Bachelor - Praktikum (Graphenlayout)

Designdokumentation Rev. 2

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

postfuse

one step beyond

Stand: 19. April 2007

`http://bp.macrolab.de`

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

- Bastian Christoph bastian.christoph@gmx.de
- Peter Schauss peter.schauss@web.de
- Martin Konrad mkon@gmx.de
- Marco Möller marco.moeller@macrolab.de

Inhaltsverzeichnis

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 Anwendungsfälle in 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, Teile der API mit JUnit zu testen und das Testen der grafischen Oberfläche zu erleichtern.

In Abbildung 2 auf Seite 7 kann man weitere wichtige Designentscheidungen erkennen. Unser Projekt baut zu großen Teilen auf Prefuse auf. 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. Während der gesamten Nutzung von Prefuse wird dieses außerdem nicht modifiziert, um eine möglichst unabhängige Version des letztendlich entstehenden Produktes zu ermöglichen.

Die API-Klassen erfüllen wie vorgesehen die im Pflichtenheft beschriebenen API Use-Cases. Die GUI Use-Cases greifen auf diese zu und viele Stellen werden somit großteils ebenfalls durch das API Design festgelegt.

Der zentrale Zugriff auf die eigentlichen API-Funktionen erfolgt über die Plugin-Klasse des Eclipse-Plugins, deren Struktur man in Abbildung 1 sieht.

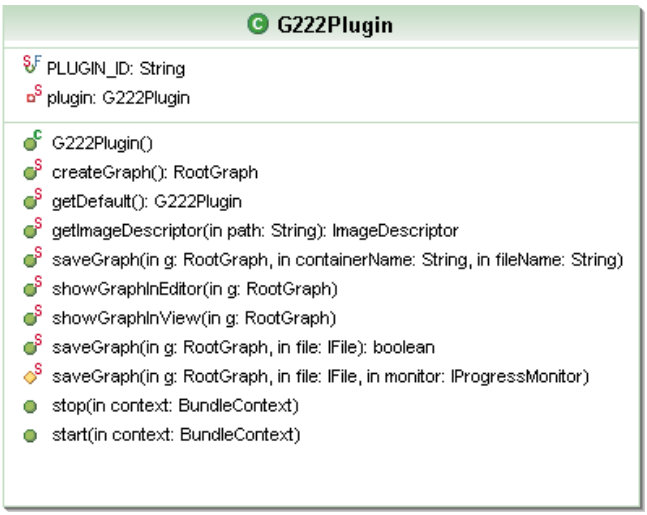

Abbildung 1: Die Plugin-Klasse

3 Design Überlegungen

3.1 Annahmen und Abhängigkeiten

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 er über die Fähigkeit, in Eclipse Plugins zu installieren und zu starten. Für die Benutzer der API gehen wir davon aus, dass sie in der Lage sind, Java-Programme mit Interaktionen zu externen APIs zu entwickeln.

Weitere Iterationen In weiteren Iterationen sollen auch die Punkte der Aufgabenstellung implementiert werden, 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

Wir werden auf XML aufbauende Dateiformate benutzen, da sich XML mitllerweile als Standard durchsetzt. 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

Mit der API sollen nur sinnvolle Graphen aufgebaut werden 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 hin 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 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 setzen 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 fr den Code. Es sollte hinterher möglich sein, den Code und das Programm für alle frei 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 ein.

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 Rücksprache 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' bzw. 'Konsistente' 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 Mehrfachkanten 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, haben wir den Vorteil, dass wir unser Produkt deutlich leichter testen können als ein Plug-In. Für jeden Testlauf des Plug-Ins muss nämlich ein neuer Eclipse-Workbench gestartet werden, was in der Regel sehr zeitaufwendig ist.

3.6 Kurzbeschreibung des Einsatzes von Design Patterns

Durch die Struktur von Prefuse, das massiv den Gebrauch von Design Patterns 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 Patterns 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.

