Bachelor - Praktikum (Graphenlayout)

Gesamt Dokumentation

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 Pflichtenheft **5**
1.1 Einleitung 6
1.2 Vision 6
1.3 Ist 6
1.4 Soll 7
1.5 Vergleich Ist/Soll 9
1.6 Andere existierende Systeme 9
1.7 Anwendungsfälle 10
1.7.1 Installations-Use-Cases 13
1.7.2 API-Use-Cases 15
1.7.3 GUI-Use-Cases 23
1.7.4 Activity Diagramme 31
1.8 Benutzeroberfläche 33
1.9 Qualitätssicherung 34
1.10 Maßnahmen zur Qualitätssicherung 34
1.10.1 RUP 34
1.10.2 Pair-Programming 34
1.10.3 Codeverwaltung und -dokumentation 34
1.10.4 Tests und Bugs 34
1.10.5 Plattformdiversität 35
1.10.6 Nutzen von Modulen 35
1.10.7 Abbruchbedingung der Tests 35
1.11 Planung 35
1.12 Änderungshistorie 37

2 Design **39**
2.1 Einleitung 40
2.2 Systemüberblick 40
2.3 Design Überlegungen 41
2.3.1 Annahmen und Abhängigkeiten 41
2.3.2 Allgemeine Vorgaben 41
2.3.3 Designziele und Richtlinien 42
2.3.4 Entwicklungsmethode 42
2.3.5 Begründung für Entscheidungen 43
2.3.6 Kurzbeschreibung des Einsatzes von Design Patterns 43
2.4 System Architektur 44
2.4.1 Subsysteme und ihre Aufgaben 44
2.4.2 Kooperation der Teilsysteme 45
2.4.3 Beschreibung der wichtigsten Klassen 54
2.5 Klassenübersicht 56
2.5.1 Packages und ihre Responsibility 56
2.5.2 Wichtigsten Designklassen und ihre Responsibility 56
2.6 Änderungshistorie 57

3 Qualitätssicherung **59**
3.1 Einleitung . . . 60
3.2 Qualitätsmerkmale . . . 60
3.2.1 Funktionalität . . . 60
3.2.2 Zuverlässigkeit . . . 60
3.2.3 Benutzbarkeit . . . 60
3.2.4 Effizienz . . . 60
3.2.5 Änderbarkeit . . . 61
3.2.6 Übertragbarkeit . . . 61
3.3 Maßnahmen zur Qualitätssicherung . . . 62
3.3.1 RUP . . . 62
3.3.2 Pair-Programming . . . 62
3.3.3 Codeverwaltung und -dokumentation . . . 62
3.3.4 Tests und Bugs . . . 62
3.3.5 Plattformdiversität . . . 63
3.3.6 Nutzen von Modulen . . . 63
3.3.7 Abbruchbedingung der Tests . . . 63
3.4 Testplan . . . 64
3.4.1 Use-Cases . . . 64
3.4.2 Komponententest . . . 65
3.4.3 Methodentest . . . 65
3.4.4 Benutzbarkeit . . . 65
3.5 Prüfung der Zeitplanung . . . 65
3.6 Testergebnisse . . . 68
3.6.1 Anwendungsfalltests . . . 70
3.6.2 Komponententest . . . 74
3.6.3 Methodentest . . . 74
3.6.4 Beispiele . . . 76
3.6.5 Benutzbarkeit . . . 76
3.6.6 Zusammenfassung . . . 76
3.7 Änderungshistorie . . . 77

4 Anhang **79**
Glossar . . . 80
Literatur . . . 83

Kapitel 1

Pflichtenheft

1.1 Einleitung

Dieses Dokument spiegelt unsere Auffassung des Projektes wieder und ist als Angebot an den Auftraggeber zu verstehen. Durch das Gegenlesen können Missverständnisse schon früh im Projekt vermieden werden.

Die Vision enthält einen groben Überblick darüber, welches Ziel mit der Entwicklung des Produktes erreicht werden soll.

Ist gibt Aufschluss darüber, welche Software bisher verfügbar ist. Deren Vor- und Nachteile werden diskutiert, um den Grund für die Entwicklung des neuen Produktes deutlich zu machen.

Soll entspricht der Ziel-Vision im Detail. Alle Features werden einzeln aufgeführt und erläutert, um einen Überblick über das Ziel der Entwicklung zu geben. entwickelte Software zum Schluss abdecken wird. Hieraus lassen sich weitere Anforderungen ableiten.

1.2 Vision

Es soll ein Eclipse-Plugin entstehen, um kleine bis mittlere Graphen (etwa 1 bis 50 Knoten) zu visualisieren. Um dieses und alle weiteren Plugins einfach und schnell verwenden zu können, sollen die Komponenten über eine Eclipse Update-Site verfügbar sein. Alle verwendeten Plug-Ins sind dabei frei verfügbar, so dass keine Probleme mit abhängigen Lizenzen entstehen.

Die Graphen werden über zwei Schnittstellen definiert. Zum einen über eine Java-API, so dass aus anderen Java-Applikationen heraus Graphen erstellt werden können. Zum anderen über XML-Dateien, da XML mittlerweile ein sehr gebräuchliches Standardformat ist und sich somit sehr gut auch zur Speicherung von Graphenstrukturen anbietet.

Hierbei ist eine vollständige Interaktion zwischen beiden Erzeugungsmethoden möglich, d.h. über die API erzeugte Graphen können im XML-Format gespeichert werden und aus dem XML-Format erzeugte Graphen können über die API manipuliert werden. Es wird auch eine Funktion angeboten, um den angezeigten Graph als Bilddatei zu exportieren.

Die Hauptanwendung wird letztendlich die Visualisierung der Graphen in einem Eclipse-Fenster sein. Dabei ist die Berechnung des Graphenlayouts jederzeit für den Benutzer unterbrechbar.

Bei der Betrachtung von mehreren Graphen werden diese überschaubar und übersichtlich angezeigt. Den Knoten können Texte beliebiger Länge zugeordnet werden und sie köennen auf- und zugeklappt werden.

Die Möglichkeit, Subgraphen und Mehrfachkanten zu verwenden, ist vorgesehen. Die bei Bedarf gerichteten Kanten können ebenfalls beschriftet werden und sind zur besseren Unterscheidung farblich markierbar.

Es werden mehrere Knoten- und Kantenpfeiltypen zur Verfügung gestellt. Die Knoten- und Kantentexte lassen sich mit Hilfe von HTML-Tags formatieren.

Zusätzlich kann man Skripte an die Knoten und Kanten hängen, die über einen Rechtsklick in dem jeweiligen Graphen ausgeführt werden. Grundbefehle, auf die die Skripte zugreifen können, sind im Plug-In implementiert. Es lassen sich mehrere verschiedene Skriptsprachen verwenden. Die Graphen werden zusätzlich in einem Overview Fenster angezeigt, das die Navigation vereinfacht, und unterstützen Pan & Zoom- Funktionen.

1.3 Ist

Bislang existiert in der Arbeitsgruppe nur ein Plug-In zur Visualisierung von Graphen, das einen Export als Grafik erlaubt. Allerdings stürzt es hin- und wieder ab. Zudem gibt es manchmal keine Anzeige der Graphen, was sich nur durch einen Eclipse Neustart beheben läßt. Außerdem ist kein XML Dateiformat spezifiziert und es ist nicht möglich, Funktionen wie Zoom in der GUI auszuführen. Auch fehlt eine Möglichkeit, Skripte einzubinden.

1.4 Soll

Das Ziel unseres Projektes kann aus der nachfolgenden Tabelle entnommen werden. Hier sind alle gewünschten Features im Detail beschrieben. Die Prioritäten sind als Schulnoten für das Endergebnis zu verstehen:

5 & 4 sind verpflichtende Features

3 optionale Features, die nach Möglichkeit implementiert werden

2 wünschenswerte Features, deren Implementation im Konzept vorgesehen wird

1 gehört zur ultimativen Ausbaustufe und eigentlich nicht erforderlich

Priorität	Beschreibung
	Eclipse Plug-in
5	Das Eclipse 3.2 Plug-In muss unter Windows und Linux mit denselben Funktionalitäten lauffähig sein. Es dürfen keine Abhängigkeiten zu Tools und Bibliotheken existieren, die nicht unter MacOS X ausführbar sind. Das Plug-In muss unter Java 5 kompilierbar sein.
5	Die Installation erfolgt über eine Eclipse Update-Site, dabei dürfen keine Abhängigkeiten zu Tools existieren, die sich nicht mit Hilfe dieser installieren lassen.
5	Die simultane Anzeige mehrerer Graphen, z.B. nebeneinander, sollte möglich sein.
4	Berechnungen sollen abbrechbar sein, falls sie zu lange dauern
3	Die Anzeige der Graphen sollte als Eclipse View erfolgen.
	Visualisierung eines Graphen
5	Knoten können mit einem mehrzeiligen Text beschriftet werden.
5	Als Zeichensatz für den Text eines Knotens sollte UTF-8 (oder UTF-16) unterstützt werden - die Zeichen, die außerhalb des ASCII Zeichensatzes definiert sind, müssen auch unterstützt werden.
5	Die Form und das Layout der Knoten sind variabel. Es lassen sich Farben und die Dicke des Rahmens variieren.
5	Mehrfachverbindungen zwischen zwei Knoten müssen unterstützt werden und auch visuell erkennbar sein
5	Mehrfachverbindungen eines Knotens mit sich selbst müssen erkennbar sein.
5	Gerichtete und ungerichtete Verbindungen müssen unterstützt werden.
4	Es sollten mindestens fünf verschiedene Typen von Verbindungen unterstützt werden, erkennbar durch unterschiedliche Farbe und Dicke der Linien
4	Es sollte möglich sein, Kanten zu beschriften.
4	Es sollten mindestens drei verschiedene Start- und Endknotentypen verfügbar sein.
3	Unterstützung von Subgraphen.
3	Als Knotentext kann HTML formatierter Text verwendet werden.
3	Zusammengesetzte Knoten, d.h. es sollte möglich sein, Text nebeneinander zu setzen.
2	Knoten sollten faltbar sein, d.h. der Benutzer sollte in der Lage sein, Knoten mit viel Text zusammenzufalten und dann nur einen Kurzbezeichner zu sehen.

Priorität	Beschreibung
	Interaktion mit dem Graphen
5	Unterstützung für ein Skript, z.B. in JavaScript, welches an einen Knoten angehängt wird
3	Unterstützung für mehr als ein Skript.
3	Filterung von verschiedenen Verbindungen sollte möglich sein, so dass man Verbindungstypen ausschalten kann
3	Es sollte möglich sein, Basisfunktionalitäten vorzudefinieren, auf die dann in den Skripten zugegriffen werden kann. Zum Beispiel eine Funktion die bei gegebenen Methodennamen und Namen der deklarierenden Klasse einen Editor öffnet und zur entsprechenden Methode navigiert.
3	Es sollte möglich sein, angezeigte Graphen als Grafiken exportieren zu können.
2	Ein Overview Fenster, das bei größeren Graphen anzeigt, wo man sich befindet.
1	Unterstützung für mehr als eine Skriptsprache.
1	Es sollte möglich sein, an Verbindungen zwischen zwei Knoten Skripte zu hängen.
	Sonstiges Anforderungen
5	Es dürfen keine offensichtlichen lizenzrechtlichen Beschränkungen existieren, die die freie zur Verfügungstellung des Plug-ins behindern.
5	Eine Webseite inkl. Eclipse Update-Site, welche das Plug-in vorstellt und die Verwendung erklärt.

1.5 Vergleich Ist/Soll

Wesentliche Neuerungen im neuentwickelten Plug-In sind die Installierbarkeit über eine Update-Site, im Gegensatz zur vorher komplizierten Installation von Hand. Außerdem war bei den alten Tools keine Unterstützung für das Anhängen von Skripten an Knoten und Kanten vorhanden. Zudem konnte die Verarbeitung der Daten nur in mehreren Einzelschritten erfolgen - zentrale Aufgabe im neuen Plugin ist es also, die wichtigen Funktionen in einem Plugin zu vereinen.

Zusammenfassend lässt sich sagen, dass die bisherige Lösung zwar schon einen großen Teil der Funktionalität bietet, aber wenig benutzerfreundlich, unzuverlässig und umständlich ist.

1.6 Andere existierende Systeme

Im Internet gibt es recht viele Systeme, die sich zum Anzeigen von Graphen eignen, viele sind allerdings kommerziell oder nicht besonders stabil bzw. nicht weit entwickelt. Es gibt aber auch einige Frameworks, die einen stabilen Zustand erreicht haben und bereits in vielen Projekten eingesetzt werden. Im Bereich der Java-Tools sind Prefuse ([11]) und das jung-Framework ([7]) zu erwähnen. Während prefuse sich eher auf die graphische Visualisierung spezialisiert, liegt bei jung der Schwerpunkt eher bei Graphenalgorithmen. Was wir allerdings nicht im Internet finden konnten, ist ein verwendbares, freies Plugin zur Visualisierung von Graphen in Eclipse. Auch die Skriptunterstützung als relativ spezielles Feature war bei keinem der gefunden Tools vorhanden.

1.7 Anwendungsfälle

Anwendungsfälle sind abstrakte Modelle von Funktionalitäten oder Diensten, welche das System den Benutzern (auch als Akteure bezeichnet) anbietet. Jeder Fall beschreibt dabei ein Szenario, wie das System mit dem Akteur interagieren kann, um ein gewisses Ziel zu erreichen. Anwendungsfälle betrachten das System als 'black box' und haben daher nur Einsicht auf Dinge, die vom System nach außen dringen.
Hier als Beispiel einen leeren Anwendungsfall, gefolgt von der Definition der einzelnen Felder:

Name		
ID		
Status		
Priorität		
Schwierigkeit		
Akteur		
Kurzbeschreibung		
Vorbedingung		
Nachbedingung		
	Aktion	Reaktion
Normaler Ablauf		
Alternativer Ablauf		

Name: Der eindeutige Name des Anwendungsfalls, der einen Hinweis auf die Funktion gibt.

ID: Eine einzigartige ID, die einen Hinweis darauf liefert, in welchem Zusammenhang der Anwendungsfall mit den Anderen steht, und in welchem Fall er eingesetzt wird (zB API).
Format: UC-[INS|API|GUI]-*

Status: Der Fortschritt der Umsetzung (In Planung, In Arbeit, Abgeschlossen)

Priorität: Wie wichtig die Umsetzung ist (Hoch, Mittel, Niedrig)

Schwierigkeit: geschätzter Aufwand zur Umsetzung: (Hoch, Mittel, Niedrig)

Akteur: Akteur, der den Anwendungsfall auslöst. Dies können bei uns folgende sein:

- **System Administrator** Benutzer, der eine Eclipseinstallation verwaltet. Es wird erwartet, dass er sich mit der Installation von Eclipse sowie der Eclipse-Komponenten über Update-Sites auskennt.
- **GUI User** Benutzer, der mit Hilfe von Maus und Tastatur in Eclipse das Plugin verwendet. Ein GUI-Benutzer kennt sich mit der Verwendung der Eclipse-Oberfläche aus und kennt die typischen Designmuster der Oberfläche von Eclipse (Kontextmenüs fast überall, Editoren, Views).
- **API User** Ein Entwickler, der ein Eclipse-Plugin-Programm schreibt, das über die von uns angebotene Schnittstelle auf die von unserem Plugin zu Verfügung gestellte Funktionalität zugreift. Bei einem Entwickler wird davon ausgegangen, dass er bereits Eclipse-Plugins entwickelt hat und sich mit der Struktur von Eclipse-Plugins auskennt.

Kurzbeschreibung: Eine kurze Beschreibung, was der eigentliche Sinn des Anwendungsfalls ist. Dient dazu, dass der Leser nicht alle Details lesen muss um zu verstehen, worum es geht.

Vorbedingung: Die Vorbedingungen geben an, welche Bedingungen erfüllt sein müssen, damit der Benutzer den Anwendungsfall initiieren kann. Sind diese Bedingungen nicht erfüllt, so funktioniert der Anwendungsfall nicht.

