KIT – Karlsruhe Institut of Technology

Praxis der Software-Entwicklung WS 2009/10

Gruppe 7

»Testdokument« v 1.0

Analysesoftware für Soziale Netzwerke

26.02.2010

Phase	Phasenverantwortlich	e-mail
Pflichtenheft	Mark Engel	mark.engel@student.kit.edu
Entwurf	Felix Stahlberg	felix.stahlberg@student.kit.edu
Implementierung	Marc Typke	marc.typke@student.kit.edu
Testen	Felix Stahlberg	felix.stahlberg@student.kit.edu
Präsentation	Mark Engel	mark.engel@student.kit.edu

Inhaltverzeichnis

1 Einleitung

In diesem Dokument werden die Ergebnisse der Testphase beschrieben. Grundprobleme beim Testen des Produktes waren vor Allem:

- **Testen des Userinterfaces**
  Testen des Webinterfaces induziert die Verwendung spezieller Testumgebungen und kann nicht mit einfachen Regressionstests wie JUnit-Tests abgedeckt werden.

- **Testen von datenbanknahem Code**
  Die unkonstante Datenbasis dieses Codes macht es schwierig, diesen zu testen.

Diesen Problemstellungen wurden in verschiedener Weise begegnet. Kapitel 2 behandelt das Testen der Userinterfaces anhand definierter Testszenarien.

Die einzige Lösung, die gesehen wurde, um das zweite Problem zu lösen, besteht darin, eine eigene Testdatenbank hochzuziehen und gegen diesen Datensätze zu testen. Aufgrund des zu hohen Aufwands gegenüber dem wenigen betroffenem Code wurde jedoch darauf verzichtet und aus Kostengründen stattdessen versucht, die entsprechenden Stellen so weit wie möglich einzugrenzen, sodass trotzdem eine möglichst hohe Testabdeckung erziehlt werden konnte.

2 Testszenarien

Zum Testen des Userinterface wird das Web application testing system *Selenium* verwendet. Mit Selenium können Testszenarien definiert werden und automatisch ausgeführt werden. Alle hier vorgestellten Testszenarien wurden mit Selenium definiert und getestet.

Die Testszenarien konnten teilweise aus dem Pflichtenheft übernommen werden. Da sich im Laufe des Projekts sich jedoch die Anforderungen geändert hatten (was durch das Ausscheiden zweier Teammitglieder und damit die Reduzierung der Teamgröße auf 60%), mussten diese Szenarien abgewandelt werden.

Die Oberflächen wird mit der Selenium Bibliothek überprüft, mit der reele Browser für das Testen benutzt werden. Es wird das rudimentäre Verhalten des Benutzerinterfaces überprüft, ohne auf die erstellten Graphen Rücksicht zu nehmen. Das korrekte Erstellen der Graphen wird bereits im Server mittels JUnit Regressionstests überprüft.

Mit den Selenium Test Fällen werden die Testszenarios des Pflichtenheftes abgedeckt.

Der erste Test *testAdminInitAndLogin* überprüft bei einem neu initialisierten Programm, bei dem kein Administrator angelegt ist, das Einfügen des Selbigen und versucht sich danach mit den eingegebenen Benutzerdaten einzuloggen. Dies ist der erste Fall der Pflichtenheft-Szenarien.

Der zweite Oberflächen Test (*testLoginLogout*) versucht sich mit existierenden Daten eines Benutzers einzuloggen (diese müssen in der Datenbank existieren) und sich wieder auszuloggen.

Der Test *testWrongCred* versucht sich mit falschen Benutzerdaten einzuloggen und überprüft die Antwort auf die auszugebende Fehlermeldung.

Der letzte Test *testAddMeasureAndMeasureRestriction* überprüft das Programm auf das Hinzufügen und Löschen von Measure und Restriction Panels.

Mit diesen Tests wird die logische Benutzbarkeit des Interfaces überprüft. Alle Validierungen des Graphen sind über die direkte JUnit Tests des Servers besser auf Korrektheit zu überprüfen und werden deshalb direkt dort und nicht in der Oberfläche überprüft (siehe nächste Kapitel).