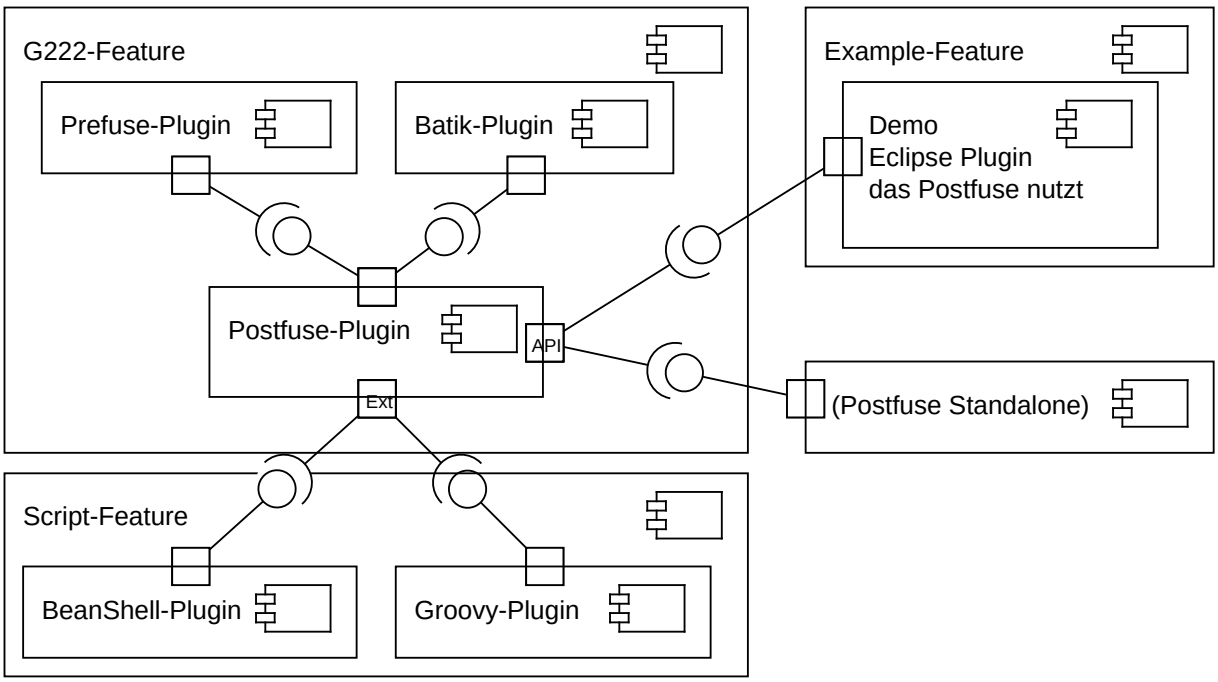

Abbildung 2: Die grobe Struktur unseres Programms

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 19 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 leichtgewichtiger Java Interpreter mit Scripting Erweiterungen.

In Abbildung 2 ist diese Struktur skizziert.

