SVG als Vektorgrafikformat im Internet allmählich durchzusetzen.

3.2.3 Geforderte Qualitätsmerkmale

Zu den Details über unsere Qualitätssicherung verweisen wir auf unser Qualitätssicherungsdokument. Hier nur einige wichtige Punkte kurz zusammengefasst.

Zuverlässigkeit und deren Messung Man soll mit der API nur sinnvolle Graphen aufbauen können. Wir versuchen dies in möglichst hohem Maß durch das Design der API zu gewährleisten.
Durch das Verwenden von vielfach erprobten Bibliotheken werden zudem weitere Fehlerquellen ausgeschlossen.
Die API und die GUI sollen unter anderen durch Black-Box Tests auf Fehler überprüft werden.

Effizienz und deren Messung Kritisch für die Performanz sind das Berechnen des Layouts und das Zeichnen des Graphen. Da beides schon von Prefuse übernommen wird, bauen wir hier bereits auf ein effizientes und erprobtes Framework auf. Messen kann man die Effizienz mit JUnit-Tests, die große Graphen generieren und die Berechnungszeiten stoppen. Wichtig ist hier vor allem, dass lang dauernde Aktionen abbrechbar sind und nicht das gesamte Programm blockieren. In den bisherigen Tests scheint sich dies allerdings alles in Sekundenbruchteilen abzuspielen.

Sicherheit Der Schutz von unbefugten Zugriffen ist in unserem Programm nicht von Bedeutung, da alle Aktionen lokal ablaufen oder die Sicherheit von anderen Programmen gewährleistet werden muss (z.B. von Eclipse beim Update).

3.2.4 Nichtfunktionale Anforderungen an die Architektur

An nichtfunktionalen Anforderungen ist in unserer Applikation vor allem die Kompatiblität zu Dateiformatstandards und die Benutzbarkeit entscheidend.
Beim Dateiformat setzten wir, wie bereits in Abschnitt 3.2.2 auf der vorherigen Seite beschrieben, an mehreren Stellen auf gängige XML Formate und können so ein hohes Maß an Kompatibilität gewährleisten.
Die GUI und die API sind beide kompakt gehalten und es wurde darauf geachtet, dass die Schnitstellen so weit wie möglich selbsterklärend sind. Ein weiterer Punkt sind lizenzrechtliche Einschränkungen an den Code. Es sollte hinterher möglich sein den Code und das Programm frei für alle zugänglich zu machen. Ziel ist es hier die BSD Lizenz zu erfüllen.

3.3 Designziele und Richtlinien

Unsere allgemeinen Ziele, die wir mit unserer Designimplementation erreichen wollen sind folgende:

- Wiederverwendbarkeit
- Effizienz
- Robustheit
- Zuverlässigkeit
- Wartbarkeit

Diese Ziele setzen wir mit unterschiedlichen Methoden um. Zum einen ist da die Modularität unseres Programmes, die das Debugging vereinfacht und auch die Wiedervewendbarkeit des Codes sichert. Effizienz, Robustheit und Zuverlässigkeit setzen wir mit Hilfe des KISS-Prinzips um. Das heißt wir versuchen die auftretenden Probleme so einfach wie möglich zu lösen ohne komplexe Programmierparadigmen anzuwenden, indem wir einen minimalistischen Lösungsweg wählen.
Sollten wir allerdings einem Problem begegnen, das sich nicht mit einem minimalem Ansatz lösen läßt, dann vermindern wir nicht unsere Anforderungen, sondern gehen den komplexen Lösungsweg.

3.4 Entwicklungsmethode

Wir nutzen als Entwicklungsmethode den Rational Unified Process (RUP). Dieser Entwicklungsprozess ist ein aktueller Standard (im Gegensatz z.B. zum V-Modell). Zudem lässt sich seine Komplexität sehr gut an unsere Projektgröße anpassen.

Dessen Eigenschaften haben wir im Folgenden kurz dargestellt.