3 Regressionstests

3.1 Code Coverage Analyse

Packagenamen werden zur Abkürzung ohne dem Präfix `edu.kit.pse.socialview` notiert.
Die Analyse der Testüberdeckung wurde mit *EclEmma* durchgeführt. Im Folgenden wird das Package `client` nicht betrachtet, da das Testen des Userinterfaces gesondert im Kapitel 2 behandelt wird..
Grundsätzliches Ziel war es, möglichst gute Anweisungsüberdeckung zu erreichen. Dies gelang bei Packages wie `server.controller.kern`, die nur sehr schwache Bindung zum Package `server.model.cosim` und damit zur Cosim-Datenbank haben, besonders gut. Betrachtet werden auf Package-Ebene jeweils neben der Anweisungsüberdeckung noch Anzahl der überdeckten *Basic Blocks*, *Lines*, *Methods* und *Types*.

3.1.1 Das Package server.controller.kern

Tester für Testate in diesem Package liegen in `server.tests.controller.kern`.

Counter	Coverage	Covered	Total
Instructions	93,4 %	770	824
Basic Blocks	85,4 %	117	137
Lines	91,3 %	179	196
Methods	97,1 %	34	35
Types	100,0 %	6	6

Abbildung 1: Code-Coverage von server.controller.kern

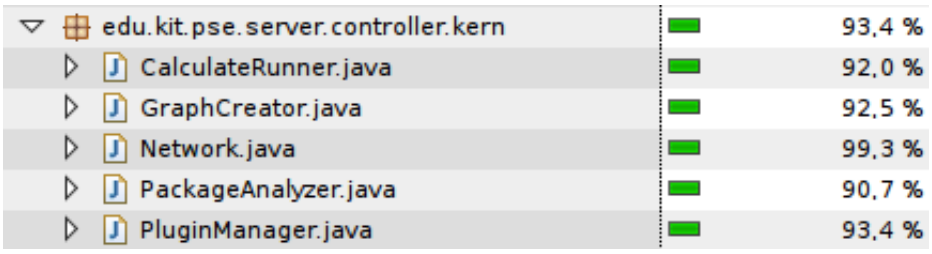

Abbildung 2: Code-Coverage der Klassen in server.controller.kern

3.1.2 Das Package server.controller.graphviz

Tester für Testate in diesem Package liegen in `server.tests.controller.graphviz`.

Counter	Coverage	Covered	Total
Instructions	92,2 %	1390	1508
Basic Blocks	91,8 %	224	244
Lines	91,0 %	252	277
Methods	82,7 %	43	52
Types	100,0 %	7	7

Abbildung 3: Code-Coverage von server.controller.graphviz

edu.kit.pse.server.controller.graphviz	92,2 %
DotEdge.java	100,0 %
DotEdgeFactory.java	98,2 %
DotElement.java	100,0 %
DotFactory.java	100,0 %
DotNode.java	100,0 %
DotNodeFactory.java	98,5 %
GraphVizGraphDisplayCreator.java	70,4 %

Abbildung 4: Code-Coverage der Klassen in server.controller.graphviz

3.1.3 Das Package server.model.datastructures

Trotz relativ hoher Bindung zur Datenbank konnte hier eine recht hohe Abdeckung erzielt werden. Tests bezüglich der Adjacence Array Darstellung des Graphen wurden standardmäßig mit einem Teilgraphen eines dreidimensionalen de-Bruijn-Graph durchgeführt.

Tester für Testate in diesem Package liegen in `server.tests.model.datastructures`.