Nachbedingung: Die Nachbedingungen geben an, in welchem Zustand sich das System nach der Ausführung befindet.

Normaler Ablauf: Der detaillierte Ablauf des Anwendungsfalls. In Listenform.

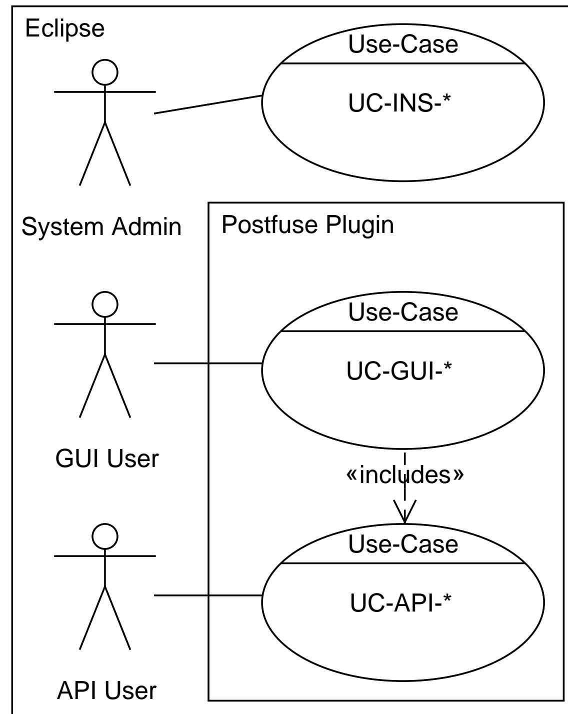
Alternativer Ablauf: Alternative Abläufe. Auch hier wieder als Liste, wobei auf Punkte aus dem normalen Ablauf verwiesen werden kann.

Aktion: Aktive Einflüße die das System beim Durchlaufen des Anwendungsfalls betreffen.

Reaktion: Das direkte Reaktion des Systems auf die unter Aktion ausgeführten Schritte.

In Abbildung 1.1 lässt sich ein guter Überblick über die Struktur der Anwendungsfälle gewinnen.

Abbildung 1.1: generisches Anwendungsfalldiagramm



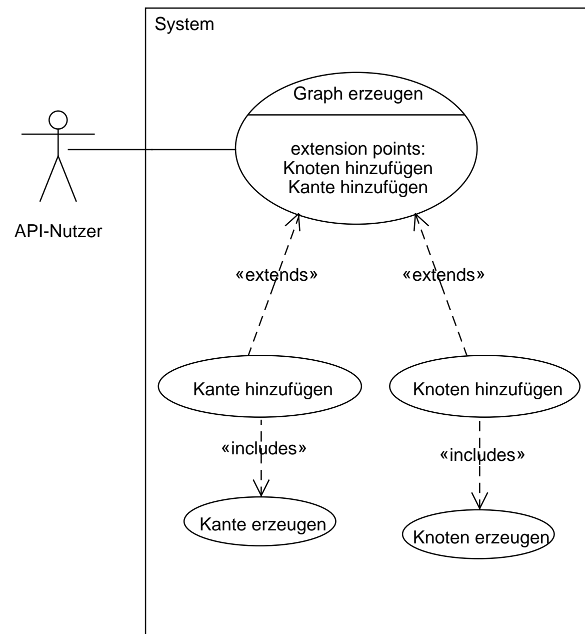
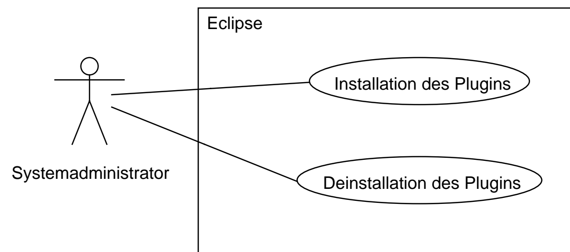


Abbildung 1.2: Erzeugen eines Graphen über API

1.7.1 Installations-Use-Cases

Die Installation unseres Plugins gehört explizit zu unserem Aufgabenbereich. Somit sind auch hier Anwendungsfälle im Folgenden vorhanden.



Installation des Plugins		
ID	UC-INS-Install	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	Systemadministrator	
Kurzbeschreibung	Das Plugin ist über eine Update-Site installierbar.	
Vorbedingung	Die Eclipse-Platform ist in Version 3.2 installiert.	
Nachbedingung	Das Plugin und alle benötigten vorausgesetzten Plugins wurden installiert.	
	Aktion	Reaktion
Normaler Ablauf	1 Der Benutzer verwendet die Update-Routine des Eclipse-Platform, um das Plugin zu installieren. 3 Der Benutzer startet Eclipse neu.	2 Das Plugin wird mit den auf der Update-Seite angegebenen Parametern installiert.
Alternativer Ablauf		

Deinstallation des Plugins		
ID	UC-INS-DeInstall	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Niedrig	
Akteur	Systemadministrator	
Kurzbeschreibung	Das Plugin lässt sich mit Eclipsemitteln deinstallieren.	
Vorbedingung	Die Eclipse-Platform ist in Version 3.2 installiert. Das Plugin ist installiert.	
Nachbedingung	Das Plugin und alle dazugehörigen Dateien sind gelöscht.	
	Aktion	Reaktion
Normaler Ablauf	1 Der Benutzer verwendet die Update-Routine des Eclipse-Platform, um das Plugin zu deinstallieren. 3 Der Benutzer startet Eclipse neu.	2 Das Plugin und alle dazugehörigen Dateien werden gelöscht.
Alternativer Ablauf		

1.7.2 API-Use-Cases

Ein Teil der Aufgabenstellung fordert, dass unser Programm eine API bereitstellen soll. Die folgenden Anwendungsfälle beschreiben, wie die Interaktion damit geschehen soll. Vorerst ist kein Löschen von Objekten wie z.B. Knoten und Kanten vorgesehen. Da die Graphen sowieso von anderen Programmen über die API erstellt werden, ist damit auch kein Zusatzaufwand für den Benutzer verbunden.

Erzeugen des Graphen		
ID	UC-API-CreateGraph	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Erzeugen eines Graph-Objekts	
Vorbedingung	Installiertes Plugin.	
Nachbedingung	Ein Graph-Objekt wurde erstellt.	
	Aktion	Reaktion
Normaler Ablauf	1a Die Methode createGraph() des Plugins wird aufgerufen.	2 Das Graph-Objekt wird erzeugt und zurückgegeben.
Alternativer Ablauf	1b Der Graph wird über den Konstruktor erzeugt.	

Hinzufügen eines Knotens		
ID	UC-API-AddNode	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Knoten in Graph einfügen	
Vorbedingung	Installiertes Plugin. Es muss ein Graph erzeugt worden sein	
Nachbedingung	Ein Knoten wurde hinzugefügt.	
	Aktion	Reaktion
Normaler Ablauf	1 Die 'addNode' Funktion des Graph-Objekts, der als Factory dient, wird aufgerufen. Optionaler Parameter ist hierbei ein NodeDesign-Objekt, das die graphische Darstellung definiert.	2 Das Graph-Objekt hat einen Knoten mehr und der erzeugte Knoten wird zurückgegeben.
Alternativer Ablauf		

Hinzufügen eines Subgraphen

ID	UC-API-AddSubgraph	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Subgraphen in Graph einfügen	
Vorbedingung	Installiertes Plugin. Es muss ein Graph erzeugt worden sein	
Nachbedingung	Ein Subgraph wurde hinzugefügt.	
	Aktion	Reaktion
Normaler Ablauf	1 Die 'addSubgraph' Funktion des Graph-Objekts, der als Factory dient, wird aufgerufen. Optionaler Parameter ist hierbei ein NodeDesign-Objekt, das die graphische Darstellung definiert.	2 Das Graph-Objekt hat einen Subgraphen mehr und der erzeugte Subgraph wird zurückgegeben.
Alternativer Ablauf		

Hinzufügen einer Kante

ID	UC-API-AddEdge	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Kante in Graph einfügen	
Vorbedingung	Installiertes Plugin. Es müssen ein Graph- und zwei Kanten-Objekte erzeugt worden sein.	
Nachbedingung	Der Graph enthält zusätzlich die eingefügte Kante	
	Aktion	Reaktion
Normaler Ablauf	1 Über die Java-API wird die Funktion 'addEdge' aufgerufen mit Ursprungs- und Zielknoten als Parameter. Optionaler Parameter ist hierbei ein NodeDesign-Objekt, das die graphische Darstellung definiert.	2a Die Kante wird im Graph-Objekt eingefügt und zurückgegeben.
Alternativer Ablauf		2b Falls ein Fehler bei der Prüfung der beiden Knoten oder eine Graphstrukturverletzung auftritt, wird keine Kante hinzugefügt, sondern eine GraphStructureException geworfen.

Erweitern der Skriptumgebung		
ID	UC-API-ScriptEnv	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Hoch	
Akteur	API User	
Kurzbeschreibung	Es können EInträge zur Skriptumgebung eines Skripts hinzugefügt werden.	
Vorbedingung	Installiertes Plugin.	
Nachbedingung	Die Skriptumgebung wurde um einen Eintrag erweitert.	
	Aktion	Reaktion
Normaler Ablauf	1 Die Skriptumgebung wird vom Skript angefordert. 2 Es wird ein Eintrag hinzugefügt.	3 Die interne Skriptumgebung wird erweitert.
Alternativer Ablauf		

Erzeugen eines Skriptes		
ID	UC-API-CreateScript	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Anlegen eines Skripts	
Vorbedingung	Installiertes Plugin.	
Nachbedingung	Es wurde ein Skript angelegt.	
	Aktion	Reaktion
Normaler Ablauf	1 Das Skript wird über die ScriptFactory, die man vom Plugin anfordern kann, erzeugt. Parameter dabei sind: Beschriftung und Code oder Datei sowie der Klassename der Skriptklasse für die gewünschte Skriptsprache.	2 Skript-Objekt wird erzeugt und zurückgegeben.
Alternativer Ablauf		

Binden eines Skriptes an Knoten oder Kante		
ID	UC-API-AddScript	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Knoten oder Kante mit Skript hinzufügen	
Vorbedingung	Installiertes Plugin. Es muss ein Skript und ein Knoten- bzw. Kanten-Objekt erzeugt worden sein.	
Nachbedingung	Das Skipt wurde mit dem Objekt verbunden.	
	Aktion	Reaktion
Normaler Ablauf	1 Die 'addScript'-Funktion des Knoten- oder Kanten-Objekts wird aufgerufen.	2 Das Knoten- oder Kanten-Objekt hat ein Skript mehr in seinem Kontextmenü.
Alternativer Ablauf		

Ausführen eines Skripts		
ID	UC-API-RunScript	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Ein Skript-Objekt wird in seiner Umgebung ausgeführt.	
Vorbedingung	Installiertes Plugin. Es ist ein Skript-Objekt definiert.	
Nachbedingung	Das Skript-Objekt wurde erfolgreich ausgeführt.	
	Aktion	Reaktion
Normaler Ablauf	1 Die 'run()' Funktion des Skriptobjekts wird aufgerufen.	2 Ein Interpreter für die entsprechende Sprache führt das Skript in einer neuen Umgebung aus unter Berücksichtigung der Skriptumgebung.
Alternativer Ablauf		

Setzen eines Filters		
ID	UC-API-SetFilter	
Status	Abgeschlossen	
Priorität	Mittel	
Schwierigkeit	Hoch	
Akteur	API User	
Kurzbeschreibung	Ein Filter-Objekt wird erzeugt und abgelegt.	
Vorbedingung	Installiertes Plugin.	
Nachbedingung	Das Filter-Objekt wurde erzeugt und gespeichert.	
	Aktion	Reaktion
Normaler Ablauf	1 Das Filter-Objekt wurde über seinen Konstruktor mit entsprechenden Parametern erzeugt.	2 Das Filter-Objekt wird erzeugt und gespeichert.
Alternativer Ablauf		

Rücksetzen eines Filters		
ID	UC-API-ClearFilter	
Status	Abgeschlossen	
Priorität	Mittel	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Der Filter wird zurückgesetzt.	
Vorbedingung	Installiertes Plugin.	
Nachbedingung	Es ist kein Filter gesetzt.	
	Aktion	Reaktion
Normaler Ablauf	1 Die Funktion clearFilter() wird aufgerufen.	2 Der Filter wird zurückgesetzt.
Alternativer Ablauf		

Berechnung des Graphenlayouts		
ID	UC-API-Layout	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Hoch	
Akteur	API User	
Kurzbeschreibung	Das Layout des Graphen wird basierend auf der vorliegenden Definition berechnet	
Vorbedingung	Installiertes Plugin. Ein Graph ist vollständig definiert	
Nachbedingung	Das Layout des Graphen ist bekannt	
	Aktion	Reaktion
Normaler Ablauf	1 Ein Algorithmus für das Layouten des Graphen wird in einem neuen Thread gestartet.	2 Das Layout wird unter berücksichtigung von Filtern und ein- bzw. ausgeklappten Knoten berechnet. 3a Das fertige Layout wird intern gespeichert.
Alternativer Ablauf		3b Bei fehlerhaft definiertem Graphen wird eine Fehlermeldung ausgegeben.

Zeichnen von Graphen in der GUI		
ID	UC-API-Draw	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Das berechnete Graphenlayout wird angezeigt.	
Vorbedingung	Installiertes Plugin. Es ist ein Graph definiert und es hat eine dazugehörige Layoutberechnung erfolgreich stattgefunden.	
Nachbedingung	Ein Fenster mit dem angezeigten Graphen ist geöffnet.	
	Aktion	Reaktion
Normaler Ablauf	1 Die Zeichenfunktion wird aufgerufen.	2 Der Graph wird gezeichnet.
Alternativer Ablauf		

Exportieren von Graphen nach SVG

ID	UC-API-Export-SVG
Status	Abgeschlossen
Priorität	Hoch
Schwierigkeit	Hoch
Akteur	API User
Kurzbeschreibung	Das berechnete Graphenlayout wird in einer SVG-Datei gespeichert.
Vorbedingung	Installiertes Plugin. Es ist ein Graph definiert und es hat eine dazugehörige Layoutberechnung erfolgreich stattgefunden.
Nachbedingung	Es existiert eine SVG-Datei, die dem Graphenlayout entspricht.

	Aktion	Reaktion
Normaler Ablauf	1 Die Exportfunktion wird unter Angabe des Dateinamens aufgerufen.	2 Der Graph wird in die entsprechnede Datei exportiert.
Alternativer Ablauf		

Exportieren von Graphen nach PNG

ID	UC-API-Export-PNG
Status	Abgeschlossen
Priorität	Hoch
Schwierigkeit	Hoch
Akteur	API User
Kurzbeschreibung	Das berechnete Graphenlayout wird in PNG Datei gespeichert.
Vorbedingung	Installiertes Plugin. Es ist ein Graph definiert und es hat eine dazugehörige Layoutberechnung erfolgreich stattgefunden.
Nachbedingung	Es existiert eine PNG-Datei, die dem Graphenlayout entspricht.

	Aktion	Reaktion
Normaler Ablauf	1 Die Exportfunktion wird unter Angabe des Dateinamens aufgerufen.	2 Der Graph wird in die entsprechnede Datei exportiert.
Alternativer Ablauf		

Speichern eines Graphen

ID	UC-API-Save
Status	Abgeschlossen
Priorität	Hoch
Schwierigkeit	Mittel
Akteur	API User
Kurzbeschreibung	Der Graph inklusive der Skripte, allerdings ohne Layout, wird gespeichert.
Vorbedingung	Installiertes Plugin. Es ist ein Graph definiert.
Nachbedingung	Der Graph inklusive der Skripte, allerdings ohne Layout, ist in die Datei geschrieben worden.