Prinzipien vom Rational Unified Process

- Iterative Entwicklung
- konstante Rcksprache mit dem Auftraggeber, ob man auch den Requirements gerecht wird
- Komponenten-basierte Architektur verwenden
- Visuelle Darstellung des Code-Konzepts (UML)
- Konstante Qualitätssicherung während der Entwicklung
- 'Sichere' Änderungen am Code (SVN)

3.5 Begründung für Entscheidungen

Wir verwenden Prefuse, weil dies bereits die meiste Funktionalität beinhaltet. Außerdem bietet es schon Möglichkeiten komplizierte Graphen mit gerichteten Kanten und Doppelkanten zu erstellen und zu visualisieren.
Alternativ war auch Draw2D in der Diskussion. Hier hätte allerdings ein sehr viel höherer Aufwand betrieben werden müssen um die Kernelemente der Applikation umzusetzen.

Da wir auch eine Standalone-Version erstellen hat es den Vorteil, dass wir diese deutlich leichter testen können als ein Plug-In. Für jeden Testdurchgang des Plug-Ins muss nämlich eine neue Eclipse-Workbench gestartet werden, was sehr zeitaufwendig ist.

3.6 Kurzbeschreibung des Einsatzes von Design Patterns

Durch die Struktur von Prefuse das massiv den Gebrauch von Design Pattern macht wie z.B.:

- Command Objekt
- Class Factory
- Programm to an Interface not to an Implementation
- Observer Pattern
- Iterator Pattern

ist in unserem Projekt automatisch die Verwendung dieser und weiterer Pattern praktisch vorgegeben.

Zu den Prinzipien von Prefuse wollen wir außerdem die Komplexität von Prefuse kapseln, um die API möglichst einfach verwendbar zu halten und den Nutzer nicht mit internen Prefuse-Methoden zu konfrontieren.

4 System Architektur

4.1 Subsysteme und ihre Aufgaben

Unsere Applikation ist in mehreren Ebenen in Subsysteme unterteilt. Dies ist zum einen die Gliederung unseres eigenen Codes, die sich in der Package-Struktur wie in Abschnitt 5.1 auf Seite 13 beschrieben, wiederspiegelt.

Zum anderen ist Postfuse durch die Benutzung von externen Komponenten und die Unterteilung in verschiedene Programme grob strukturiert.
Postfuse ist einmal als Standalone Programm und einmal als Eclipse Plugin verfügbar. Beide Applikationen platzieren das selbe Swing-Widget in einem Formular. Dieses Widget und die beiden Applikationen sind alle als eigene Java-Projekte realisiert. Das Widget stellt zum einen die GUI sowie die API zur Verfügung. Hierzu kapselt unser Postfuse-Widget mehrere andere externe Komponenten. Dies sind:

Prefuse implementiert die Graphenanzeige sowie dessen Layoutberechnung.

BeanShell ist leichtgwichtiger Java Interpreter mit Scripting Erweiterungen.

RubyInterpreter ist ein Ruby Interpreter.

In Abbildung 2 auf der nächsten Seite ist dieses Struktur skizziert.

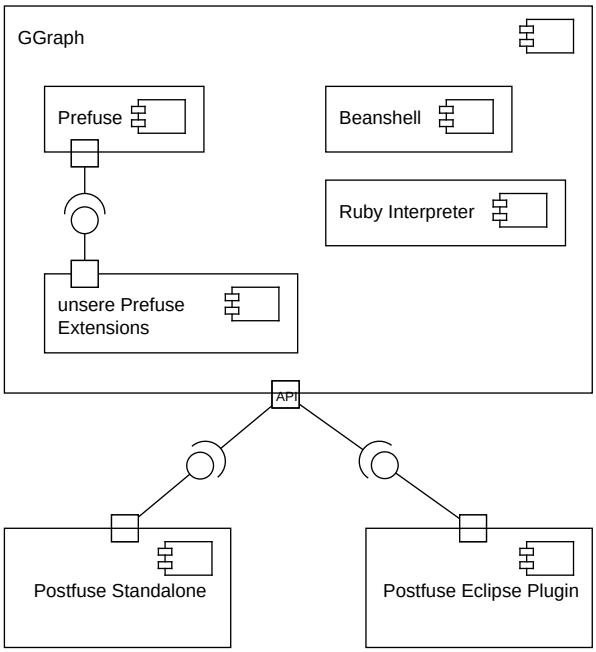

Abbildung 2: Die grobe Struktur unseres Programms

4.2 Kooperation der Teilsysteme

4.2.1 Klassenübersicht

In den folgenden Abbildungen ist eine Übersicht unserer verwendeten Klassen und deren Schnitstellen zu sehen:

