Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 67

Grafische Benutzeroberfläche zur Analyse von Abhängigkeitsgraphen

Yannic Noller

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer:	Dipl.-Inf. Ivan Bogicevic
	Dipl.-Inf. Jonathan Streit (itestra GmbH)
Beginn am:	1. Mai 2013
Beendet am:	7. August 2013
CR-Nummer:	D.2.6, H.5.2, K.6.3

Kurzfassung

In dieser Bachelorarbeit wurde in Kooperation mit der itestra GmbH eine Eclipse Plug-in zur Visualisierung von Abhängigkeitsgraphen entworfen und implementiert. Als Basis diente das von der itestra GmbH selbst entwickelte Analysewerkzeug, das Abhängigkeiten in Softwaresystemen analysiert. Die Software wurde mit der Vorgehensweise der evolutionären Software-Entwicklung entwickelt, wobei jede Evolutionsstufe durch eine Evaluation mit den Mitarbeitern der itestra GmbH abgeschlossen wurde. Aus der Entwicklung resultierten insgesamt drei Eclipse Plug-ins: DependencyVisualizer, DependencyLayoutAlgorithms und DependencyModelAdapter. Das Plug-in DependencyVisualizer sorgt für die eigentliche Visualisierung des Abhängigkeitsgraphen und nutzt dabei die Layout-Algorithmen aus dem Plug-in DependencyLayoutAlgorithms. Das Plug-in DependencyModelAdapter stellt die Anbindung an das bestehende Analysewerkzeug der itestra GmbH bereit. Das Werkzeug DependencyVisualizer wird in Zukunft von den Mitarbeitern der itestra GmbH eingesetzt.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Vorstellung des Industriepartners

Das Unternehmen itestra GmbH ist ein Software-Dienstleister im Bereich unternehmenskritischer Prozesse, Systeme und Anwendungen. Die itestra GmbH bietet Neu- und Weiterentwicklung von Software, Beratungs- und Reengineeringsleistungen und Seminare an. Das Unternehmen hat seinen Hauptsitz in München und ist international tätig. Unter anderem hat es Kompetenzzentren in Madrid, Tallinn und San Francisco. Die itestra GmbH ist an der Forschung und Weiterentwicklung von bestehenden Methoden interessiert und arbeitet in diesem Rahmen mit Universitäten zusammen. [ite13]

1.2 Motivation

Das Unternehmen itestra GmbH besitzt ein selbst entwickeltes Analysewerkzeug, das Abhängigkeiten von Softwaresystemen analysiert. Dieses Werkzeug wird u.a. für die Identifikation von Unused Code und die Planung von Restrukturierungsmaßnahmen eingesetzt. Es erzeugt ein Datenmodell für einen Graphen, das bisher mit dem Werkzeug dot (siehe Kapitel 2.2.1) zu einem statischen Bild visualisiert wird. Bei großen Graphen ist diese Visualisierungstechnik nicht brauchbar, da im resultierenden Bild nur wenig erkennbar ist (siehe Abbildung 1.1).

Abbildung 1.1: Auszug einer Visualisierung eines Abhängigkeitsgraphen mit ca. 6000 Knoten und 10.000 Kanten mit dem Werkzeug dot

Des Weiteren sind die resultierenden Graphen aus einer Abhängigkeitsanalyse oft zu groß für eine Bildschirmseite. Eine sinnvolle Verwendung dieser Graphen erfordert daher eine dynamische Repräsentation und Möglichkeit zur Interaktion. Aus diesem Grund hat itestra GmbH die Erweiterung ihres Werkzeugs als Bachelorarbeit (siehe Kapitel 1.3) an der Universität Stuttgart ausgeschrieben.