	Aktion	Reaktion
Normaler Ablauf	1 Die Speichern-Funktion wird unter Angabe des Dateinamens gestartet.	2 Der Graph inklusive der Skripte, allerdings ohne Layout, wird in die Datei geschrieben.
Alternativer Ablauf		

Laden eines Graphen		
ID	UC-API-Load	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	API User	
Kurzbeschreibung	Basierend auf einer gespeicherten Graphendefinition werden Objekte erzeugt, die diesen repräsentieren	
Vorbedingung	Installiertes Plugin. Eine Datei im passenden Format zum Laden ist vorhanden.	
Nachbedingung	Der Graph wurde intern erzeugt.	
	Aktion	Reaktion
Normaler Ablauf	1 Die Lade-Funktion wird unter Angabe des Dateinamens gestartet.	2a Die entsprechenden Objekte werden aus der Datei geladen und angelegt.
Alternativer Ablauf		2b Die Struktur der Datei ist fehlerhaft und es wird eine Exception geworfen.

1.7.3 GUI-Use-Cases

Alle übrigen Use-Cases beziehen sich auf die Interaktion mittels der graphischen Oberfläche mit unserem Plugin.

Zoomen der Ansicht		
ID	UC-GUI-Zoom	
Status	Abgeschlossen	
Priorität	Mittel	
Schwierigkeit	Niedrig	
Akteur	GUI User	
Kurzbeschreibung	Die Graphenansicht kann vergrößert und verkleinert werden.	
Vorbedingung	Im Anzeigefenster angezeigter Graph.	
Nachbedingung	vergrößerter bzw. verkleinerter Graph im Anzeigefenster.	
	Aktion	Reaktion
Normaler Ablauf	1a Der Nutzer rollt das Rollrad seiner Maus.	2 Die Größe des Graphens im Fenster ändert sich.
Alternativer Ablauf	1b Der Nutzer klickt auf den Zoomin/out-Button in der Toolbar. 1c Der Nutzer klickt auf Zoomin/out im Popup-Menu, das per Rechtsklick auf den Hintergund des Graphen zu erreichen ist.	

Verschieben der Ansicht (‚Pan‘)		
ID	UC-GUI-Pan	
Status	Abgeschlossen	
Priorität	Mittel	
Schwierigkeit	Niedrig	
Akteur	Benutzer	
Kurzbeschreibung	Die Graphenansicht kann verschoben werden.	
Vorbedingung	Im Anzeigefenster angezeigter Graph.	
Nachbedingung	Graph ist im Anzeigefenster verschoben	
	Aktion	Reaktion
Normaler Ablauf	1 Der Nutzer klickt auf eine leere Position im Graphen und bewegt den Mauszeiger bei gedrückter linker Maustaste.	2 Die Position des Graphen im Fenster ändert sich.
Alternativer Ablauf		

Graphgröße an Fenster anpassen		
ID	UC-GUI-ZoomToFit	
Status	Abgeschlossen	
Priorität	Mittel	
Schwierigkeit	Mittel	
Akteur	GUI User	
Kurzbeschreibung	Die Zoom und Pan Einstellungen werden zurückgesetzt.	
Vorbedingung	Im Anzeigefenster angezeigter Graph.	
Nachbedingung	Der Graph ist vollständig im Anzeigefenster sichtbar.	
	Aktion	Reaktion
Normaler Ablauf	1a Der Nutzer klickt auf den ZoomToFit Button.	2 Der Graph wird vollständig im Fenster angezeigt.
Alternativer Ablauf	1a Der Nutzer benutzt den Eintrag im Kontextmenü des Graphenhintergrunds.	

Anzeigen des Graphen-Overviews		
ID	US-GUI-OverviewShow	
Status	Abgeschlossen	
Priorität	Niedrig	
Schwierigkeit	Hoch	
Akteur	GUI User	
Kurzbeschreibung	Der GUI User kann sich einen Overview des Graphen anzeigen lassen.	
Vorbedingung	Das Graphenlayout wurde berechnet und es wird in der View angezeigt.	
Nachbedingung	Man sieht zusätzlich ein kleines Overview-Fenster mit dem Graph.	
	Aktion	Reaktion
Normaler Ablauf	1 Der GUI-User fordert über die Eclipse-Plattform ein Overview-Fenster als View an.	2 Ein Overview-Fenster wird geöffnet. 3 Der Graph wird im Overview-Fenster so skaliert, dass er komplett sichtbar ist.
Alternativer Ablauf		

Ein- und Ausfalten eines Knoten		
ID	UC-GUI-Fold	
Status	Abgeschlossen	
Priorität	Niedrig	
Schwierigkeit	Niedrig	
Akteur	GUI User	
Kurzbeschreibung	Knoten wird zusammengefaltet, falls er ausgefaltet war bzw. eingefaltet, falls er ausgefaltet war.	
Vorbedingung	Es wird ein Graph angezeigt mit dem Knoten im anderen Faltungszustand.	
Nachbedingung	Der Faltungszustand des Knoten wurde gewechselt.	
	Aktion	Reaktion
Normaler Ablauf	1 Der GUI User klickt auf 'folded' im Kontextmenü des Knoten.	2 Der Knoten wird gefaltet bzw. ausgefaltet. 3 Die Anzeige wird aktualisiert mit UC-API-Draw.
Alternativer Ablauf		

Anzeige zweier Graphen nebeneinander		
ID	UC-GUI-Dual	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Niedrig	
Akteur	GUI User	
Kurzbeschreibung	Es können zwei Graphen nebeneinander angezeigt werden	
Vorbedingung	Es sind zwei Editoren offen, die Graphen zeigen.	
Nachbedingung	Zwei Fenster nebeneinander, in denen jeweils ein Graph zu sehen ist.	
	Aktion	Reaktion
Normaler Ablauf	1 Über die Leiste des Editors wird dieser mit der Maus neben den anderen gezogen.	2 Der Editor wird von Eclipse neu positioniert.
Alternativer Ablauf		

Setzen eines Filters		
ID	UC-GUI-SetFilter	
Status	Abgeschlossen	
Priorität	Mittel	
Schwierigkeit	Mittel	
Akteur	GUI User	
Kurzbeschreibung	Nur gewisse Graphelemente werden angezeigt.	
Vorbedingung	Ein Graph wird angezeigt.	
Nachbedingung	Der Graph wird entsprechend dem Filter gefiltert angezeigt.	
	Aktion	Reaktion
Normaler Ablauf	1 Im Filtermenü werden Filter gesetzt.	2 Es wird UC-API-SetFilter angewendet. 3 Die Anzeige wird aktualisiert mit UC-API-Draw.
Alternativer Ablauf		

Rücksetzen eines Filters		
ID	UC-GUI-ClearFilter	
Status	Abgeschlossen	
Priorität	Mittel	
Schwierigkeit	Niedrig	
Akteur	GUI User	
Kurzbeschreibung	Der Filter wird zurückgesetzt.	
Vorbedingung	Ein Graph wird angezeigt.	
Nachbedingung	Der Graph wird mit allen Kanten angezeigt.	
	Aktion	Reaktion
Normaler Ablauf	1 Im Filtermenü wird auf 'Filter zurücksetzen' geklickt.	2 Es wird UC-API-ClearFilter angewendet. 3 Die Anzeige wird aktualisiert mit UC-API-Draw.
Alternativer Ablauf		

Berechnung des Graphenlayouts		
ID	UC-GUI-Layout	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Hoch	
Akteur	GUI User	
Kurzbeschreibung	Das Layout des Graphen wird, basierend auf dem vorliegenden PreLayoutFilter und ein/ausgklappten Knoten, berechnet.	
Vorbedingung	Ein Graph wird angezeigt.	
Nachbedingung	Das Layout des Graphen ist aktualisiert.	
	Aktion	Reaktion
Normaler Ablauf	1a Klicken auf den Button zur Neuberechnung des Layouts.	2 Ein Fenster mit einem Abbruch-Button öffnet sich. 3 Es wird UC-API-Layout aufgerufen. 4 Das Fenster mit dem Abbruch-Button wird geschlossen. 5 Es wird UC-API-Draw aufgerufen.
Alternativer Ablauf	1a Es wird auf den entsprechenden Eintrag im Kontextmenü, welches durch Rechtsklick auf den Graphenhintergrund erreichbar ist, geklickt.	

Abbruch der Berechnung des Graphenlayouts

ID	UC-GUI-LayoutAbort
Status	In Arbeit
Priorität	Hoch
Schwierigkeit	Hoch
Akteur	GUI User
Kurzbeschreibung	Die Berechung des Graphenlayouts wird abgebrochen.
Vorbedingung	Es findet im Moment eine Berechnung eines Graphenlayouts statt.
Nachbedingung	Es findet keine Berechnung mehr statt.

	Aktion	Reaktion
Normaler Ablauf	1 Es wird auf den Schließen-Button des aktuellen Editors geklickt.	2 Das aktuelle Editor-Fenster wird geschlossen.
Alternativer Ablauf		

Exportieren von Graphen nach SVG

ID	UC-GUI-Export
Status	Abgeschlossen
Priorität	Hoch
Schwierigkeit	Hoch
Akteur	GUI User
Kurzbeschreibung	Das angezeigte Graphenlayout wird in einer SVG Datei gespeichert.
Vorbedingung	Ein Graph wird angezeigt.
Nachbedingung	Es existiert eine SVG Datei die dem Graphenlayout entspricht.

	Aktion	Reaktion
Normaler Ablauf	1 Es wird auf den Export-Button geklickt. 3 Eingabe des Dateinamens und bestätigen.	2 Ein Dateiauswahldialog wird geöffnet. 4 Es wird UC-API-Export-SVG aufgerufen.
Alternativer Ablauf		

Exportieren von Graphen nach PNG

ID	UC-GUI-Export-PNG
Status	Abgeschlossen
Priorität	Hoch
Schwierigkeit	Hoch
Akteur	GUI User
Kurzbeschreibung	Das angezeigte Graphenlayout wird in einer PNG Datei gespeichert.
Vorbedingung	Ein Graph wird angezeigt.
Nachbedingung	Es existiert eine SVG Datei die dem Graphenlayout entspricht.

	Aktion	Reaktion
Normaler Ablauf	1 Es wird auf den Export-Button geklickt. 3 Der Dateinamen wird eingegeben und bestätigt.	2 Ein Dateiauswahldialog wird geöffnet. 4 Es wird UC-API-Export-PNG aufgerufen.
Alternativer Ablauf		

Ausführen eines Skripts		
ID	UC-GUI-RunSkript	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	GUI User	
Kurzbeschreibung		
Vorbedingung	Ein Graph wird angezeigt. Einem Knoten / einer Kante ist ein Skript zugeordnet.	
Nachbedingung	Das Skript wurde erfolgreich ausgeführt.	
	Aktion	Reaktion
Normaler Ablauf	1 Ein Knoten oder eine Kante wird mit dem rechten Mausknopf angeklickt. 3 Das gewünschte Skript wird aus dem Kontextmenü ausgewählt.	2 Ein Kontextmenü mit der Auflistung der dem Knoten / der Kante zugeordneten Skripte öffnet sich. 4 Es wird UC-API-RunScript aufgerufen.
Alternativer Ablauf		

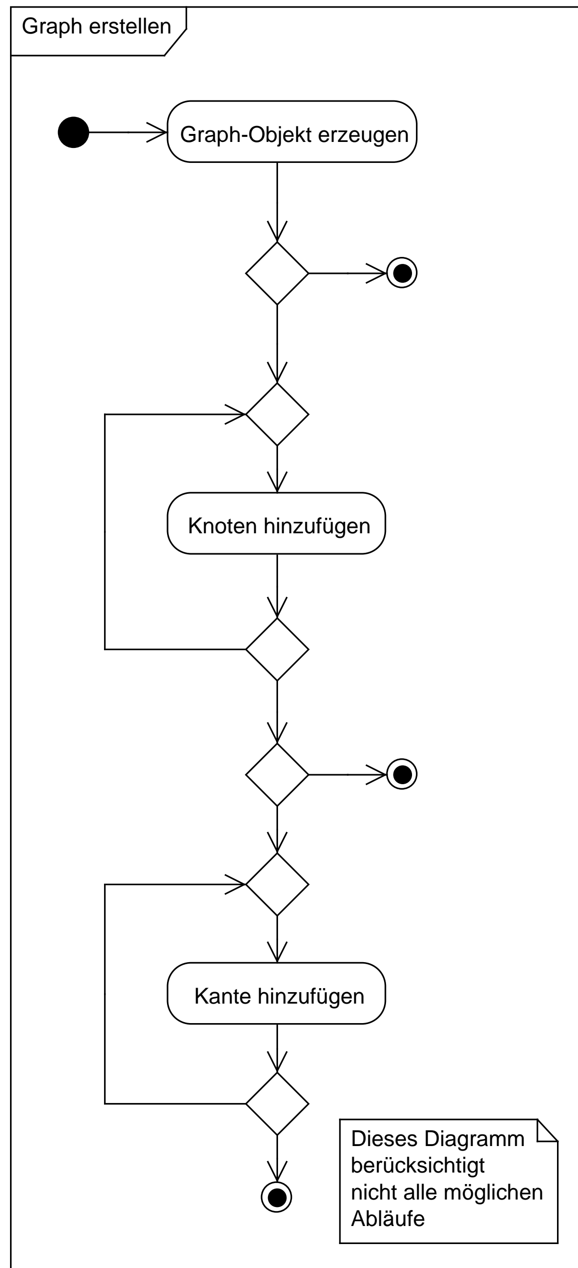
Speichern eines Graphen		
ID	UC-GUI-Save	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	GUI User	
Kurzbeschreibung	Der Graph inklusive der Skripte, allerdings ohne Layout, wird gespeichert.	
Vorbedingung	Ein Graph wird im aktiven Editor angezeigt.	
Nachbedingung	Der Graph inklusive der Skripte, allerdings ohne Layout, ist in die Datei geschrieben worden.	
	Aktion	Reaktion
Normaler Ablauf	1a Der Speichern-Button von Eclipse wird angeklickt. 3 Der Dateinamens wird eingegeben und bestätigt.	2 Ein Dateiauswahldialog wird geöffnet. 4 Es wird UC-API-Save aufgerufen.
Alternativer Ablauf	1b Klick auf 'Speichern unter' in Eclipse.	

Laden eines Graphen		
ID	UC-GUI-Load	
Status	Abgeschlossen	
Priorität	Hoch	
Schwierigkeit	Mittel	
Akteur	GUI User	
Kurzbeschreibung	Der Graph wird aus einer Datei geladen.	
Vorbedingung	Installiertes Plugin. Eine Datei im passenden Format zum laden ist vorhanden.	
Nachbedingung	Der Graph wird angezeigt.	
	Aktion	Reaktion
Normaler Ablauf	1 Es wird auf den Laden-Button geklickt. 3 Der Dateinamen wird eingegeben und bestätigt.	2 Ein Dateiauswahldialog wird geöffnet. 4 Es wird UC-API-Load aufgerufen. 5 Es wird UC-API-Layout aufgerufen. 6 Es wird UC-API-Draw aufgerufen.
Alternativer Ablauf		

1.7.4 Activity Diagramme

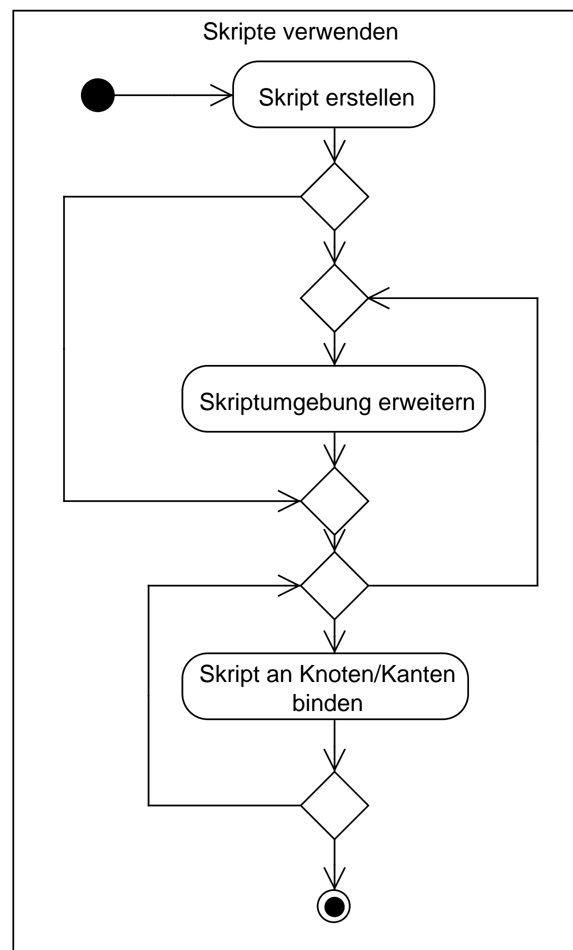
In Abbildung 1.3 lässt sich ein Überblick über den kompletten Ablauf, der zum Erstellen eines Graphen notwendig ist, gewinnen.