Counter	Coverage	Covered	Total
Instructions	79,1 %	1968	2489
Basic Blocks	79,8 %	288	361
Lines	77,8 %	430	553
Methods	79,1 %	102	129
Types	91,7 %	11	12

Abbildung 5: Code-Coverage von server.model.datastructures

edu.kit.pse.server.model.datastructures	79,1 %
AdjacenceArrayEdge.java	97,1 %
AdjacenceArrayGraph.java	69,9 %
AdjacenceArrayNode.java	84,1 %
Admin.java	100,0 %
AppEntityManager.java	83,3 %

Abbildung 6: Code-Coverage der Klassen in server.model.datastructures

3.1.4 Das Package server.model.cosim

Dieses Package weist die mit Abstand stärkste Bindung zur Cosim-Datenbank auf. Hier konnten nur untergeordnete Tests der Primary Key Classes implementiert werden.

Counter	Coverage	Covered	Total
Instructions	58,3 %	442	758
Basic Blocks	46,9 %	84	179
Lines	55,8 %	110	197
Methods	33,7 %	35	104
Types	40,0 %	4	10

Abbildung 7: Code-Coverage von server.model.cosim

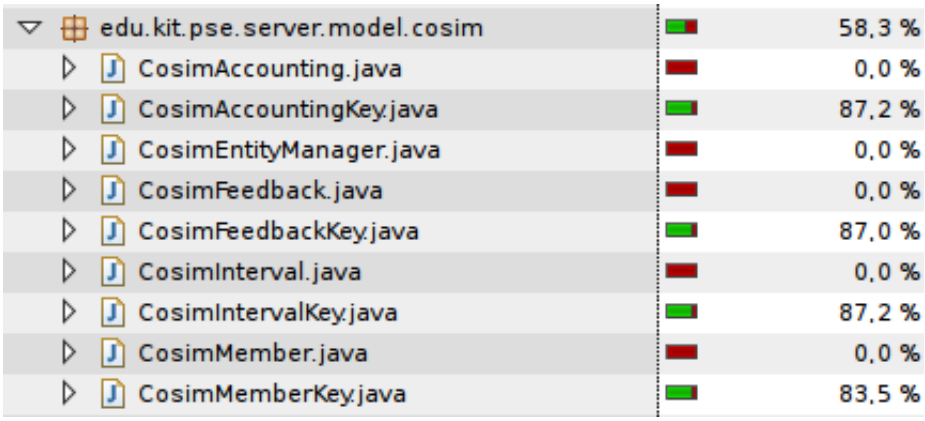

Abbildung 8: Code-Coverage der Klassen in server.model.cosim

4 Probleme

Neben einigen trivialeren Fehlern wurden auch nichttriviale Fehler zu Tage gebracht, von denen hier ein Beispiel beschrieben werden soll.

4.1 Indexverschiebungen nach Graph.delDeactivatedNodes()

Ohne detailiert auf die Adjazenzarraydarstellung einzugehen, die für die interne Graphenrepräsentation genutzt wurde, einzugehen, sei gesagt, dass jeder Knoten durch seinen Index in einem Array vollständig beschrieben wird. Die Klasse AdjacenceArrayNode nutzt diesen Index intern, um Instanzen zu identifizieren. Dies führt jedoch zu Problemen im Zusammenhang mit Graph.delDeactviatedNodes(). Nach der Ausführung dieser Methode werden alle deaktivierten Knoten gelöscht, die Indizes der Knoten ändern sich also im Allgemeinen. Dauert die Lebenszeit einer AdjacenceArrayNode-Instanz die Ausführung von Graph.delDeactivatedNodes() wird sie ungültig oder repräsentiert einen anderen Knoten. Graph.delDeactivatedNodes() wurde im PluginManager nach der Zentralitätsmaßberechnung der Plugins ausgeführt, bei der angenommen wurde, dass keine AdjacenceArrayNode-Instanzen mehr existieren. Die Plugins legen ihre Berechnungen jedoch in einem MeasureValueModel ab, das später bei der dot-Source Generierung erst wieder ausgelesen wird. MeasureValueModel verwendet intern zur Ablage der Daten eine HashMap mit Nodes als Keys. Da die HashMap zum Vergleich von Schlüsseln sich der Methoden equals() und hashCode() bedient und diese beide mit dem gespeicherten Index arbeiten kam es zu fehlerhaften Werten in MeasureValueModel und damit falschen Darstellungen aufgrund falscher Datenbasis.