1.3 Aufgabenstellung

Ziel der Arbeit ist der Entwurf und die Implementierung einer grafischen Benutzeroberfläche zur Analyse von Abhängigkeitsgraphen in Kooperation mit dem Unternehmen itestra GmbH. Da diese Benutzeroberfläche das bestehende Analysewerkzeug von itestra GmbH erweitern soll, muss das Werkzeug als Eclipse Plug-in realisiert werden. An das Plug-in werden fünf Kernanforderungen gestellt:

- Anzeige des Graphen in beliebiger Vergrößerung mit verschiebbarem Ausschnitt
- Anzeige von Zusatzinformationen per Mouse-Over
- Fokussieren einzelner Knoten im Graph und Ausblenden aller Knoten, die weiter als n Knoten entfernt sind
- Manuelles Aus- und Einblenden einzelner Knoten per Maus oder durch Eingabe ihres Namens
- Suchen nach Knoten anhand ihres Namens

Folgende Funktionen können optional implementiert werden:

- Suchen nach Verbindungswegen zwischen zwei Knoten
- Markieren der erreichbaren Knoten ausgehend von einem bestimmten Knoten

Des Weiteren soll das entwickelte Werkzeug mit den Mitarbeitern der itestra GmbH evaluiert werden.

1.4 Überblick über das Dokument

Das Dokument ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen : geht auf die Grundlagen dieser Arbeit ein. Insbesondere werden die Ergebnisse der Literaturrecherche vorgestellt.

Kapitel 3 – Entwicklung : erklärt den Ablauf des durchgeführten Softwareprojekts und stellt die entstandene Software vor.

Kapitel 4 – Evaluation : beschreibt die abschließende Evaluation und deren Ergebnisse.

Kapitel 5 – Benutzerhandbuch : erklärt die Bedienung der entwickelten Software.

Kapitel 6 – Zusammenfassung und Ausblick : fasst die Ergebnisse der Arbeit zusammen und stellt Erweiterungsmöglichkeiten vor.

2 Grundlagen

In diesem Kapitel werden die verwendeten Technologien und die Ergebnisse der Literaturrecherche vorgestellt.

2.1 Verwendete Technologien

Die Anforderung 5 (siehe Kapitel 3.1) spezifiziert, dass die zu entwickelnde Software ein *Eclipse*[1] Plug-in sein muss. Im folgenden wird geklärt, was genau ein *Eclipse* Plug-in ist und welche Bestandteile es besitzt. Des Weiteren wird geklärt, wie in einem *Eclipse* Plug-in ein Graph visualisiert werden kann.

2.1.1 Eclipse Plug-in

Ein *Eclipse* Plug-in (im folgenden nur Plug-in genannt) erweitert die Entwicklungsumgebung *Eclipse* um weitere Funktionalitäten. *Eclipse* selbst besteht nur aus einer relativ kleinen Codebasis und wird von vielen Plug-ins ergänzt.

Ein Plug-in kann neue Oberflächenelemente oder eine Klassenbibliothek zur Verfügung stellen und kann von anderen Plug-ins verwendet oder ergänzt werden. Alle Oberflächenelemente in einem Plug-in sind Teil des *Standard Widget Toolkit (SWT)* von Java. Plug-ins werden in *Eclipse* selbst entwickelt, damit sie durch eine emulierte Ausführung von *Eclipse* direkt getestet werden können. Es kann so gut wie jedes Oberflächenelement in *Eclipse* erweitert werden: Menüs, Kontextmenüs, Fenster, usw. Für eine Erweiterung wird lediglich die ID der zu erweiternden Komponente benötigt.

Kern eines Plug-ins ist die Manifest-Datei, die alle wichtigen Informationen des Plug-ins enthält. Dort werden alle Abhängigkeiten und Erweiterungen definiert. Eine spezielle Erweiterung von *Eclipse* ist ein neues Fenster. Es gibt zwei Möglichkeiten ein Fenster in *Eclipse* zu erstellen: eine *View* oder ein *Editor*. *Views* und *Editors* werden in einer *Perspective* angeordnet.

[1] http://eclipse.org/

View

Eine *View* ist die einfachste Art Informationen in *Eclipse* darzustellen. Es ist eine Art Fenster in *Eclipse*, von dem es nur eine Instanz geben kann. Dort können benutzerdefiniert *SWT* Oberflächenelemente platziert werden. Eine *View* kann durch ihre ID eindeutig identifiziert werden, die in der Manifest-Datei definiert ist.