Abbildung 1.3: Activity Diagramm zum Erstellen eines Graphens

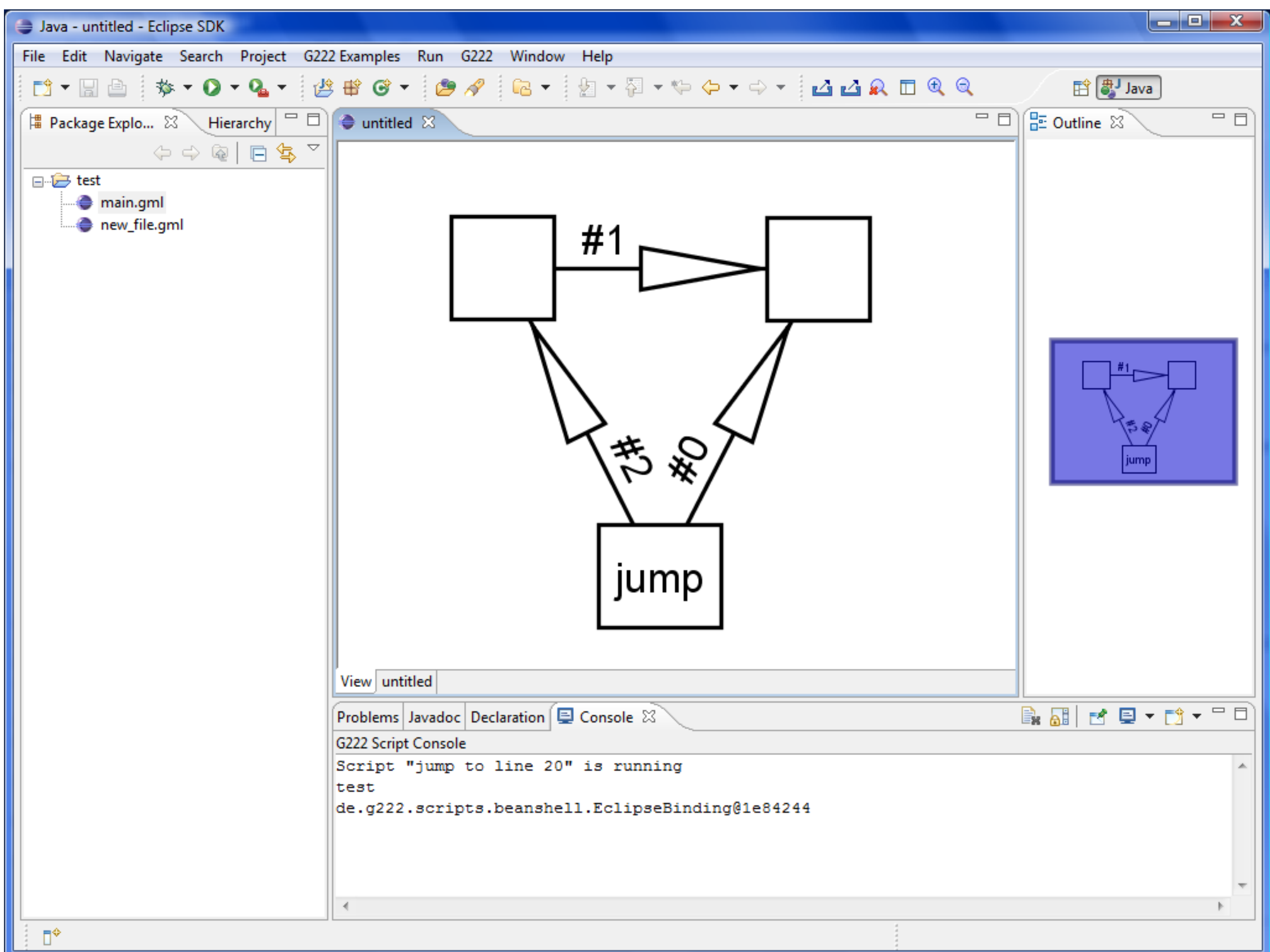


In Abbildung 1.4 auf der nächsten Seite ist der Ablauf beim Erstellen und Hinzufügen von Skripten dargestellt.

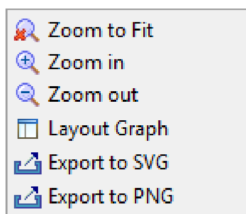
Abbildung 1.4: Activity Diagramm zum Verwenden von Skripten



1.8 Benutzeroberfläche



Dieses Bild zeigt die Benutzeroberfläche unseres Plug-Ins. In der Mitte wird der Graph in einem Eclipse-Editor angezeigt. Rechts in der Outline wird eine Übersicht des gesamten Graphen gezeigt, zusammen mit einem Rechteck über dem Bereich, der im Eclipse-Editor angezeigt wird.



Dies ist das Popup-Menü, das erscheint, wenn man mit dem rechten Mausknopf auf einen leeren Bereich im Bild des Graphen klickt.

1.9 Qualitätssicherung

1.10 Maßnahmen zur Qualitätssicherung

1.10.1 RUP

Als grundlegendes Vorgehensmodell verwenden wir das (RUP)-Modell. Das heißt, wir verwenden eine iterative Entwicklung, die auf komponentenbasierter Architektur aufbaut. Das gewährleistet eine konstante Qualitätssicherung während der Entwicklung. Die einzelnen Disziplinen des RUP-Modells stellen die wichtigsten Stationen dar, welche das Projekt während der Entwicklung durchläuft. Beginnend bei der Analysephase bis zum fertigen Produkt kann man dank der Disziplinen und Iterationen bereits frühzeitig anfangen zu testen.

1.10.2 Pair-Programming

Der Code selbst wird zum Teil mit Hilfe des Pair-Programmings entwickelt. Dadurch können wir problematische Lösungen vermeiden und verbreiten das Wissen des Quellcodes unter uns, was die Qualität des Endproduktes verbessert.

1.10.3 Codeverwaltung und -dokumentation

An technischen Mitteln zur Qualitätssicherung verwenden wir SVN als Versionsverwaltungstool. Zudem wird bei einer Code-Änderung eine EMail an die Team-Mailingliste verschickt und der Code automatisch auf dem Server kompiliert. Die kompilierten Dokumente werden automatisch in den Downloadbereich und das kompilierte Plugin auf die Updatesite der Projekthomepage gestellt. So hat jeder immer die aktuelle, lauffähige Version. Es besteht hierdurch immer die Möglichkeit, mit einer realen Installation über die Updatesite den aktuellen Codestand zu testen.
Javadoc und Code Conventions dienen dazu, dass der Code auch für andere Personen als den Ersteller gut lesbar ist. Die Code Conventions sind z.B.:

- Verwendung von Eclipse Standard-Code-Style
- Methodennamen klein, jedes weitere Wort groß
- kurze und aussagekräftige Variablennamen
- Definitionen (von Variablen, Konstanten) immer am Anfang
- Variablen/Methodennamen auf englisch, Kommentare auf deutsch

1.10.4 Tests und Bugs

Unit Test

Während jeder Iteration unseres Programmes werden immer wieder Tests durchgeführt. Dazu verwenden wir das JUnit Test-Framework, welches uns Hinweise auf die Art des Fehlers liefert (falsches Ergebnis / auftreten eines Fehlers). Die graphische Oberfläche testen wir von Hand, da eine Automatisierung der Tests den Rahmen des Projekts sprengen würde.

Überdeckung

Damit wir auch sicherstellen können, dass die unsere Tests den gesamten Code Abdecken, messen wir dies mit dem Coverlipse Eclipse-Plugin. Dadurch können wir garantieren, dass nichts implementiert wird, was nicht fehlerfrei ist.

Bugtracking

Fehler, die während unseren Erprobungen auftreten, dokumentieren wir mit Mantis. Hiermit wird sichergestellt, dass ein Bug auch dann nicht in Vergessenheit gerät, wenn es nicht direkt eine Möglichkeit gibt, bzw. genügend Zeit zur Verfügung steht, ihn zu beheben.

1.10.5 Plattformdiversität

Die Anforderung der System- und Plattformunabhängigkeit wird bei uns immer implizit mitgetestet, da bei uns in der Projektgruppe eine sehr heterogene Hardware- und Systemumgebung anzutreffen ist.

1.10.6 Nutzen von Modulen

In unserem Projekt wird intensiv von bereits etablierten Komponenten Gebrauch gemacht. Z.B. verwenden wir als Grundgerüst der Graphen Prefuse und für die Skripte entsprechende Interpreter aus dem Umfeld des BSF. Dies stellt sicher, dass es in diesen Bereichen unseres Programms sehr wenige Fehler liegen können, da dieser Code schon in anderen Projekten vielfach genutzt wurde und somit einen gewissen Reifegrad aufweist.

1.10.7 Abbruchbedingung der Tests

Wir werden die Tests abbrechen, wenn wir keine Fehler mehr finden, die die Stabilität des Plugins gefährden und wir keine Zeit mehr haben, kleinere Probleme in der Oberfläche oder bei Features niedriger Priorität zu beheben. Da die Zeit sowieso für ausführliche Tests recht knapp bemessen ist, werden wir bis zuletzt auf Fehler prüfen.

1.11 Planung

Um die vielen Abgabezeitpunkte der Dokumentiterationen fristgerecht einzuhalten, haben wir uns eine Zeitplanung überlegt, die in Abbildung 1.5 zu sehen ist. Die zunächst festgelegte Aufgabenverteilung nach Themen: Marco übernimmt die Einarbeitung in XML-Datenstrukturen. Andreas und Peter evaluieren verschiedene Tools zur Darstellung von Graphen. Martin arbeitet sich in die Eclipse-Plugin-Entwicklung ein. Die Projektleitung übernimmt zunächst Marco, der auch den ersten Review-Vortrag halten wird. Insgesamt ist eine Arbeitszeit von 1000 Mannstunden, d.h. 250 Stunden pro Person von uns für das Projekt veranschlagt.

KW	Datum		Pflichtenheft	Designdokument	Qualitätssicherungs dokument	Implementierung	Test
42	16.10.06						
43	23.10.06						
44	30.10.06		Version 0				
45	06.11.06						
46	13.11.06						
47	20.11.06						
48	27.11.06		Version 1				
49	04.12.06			Version 0			
50	11.12.06		Review				
51	18.12.06						
52	25.12.06						
1	01.01.07						
2	08.01.07			Version 1			
3	15.01.07						
4	22.01.07			Review	Version 0		
5	29.01.07						
6	05.02.07						
7	12.02.07				Version 1	Iteration 1	
8	19.02.07						
9	26.02.07						
10	05.03.07						
11	12.03.07						
12	19.03.07						
13	26.03.07			Überarbeitung	Überarbeitung		
14	02.04.07					Iteration 1.1	
15	09.04.07						
16	16.04.07					Endprodukt	
17	23.04.07						

Abbildung 1.5: Unsere Zeitplanung

1.12 Änderungshistorie

Datum	Thema	Inhalt	seite
05.12.2006	alles	Beginn der History mit auslieferung Revison 1	*
03.04.2007	Korrektur	Überarbeitung der Usecases	*
08.04.2007	alles	finales Release 2	*

Kapitel 2

Design

2.1 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.

2.2 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 2.2 auf Seite 44 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 2.1 sieht.

Abbildung 2.1: Die Plugin-Klasse

2.3 Design Überlegungen

2.3.1 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.

2.3.2 Allgemeine Vorgaben

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.

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.

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).

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 2.3.2 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.

2.3.3 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.

2.3.4 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)

2.3.5 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.

2.3.6 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.

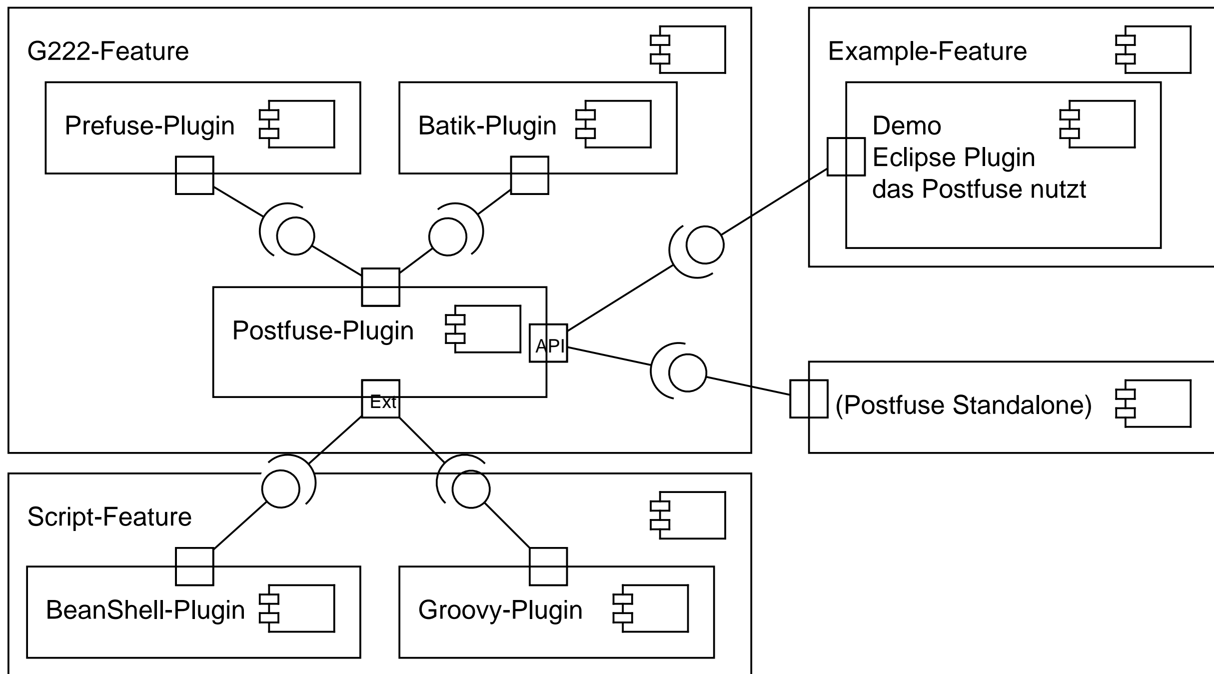


Abbildung 2.2: Die grobe Struktur unseres Programms

2.4 System Architektur

2.4.1 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 2.5.1 auf Seite 56 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 2.2 ist diese Struktur skizziert.