4.2 Kooperation der Teilsysteme

4.2.1 Klassenübersicht

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

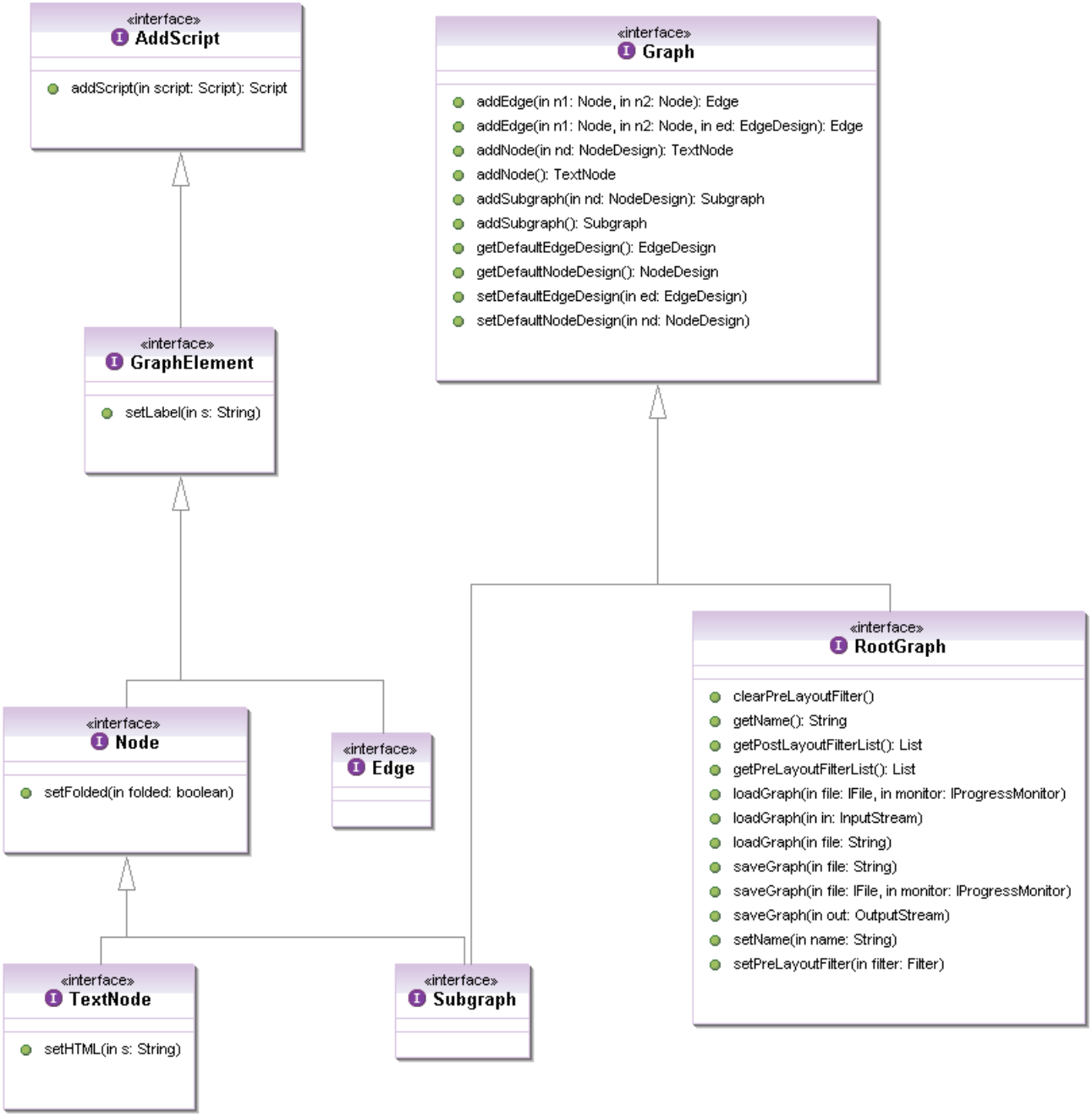

Abbildung 3: Die API Klassen 1

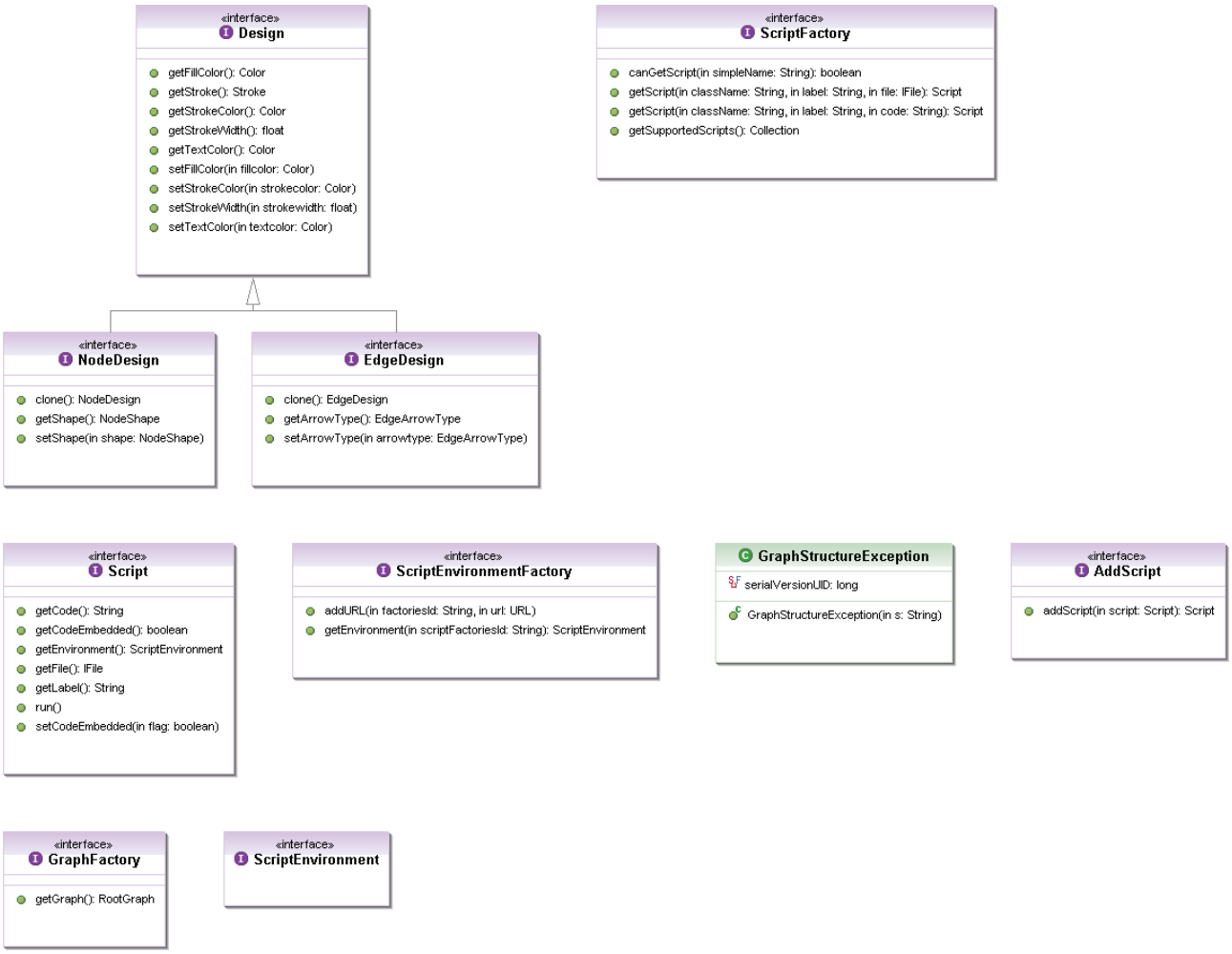

Abbildung 4: Die API Klassen 2