Lösung: Knoten werden nicht mehr gelöscht, sondern nur noch deaktiviert.

5 Performancetests

Die Laufzeit der Verarbeitung von Graphenanfragen hängt im Wesentlichen zum Einem von der Datenbankverbindung zur Cosim-Datenbank, zum Anderen von der Effizienz der internen Graphenrepräsentation ab. Da die Verbindungsgeschwindigkeit zur Datenbank und deren Performance stark variieren kann, konzentrieren sich die durchgeführten Performancetests meistens auf die Adjazenzarraydarstellung des Graphen. Diese Datenstruktur ist eine sehr effiziente und schnelle Art und Weise, Graphen im Rechner zu repräsentieren, wodurch bei den Tests gute Werte erzielt werden konnten. Getestet wurde auf einem Intel Core2 Duo CPU mit 2 GHz und 2GB RAM unter einem Ubuntu 8.10 auf einem Linux 2.6.27-14-generic Kernel (Sony Vaio BX61XN). Die Java VM wurde in der Version 1.6.0_14 verwendet. Die Ergebnisse wurden mit *gnuplot* geplottet.

5.1 Graphtraversierung

Traversiert wurde über vollständige Graphen mit n Knoten und Schlingen (also n^2 Kanten). Das Diagramm zeigt die durchschnittliche Laufzeit der Traversierung eines Knoten (bzw einer Kante) mit aufsteigendem n.

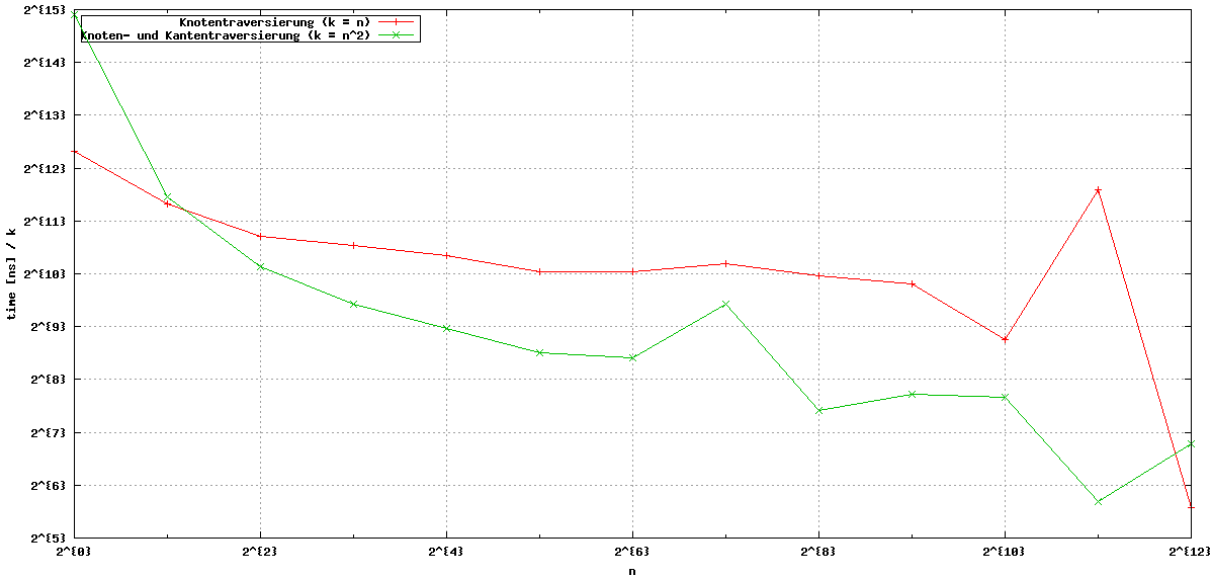

Abbildung 9: Traversierung über vollständige Graphen

Die durchschnittliche Traversiungslaufzeit eines Graphenelements bleibt mit ansteigendem n nicht nur konstant, sondern nimmt sogar ab. Dies ist wahrscheinich auf die effizientere Ausnutzung des Caches bei großen Arrays oder Optimierungsmaßnahmen der Java VM zurückzuführen.