2.4.2 Kooperation der Teilsysteme

Klassenübersicht

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

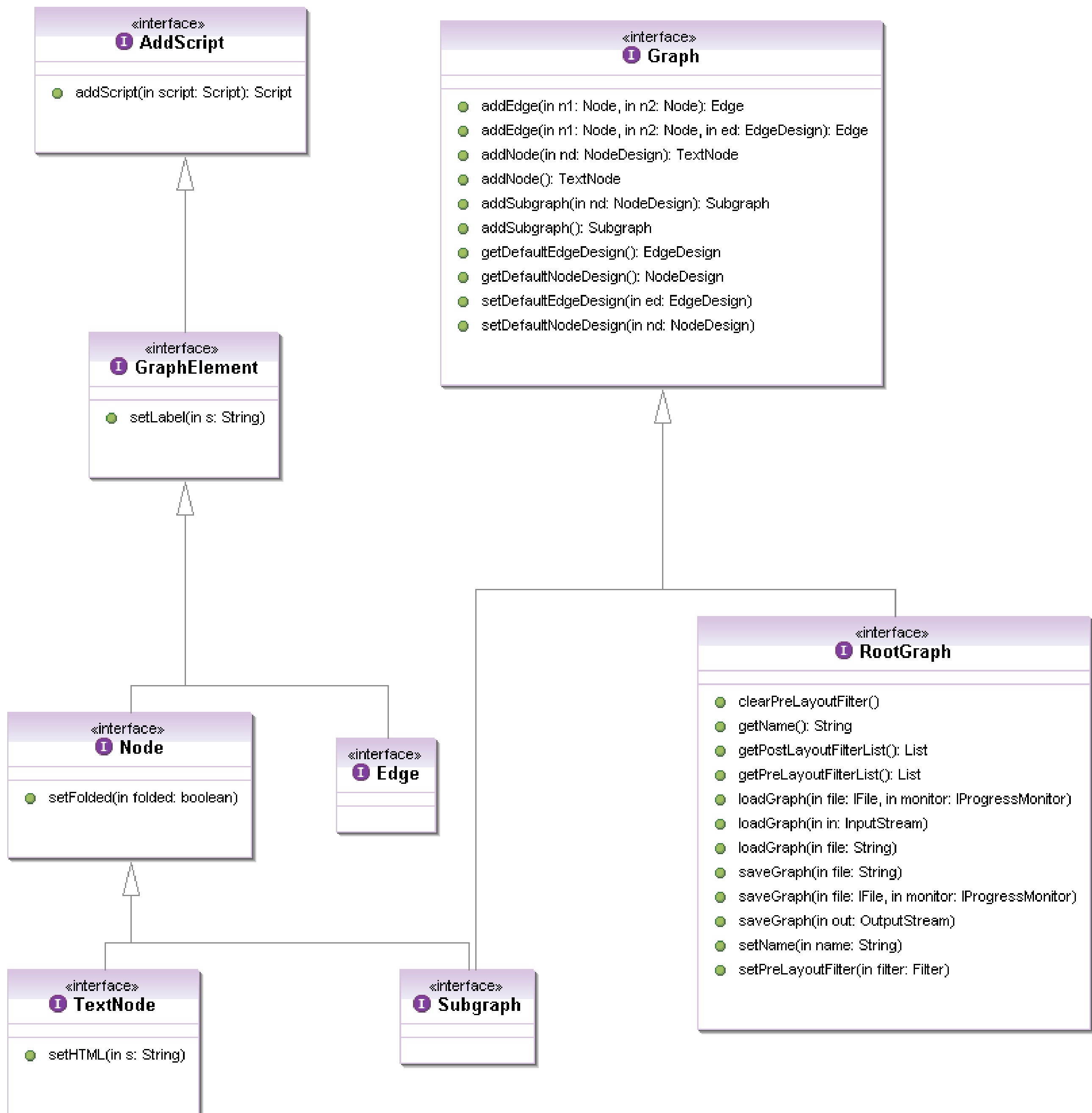


Abbildung 2.3: Die API Klassen 1

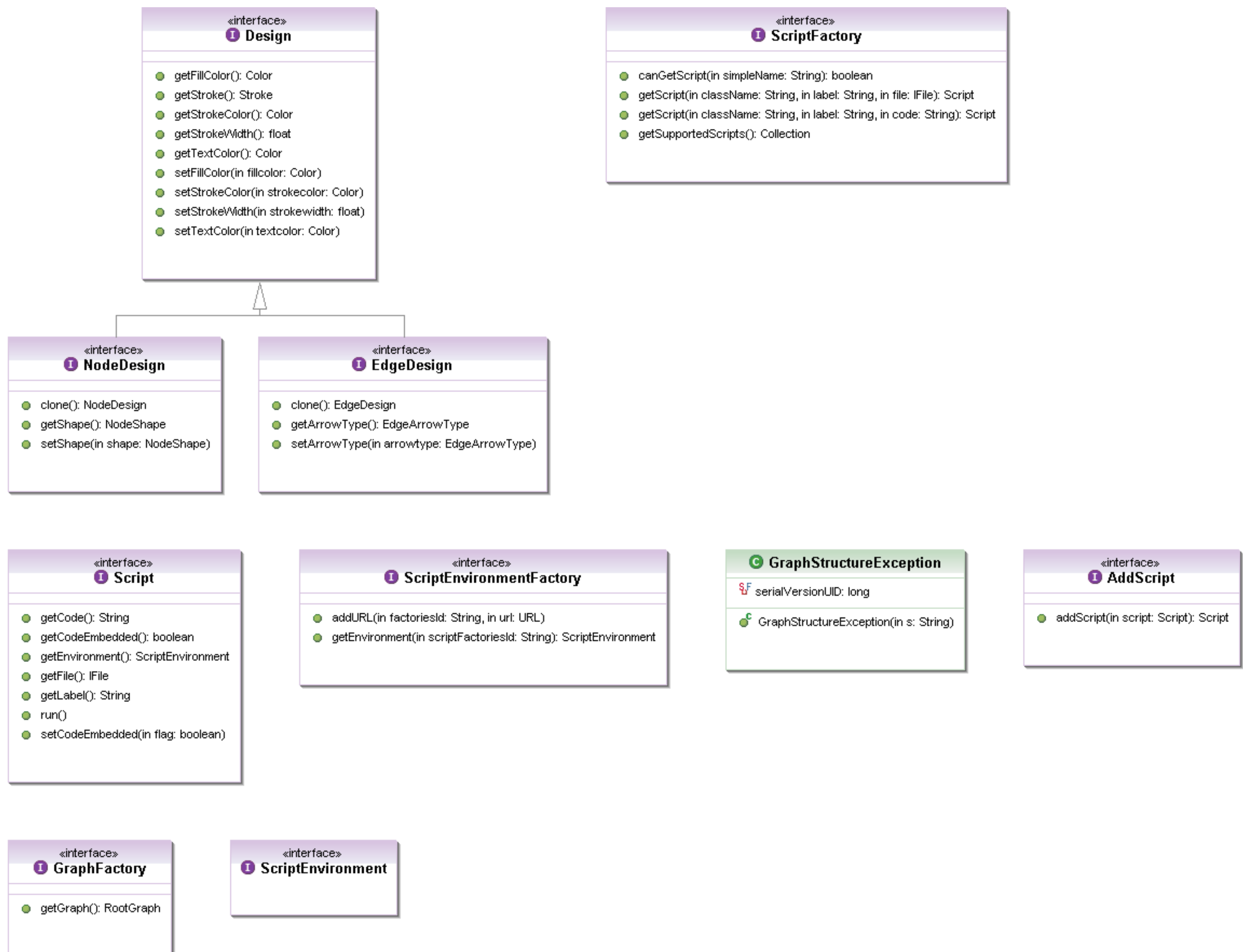


Abbildung 2.4: Die API Klassen 2

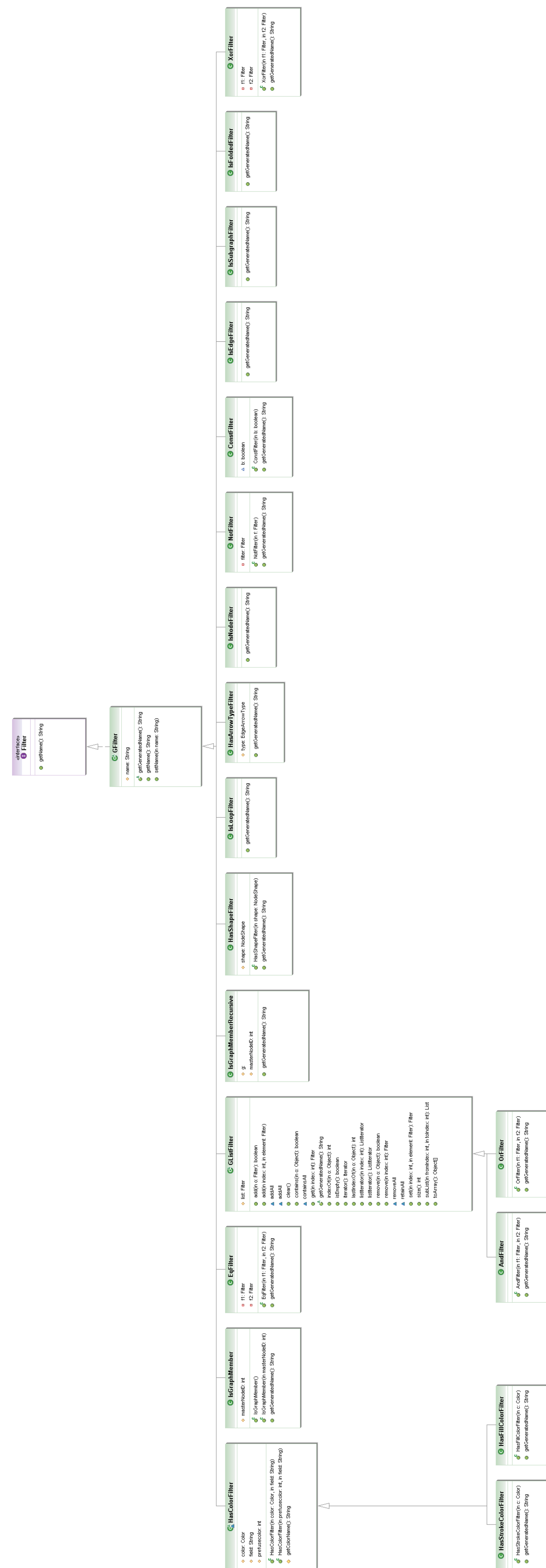


Abbildung 2.5: Die API Klassen 3 - Filter

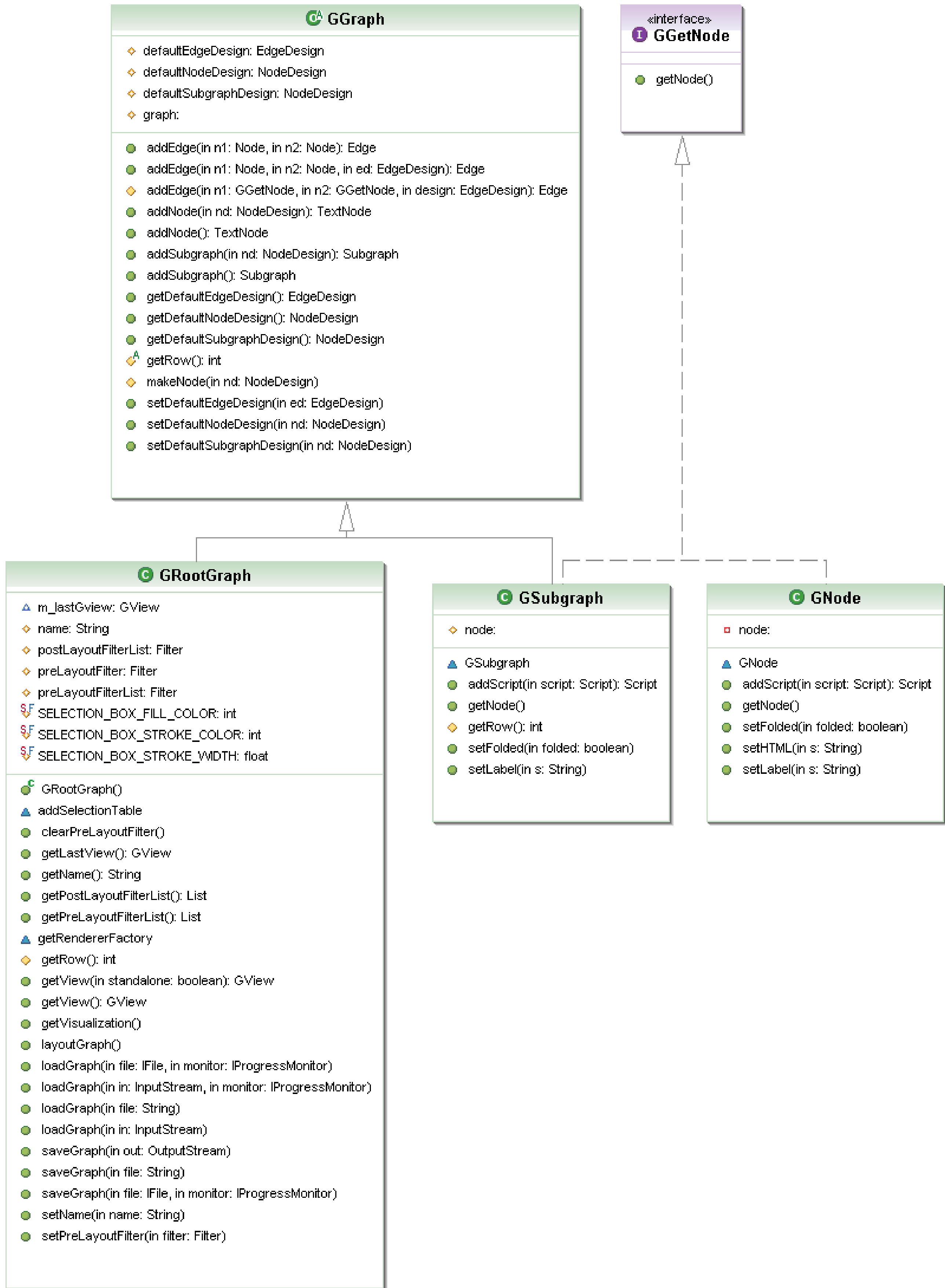


Abbildung 2.6: Die Implementierungen des API Interfaces

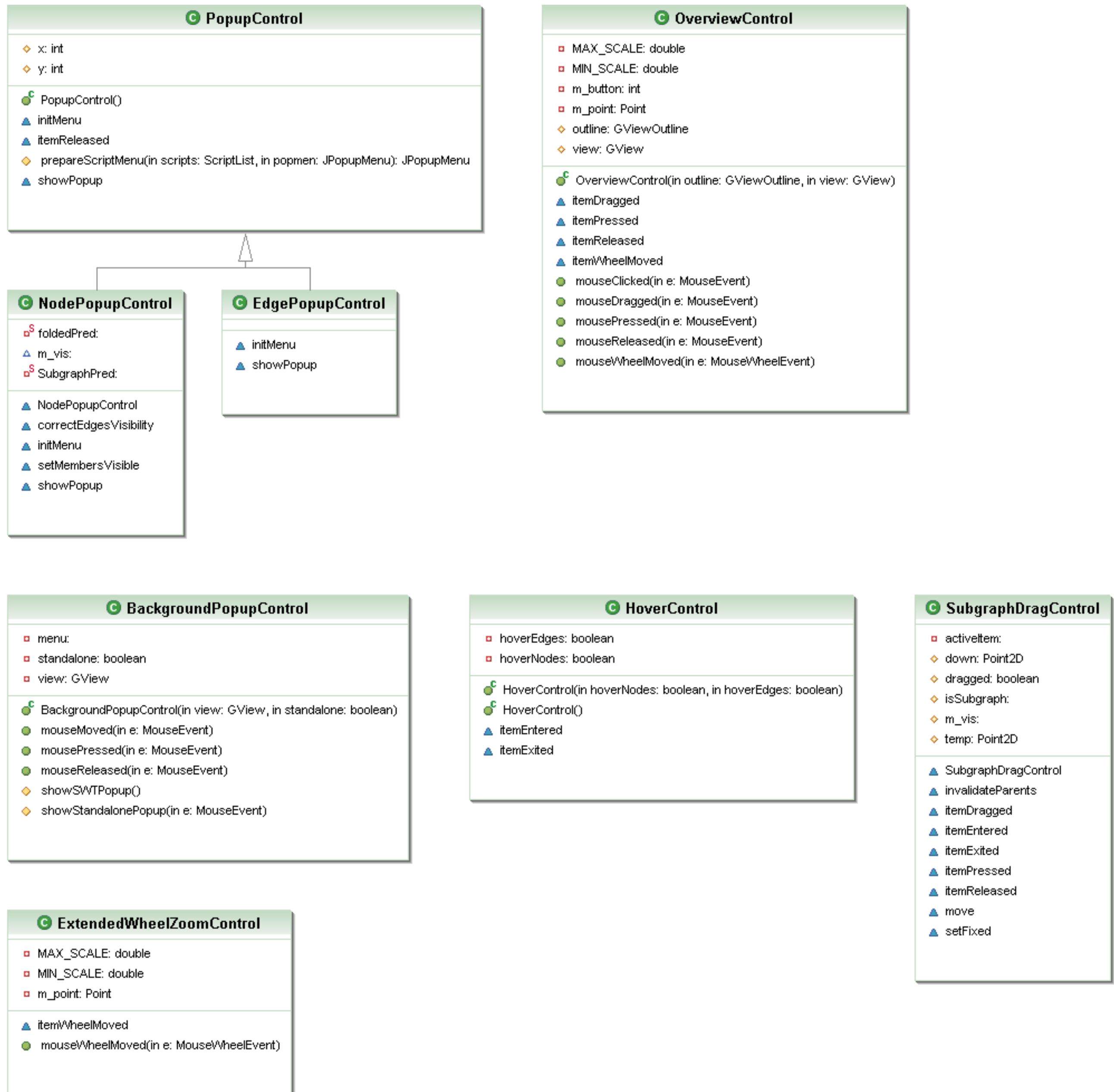


Abbildung 2.7: Die Controls um die Graph GUI zu bedienen

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 2.8 illustriert.
Dies betrifft die Use-Cases:

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

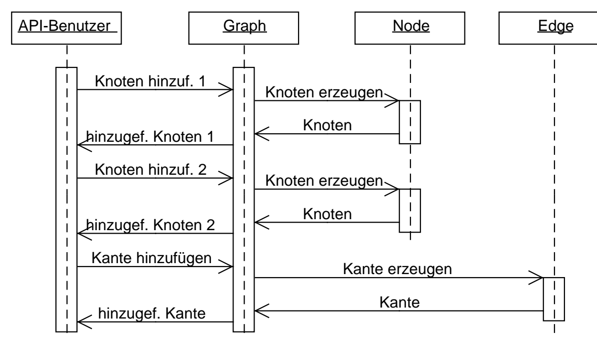


Abbildung 2.8: Erstellen und Hinzufügen eines Knotens und einer Kante

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 2.9 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

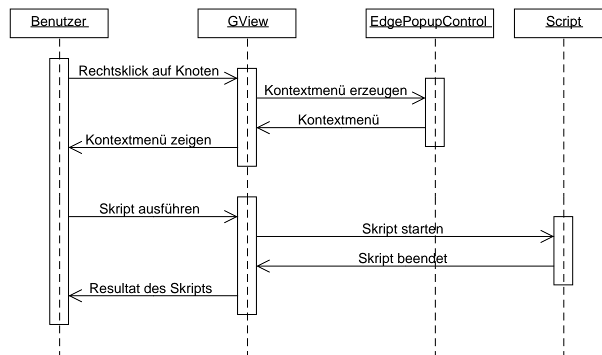


Abbildung 2.9: Ausführen eines Skriptes

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 2.10.
Dies betrifft die Use-Cases:

- Setzen eines Filters
- Rücksetzen eines Filters

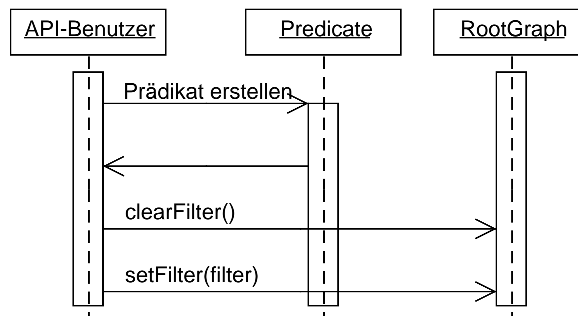


Abbildung 2.10: Anwenden eines Filters auf einen Graphen

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 2.11 auf der nächsten Seite.
Dies betrifft die Use-Cases:

- Speichern eines Graphen
- Laden eines Graphen

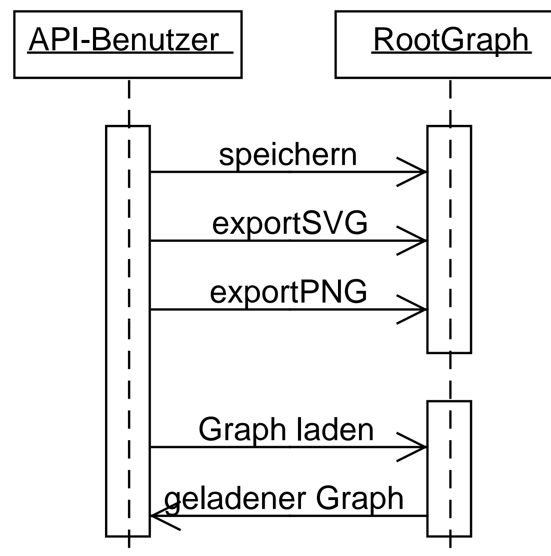


Abbildung 2.11: Speichern des Graphen

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 2.12 zusammen- oder ausgefaltet werden.

- Ausfalten eines Knoten
- Zusammenfalten eines Knoten

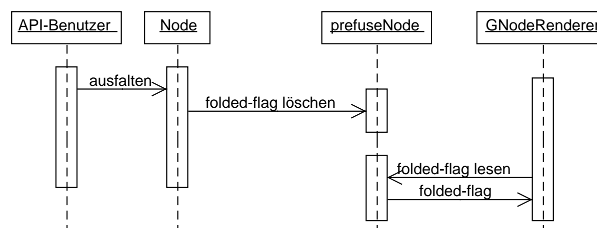


Abbildung 2.12: Ausfalten eines Knotens

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

2.4.3 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.

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();

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();

2.5 Klassenübersicht

2.5.1 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.

2.5.2 Wichtigsten Designklassen und ihre Responsibility

Die wichtigsten Klassen sind GGraph und Script. Zur genaueren Beschreibung s. 2.4.3.

2.6 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	*

Kapitel 3

Qualitätssicherung

3.1 Einleitung

Das Qualitätssicherungsdokument soll beschreiben, wie die Qualitätsmerkmale sichergestellt und im Sinne des Auftraggebers umgesetzt werden. Dabei werden die verwendeten Tests aufgelistet und einige zusätzlich detailliert beschrieben.
Außerdem gibt das Qualitätssicherungsdokument einen Einblick darüber, welche Qualitätsmerkmale berücksichtigt werden, und welche Priorität sie haben. Dabei wird die ISO/IEC 9126 - Norm als Modell für unsere Qualitätsmerkmale genutzt.

3.2 Qualitätsmerkmale

3.2.1 Funktionalität

Unter Funktionalität verstehen wir, inwiefern das Produkt die geforderten Funktionen unterstützt. Die Funktionen unterliegen diversen Anforderungen, z.B.

Richtigkeit: Hier geht es bei unserem Programm im Wesentlichen darum, dass die Graphen richtig abgespeichert werden können und nach dem Laden in ihrer logischen Struktur und in Form und Farbe der Komponenten wiederhergestellt wird.

Sicherheit: Hier sollte z.B. verhindert werden, dass beim Laden des Graphen oder beim Ausführen der Skripte Sicherheitslecks auftreten, die z.B. zum Einschleusen von Viren geeignet wären.

3.2.2 Zuverlässigkeit

Unter Zuverlässigkeit verstehen wir, ob, abhängig von vordefinierten Bedingungen, unser Produkt auch einen längeren Zeitraum über stabil laufen kann. Darunter fallen

Robustheit: Der Graph darf durch keinen API-Zugriff in einen inkonsistenten Zustand versetzt werden. Andererseits soll die Oberfläche auch bei wildem Klicken und ausführen von Skripten ohne Einschränkung oder Absturz bedienbar sein.

Fehlertoleranz: Die Fehlertolerzanz beschreibt die Fähigkeit, dass trotz Software-Fehlern oder Fehlern bei der Einhaltung der vorher definierten Schnittstellen ein bestimmtes Leistungsniveau gehalten wird. Fehlertoleranz ist bei unserem Programm vor allem beim Laden von Graphen zu beachten. Hierbei sollen zusätzliche, aber von uns nicht verwendete, GraphML-konforme Tags in der XML-Datei nicht zu einem Abbruch des Ladevorgangs führen.

3.2.3 Benutzbarkeit

Unter Benutzbarkeit verstehen wir, inwiefern sich der Benutzer in das Produkt einarbeiten muss, damit er es nach seinen Wünschen bedienen kann.

Bedienbarkeit: Bei unserem Programm geht es dabei um den Installationsaufwand, den Aufwand das Plugin als Entwickler zu benutzen und zudem die Oberfläche, die in Eclipse eingebettet sein soll.

Attraktivität: Das Plugin ist attraktiv, wenn es vollständig mit allen Abhängigkeiten von einer Update-Site installierbar ist und mit einem übersichlichen Interface alle vorhandenen Funktionen zu Verfügung stellt.

3.2.4 Effizienz

Die Effizienz beschreibt, wie das Verhältnis zwischen Ressourcen und Leistungsniveau für das Produkt ist. Eine hohe Effizienz ist gewährleistet, wenn das Produkt wenig Ressourcen verbraucht.

Zeitverhalten: Bei der Visualisierung von Graphen sind die zeitkritischen Punkte das Layout und das Rendern des Graphen. Hierbei wird, falls ein schnelles Terminieren nicht sichergestellt werden kann, Wert auf Abbrechbarkeit mittels Benutzerfunktion gelegt.

Verbrauchsverhalten: Zudem sollte die Benutzung nicht die Gesamtleistung des Systems zu stark beeinflussen.

3.2.5 Änderbarkeit

Die Änderbarkeit sagt aus, wieviel Aufwand betrieben werden muss, wenn man Änderungen am Produkt, seien es Korrekturen, Verbesserungen oder Erweiterungen, durchführen möchte.

Analysierbarkeit: Die Analysierbarkeit sagt aus, wie hoch der Aufwand ist, um Mängel und deren Ursachen am Programm zu finden.

Modifizierbarkeit: Die Modifizierbarkeit beschreibt, wie hoch der Aufwand ist, wenn man Änderungen zur Verbesserung des Programms durchführen will. Falls z.B. ein neuer Knotentyp oder eine andere Art von Interaktion gewünscht ist, lässt sich dies durch die Generizität in der genutzen Prefuse Komponente sehr leicht anpassen.

Prüfbarkeit: Die Prüfbarkeit ist in unserem Fall generell aufgrund der Plugin Struktur etwas erschwert. Um diese zu verbessern, wird auf der gleichen Codebasis auch eine standalone-Version des Programms zusammengestellt, da hier eine leichtere Prüfbarkeit besteht, da die standalone-Version wesentlich schneller geladen wird als Eclipse.

3.2.6 Übertragbarkeit

Unter Übertragbarkeit bzw. Portierbarkeit verstehen wir, inwiefern sich unser Produkt auf andere Systeme/Plattformen übertragen läßt.

Installierbarkeit: Der Aufwand, der zum Installieren des Produktes benötigt wird, abhängig vom Zielsystem. Hier wäre der Idealfall eine reine Eclipse-Plugin-Installation von einer Update-Site, dieser wird bei installierter Eclipse Version 3.2 und neuester Java-Version auch erreicht.

3.3 Maßnahmen zur Qualitätssicherung

3.3.1 RUP

Als grundlegendes Vorgehensmodell verwenden wir das (RUP)-Modell. Das heißt, wir verwenden eine iterative Entwicklung, die auf komponentenbasierter Architektur aufbaut. Das gewährleistet eine konstante Qualitätssicherung während der Entwicklung. Die einzelnen Disziplinen des RUP-Modells stellen die wichtigsten Stationen dar, welche das Projekt während der Entwicklung durchläuft. Beginnend bei der Analysephase bis zum fertigen Produkt kann man dank der Disziplinen und Iterationen bereits frühzeitig anfangen zu testen.

3.3.2 Pair-Programming

Der Code selbst wird zum Teil mit Hilfe des Pair-Programmings entwickelt. Dadurch können wir problematische Lösungen vermeiden und verbreiten das Wissen des Quellcodes unter uns, was die Qualität des Endproduktes verbessert.

3.3.3 Codeverwaltung und -dokumentation

An technischen Mitteln zur Qualitätssicherung verwenden wir SVN als Versionsverwaltungstool. Zudem wird bei einer Code-Änderung eine EMail an die Team-Mailingliste verschickt und der Code automatisch auf dem Server kompiliert. Die kompilierten Dokumente werden automatisch in den Downloadbereich und das kompilierte Plugin auf die Updatesite der Projekthomepage gestellt. So hat jeder immer die aktuelle, lauffähige Version. Es besteht hierdurch immer die Möglichkeit, mit einer realen Installation über die Updatesite den aktuellen Codestand zu testen.
Javadoc und Code Conventions dienen dazu, dass der Code auch für andere Personen als den Ersteller gut lesbar ist. Die Code Conventions sind z.B.:

- Verwendung von Eclipse Standard-Code-Style
- Methodennamen klein, jedes weitere Wort groß
- kurze und aussagekräftige Variablennamen
- Definitionen (von Variablen, Konstanten) immer am Anfang
- Variablen/Methodennamen auf englisch, Kommentare auf deutsch

3.3.4 Tests und Bugs

Unit Test

Während jeder Iteration unseres Programmes werden immer wieder Tests durchgeführt. Dazu verwenden wir das JUnit Test-Framework, welches uns Hinweise auf die Art des Fehlers liefert (falsches Ergebnis / auftreten eines Fehlers). Die graphische Oberfläche testen wir von Hand, da eine Automatisierung der Tests den Rahmen des Projekts sprengen würde.

Überdeckung

Damit wir auch sicherstellen können, dass die unsere Tests den gesamten Code Abdecken, messen wir dies mit dem Coverlipse Eclipse-Plugin. Dadurch können wir garantieren, dass nichts implementiert wird, was nicht fehlerfrei ist.

Bugtracking

Fehler, die während unseren Erprobungen auftreten, dokumentieren wir mit Mantis. Hiermit wird sichergestellt, dass ein Bug auch dann nicht in Vergessenheit gerät, wenn es nicht direkt eine Möglichkeit gibt, bzw. genügend Zeit zur Verfügung steht, ihn zu beheben.

3.3.5 Plattformdiversität

Die Anforderung der System- und Plattformunabhängigkeit wird bei uns immer implizit mitgetestet, da bei uns in der Projektgruppe eine sehr heterogene Hardware- und Systemumgebung anzutreffen ist.

3.3.6 Nutzen von Modulen

In unserem Projekt wird intensiv von bereits etablierten Komponenten Gebrauch gemacht. Z.B. verwenden wir als Grundgerüst der Graphen Prefuse und für die Skripte entsprechende Interpreter aus dem Umfeld des BSF. Dies stellt sicher, dass es in diesen Bereichen unseres Programms sehr wenige Fehler liegen können, da dieser Code schon in anderen Projekten vielfach genutzt wurde und somit einen gewissen Reifegrad aufweist.

3.3.7 Abbruchbedingung der Tests

Wir werden die Tests abbrechen, wenn wir keine Fehler mehr finden, die die Stabilität des Plugins gefährden und wir keine Zeit mehr haben, kleinere Probleme in der Oberfläche oder bei Features niedriger Priorität zu beheben. Da die Zeit sowieso für ausführliche Tests recht knapp bemessen ist, werden wir bis zuletzt auf Fehler prüfen.

3.4 Testplan

3.4.1 Use-Cases

Wir führen einen Blackbox-Test für die zwei Use Cases durch. Für jeden dieser Tests gibt es normale und alternative Abläufe. Diese Tests werden zum Teil durch JUnit Tests und zum anderen Teil mit manueller, aber vorher spezifizierter, Interaktion durchgeführt.

Laden eines Graphen (API)

Dieser Blackbox-Test soll den Use-Case "UC-API-Load" testen, bei dem ein Graph über die Methode Graph.load(...) geladen wird.