Editor

Ein *Editor* ist etwas komplexer als eine *View* und ist für die Darstellung von Informationen vorgesehen, die geändert werden können. Zum Beispiel wird mit einem *Editor* eine Java-Datei angezeigt, die direkt in diesem Fenster in *Eclipse* bearbeitet werden kann. Ein *Editor* hat immer eine Eingabe, die als eigene Klasse definiert werden muss, die von der Klasse `EditorInput` erbt. Ein *Editor* kann in einer *Eclipse* Instanz für verschiedene Eingaben mehrmals geöffnet sein und kann durch seine, in der Manifest-Datei definierten, ID identifiziert werden. Allerdings haben die verschiedenen *Editor* Instanzen dieselbe ID und können nur durch ihre hinterlegten Informationen unterschieden werden.

Perspective

Eine *Perspective* bestimmt welche *Views* in einer *Eclipse* Instanz standardmäßig angezeigt werden und wie sie dort angeordnet und positioniert sind. Die *Views* werden dabei relativ zur *EditorArea* (immer links oben) oder anderen *Views* positioniert. Je nach Einstellungen in der *Perspective* kann der Benutzer die Anordnung noch selbst anpassen. In einer *Perspective* können alle verfügbaren *Views* angezeigt werden, d. h. auch *Views*, die nicht selbst entwickelt wurden.

2.1.2 JFace

JFace ist ein UI Toolkit, das auf *SWT* aufsetzt. Es enthält Vereinfachungen für die Entwicklung, wie zum Beispiel vorgefertigte Viewer, die verschiedene Oberflächenelemente zur Verfügung stellen (TableViewer, ListViewer, ComboViwer) oder Dialogfenster, die zur schnellen Informationsweitergabe an der Benutzer hilfreich sind. Mit *JFace* kann eine Datenbindung realisiert werden, die es a priori nicht gibt.

Eine besondere Vereinfachung ist der GraphViewer von *JFace*. Damit können einfach Graphen aus dem *Zest Framework* (siehe Kapitel 2.1.3) dargestellt werden. Bestimmte Funktionen wie das Zoomen im Graphen und das Filtern von Knoten wird dadurch einfacher. Leider ist die Implementierung des GraphViewers an verschiedenen Stellen nicht effizient und transparent genug und muss daher eventuell durch eine eigene Implementierung ersetzt werden.

2.1.3 Zest: Eclipse Visualization Toolkit

Das *Graphical Editing Framework (GEF)* bietet Technologien um *Eclipse* grafisch zu erweitern und besteht aus drei Komponenten: *Draw2D*, *GEF (MVC)* und *Zest*. [The13b] *Zest* ist eine Bibliothek bestehend aus Komponenten zur Visualisierung von Graphen. Die Komponenten basieren auf *SWT* und *Draw2D*. [The13d]

Mit *Zest* kann leicht ein Graph visualisiert werden. Das Verschieben von Elementen des Graphen und Funktionen wie das Zoomen im Graphen sind schon vorhanden oder schnell realisierbar. Es sind vorgefertigte Layout-Algorithmen vorhanden und es gibt Vorlagen für neue Layout-Algorithmen. Des Weiteren verletzt *Zest* nicht das Look & Feel von *Eclipse*, da es die gleichen Oberflächenelemente verwendet.

Knoten und Kanten werden in *Zest* durch eigene Klassen repräsentiert, die jeweils ein eigenes Datenattribut haben. Dadurch kann das entsprechende Objekt des Datenmodells hinterlegt werden.

2.2 Ähnliche Werkzeuge

Folgende Werkzeuge/Bibliotheken wurden in der Literaturrecherche identifiziert. Sie bieten Ideen und Ansätze zur Entwicklung des geforderten Werkzeugs.