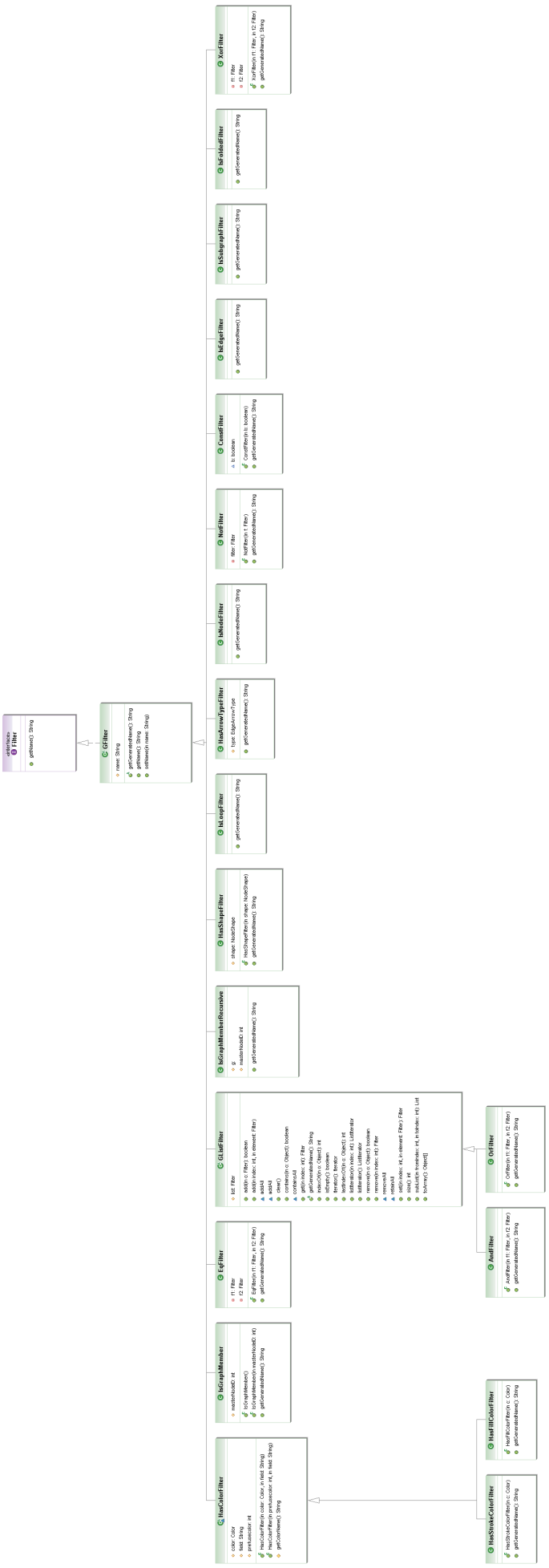

Abbildung 5: Die API Klassen 3 - Filter

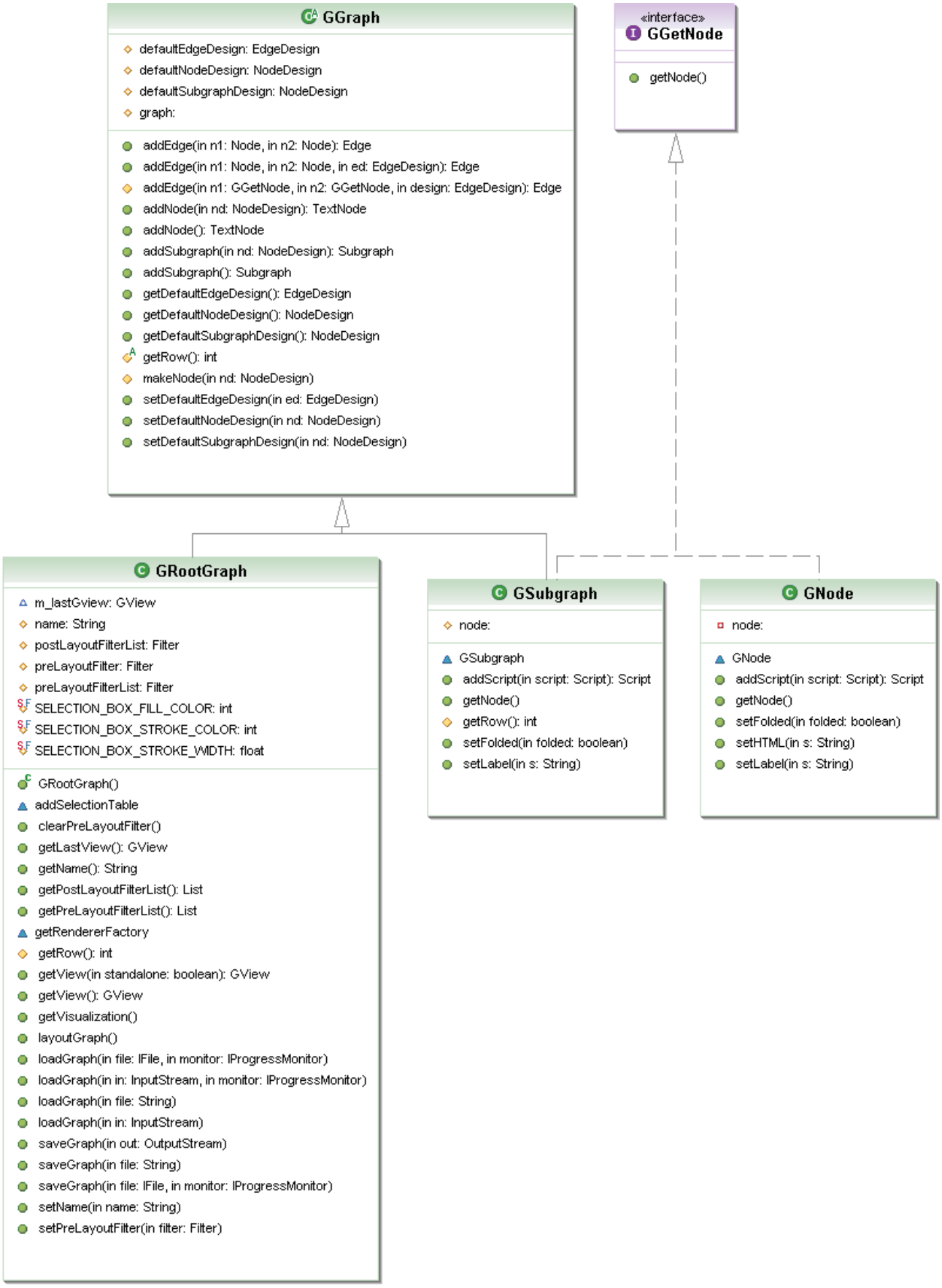

Abbildung 6: Die Implementierungen des API Interfaces

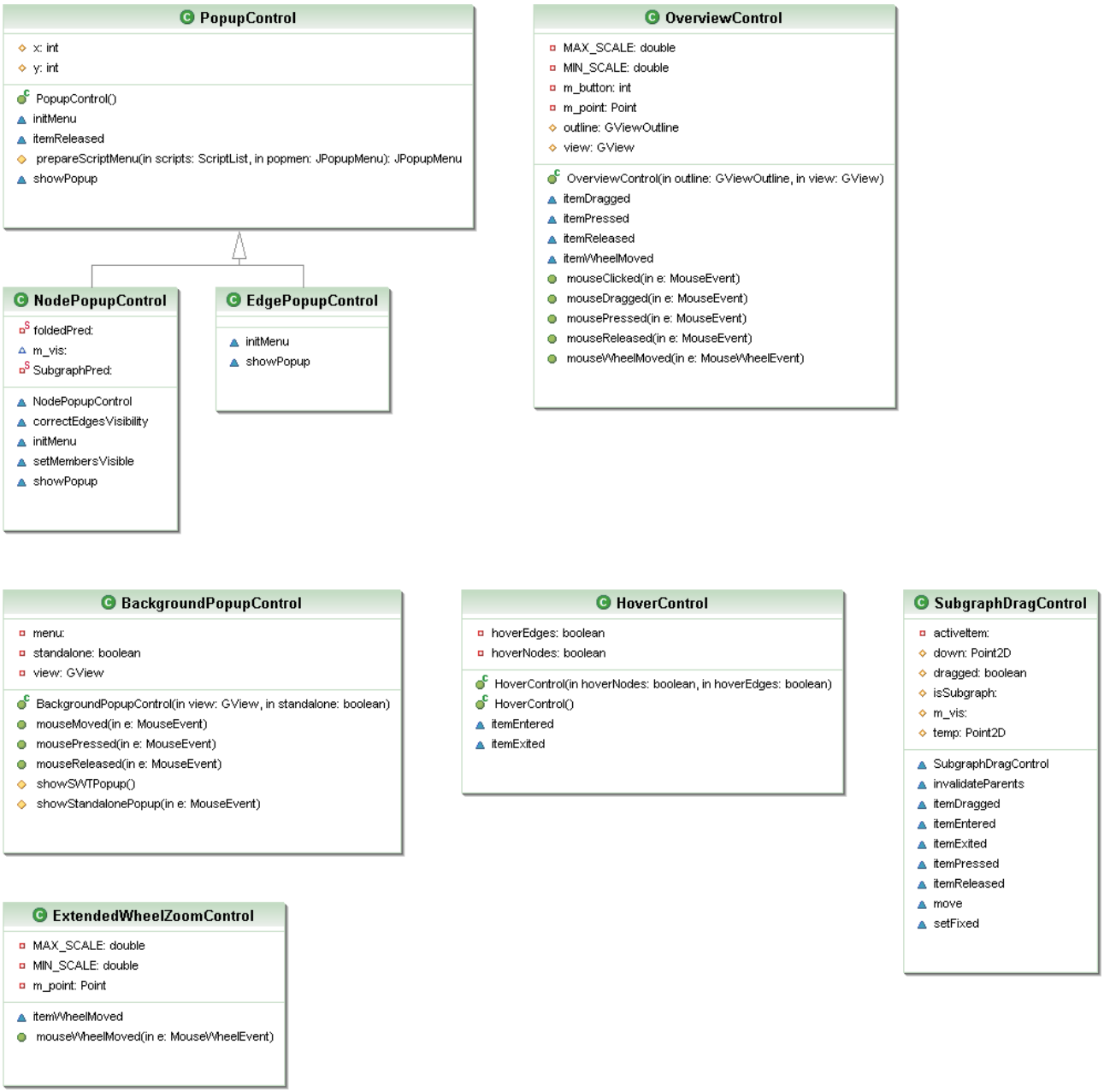

Abbildung 7: Die Controls um die Graph GUI zu bedienen

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 illustriert.
Dies betrifft die Use-Cases:

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

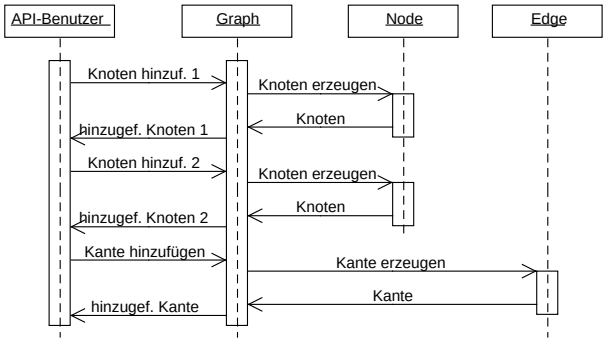

Abbildung 8: Erstellen und Hinzufügen eines Knotens und einer Kante

Skripte Ein neues Skript lässt sich einfach über eine ScriptFactory als Link auf eine Datei erzeugen. Auch das Erstellen aus einem Dateipfad ist möglich. Den Knoten und Kanten kann man mit der Funktion *add(Script 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