- *Normaler Ablauf:* Fehlerfreie Graph-Datei
  Es wird eine fehlerfreie Graph-Datei geladen. Die Graph-Objekte werden erstellt, wie sie in der Datei spezifiziert wurden, und können benutzt werden.
- *Alternativer Ablauf 1:* Die Graph-Datei existiert nicht
  Es wird versucht, eine nicht existierende Graph-Datei zu laden. Als Ergebnis muss eine FileNotFound-Exception geworfen werden.
- *Alternativer Ablauf 2:* Syntax-Fehler
  Die Graph-Datei enthält irgendwo einen Syntax-Fehler, ist leer, oder gar keine XML-Datei.
- *Alternativer Ablauf 3:* Semantik-Fehler
  Der Syntax der Graph-Datei ist fehlerfrei, aber sie enthält einen oder mehrere Semantik-Fehler.

  Semantikfehler können z.B. sein:

  - Eine Kante wird zwischen zwei nicht existierenden Knoten hinzugefügt.
  - Eine negative Linienbreite wird angegeben.
- *Alternativer Ablauf 4:* Nicht genügend Speicher
  Es ist nicht genügend Speicher vorhanden, um den Graphen einzuladen. Es muss eine OutOfMemory-Exception geworfen werden.

Zum Erproben dieser Blackboxtests wird eine Liste mit GraphML Testdateien erstellt, die genau die erwarteten Ergebnisse liefern. Die defekten Dateien werden zum Teil durch Manipulation aus korrekten Dateien erstellt, zum anderen Teil aus ausgewählten exemplarischen, nicht- GraphML Dateien.

Starten eines Skripts (GUI)

Dieser Blackbox-Test soll den Use-Case "UC-GUI-RunSkript" testen, bei dem ein Skript über die GUI gestartet wird. Hierzu muss mit dem rechten Mausknopf auf einen Knoten bzw. einer Kante geklickt und ein Skript auswählt werden.

- *Normaler Ablauf:* Skript wird fehlerfrei ausgeführt
  Das Skript wird fehlerfrei ausgeführt.
- *Alternativer Ablauf 1:* Skript-Datei existiert nicht
  Es wird versucht, ein Skript aus einer nicht vorhandenen Datei auszuführen. Als Ergebnis muss eine FileNotFoundException geworfen werden.
- *Alternativer Ablauf 2:* Skriptinterpreter wirft eine Exception
  Der Skriptinterpreter wirft eine Exception. Diese Exception muss aufgefangen werden und im Error-Log von Eclipse erscheinen.

  Bei der Exception kann es sich z.B. um einen Syntaxfehler, um einen Fehler bei der Ausführung eines Skriptbefehls oder einem Datei- Lesefehler handeln.

Zum Erproben werden einige Exemplarische Scripts an Knoten und Kanten gehängt, die sowohl aus Dateien als auch aus in den GraphML eingebetteten Code stammen.

3.4.2 Komponententest

Wir führen einen Komponententest für die Klassen GGraph und GScript mit Hilfe von JUnit durch.

3.4.3 Methodentest

Wir führen einen White Box Test für die zwei Methoden GNodeRenderer.render() und ExtGraphMLWriter.writeGraph() durch. Dabei müssen alle Code-Teile mindestens einmal ausgeführt worden sein (Zweigüberdeckung). Diese Überdeckung stellen wir durch manuelle Analyse des Codes und darauf abgestimmte Testfälle sicher.

GNodeRender.render()

Diese Funktion zeichnet einen Knoten mit Hilfe eines AWT-Graphics2D-Objektes. Um alle Zweige abzudecken, müssen folgende Features getestet werden:

- Alle NodeShapes
- Das Darüberbewegen mit der Maus
- Normaler Text und HTML-Text

ExtGraphMLWriter.writeGraph(Graph graph, OutputStream os)

Diese Funktion speichert einen Graphen in einer XML-Datei. Hierzu muss sichergestellt werden, dass alle speicherbaren Objekte vorkommen und in eine semantisch und syntaktisch korrekte XML-Datei geschrieben werden. Auch Fehler beim Schreiben auf das Medium müssen berücksichtigt werden.

3.4.4 Benutzbarkeit

Wir führen einen Benutzbarkeitstest sowohl für die GUI als auch für die API durch.

Benutzbarkeit der GUI

Wir lassen verschiedene Personen die GUI benutzen. Zu den Testpersonen sollen auch Personen gehören, die nicht aus unserer Gruppe sind. Hierzu sollen sowohl Programmierer als auch Enduser gehören, um ein breites Spektrum an Problemen finden zu können.

Benutzbarkeit der API

Wir testen die Benutzbarkeit der API, indem mehrere Leute verschiedene Testprogramme schreiben, die die API benutzen. Es werden auch Leute miteinbezogen, die nicht zu unserer Gruppe gehören. Wir bewerten, ob die Aufgaben, für die die API gedacht ist, leicht durchzuführen sind und wie verständlich ein Testprogramm von jeweils jemand anderem ist. Zudem werden wir eigene Demo-Programme schreiben, um die Benutzbarkeit exemplarisch vorzuführen. Diese Demos werden einen Teil der Dokumentation des Plugins ausmachen.

3.5 Prüfung der Zeitplanung

Insgesamt kommen wir auf eine gesamte Zahl geleisteter Stunden von 714. Wenn man dies der Planung von 1000 Stunden gegenüberstellt und die eine noch verbleibende Woche berücksichtigt, kommt man zu dem Ergebnis, dass wir etwa 80% der geplanten Zeit benötigt haben. Die Gesamtzahl der Stunden verteilt sich wie in Diagramm 3.1 gezeigt auf die verschiedenen Gebiete. Die Gesamtlast war nicht absolut gleichverteilt auf

alle Gruppenmitglieder, wie man in Diagramm 3.2 sieht. Hierbei muss man allerdings beachten, dass Klausuren während der Praktikumszeit lagen, Andreas nach etwa der Hälfte ausgestiegen und Bastian erst kurz nach der Hälfte hinzugekommen ist. Die effektive Aufgabenverteilung sah so aus: Marco hielt die Reviewvorträge und übernahm während dieser Zeit auch die Organisation. Zudem betreute er den Server war bei der Programmierung im im Wesentlichen für den Layout und das Laden/Speichern in gml-Dateien zuständig. Andreas war an der Erstellung der Dokumentation beteiligt, konnte auf Grund seiner Ausstiegs nicht mehr an der Implementierung mitwirken. Bastian übernahm gegen Ende einige Aufgaben bei der Dokumentation. Martin war zunächst für die Einarbeitung in die Eclipse-Plugin-Entwicklung zuständig, übernahm später aber stattdessen die vollständige Verantwortung für die Renderer, also die graphische Gestaltung der Knoten und Kanten und half hin- und wieder bei der Dokumentation aus. Peter war zunächst für den Aufbau der Dokumentinfratruktur zuständig und übernahm bei der Implementation die Plugin-Einbindung, also den Eclipse-spezifischen Teil der Programmierung. Gegen Ende übernahm er die Organisation und die Testdokumentation der wechselseitigen Tests mit der Gruppe SDG.

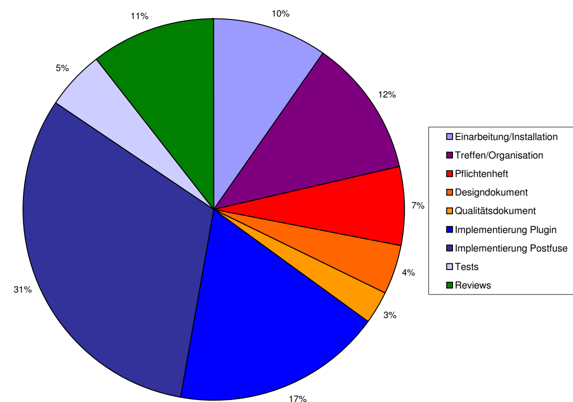


Abbildung 3.1: Verteilung der Stunden auf die verschiedenen Aufgabenbereiche. Die Stunden für die Programmierung sind nochmal aufgegliedert in den Teil, der direkt die Eclipse-Integration betrifft, und die Erweiterungen von prefuse.

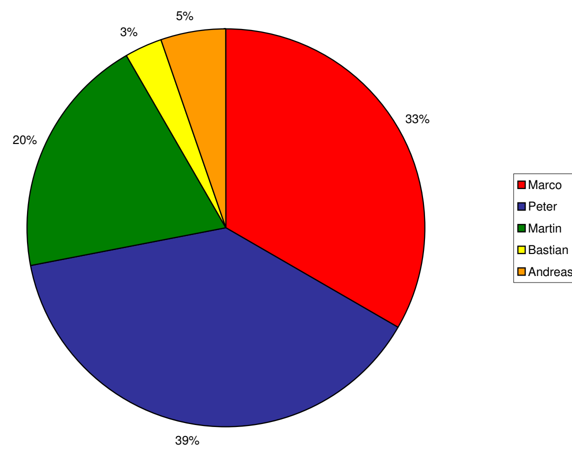


Abbildung 3.2: Verteilung der Stunden auf die Gruppenmitglieder

3.6 Testergebnisse

Im folgenden Kapitel werden wir unsere Testergebnisse ablegen. Hierzu wird wir für jeden Test eine Tabelle mit folgendem Format ausgefüllt:

Name	
ID	-
Datum - Zeit	
Status	
Priorität	
Akteur	
Fehlerbeschreibung	
Korrekturaktion	

Name: Der Name des angewendeten Testsplans. Diese Tests sind im vorherigen Kapitel beschrieben.

ID: Eine einzigartige ID des jeweiligen Testdurchlaufs.

Datum - Zeit: Datum und Zeit, an dem der Test durchgeführt wurde.

Status: Ausgang der Messung. Mögliche Werte sind:

- **passed** Alle Ergebnisse entsprechen der Spezifikation. Der Test ist bestanden.
- **failed** Es sind (erhebliche) Abweichungen vom geplanten Verhalten. Der Test ist nicht bestanden.
- **1-6** Diese Werte (Schulnoten) können nur bei subjektiven Tests wie Benutzerakzeptanztests vergeben werden.

Priorität: Wie wichtig ist das Bestehen dieses Tests für das Endprodukt (Arten: Hoch, Mittel, Niedrig)

Akteur: Name der Person, die getestet hat.

Fehlerbeschreibung: Eine kurze Beschreibung des aufgetretenden Fehlers (falls vorhanden) bzw. eine Beschreibung des subjektiven Eindrucks bei Benutzerakzeptanztests.

Korrekturaktion: Im Falle eines Fehlers die Aktion zur Behebung des Problems.

Im Folgenden werden die einzelnen Testdurchläufe nach diesem Schema beschrieben. Bislang wurden bis auf Ausprobieren während der Implementierung noch keine systematischen Tests durchgeführt. Die Folgenden Überschriften geben die geplante Gliederung wieder, die sich an den im vorherigen Abschnitt aufgeführten Testplänen orientiert.

Erlaubte Kanten	
ID	TEST-EDGE
Datum - Zeit	27.3.2007
Status	failed
Priorität	Hoch
Akteur	API-Benutzer
Fehlerbeschreibung	Es wird getestet, ob Kanten von Subgraphen zu enthaltenen Knoten zu einer GraphStructureException führen.
Korrekturaktion	Beim Hinzufügen einer Kanten zwischen zwei verschachtelten Subgraphen wurde keine Exception geworfen.

Erlaubte Kanten - Wiederholung	
ID	TEST-EDGE-2
Datum - Zeit	4.4.2007
Status	passed
Priorität	Hoch
Akteur	API-Benutzer
Fehlerbeschreibung	Es wird getestet, ob Kanten von Subgraphen zu enthaltenen Knoten zu einer GraphStructureException führen.
Korrekturaktion	

ZoomToFit beim Start	
ID	TEST-ZOOMTOFIT
Datum - Zeit	28.3.2007
Status	failed
Priorität	Hoch
Akteur	GUI-Benutzer
Fehlerbeschreibung	Es wird getestet, ob der Graph beim Öffnen des Fensters sinnvoll dargestellt wird.
Korrekturaktion	Layouter synchronisiert, ZoomToFit wird nach dem ersten Paint aufrufen, Deadlockmöglichkeiten bei verschieden Actions verhindert.

ZoomToFit beim Start (Wiederholung)	
ID	TEST-ZOOMTOFIT-2
Datum - Zeit	03.04.2007
Status	passed
Priorität	Hoch
Akteur	GUI-Benutzer
Fehlerbeschreibung	Es wird getestet, ob der Graph beim Öffnen des Fensters sinnvoll dargestellt wird.
Korrekturaktion	

Neuzeichnen der Schleifen beim Bewegen eines Knoten	
ID	TEST-LOOP
Datum - Zeit	21.03.2007
Status	failed
Priorität	Hoch
Akteur	GUI-Benutzer
Fehlerbeschreibung	Graphikfehler beim Knotenverschieben bei Schleifen
Korrekturaktion	Es wird ein Neuzeichnen der Schleifen benachbarten Knoten des gezogenen Knoten veranlasst.

Neuzeichnen der Schleifen beim Bewegen eines Knoten (Wdh)	
ID	TEST-LOOP-2
Datum - Zeit	02.04.2007
Status	passed
Priorität	Hoch
Akteur	GUI-Benutzer
Fehlerbeschreibung	Graphikfehler beim Knotenverschieben bei Schleifen
Korrekturaktion	

3.6.1 Anwendungsfalltests

Diese Tests wurden wechselseitig mit der Gruppe SDG durchgeführt und weichen deshalb leicht von der Planung ab.

Graph aus einer GML-Datei laden	
UC-ID	UC-GUI-Load
Akteur	Tester der anderen Gruppe
Kurzbeschreibung	Laden eines Graphen aus einer bereits vorhandenen GML-Datei
Vorbedingung	G222 ist installiert und es liegt eine GML-Datei vor.
Nachbedingung	Der gewünschte Graph wird angezeigt.
Aktion	1 Einbinden des Projekts, welches die GML-Dateien enthält 3 Auswählen einer gml-Datei innerhalb dieses Projekts 4 mittels 'open' oder Doppelklick geöffnet
Reaktion	2 GML-Dateien stehen für die Anzeige zur Verfügung 5 neues Fenster wird geöffnet, in dem der Graph angezeigt wird.
Bewertung:	
Reife	100%
Fehlertoleranz	100%
Erlernbarkeit	100%
Bedienbarkeit	90%
Kommentar	Zunächst dachte ich, man muss zwangsläufig ein Projekt haben, indem die Dateien eingebunden sind. Ich habe dann jedoch nach dem alternativen Testablauf festgestellt, dass das nicht notwendig ist, wenn man die Datei direkt öffnet.

Graph mittels der API erzeugen	
UC-ID	Viele
Akteur	Tester der anderen Gruppe
Kurzbeschreibung	Es wird die Verwendbarkeit des Plugins in neuen Plugins getestet.
Vorbedingung	G222 ist installiert und es liegt eine GML-Datei vor.
Nachbedingung	Der gewünschte Graph wird angezeigt.
Aktion	1 Überschreiben einer schon vorhandenen Action-Datei 2 Erzeugen eines neuen Graphen 3 Hinzufügen von Kanten, Knoten Skripten und Formatierungen 4 Anzeige des Graphs, über 'G222Examples'
Reaktion	5 erzeugter Graph wird angezeigt.
Bewertung:	
Reife	70%
Fehlertoleranz	60%
Erlernbarkeit	60-70%
Bedienbarkeit	60%
Kommentar	Nach dem Erzeugen einer neuen Test-Klasse, die den gleichen Aufbau hatte, wie die anderen Beispielklassen, wurde diese Testklasse bei G222Examples leider nicht aufgeführt. Woran das lag, konnte nicht festgestellt werden, weswegen der neu erzeugte Graph auch nicht angezeigt werden konnte. Nur durch das Überschreiben der run-Methode einer schon vorhandenen Klasse mit dem neuen Code, konnte der Graph dann angezeigt werden. Das Erzeugen des Graphen selbst verlief aufgrund der gegebenen Beispiele ohne Probleme. Jedoch würde eine Dokumentation der API-Methoden dem Benutzer das Erstellen von Graphen sehr erleichtern. Anmerkung der Gruppe G222: Die Verwendung der Actionsets ist optional, man kann die Anzeige des Graphs auch über eine selbstdefinierte View oder irgendetwas anderes bewerkstelligen. Die neue Action wurde nicht angezeigt, da man sich erst in den entsprechenden Extension Point der Eclipse Workbench einklinken muss. Diese sollte aber einem Eclipse-Plugin-Entwickler bekannt sein.