2.2.1 GraphViz

Bezeichnung:	GraphViz
Autor:	AT&T Research
Version:	2.30.1
Lizenz:	Eclipse Public License v1.0[2]
Beschreibung:	*GraphViz* ist eine Sammlung von Programmen zur Graphvisualisierung. Darin enthalten sind: *dot*, *neato*, *fdp*, *sfdp*, *twopi* und *circo*. Die Programme implementieren unterschiedliche Layout-Algorithmen. Zum Beispiel ordnet *dot* die Knoten des Graphen in einer hierarchischen Struktur an, *neato* versucht ein Energiefunktional zu optimieren und *twopi* ordnet die Knoten in einer radialen Struktur an. Die Komponenten von *GraphViz* sind abgeschlossene Programme, die nicht in ein Plug-in direkt eingebunden werden können, sondern extra aufgerufen werden müssen. Ergebnis der Visualisierung ist eine statische Repräsentation des Graphen. [AT 13]
Bewertung:	*GraphViz* bietet viele verschiedene Möglichkeiten zur Visualisierung von Graphen, ist gut dokumentiert und weit verbreitet. Das Programm *dot* wurde bisher von der itestra GmbH eingesetzt, um Abhängigkeitsgraphen darzustellen. *GraphViz* ist aber prinzipiell nicht für ein Plug-in geeignet, da es eigenständige Programme enthält, die nicht in Java entwickelt wurden. Entscheidendes Problem bei *GraphViz* ist das Resultat der Visualisierung: Eine statische Repräsentation ist nicht ausreichend für die Analyse von Abhängigkeitsgraphen. Dennoch können die Layoutansätze, im speziellen von *dot*, betrachtet und wiederverwendet werden.

[2] http://www.eclipse.org/legal/epl-v10.html

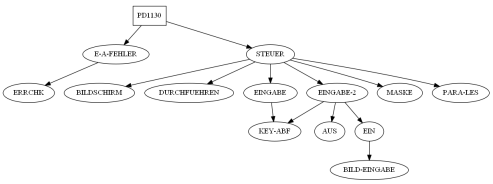

Abbildung 2.1: Visualisierung eines Graphen mit *dot* von *GraphViz*

2.2.2 JGraph

Bezeichnung:	JGraph
Autor:	JGraph Ltd
Version:	1.13.00
Lizenz:	OpenSource BSD License[3]
Beschreibung:	*JGraph* ist eine Java Bibliothek zur Visualisierung von Knoten-Kanten-Diagrammen. *JGraph* ist eine Swing Komponente. Für die Anordnung der Knoten sind verschiedene Standardalgorithmen vorhanden, zum Beispiel für Bäume und hierarchische Strukturen, force-directed Layouts, BPMN, UML und Netzwerke. *JGraph* bietet auch Funktionen zur Analyse eines Graphen, z. B. Finden des kürzesten Weges zwischen zwei Knoten. *JGraph* wird oft eingesetzt, um einen Grapheditor zu entwickeln. Viele Firmen nutzen diese Bibliothek für ihre Anwendungen. [JGr13]
Bewertung:	*JGraph* ist eine ausgereifte Bibliothek zur Visualisierung von Graphen, die alle wichtigen Komponenten enthält. Allerdings ist *JGraph* nur in einer Swing Anwendung verwendbar. Ein Plug-in setzt aber eine *SWT* Anwendung voraus und daher ist *JGraph* für diese Bachelorarbeit nicht geeignet.

[3]http://opensource.org/licenses/BSD-3-Clause

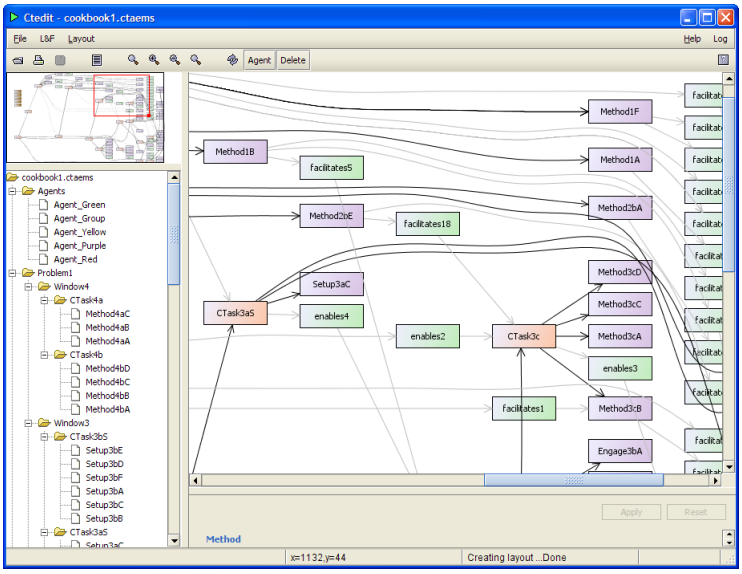

Abbildung 2.2: Beispiel einer Anwendung, die *JGraph* zur Visualisierung des Graphen verwendet [JGr13]

