Bachelor - Praktikum (Graphenlayout)

Qualitätdokumentation Rev. 2

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

postfuse

one step beyond

Stand: 8. April 2007

`http://bp.macrolab.de`

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

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

Inhaltsverzeichnis

1 Einleitung

Das 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.

2 Qualitätsmerkmale

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.

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.

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.

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.

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.

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 Maßnahmen zur Qualitätssicherung

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.

3.2 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 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

3.4 Tests und Bugs

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.5 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.

3.6 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.

3.7 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.

4 Testplan

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.

4.1.1 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.

4.1.2 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.

4.2 Komponententest

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

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.

4.3.1 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

4.3.2 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.

4.4 Benutzbarkeit

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

4.4.1 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.

4.4.2 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.

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 1 gezeigt auf die verschiedenen Gebiete. Die Gesamtlast war nicht absolut gleichverteilt auf alle Gruppenmitglieder, wie man in Diagramm 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.

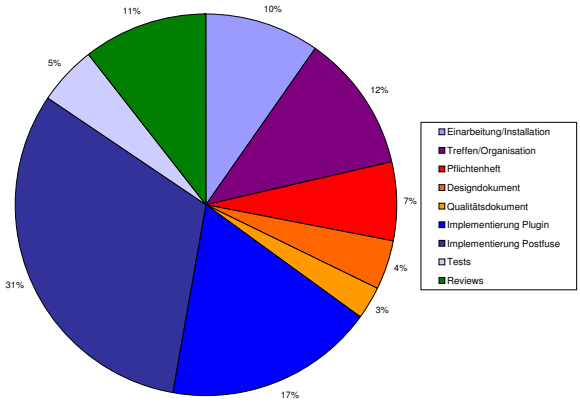


Abbildung 1: Verteilung der Stunden auf die verschiedenen Aufgabenbereiche. Die Stunden für die Programmierung sind nochmal aufgegliedert in den Teil, der direkt die Eclipse-Integration betrifft, und die Erweiterungen von prefuse.

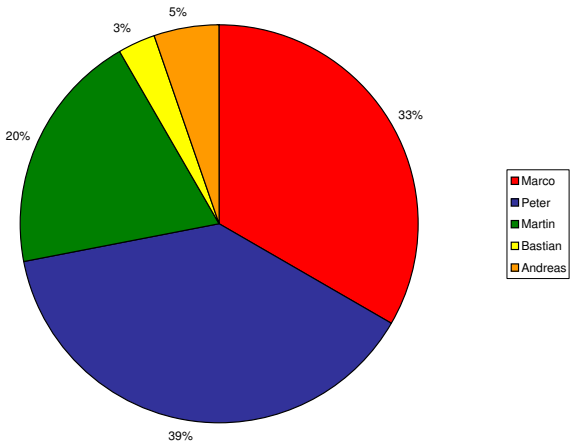


Abbildung 2: Verteilung der Stunden auf die Gruppenmitglieder

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	

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.