Grafik-Export	
UC-ID	UC-GUI-Export-SVG und UC-API-Export-PNG
Akteur	Tester der anderen Gruppe
Kurzbeschreibung	Test der Export-Funktionen in SVG und PNG.
Vorbedingung	Ein Graph wird angezeigt, Testprojekt wurde erzeugt.
Nachbedingung	Graph wurde als Grafik exportiert.
Aktion	1 'Export to SVG' geöffnet 3 Speicherort ausgewählt (Projekt) 5 Name für die Datei angegeben 7 Bestätigen 8 'Export to PNG' geöffnet 10 Speicherort ausgewählt (Projekt) 12 Name für die Datei angegeben 14 Bestätigen
Reaktion	2 Exportdialog geöffnet 4 Speicherort & Name ausgewählt 6 SVG-Datei wird erzeugt 9 Exportdialog geöffnet 11 Speicherort & Name ausgewählt 13 PNG-Datei wird erzeugt
Bewertung:	
Reife	100%
Fehlertoleranz	100%
Erlernbarkeit	90%
Bedienbarkeit	90%
Kommentar	Das Speichern des Graphen als Grafik verlief ohne Probleme. Jedoch wäre es von Vorteil, wenn nicht für die Speicherung ein extra Projekt angelegt werden müsste, sondern wenn die Dateien auch unabhängig davon erzeugt werden könnten.

Starten eines Skriptes	
UC-ID	Viele
Akteur	Tester der eigenen Gruppe
Kurzbeschreibung	Es wird ein Skript aufgerufen.
Vorbedingung	Es wird ein Graph angezeigt mit mindestens einem an einen Knoten oder eine Kante angehängten Skript.
Nachbedingung	Das Skript wurde ausgeführt.
Aktion	1 Rechtklick auf Knoten/Kanten mit Skript 3 Auswahl der Skripts im Menü
Reaktion	2 Popup-Menü öffnet sich. 4 Skript wird ausgeführt und Ausgaben des Skripts werden in einer Konsole angezeigt.
Bewertung:	
Reife	90%
Fehlertoleranz	80%
Erlernbarkeit	100%
Bedienbarkeit	90%
Kommentar	Interaktionen mit Eclipse, z.B. das öffnen einer Datei im Editor, sind möglich.

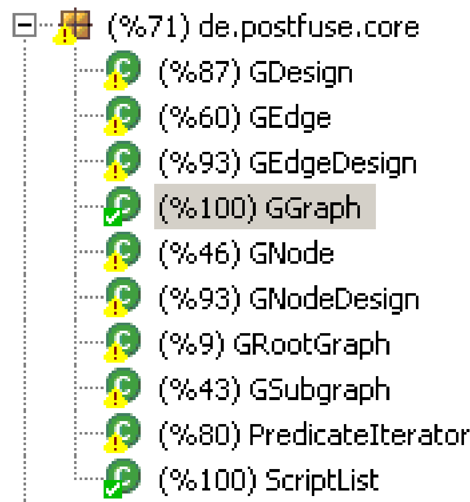


Abbildung 3.3: Überdeckung von GGraph

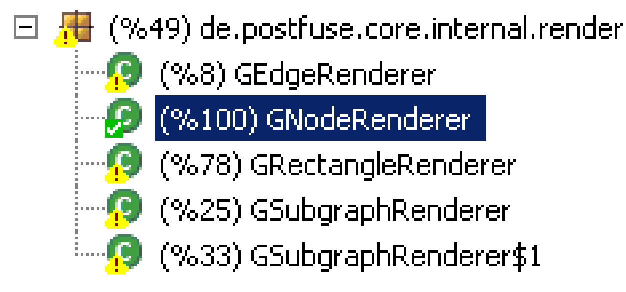


Abbildung 3.4: Überdeckung des NodeRenderers

3.6.2 Komponententest

GGraph

Hier haben wir einen Überdeckungstest mit Coverlipse Eclipse-Plugin von GGraph durchgeführt. Da diese Klasse schon während der gesamten Entwurfszeit immer implizit mitgetestet wurde, konnten keine Fehler entdeckt werden.

GScript

3.6.3 Methodentest

GNodeRender.render()

Bei diesen Tests erzeugten wir durch Zusammenstellung eines Graphen mit besonders dekorierten Knoten eine Überdeckung und prüften dann in der Ansicht die richtige Darstellung, welche durch automatische Tests nicht praktikabel getestet werden kann. Es wurden keine schwerwiegenden neuen Fehler gefunden, nur Probleme, falls das Label null gesetzt wird. Diese waren schnell zu beseitigen.

ExtGraphMLWriter.writeGraph()

Wie man in Abbildung 3.5 erkennen kann, wurde eine 100-prozentige Überdeckung der Methode erreicht. Wir haben auch bei der Überdeckung der Untermethoden über 80% Überdeckung. Durch den Test in einem vollen Speichern-Laden-Zyklus konnte die Konsistenz der beiden Methoden hervorragend überprüft werden.

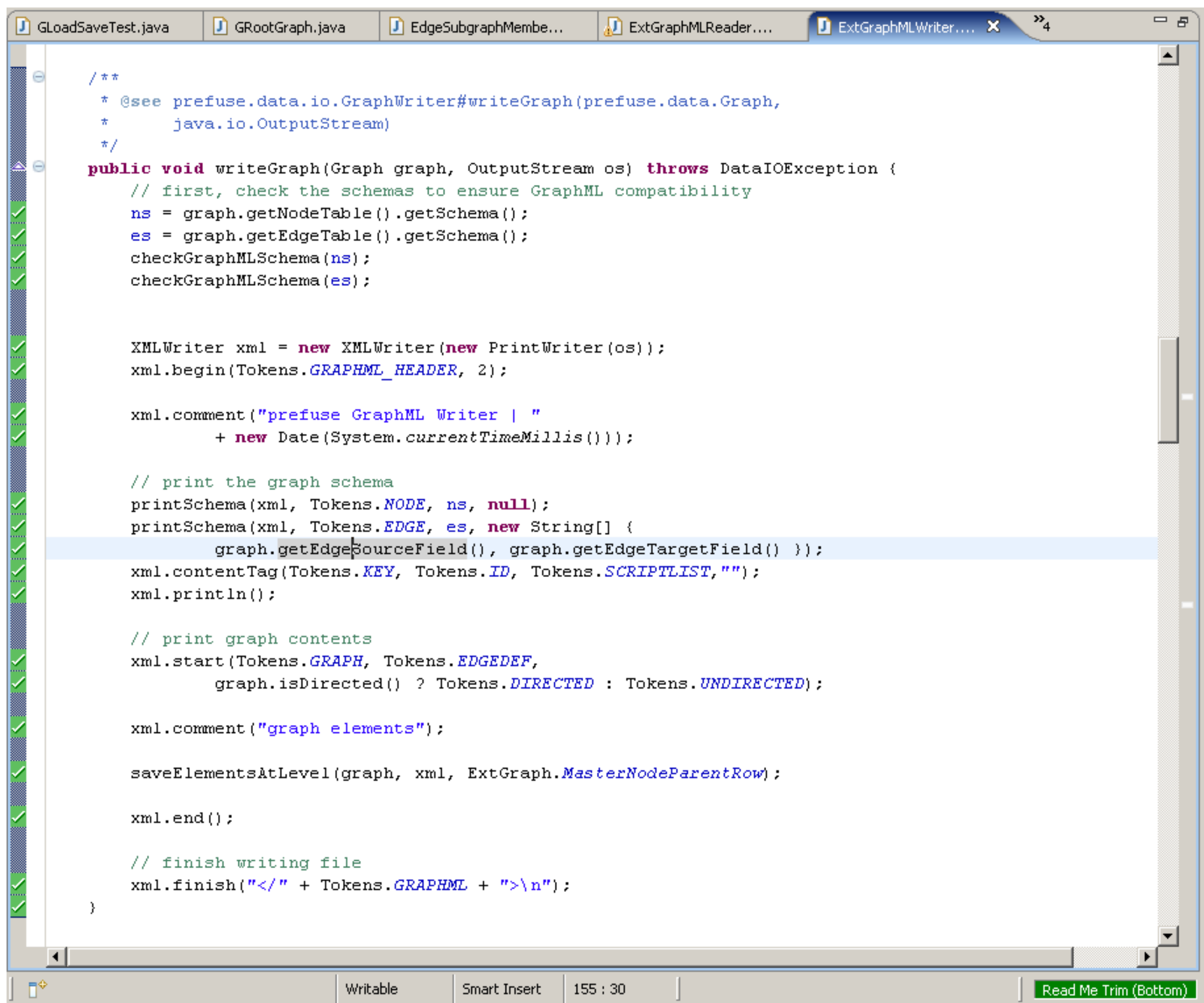


Abbildung 3.5: Überdeckung der writeGraph-Methode

3.6.4 Beispiele

Wir haben verschiedene Beispiele zur Benutzung der API zur Erstellung von Graphen erstellt. Vor allem die dynamischen Beispiele, die zufällige Graphen mit Subgraphen und zufälligen Designkomponenten erzeugen waren beim Test des Layouters sehr erfolgreich.

3.6.5 Benutzbarkeit

Benutzbarkeit der GUI

Die Benutzbarkeit der GUI wurde durch unsere andauernden Tests der Eclipse-Integration fortwährend mitgetestet und alles, was die anderen Tests auf Grund von Umständlichkeiten behinderte, wurde behoben. Weitere Tests waren für die GUI nicht nötig, da sowohl die Bewertung der anderen Gruppe sehr gut ausfiel, als auch durch Inspektion des Eclipse-Source-Codes altbewährte Muster der Eclipse-Oberfläche benutzt wurden.

Benutzbarkeit der API

Die Benutzbarkeit der API stellen wir durch die große Anzahl an Beispiele sicher (bisher über 15). Beim Design der Beispiele wurden dieselben Schnittstellen, die der Benutzer später sieht, verwendet. Um die Benutzbarkeit weiter zu steigern werden wir noch den kommentierten Quellcode der API dem Plugin beilegen, so dass ein Entwickler in Eclipse auch die erweiterten Content-Assist-Möglichkeiten von Eclipse nutzen kann.

3.6.6 Zusammenfassung

Zusammenfassend lässt sich sagen, dass durch häufige kleinere Tests während der Entwicklungszeit böse Überraschungen bei den abschließenden Test ausblieben. Auch die Wiederherstellungsfähigkeiten unseres Versinsverwaltungssystems haben uns einmal vor der Beinahe-Katastrophe bewahrt. Insgesamt waren die Tests ’von Hand’ am erfolgreichsten und konnten die meisten Fehler aufspüren. Nur bei der komplizierten Layout-Berechnung war ein JUnit-Test unverzichtbar. Vor allem verschiedene Thread- und Eclipseprobleme wären nicht durch automatische Tests zu finden gewesen.

3.7 Änderungshistorie

Datum	Thema	Inhalt	Seite
12.12.2006	alles	Beginn der History	77
26.01.2007	alles	Ausgabe Iteration 0	*
15.02.2007	Blackbox-Test	detaillierter beschrieben	*
16.02.2007	Testergebnisse	Einleitung und Format	*
05.04.2007	Testergebnisse	die Tests der anderen Gruppe eingefügt	70
06.04.2007	Testergebnisse	Ergebnisse der Überdeckungstests	74
08.04.2007	alles	finales Release 2	*

Kapitel 4

Anhang

Glossar

ASCII

Abkürzung für American Standard Code for Information Interchange. Ein weit verbeitete Zeichenkodierung, auf die viele andere aufbauen. `de.wikipedia.org/wiki/Ascii` *7*

BSF

Das Bean Scripting Framework (BSF) ist eine standardisierte Schnittstelle für Interpreter von Skriptsprachen, die eine Interaktion mit Java erlauben. `http://jakarta.apache.org/bsf/` *35*, *63*

Coverlipse Eclipse-Plugin

Coverlipse ist eine Erweiterung des JUnit Test-Framework. Es zeigt zu den Tests jeweils an, welche Codezeilen beim Test durchlaufen wurden und kann somit Hinweise auf ungetesteten Code geben. *34*, *62*, *74*

Eclipse

Eclipse ist eine offene Entwicklungsplatform, die auf Java-Technologie beruht. Es wird sowohl als IDE, als auch für die Entwicklung neuer Programme, verwendet. Mit Plugins lässt es sich beliebig erweitern. *34*, *62*

Eclipse Update-Site

Eclipse bietet die Möglichkeit Erweiterungen aus dem Internet nachzuinstallieren. Die unterstützten Update-Sites sind spezielle Webseiten, die dies erlauben. *6*

Eclipse-Plugin

Eine Programm oder Programmpaket, dass sich in der Eclipse-Oberfläche einklinkt und den Funktionsumfang erweitert [13]. *6*

GraphML

Ein XML-Format zum Abspeichern von Graphen. Den Knoten und Kanten können beliebige eigene Attribute zugewiesen werden. *41*

GUI

Eine GUI (Graphical User Interface) erlaubt dem Benutzer eines Programmes, mit diesem zu interagieren. Dies geschieht mit Hilfe von grafischen Elementen wie Fenster, Knöpfe und Listen. *6*

Java

Java ist eine moderne weit verbreitete Programmiersprache.
`de.wikipedia.org/wiki/Java_(Programmiersprache)` *6*

Java-API

Eine (Java)-API, kurz für Aplication Programming Interface, ist eine Schnitstelle mit der ein Programm auf die Funktionen eines anderen zugreifen kann.
`de.wikipedia.org/wiki/Programmierschnittstelle` *6*

JUnit Test-Framework

JUnit ist ein Framework zum Testen von Java-Programmen, das besonders für automatisierte Unit-Tests einzelner Units geeignet ist *34*, *62*

Mantis

Mantis ist ein quellenoffenes, auf MySQL und PHP basierendes Bug-Trackingsystem. Es gibt uns die Möglichkeit, einen Überblick über alle noch zu behebenden Probleme und alle Featurewünsche zu bekommen. *34*, *62*

Pair-Programming

Paarprogrammierung bedeutet, dass bei der Erstellung des Quellcodes jeweils zwei Programmierer an einem Rechner arbeiten. Dabei ist einer für die Erstellung des Codes zuständig, wahrend der andere ständig kontrolliert. *34*, *62*

Pan & Zoom

Verschieben, verkleinern und vergößern der aktuellen Bildansicht. *6*

PNG

Portable Network Graphics ist ein im Internet häufig verwendetes verlustbehaftetes Dateiformat für Grafiken. `de.wikipedia.org/wiki/Portable_Network_Graphics` *15*

Prefuse

Ein Java-Framework zum Erstellen und Visualisieren von Graphen, Tabellen und Bäumen. Die Datenfelder lassen sich beliebig mit eigenen Daten erweitern. *9*, *35*, *41*, *61*, *63*

RUP

RUP (Rational Unified Process) ist ein Vorgehensmodell für die Entwicklung objektorientierter Programme. Es verwendet UML als Notationssprache. *34*, *62*

SVG

Ein auf XML basiertendes 2D-Vektorgrafikformat. Es unterstützt sowohl die üblichen grafischen Primitive, als auch Rastergrafiken, Text und Animationen. *15*, *41*

SVN

SVN (Subversion) ist eine Open-Source-Software zur Versionsverwaltung. Es kontrolliert den gemeinsamen Zugriff von mehreren Entwicklern auf die Dateien eines Projektes. *34*, *42*, *62*

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. *42*

UTF-8

Eine Zeichenkodierung, unter Java der Standard, die eine wesentlich höhere Anzahl von Zeichen abdeckt, als ASCII. `de.wikipedia.org/wiki/Utf8` *7*

XML

eXtensible Markup Language. Eine Gerüsst um Datenformate einheitlich zu spezifizieren. XML beruht auf der Syntax von HTML-Tags
`de.wikipedia.org/wiki/Extensible_Markup_Language` *6*

Literaturverzeichnis

[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.