2.2.3 PDE Visualization

Bezeichnung:	PDE Incubator Dependency Visualization
Autor:	The Eclipse Foundation
Version:	M1 - 13.04.2009
Lizenz:	Eclipse Public License v1.0
Beschreibung:	*PDE Incubator Dependency Visualization (PDE)* ist Teil des *Eclipse Incubator* Projekts und visualisiert die Abhängigkeiten zwischen verschiedenen Plug-ins. *PDE* basiert auf *Zest* und *Draw2D*. *PDE* bietet unter anderem folgende Funktionalitäten: Darstellung von Knoten-Kanten-Diagrammen, Färbung von Knoten und Kanten, Screenshots des Graphen als Bildexport, verschiedene Zoom Stufen, Berechnung kürzester Wege und Suche von Knoten. [The13c]
Bewertung:	Grundsätzlich ist *PDE* ein guter Ausgangspunkt für diese Bachelorarbeit: es werden Abhängigkeiten in Knoten-Kanten-Diagrammen mit *Zest* dargestellt. *PDE* bietet keine Möglichkeit eingehende Abhängigkeiten anzuzeigen und es können keine Knoten aus dem Graph gefiltert werden. Der Code könnte demnach als Basis verwendet werden, müsste aber erweitert werden. Da der Link zu den Quelldateien des Projekts fehlerhaft ist und damit der Code nicht eingesehen werden kann, ist eine umfassende Bewertung nicht möglich. Die Ansätze von *PDE* sind dennoch interessant und hilfreich.

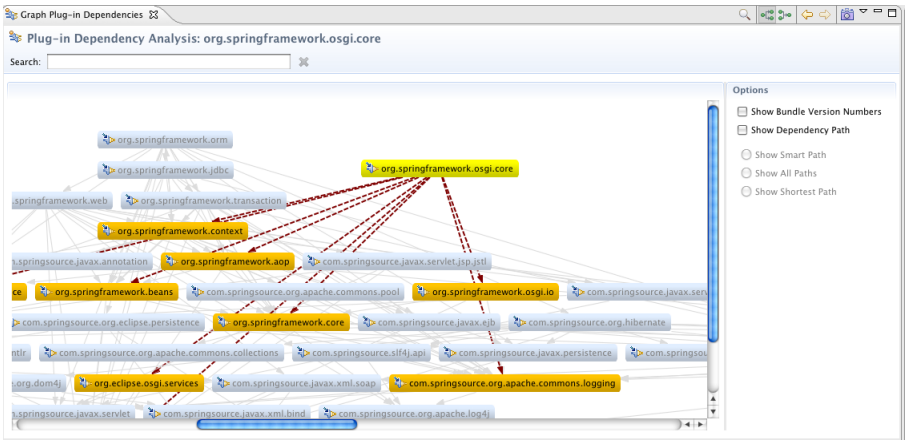

Abbildung 2.3: Visualisierung eines Graphen mit *PDE* [The13c]