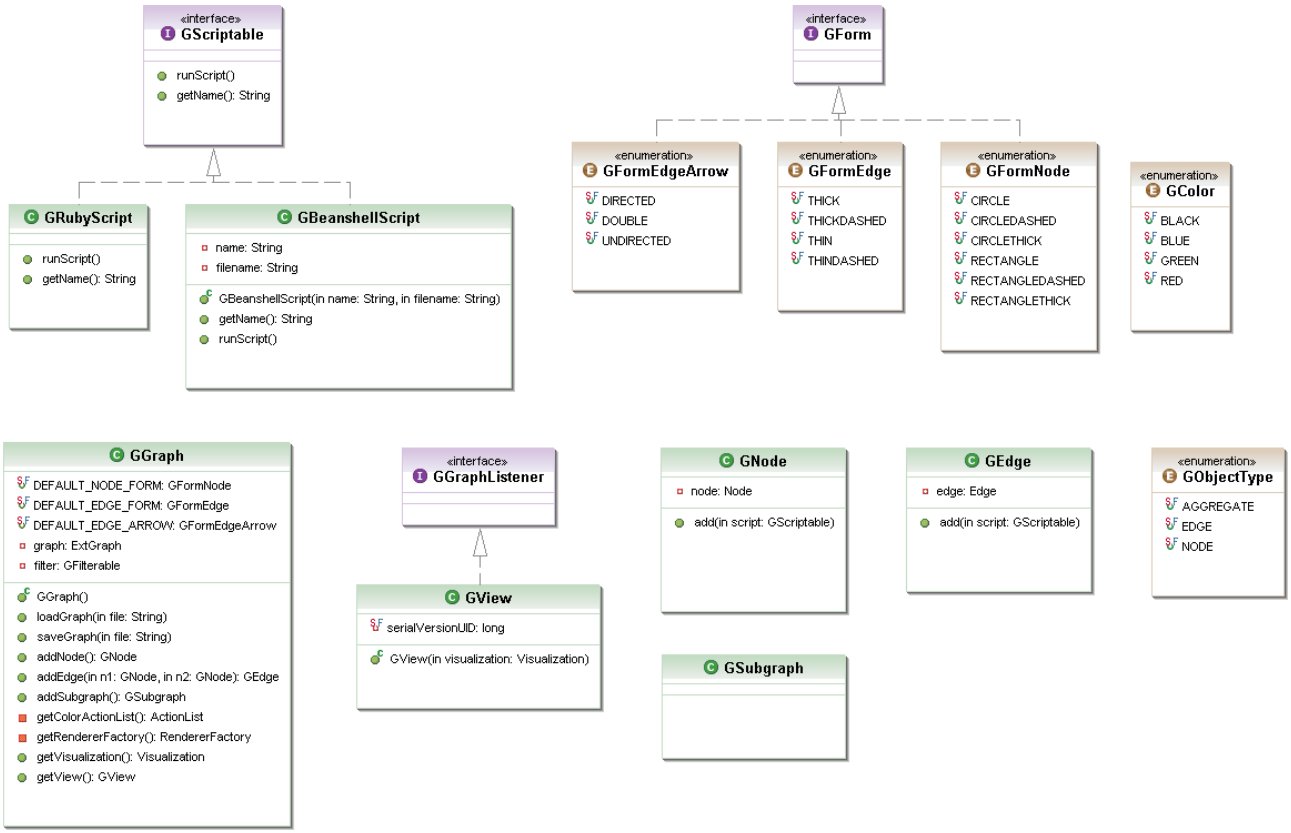

Abbildung 3: Die Klassen von de.postfuse.api

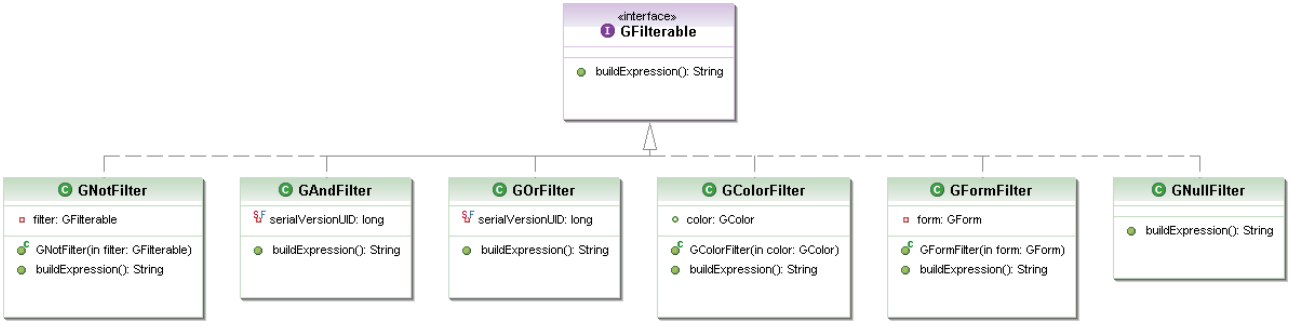

Abbildung 4: Die Klassen von de.postfuse.api.filter

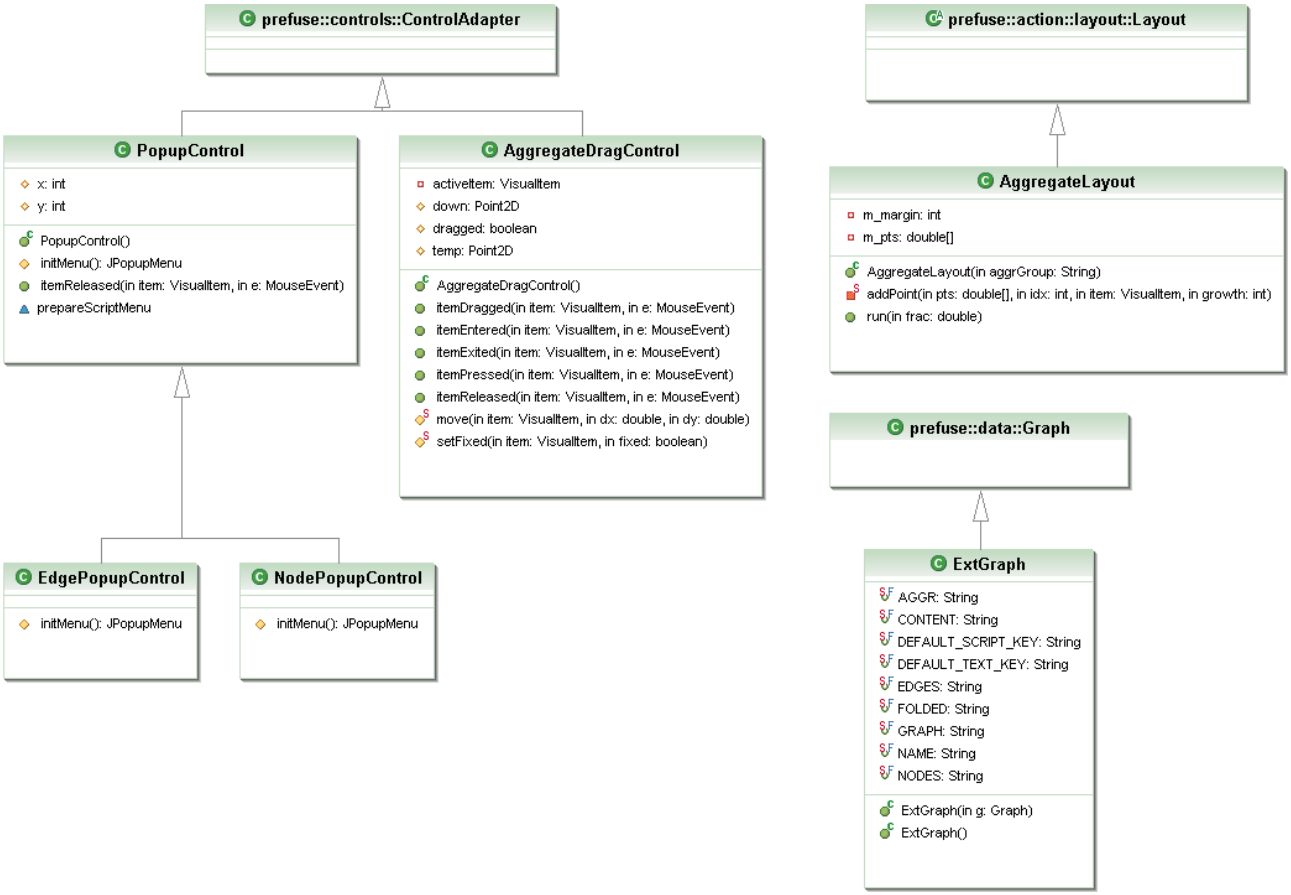

Abbildung 5: Die Klassen von de.postfuse.api.internal

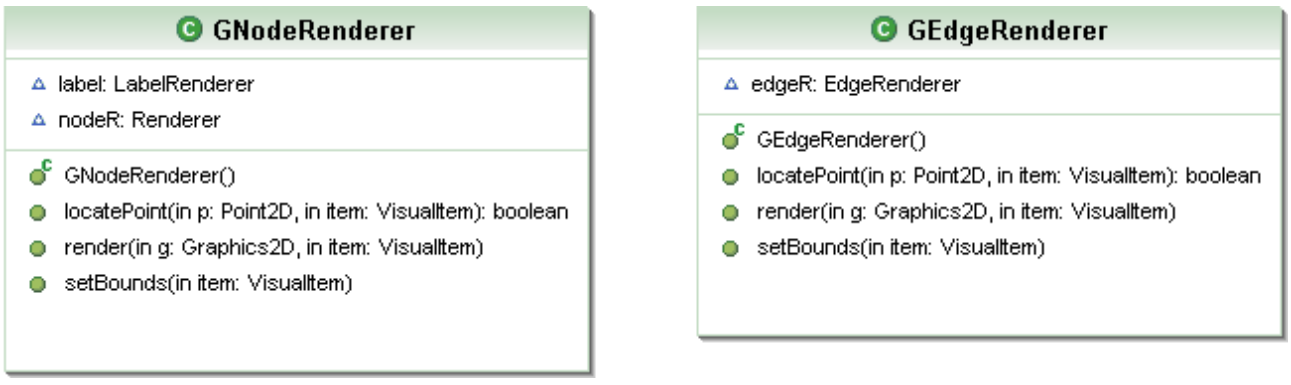

Abbildung 6: Die Klassen von de.postfuse.api.internal.render

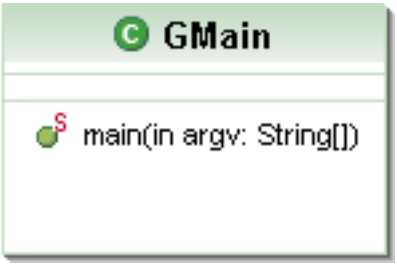

Abbildung 7: Die Klassen von de.postfuse.standalone

4.2.2 Use Case Realizations

Installations-Use-Cases werden über die in Eclipse enthaltenen Mechanismen in zusammenarbeit mit unserer Updatesite gewährleistet.