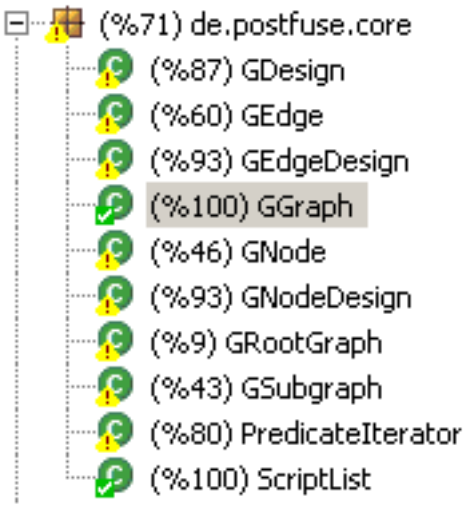


Abbildung 3: Überdeckung von GGraph

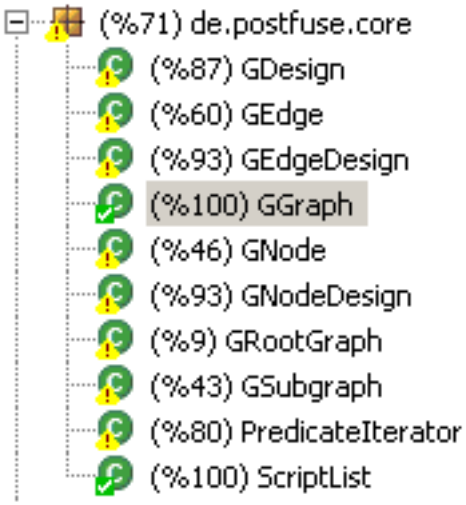


Abbildung 4: Überdeckung des NodeRenderers

6.2 Komponententest

6.2.1 GGraph

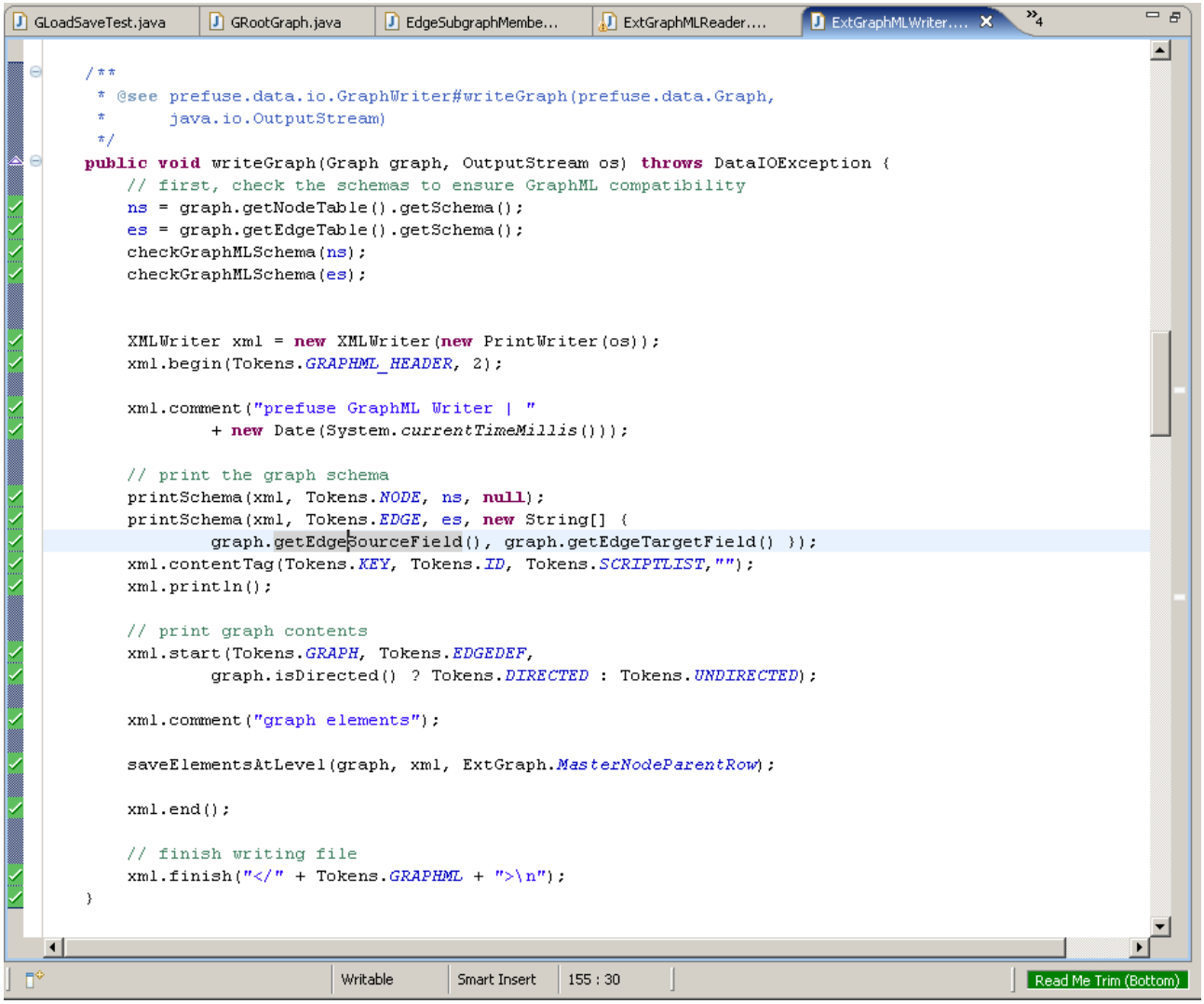
Hier haben wir einen Überdeckungstest mit Coverlipse Eclipse-Plugin von GGraph durchgeführt. Da diese Klasse schon während der gesamten Entwurfszeit immer implizit mitgetestet wurde, konnten keine Fehler entdeckt werden.