2.2.4 Sotograph

Bezeichnung:	Sotograph
Autor:	hello2morrow
Version:	7.1.9
Lizenz:	proprietär
Beschreibung:	Der *Software-Tomograph (Sotograph)* ist ein Werkzeug zur Qualitätsanalyse von Software. Es kann eigenständig betrieben werden oder in Kombination mit anderen Werkzeugen von *hello2morrow*. *Sotograph* unterstützt folgende Programmiersprachen: ABAP, ABAPObjects, C, C++, C# und Java. *Sotograph* kann in folgende Entwicklungsumgebungen integriert werden: *Eclipse*, *Visual Studio*, *UltraEdit*, *SlickEdit* und *Emacs*. *Sotograph* erweitert *Sotoarc*, das Basisprodukt von *hello2morrow*, um folgende Funktionalitäten: die Visualisierung der Ergebnisse mit verschiedenen Layout-Algorithmen und Färbung des Graphen, das Finden von Eigenschaften wie zyklische Abhängigkeiten, dupliziertem Code, usw. und das Überwachen und Verwalten der Software-Evolution. [hel13]
Bewertung:	*Sotograph* ist ein sehr umfangreiches Werkzeug zur Analyse von Software. Da es aber nicht unter einer OpenSource Lizenz veröffentlicht wurde, ist es in dieser Bachelorarbeit nicht verwendbar. Des Weiteren besitzt die itestra GmbH schon ein Werkzeug zur Analyse, es fehlt lediglich eine effiziente Visualisierungskomponente.

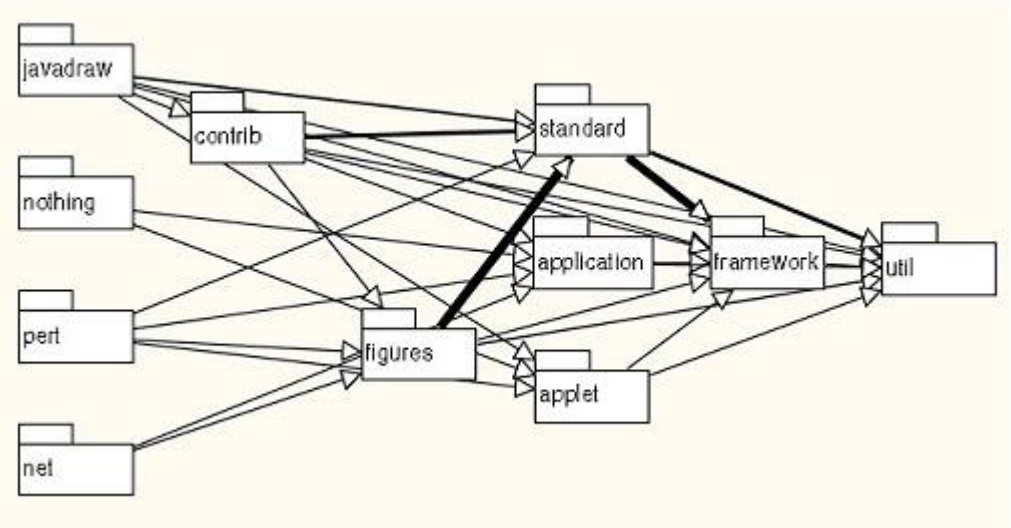

Abbildung 2.4: Visualisierung eines Graphen mit *Sotograph* [hel13]

2.2.5 yEd

Bezeichnung:	yEd
Autor:	yWorks
Version:	3.10.2
Lizenz:	Freeware
Beschreibung:	*yEd* ist ein kostenloses Werkzeug zur Erstellung und Visualisierung von Graphen. *yEd* unterstützt Diagramme wie BPMN, Flussdiagramme, Bäume, Netzwerke und UML. Es gibt viele Layout-Algorithmen zum automatischen Anordnen der Knoten und Kanten. Es können Daten in Formaten wie xls, xml oder graphml importiert werden. *yEd* bietet einen Export in diverse Bildformate. *yEd* ist vielseitig anpassbar (Farben, Formen, usw.) und enthält eine geringe Anzahl Funktionen zur Analyse von Graphen. [yWo13a]
Bewertung:	*yEd* ist ein sehr umfangreicher und benutzerfreundlicher Grapheditor. Durch viele verschiedene Sichten auf den Graphen erhält der Benutzer einen guten Überblick. Dennoch ist *yEd* eine abgeschlossene Anwendung wie *GraphViz* und kann daher nicht direkt in ein Plug-in integriert werden. Die umfangreichen Funktionen zur Bearbeitung von Graphen werden von der itestra GmbH nicht benötigt, stattdessen fehlen Funktionen wie die Suche nach kürzesten Wegen zwischen zwei Knoten.