Dies betrifft die Use-Cases:

- Installation des Plugins
- Deinstallation des Plugins

API-Use-Cases

Graph erstellen Als erstes muss der Graph erstellt werden. Dies geschieht durch instanzieren der Klasse GGraph.
Dies betrifft die Use-Cases:

- Erzeugen des Graphen

Knoten und Kanten erstellen und hinzufügen Um Knoten in den Graphen Hinzuzufügen wird die Funktion *addNode()* des GGraph Objekts aufgerufen. Um anschließend Kanten hinzuzufügen oder Skripte an den Knoten zu binden wird dabei zudem eine Referenz auf den Knoten zurückgegeben.
Analog hierzu kann über die Funktion *addEdge(GNode n1, GNode n2)* eine Kante hinzugefügt werden.
Diese Vorgänge sind in Abbildung 8 auf der nächsten Seite illustriert.
Dies betrifft die Use-Cases:

- Erzeugen eines Knotens
- Erzeugen einer Kante
- Hinzufügen eines Knotens
- Hinzufügen einer Kante

Skripte Ein neues Skript lässt sich einfach über den Konstruktor als Link auf eine Datei erzeugen. Auch das Erstellen aus einem String ist möglich. Knoten und Kanten kann man mit der Funktion *add(GScriptable s)* ein neues Skript hinzufügen. Die Skripte werden über das Kontextmenü der Knoten und Kanten auf der GUI wie in Abbildung 9 auf der nächsten Seite gezeigt aufgerufen.
Dies betrifft die Use-Cases:

- Erzeugen einer Skriptumgebung
- Erzeugen eines Skriptes
- Binden eines Skriptes an Knoten oder Kante
- Ausführen eines Skripts

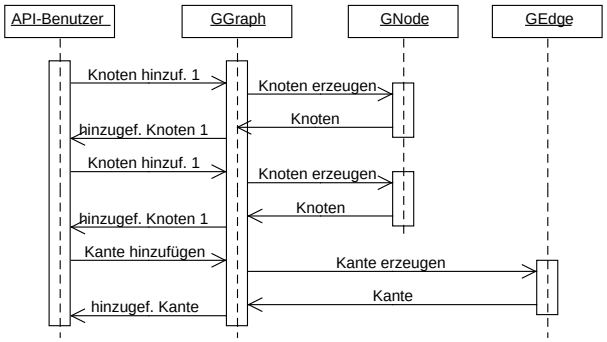

Abbildung 8: Erstellen und Hinzufügen eines Knotens und einer Kante

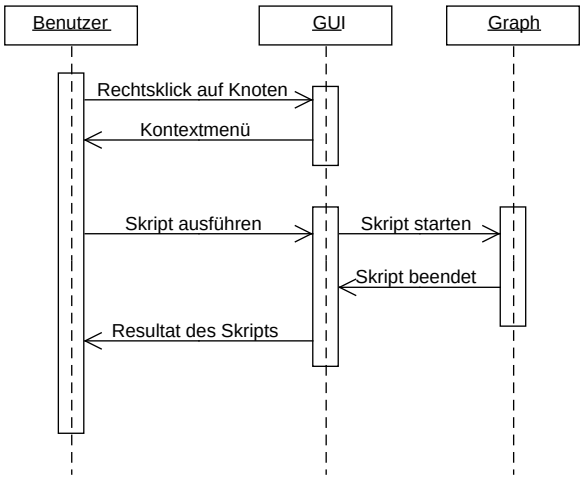

Abbildung 9: Ausführen eines Skriptes

Filter Filter sind Objekte mit dem Interface *GFilterable*. Dieser können einem GGraph Objekt über *setFilter(GFilterable f)* hinzugefügt werden. Rückgesetzt wird ein Filter über *clearFilter()*. Siehe hierzu auch Abbildung 10 auf der nächsten Seite.
Dies betrifft die Use-Cases:

- Setzen eines Filters
- Rücksetzen eines Filters

Laden und Speichern Die Klasse GGraph verfügt über die methoden *loadGraph(String file)* und *saveGraph(String file)* zum laden uns speichern eines Graphen aus bzw. in eine Datei. Siehe hierzu auch Abbildung 11 auf der nächsten Seite.
Dies betrifft die Use-Cases:

- Speichern eines Graphen
- Laden eines Graphen

Sonstiges Die übrigen API-Use-Cases sind übere weitere methoden der Klasse GGraph realisiert. Diese Use-Cases sind:

- Berechnung des Graphenlayouts

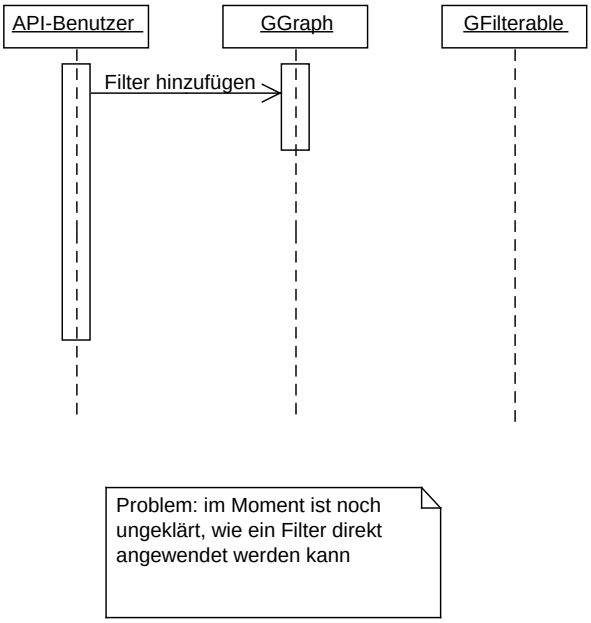

Abbildung 10: Anwenden eines Filters auf einen Graphen

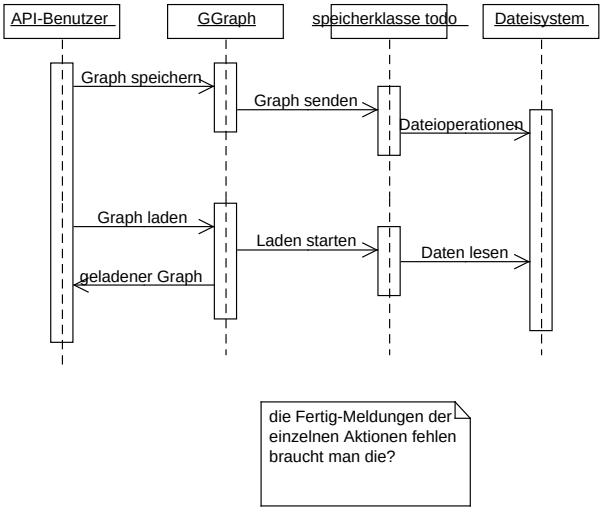

Abbildung 11: Speichern des Graphen

- Abbruch der Berechnung des Graphenlayouts
- Zeichnen von Graphen in GUI
- Exportieren von Graphen nach SVG

GUI-Use-Cases

Knoten falten Knoten können analog zu Abbildung 12 auf der nächsten Seite zusammen- oder ausgefaltet werden.

- Ausfalten eines Knoten
- Zusammenfalten eines Knoten

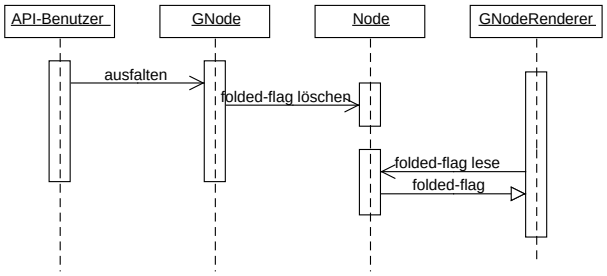

Abbildung 12: Ausfalten eines Knotens

Über Prefuse realisiert Die folgenden GUI-Use-Cases sind intern in Prefuse realisiert:

- Zoomen der Ansicht
- Verschieben der Ansicht (‚Pan‘)
- Reset Zoom + Pan
- Anzeigen des Graphen-Overviews

Über Eclipse realisiert Die Anzeige von mehreren Graphen nebeneinander ist Eclipse-intern über das mehrfache starten unseres Plugins realisiert. Dies betrifft den Use-Case:

- Anzeige zweier Graphen nebeneinander

Über API-Use-Cases realisiert Die folgenden Use-Cases sind, wie bereits im Pflichtenheft beschrieben, über die API-Usescases realisiert:

- Setzen eines Filters
- Rücksetzen eines Filters
- Berechnung des Graphenlayouts
- Abbruch der Berechnung des Graphenlayouts
- Exportieren von Graphen nach SVG
- Ausführen eines Skripts
- Speichern eines Graphen
- Laden eines Graphen

4.3 Beschreibung der wichtigsten Klassen

Zwei der wichtigsten Klassen in unserer Applikation sind die Klassen GGraph und GScriptable.
GGraph ist deshalb so wichtig, weil es die Kernklasse unseres Graphen darstellt. In ihr werden die Standard Filter-/Knoten-/Kantentypen definiert. Zudem kann man genannte Typen auch in GGraph verändern. Zudem kann man Knoten, Kanten und Subgraphen erstellen und in den Graphen einbinden. Das Speichern und Laden, sowie der Export von Graphen nach SVG läuft über GGraph. Außerdem kann man in GGraph die Farben definieren und die Renderer werden mit der View initialisiert.
GScriptable ist unsere Basisklasse die als Interface für alle Skriptklassen dient. Jedes Skript hat einen spezifischen Namen und einen dazugehörigen Dateipfad. Zudem kann es ausgeführt werden. Dabei wird der Code an den entsprechenden Interpreter übergeben, der dann von diesem ausgeführt wird. Dabei werden eventuell auftretende Exceptions abgefangen, und verarbeitet.