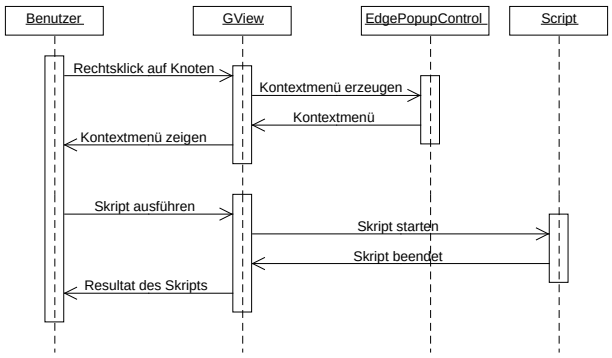

Abbildung 9: Ausführen eines Skriptes

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

- Setzen eines Filters
- Rücksetzen eines Filters

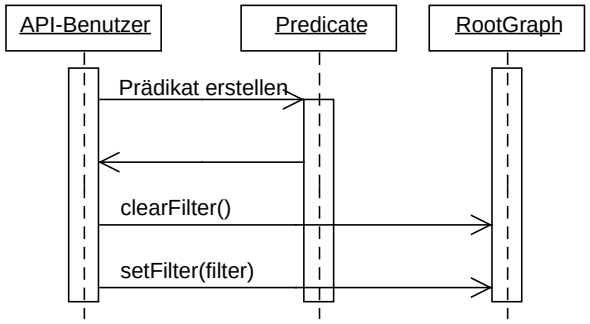

Abbildung 10: Anwenden eines Filters auf einen Graphen

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

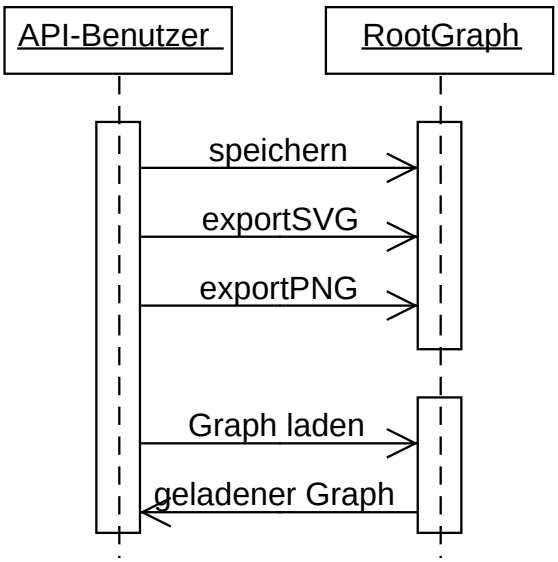

Abbildung 11: Speichern des Graphen

Berechnung des Graphenlayouts Im Package `de.postfuse.core.internal.layout` sind die wesentlichen zum Layout nötigen Klassen enthalten. Zum Layouten des Graphen werden verschiedene Algorithmen kombiniert. Dies sind Baum-, Kräfte- und Subgraphenlayouter. Der Subgraphenlayoutalgorithmus ruft die beiden anderen auf und geht rekursiv wie folgt vor:

1. aktueller Graph := gesamter Graph
2. Layoute alle Subgraphen, die im aktuellen Graphen enthalten sind. Setze diese jeweils als aktuellen Graphen und fahre bei 2. fort.
3. Berechne für jeden Knoten oder Subgraphen die Ausmaße mit Hilfe der entsprechenden Renderer.
4. Markiere alle Graphelemente, die nicht im aktuellen Subgraphen enthalten sind, als unsichtbar.
5. Falls es sich um ein initiales Layouting (Der Layouter kann mehrfach aufgerufen werden.) handelt, layoute alle sichtbaren Elemente mit dem Baumlayouter.
6. Layoute alle sichtbaren Elemente mit dem Kräftelayouter.
7. Passe die Position der Kindelemente des Subgraphen an den verschobenen Subgraphen an.

Das Kräftelayout simuliert die Subgraphen und Knoten als bewegliche Punktmassen anhand einer numerischen Physiksimulation. Hier herrschen zusätzlich zur Trägheit folgende Kräfte:

- Viskosität des Mediums → Bewegungen werden abgebremst.
- Gravitation → alle Knoten ziehen sich gegenseitig an.
- Federkraft → die Kanten zwischen den Knoten (auf der gleichen Subgraphenhierarchieebene) werden als Feder interpretiert und üben Federkräfte aus.
- Max. Abstandskraft → sobald sich Elemente weiter als ein Maximum voneineander entfernen, beginnt eine mit dem Abstand linear ansteigende Kraft sie gegenseitig anzuziehen.
- Antiüberlappungskraft → sollten zwei Elemente sich gegenseitig überlappen oder näher als ein gewisser Mindestabstand kommen, beginnt eine mit dem Abstand linear ansteigende Kraft sie gegenseitig abzustoßen.

Da es sich bei Knoten und Subgraphen um ausgedehnte Elemente handelt, wurde bei den drei zuletztgenannten Kräften nicht der Abstand der Mittelpunkte sondern der Abstand der Ränder einer rechteckigen Boundingbox genommen. Die Simulation endet nach einer festgelegten Zeit (nicht Rechenzeit) im simulierten System.

Durch erneutes Aufrufen des Layouters kann die Kräftesimulation fortgeführt werden.

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

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

GUI-Use-Cases

Knoten falten Knoten können analog zu Abbildung 12 zusammen- oder ausgefaltet werden.

- Ausfalten eines Knoten
- Zusammenfalten eines Knoten

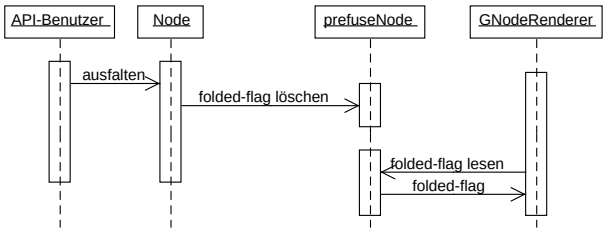

Abbildung 12: Ausfalten eines Knotens

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

- Zoomen der Ansicht
- Verschieben der Ansicht (‚Pan‘)
- Reset Zoom + Pan (mittlerweile aber selbst neuprogrammiert)
- Anzeigen des Graphen-Overviews (nur die Grundfunktionalität der Anzeige einer zweiten Ansicht)

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
- Verwaltung von Installation/Deinstallation des Plugins

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
- Exportieren von Graphen nach PNG
- 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 Script.

GGraph

GGraph ist deshalb so wichtig, weil es die Kernklasse unseres Graphen darstellt. In ihr werden die Standard-Knoten- und Kantentypen definiert. Man kann Knoten, Kanten und Subgraphen zum Graphen hinzufügen.. Außerdem kann man in GGraph die Farben definieren und die Renderer werden mit der View initialisiert.