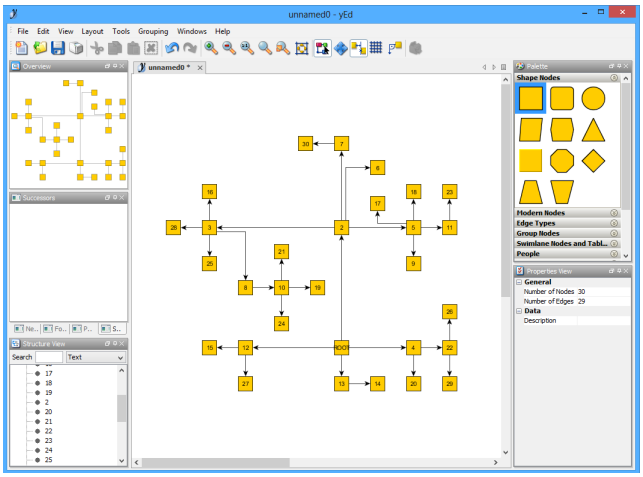

Abbildung 2.5: Visualisierung eines Graphen mit *yEd*

2.3 Bibliotheken zur Analyse von Graphen

Folgende Bibliotheken wurden dahingehend geprüft, ob sie für die Berechnung von Verbindungswegen zwischen zwei Knoten (siehe Anforderung 32 Kapitel 3.1) eingesetzt werden können.

2.3.1 JGraphT

Bezeichnung:	JGraphT
Autor:	Barak Naveh and Contributors
Version:	0.8.3
Lizenz:	GNU Lesser General Public License[4]
Beschreibung:	*JGraphT* ist eine quelloffene Java Bibliothek, die verschiedene graphentheoretische Objekte und Algorithmen zur Verfügung stellt. Es gibt eine Vielzahl von verschiedenen Graphentypen, Datenstrukturen und Algorithmen (Dijkstra Shortest Path, Bellman Ford, K-Shortest-Path, usw.). Optional kann *JGraph* zur Visualisierung des Graphen verwendet werden. [Bar13]
Bewertung:	*JGraphT* ist eine weitverbreitete und ausgereifte Bibliothek zur Analyse von Graphen. Da *JGraphT* aber unter LGPL veröffentlicht wurde, ist es zur Verwendung in einem Plug-in nicht geeignet, da dieses gewöhnlich unter EPL veröffentlicht wird und LGPL und EPL nicht kompatibel sind [The13a].

[4] http://www.gnu.org/licenses/lgpl-3.0.en.html

2.3.2 JUNG

Bezeichnung:	JUNG
Autor:	Joshua O'Madadhain, Danyel Fisher, Tom Nelson
Version:	2.0.10
Lizenz:	BSD Open Source License
Beschreibung:	*Java Universal Network/Graph Framework (JUNG)* ist eine quelloffene Java Bibliothek zur Analyse von Graphen. Unter Anderem gibt es eine Vielzahl an Algorithmen zur Graphentheorie, Data Mining, Netzwerkanalyse, Clustering, Optimierung, statistische Analyse und wichtigen Metriken (PageRank, HITS, usw.). [Jos13]
Bewertung:	*JUNG* bietet eine Vielzahl an Möglichkeiten, ist aber nicht so weit verbreitet wie *JGraphT*. Für die Anforderungen an die Bachelorarbeit (siehe Kapitel 3.1) genügen die vorhandenen Funktionen und bieten auch Erweiterungsmöglichkeiten. Die Lizenzierung von *JUNG* ist kompatibel mit EPL [The13a].

2.3.3 yFiles

Bezeichnung:	yFiles
Autor:	yWorks
Version:	2.10
Lizenz:	proprietär
Beschreibung:	*yFiles for Java (yFiles)* ist eine umfangreiche Java Bibliothek, die Algorithmen (Bellman Ford, Dijkstra, K-Shortest-Path, usw.), Visualisierung und das automatische Anordnen der Knoten und Kanten der Graphen anbietet. [yWo13b]
Bewertung:	*yFiles* ist sehr umfangreich und gut dokumentiert. Da die Bibliothek aber kostenpflichtig und nicht quelloffen ist, kann sie in dieser Bachelorarbeit nicht verwendet werden.

