Bachelor - Praktikum (Graphenlayout)

Qualitätdokumentation Rev. 1

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

postfuse

one step beyond

Stand: 18. Februar 2007

http://bp.macrolab.de

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

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

Inhaltsverzeichnis

1 Einleitung

Das Qualitätssicherungsdokument dient dazu, zu beschreiben, wie wir die Qualitätsmerkmale sicherstellen und im Sinne des Auftraggebers umsetzen wollen. Dabei werden die verwendeten Tests aufgelistet, und einige detailliert beschrieben.
Außerdem gibt das Qualitätssicherungsdokument einen Einblick darüber, welche Qualitätsmerkmale berücksichtigt werden, und welche Priorität sie haben. Dabei verwenden wir die ISO/IEC 9126 Norm als Modell für unsere Qualitätsmerkmale.

2 Qualitätsmerkmale

2.1 Funktionalität

Unter Funktionalität verstehen wir, inwiefern unser 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 keine 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 läuft. Darunter fallen

Robustheit: Hier ist einmal die Robustheit der API zu nennen. Der Graph soll durch keinen API-Zugriff in einen inkonsistenten Zustand versetzt werden können. Andererseits soll die Oberfläche auch wildes Klicken und ausführen von Skripten ohne Absturz verkraften.

Fehlertoleranz: Die Fehlertolerzanz beschreibt die Fähigkeit, dass, trotz Software-Fehlern oder Fehlern bei der Einhaltung der vorher definierten Schnittstellen, ein bestimmtes Leistungsniveau gehalten wird. Fehlertolrenz ist bei unserem Programm vor allem bei Laden von Graphen zu beachten. Hierbei sollten unbedeutende Fehler in der XML-Datei nicht zu einem Abbruch der Ladevorgangs führen.

2.3 Benutzbarkeit

Unter Benutzbarkeit verstehen wir, inwiefern sich der Benutzer in das Produkt einarbeiten muss, damit er es nach seinen Wünschen benutzen 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 ist der zeitkritische Punkt das Layout und das Rendern des Graphen. Hierbei wird, falls ein schnelles Terminieren nicht sichergestellt werden kann, Wert auf Abbrechbarkeit gelegt.

Verbrauchsverhalten: Zudem sollte die Benutzung nicht die Gesamtleistung des Systems zu starkt beeinflussen. Bei dem dynamischen Layout gengt es z.B. wenn dies alle 20 ms neu berechnet wird, und nicht wie in dem verwendeten Package vorgesehen in einer Endlosschleife läuft.

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 eher ein Problem aufgrund der Plugin Struktur. Um diese zu verbessern, wird auf der gleichen Codebasis auch eine standalone Bersion des Programms gebaut, da hier eine leichtere prüfbarkeit besteht.

2.6 Übertragbarkeit

Unter Übertragbarkeit 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. Zumindest bei MacOsX wird aber eine Installation weiterer Komponenten erforderlich sein.

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 da, welche das Projekt während der Entwicklung durchläuft. Beginnend bei der Analysephase bis zum fertigen Produkt kann man dank der Disziplinen und Iterationen bereits früzeitig 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 fr 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

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). Dadurch können wir garantieren, dass nichts implementiert wird, was nicht fehlerfrei ist.
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.
Sobald es uns möglich ist, werden wir auch mehrere Systemtests durchführen, um zu garantieren, dass die einzelnen Komponenten fehlerfrei interagieren.

3.5 Plattform Diversität

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 invensiv Gebrauch gemacht von bereits etablierten Komponenten z.B. zum Umgang mit Graphen oder Skripten. 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.

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 wurden erstellt, so 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 genau die erwarteten Ergebnisse liefern. Die defekten Dateien werden zum Teil durch manipulation aus korrekten Dateien erstellt, zum anderen Teil aus wahrlos ausgewhlten 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 FileNotFound-Exception 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 Dateilesefehler handeln.

Zum Erproben werden einige Exemplarische Scripts an Knoten und Kanten gehngt, 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 GGraph.write() 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 alle NodeShapes getestet werden.

4.3.2 GGraph.write()

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 geschieben 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 sollen einen Teil der Dokumentation des Plugins ausmachen.

5 Testergebnisse

Im folgenden Kapitel werden wir in der nun folgenden Testphase unserer Testergebnisse ablegen. Hierzu werden wir fuer jeden Test eine Tabelle mit folgendem Format ausfüllen:

Name auf dieser Seite	
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: Wann wurde der Test durchgeführt.

Status: Ausgang der Messung. Mgliche Werte sind:

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

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

Akteur: Name der testenden Person.

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.

5.1 Use-Cases

5.1.1 Laden eines Graphen (API)

5.1.2 Starten eines Skripts (GUI)

5.2 Komponententest

5.2.1 GGraph

5.2.2 GScript

5.3 Methodentest

5.3.1 GNodeRender.render()

5.3.2 GGraph.write()

5.4 Benutzbarkeit

5.4.1 Benutzbarkeit der GUI

5.4.2 Benutzbarkeit der API

5.5 Zusammenfassung

6 Änderungshistorie

Datum	Thema	Inhalt	Seite
12.12.2006	alles	Beginn der History	*
26.01.2007	alles	Ausgabe Iteration 0	*
15.02.2007	Blackbox-Test	detailierter Beschrieben	*
16.02.2007	Testergebnisse	Einleitung und Format	*

Glossar

Eclipse

Eclipse ist eine offene Entwicklungsplatform, die auf Java-Technologie beruht. *5*

JUnit Test-Framework

JUnit ist ein Framework zum Testen von Java-Programmen, das besonders für automatisierte Unit-Tests einzelner Units geeignet ist *6*

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 zu bekommen. *6*

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, der andere dient zum Kontrollieren *5*

Prefuse

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

RUP

Rational Unified Process *5*

SVN

Subversion; Team Code Repository *5*

Literatur

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

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

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

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

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

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

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

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