Script

Script ist das Interface für alle Skriptklassen. Jedes Skript hat einen spezifischen Namen und einen dazugehörigen Dateipfad. Beim Ausführen eines Skripts, wird dessen Code an den entsprechenden Interpreter übergeben, der dann von diesem ausgeführt wird. Dabei werden eventuell auftretende Exceptions abgefangen und geloggt.

4.3.1 Spezifikation von GGraph

public Elemente:

- public TextNode addNode();
- public TextNode addNode(NodeDesign nd);
- public Edge addEdge(Node n1, Node n2);
- public Edge addEdge(Node n1, Node n2, EdgeDesign ed);
- public Subgraph addSubgraph();
- public Subgraph addSubgraph(NodeDesign nd);
- public void setDefaultEdgeDesign(EdgeDesign ed);
- public void setDefaultNodeDesign(NodeDesign nd);

- public void setDefaultSubgraphDesign(NodeDesign nd);
- public EdgeDesign getDefaultEdgeDesign();
- public NodeDesign getDefaultNodeDesign();
- public NodeDesign getDefaultSubgraphDesign();
- public GView getView();

4.3.2 Spezifikation von Script

public Elemente:

- public void run();
- public String getLabel();
- public String getCode();
- public boolean getCodeEmbedded();
- public void setCodeEmbedded(boolean flag);
- public IFile getFile();
- public ScriptEnvironment getEnvironment();

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 wird es dadurch leichter, einen modularen entkoppelten Code zu schreiben. Die einzelnen Packages sind die folgenden:

de.postfuse.ui
de.postfuse.ui.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.core.internal
de.postfuse.core.internal.controls
de.postfuse.core.internal.export
de.postfuse.core.internal.layout
de.postfuse.core.internal.render

Diese fünf Packages enthalten die internen Teile unserer API, eingeteilt in GUI-Controls, Export, Layout und Rendering und den Rest. Diese werden von Benutzern unserer API nicht direkt benutzt. Nur die API selbst greift darauf dazu.

de.postfuse.samples

In diesem Package sind Beispiele für Graphen zu finden.

de.g222.plugin
de.g222.plugin.actions
de.g222.plugin.editors
de.g222.plugin.editors.xml
de.g222.plugin.wizards

Diese Packages enthalten unser Eclipse-Plugin. In ’actions’ sind die Eclipse Aktionen enthalten, die z.B. für den Export zuständig sind. In ’editors’ ist der eigtl. Editor enthalten, der die vom prefuse Display erbende und erweiterte View kapselt. In ’wizards’ ist ein Eclipse-Dialog zum Erzeugen einer neuen GraphML-Datei.

de.g222.example
de.g222.example.actions
de.g222.example.views

Diese Packages enthalten Beispiel-Plugins für Eclipse, die unsere Graph-API und unser Eclipse-Plugin benutzen.

5.2 Wichtigsten Designklassen und ihre Responsibility

Die wichtigsten Klassen sind GGraph und Script. Zur genaueren Beschreibung s. 4.3.

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	*
04.04.2007	alles	Inhalte geprüft und überarbeitet	*
08.04.2007	alles	finales Release 2	*

Glossar

GraphML

Ein XML-Format zum Abspeichern von Graphen. Den Knoten und Kanten können beliebige eigene Attribute zugewiesen werden. *4*

Prefuse

Ein Java-Framework zum Erstellen und Visualisieren von Graphen, Tabellen und Bäumen. Die Datenfelder lassen sich beliebig mit eigenen Daten erweitern. *4*

SVG

Ein auf XML basiertendes 2D-Vektorgrafikformat. Es unterstützt sowohl die üblichen grafischen Primitive, als auch Rastergrafiken, Text und Animationen. *4*

SVN

SVN (Subversion) ist eine Open-Source-Software zur Versionsverwaltung. Es kontrolliert den gemeinsamen Zugriff von mehreren Entwicklern auf die Dateien eines Projektes. *5*

UML

UML (Unified Modelling Language) ist eine Sprache, um die Struktur und das Verhalten eines objektorientierten Programmes darzustellen. Es unterstützt verschiedene Diagrammarten, z.B. Klassendiagramme und Objektdiagramme. *5*

Literatur

[1] Batik SVG Toolkit. http://xmlgraphics.apache.org/batik/.

[2] Bean Scripting Framework. http://jakarta.apache.org/bsf/.

[3] BeanShell. http://www.beanshell.org/home.html.

[4] Coverlipse. http://coverlipse.sourceforge.net/index.php.

[5] Developing Eclipse plug-ins. http://www-128.ibm.com/developerworks/opensource/library/os-ecplug/.

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

[7] Java Universal Network/Graph Framework. http://jung.sourceforge.net/.

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

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

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

[11] the prefuse visualization toolkit. http://prefuse.org/.

[12] UMLet 7.1. http://www.umlet.com/.

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

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

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