4.3.1 Spezifikation von GGraph:

public Elemente:

- GFormNode DEFAULT_NODE_FORM
- GFormEdge DEFAULT_EDGE_FORM
- GFormEdgeArrow DEFAULT_EDGE_ARROW
- GGraph()
- setFilter(GFilterable filter)
- clearFilter()
- exportImage(String format)
- loadGraph(String file)
- saveGraph(String file)
- getVisualization()
- GView getView()
- GNode addNode()
- GEdge addEdge(GNode n1, GNode n2)
- GSubgraph addSubgraph()

private Elemente:

- ExtGraph graph
- GFilterable filter
- ActionList getColorActionList()
- RendererFactory getRendererFactory()

4.3.2 Spezifikation von GScriptable:

public Elemente:

- ScriptConstructor(String name, String filename)
- String getName()
- runScript()

private Elemente:

- String name
- String filename

5 Klassenübersicht

5.1 Packages und ihre Responsibility

Wir haben unsere Kernapplikation in mehrere Packages unterteilt. Hierdurch wird zum einen der gesamte Code übersichtlicher und zum anderen erleichtert es einen modularen entkoppelten Code zu schreiben. Die einzelnen Packages sind im Folgendem:

de.postfuse.api
de.postfuse.api.filter

Diese beiden Packages enthalten die öffentlichen Teile unserer API, eingeteilt in die Filter-Klassen und den Rest. Diese Teile können von Benutzern unserer API gesehen und in ihren Programmen direkt verwendet werden.

de.postfuse.api.internal
de.postfuse.api.internal.render
de.postfuse.api.internal.export

Diese drei Packages enthalten die internen Teile unserer API, eingeteilt in einen Rendering-Teil, einen Export-Teil, und den Rest. Diese werden von anderen Leuten nicht direkt benutzt. Nur die API selbst greift darauf dazu.

de.postfuse.standalone

Dieses Package bzw. Projekt, enthält ein kleines Programm, das die API und daraus insbesondere GView benutzt, um einen Graphen anzuzeigen und den Benutzer damit interagieren zu lassen.

de.postfuse.plugin

Dieses Package bzw. Projekt ist ähnlich wie das Standalone-Projekt, bloß daß es kein eigenständiges Programm ist, sondern ein Eclipse-PlugIn.

5.2 Wichtigsten Designklassen und ihre Responsibility

Die wichtigsten Klassen sind GGraph und GView.

GGraph Die Klassen GNode, GEdge, GFilterable, GScriptable brauchen alle ein GGraph-Objekt, zu dem sie hinzugefügt werden oder auf dem sie operieren. Die GGraph-Klasse stellt Methoden bereit, um Graphen zu erstellen und deren Layout zu berechnen. Graphen können aus Dateien geladen und wieder in Dateien gespeichert werden. Außerdem können Scripte ausgeführt werden, die auf dem Graphen operieren.

GView GView zeigt den Graphen inklusive aller Steruerungselemente in einem Swing-Widget an. Um das fertige Plugin bzw. Standalone Programm zu implementieren ist es nur noch nötig dieses Widget auf eine Form zu platzieren und anzuzeigen.

6 Änderungshistorie

Datum	Thema	Inhalt	seite
14.12.2006	alles	Beginn der History mit Revision 0	*
12.01.2007	alles	Inhalt komplett Umgebaut auf Revision 1	*

Glossar

GraphML

Ein XML-Format zum Abspeichern von Graphen. *4*

Prefuse

Ein Java-Framework zu Visualisierung von Daten wie z.B. Graphen. *5*

SVG

Ein XML-basiertes Vektorgrafikformat. *4*

SVN

Subversion; Team Code Repository *6*

UML

Unified Modelling Language *6*

Literatur

[1] http://www-128.ibm.com/developerworks/opensource/library/os-ecplug/.

[2] Eclipse 3.2 Documentation. http://help.eclipse.org/help32/index.jsp.

[3] PDE Does Plug-ins. http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html.

[4] SVG Eclipse Plugin. http://sourceforge.net/projects/svgplugin/.

[5] The GraphML File Format. http://graphml.graphdrawing.org/.

[6] Eclipse.org home. http://www.eclipse.org/, 2006.

[7] StarUML. http://staruml.sourceforge.net/, 2006.

[8] subversion. http://subversion.tigris.org/, 2006.