6.2.2 GScript

6.3 Methodentest

6.3.1 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.




```
    /**
     * @see prefuse.data.io.GraphWriter#writeGraph(prefuse.data.Graph,
     *      java.io.OutputStream)
     */
    public void writeGraph(Graph graph, OutputStream os) throws DataIOException {
        // first, check the schemas to ensure GraphML compatibility
        ns = graph.getNodeTable().getSchema();
        es = graph.getEdgeTable().getSchema();
        checkGraphMLSchema(ns);
        checkGraphMLSchema(es);


        XMLWriter xml = new XMLWriter(new PrintWriter(os));
        xml.begin(Tokens.GRAPHML_HEADER, 2);

        xml.comment("prefuse GraphML Writer | "
                + new Date(System.currentTimeMillis()));

        // print the graph schema
        printSchema(xml, Tokens.NODE, ns, null);
        printSchema(xml, Tokens.EDGE, es, new String[] {
                graph.getEdgeSourceField(), graph.getEdgeTargetField() });
        xml.contentTag(Tokens.KEY, Tokens.ID, Tokens.SCRIPTLIST,"");
        xml.println();

        // print graph contents
        xml.start(Tokens.GRAPH, Tokens.EDGEDEF,
                graph.isDirected() ? Tokens.DIRECTED : Tokens.UNDIRECTED);

        xml.comment("graph elements");

        saveElementsAtLevel(graph, xml, ExtGraph.MasterNodeParentRow);

        xml.end();

        // finish writing file
        xml.finish("</" + Tokens.GRAPHML + ">\n");
    }
```

Abbildung 5: Überdeckung der writeGraph-Methode

6.3.2 ExtGraphMLWriter.writeGraph()

Wie man in Abbildung 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.

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.

6.5 Benutzbarkeit

6.5.1 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.

6.5.2 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.

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.

7 Änderungshistorie

Datum	Thema	Inhalt	Seite
12.12.2006	alles	Beginn der History	21
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	14
06.04.2007	Testergebnisse	Ergebnisse der Überdeckungstests	18
08.04.2007	alles	finales Release 2	

Glossar

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/` *6*

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. *5*, *17*

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. *5*

JUnit Test-Framework

JUnit ist ein Framework zum Testen von Java-Programmen, das besonders für automatisierte Unit-Tests einzelner Units geeignet ist *5*

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. *5*

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. *5*

Prefuse

Ein Java-Framework zum Erstellen und Visualisieren von Graphen, Tabellen und Bäumen. Die Datenfelder lassen sich beliebig mit eigenen Daten erweitern. *4*, *6*

RUP

RUP (Rational Unified Process) ist ein Vorgehensmodell für die Entwicklung objektorientierter Programme. Es verwendet UML als Notationssprache. *5*

SVN

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

Literatur

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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