Rechenbeispiel: die Knoten und Kanten eines vollständigen Graphen mit Schlingen der Größe 2^{12} (das heißt 2^{12} = 4096 Knoten und 2^{24} = 16777216 Kanten) kann innerhalb von 1.837 Sekunden traversiert werden.

5.2 Plugin Manager

SocialView bietet zwei Möglichkeiten an, wie der PluginManager die Berechnungen der Zentralitätsmaße durch die Plugins organisiert: sequentiell und nebenläufig durch Threads (in properties.ini einstellbar). Diese beiden Herangehensweise seien im Folgenden gegenübergestellt, wobei die Zentralitätsmaße *Hops* (Entfernung zu einem Startknoten) und *Outdegree* (Ausgangsgrad) auf Graphen mit n Knoten und n*(n * 0.4) Kanten berechnet wurden.

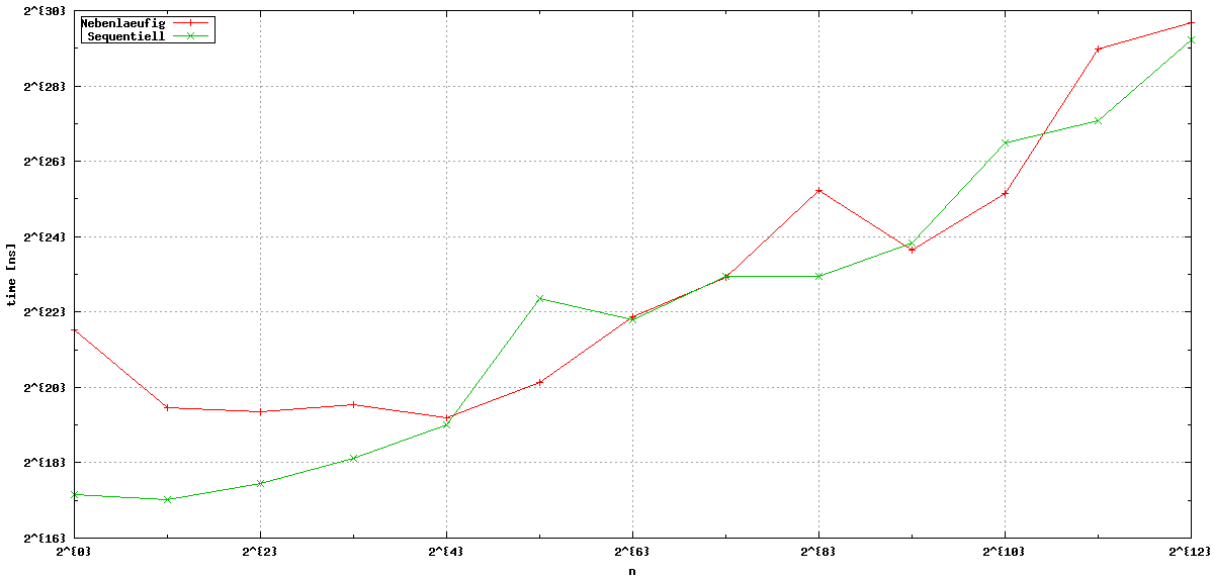

Abbildung 10: Sequentielle und Parallele Zentralitätsmaßberechnung (Ausgangsgrad n*0.4)

Bei kleinem n ist die sequentielle Berechnung schneller, da bei der nebenläufigen Berechnung zusätzlicher Aufwand durch die Erstellung der Threads entsteht. Bei größerem n fallen die Ergebnisse unterschiedlich aus. Die Ursache liegt wahrscheinilch in der komplexen Wechselwirkung zwischen den Plugins untereinander. Bei den Berechnungen kann es vorkommen, dass ein Plugin einen Knoten deaktiviert und das andere Plugin die Berechnung des Zentralitätsmaßes auf diesen Knoten überspringen kann. Dies kann bei paralleler Berechnung in beide Richtungen geschehen, bei sequentieller nur in chronologisch aufsteigender Reihenfolge. So ist es bei sequentieller Berechnung besonders günstig, wenn das erste Plugin viele Knoten deaktiviert und dem zweiten Plugin Arbeit erspart. Parallele Berechnung punktet zum Beispiel, wenn sehr wenige bis keine Knoten deaktiviert werden und beide Plugins Werte auf alle Knoten berechnen.

5.3 Pluginberechnung

Um die Performance der Berechnung von Zentralitätsmaßen auf unterschiedlichen Graphen zu visualisieren, wurden wieder Laufzeiten der Zentralitätsmaßberechnungen für *Hops* und *Outdegree* auf Graphen mit n Knoten des Ausgangsgrads k * n, also kn^2 Kanten in einem Diagramm dargestellt. Die Berechnungen wurden sequentiell durchgeführt.

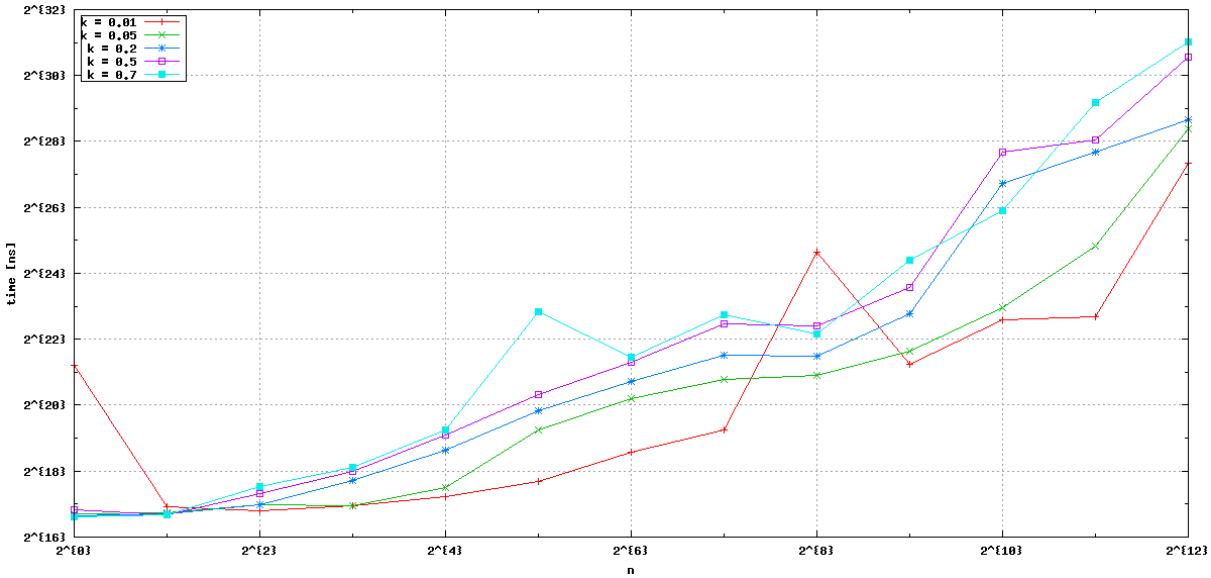

Abbildung 11: Berechnung von Zentralitätsmaßen (Ausgangsgrad k * n)

Rechenbeispiel (Grüner Graph, k = 0.05): Auf einem Graphen mit 2^{12} = 4096 Knoten mit Ausgangsgrad 0.05 * 2^{12} = 204, also 2^{12} * 204 = 835584 Kanten können Ausgangsgrad und Entfernung zu einem Startknoten innerhalb von 0.347 Sekunden berechnet werden.