3 Entwicklung

3.1 Anforderungssammlung

Im folgenden werden alle Anforderungen an das Softwareprodukt aufgezählt. Nicht alle Anforderungen waren von Beginn dieser Arbeit an klar und definiert. Viele haben sich während dem Analysegespräch ergeben, andere sind erst nach der Entwicklung des ersten oder zweiten Prototyps erkannt worden.

Die Nummerierung der Anforderungen wird fortlaufend erhöht, damit jede Anforderung durch die entsprechende Nummer identifiziert werden kann.

Anforderungen an die Durchführung des Softwareprojekts

1. Zur Entwicklung der Software soll ein evolutionäres Vorgehensmodell (siehe Kapitel 3.2.1) eingesetzt werden.
2. Die finale Version sollte Mitte/Ende Juni 2013 ausgeliefert werden.
3. Es müssen keine Spezifikation oder sonstige Dokumente angefertigt werden, die Software soll dennoch im Code ausreichend kommentiert sein.
4. Die Software muss von den Mitarbeitern der itestra GmbH evaluiert werden und die Ergebnisse müssen schriftlich festgehalten werden.

Allgemeine Anforderungen an das Werkzeug

5. Die Software muss als *Eclipse* Plug-in vorliegen.
6. Zur Entwicklung muss Java in der Version 1.6 verwendet werden.
7. Die *Eclipse*-Zielversion muss zwischen 3.4 und 3.7 liegen.
8. Die Software muss unter Windows 7 lauffähig sein.
9. Die Sprache der Benutzeroberfläche soll zunächst Englisch sein, muss aber erweiterbar gestaltet werden (Internationalisierung).

10. Es sind die Java Code Conventions[1] einzuhalten.
11. Es muss eine eigene *Perspective* für die Benutzung des zu entwickelnden Plug-ins erstellt werden.
12. Das Werkzeug muss unabhängig von Komponenten der itestra GmbH lauffähig sein. Damit muss ein eigenes Datenmodell entwickelt werden.

Anforderungen an die Anbindung an das Analysewerkzeug von der itestra GmbH

13. Es muss eine Eingabemaske geben, mit der man durch Eingabe der Kommandozeilenparameter des Analysewerkzeugs die Analyse starten kann und der resultierende Graph visualisiert wird.
14. Das Datenmodell des Analysewerkzeugs muss in das eigene Datenmodell umgewandelt werden können.
15. Die eingegeben Kommandozeilenparameter sollen permanent gespeichert werden und wiederverwendet werden können.
16. Die Log-Ausgaben des Analysewerkzeugs sollen abgefangen und in der Benutzeroberfläche angezeigt werden.

Anforderungen an die Darstellung des Graphen

17. Der Graph muss in einem Knoten-Kanten-Diagramm dargestellt werden.
18. Die Darstellung eines Graphen muss mit bis zu 10.000 Knoten möglich sein.
19. Der Graph muss in beliebiger Vergrößerung dargestellt werden können.
20. Der sichtbare Ausschnitt des Graphen muss verschiebbar sein.
21. Zusatzinformationen zu einem Knoten müssen per Mouse-Over angezeigt werden können.
22. Zusatzinformationen zu einem Knoten müssen in der schon existierenden PropertySheet von *Eclipse* angezeigt werden können.
23. Die Anordnung der Knoten und Kanten (Layout) soll die Hierarchie im Graphen repräsentieren.
24. Es müssen mehrere Graphen gleichzeitig geöffnet sein können. Dabei soll man zwischen ihnen wechseln können und die Anzeige muss sich entsprechend aktualisieren.

[1] http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

Anforderungen an Graphtransformationen

25. Die Graphtransformationen sollen, wenn sinnvoll, in einem gemeinsamen Fenster verfügbar gemacht werden.
26. Einzelne Knoten müssen im Graphen fokussiert werden können.
27. Ausgehend von einem fokussierten Knoten, müssen alle Knoten ausblendbar sein, die mehr als n Knoten entfernt sind.
28. Einzelne bzw. mehrere Knoten müssen per Maus manuell aus- und einblendbar sein.
29. Einzelne bzw. mehrere Knoten müssen durch Eingabe ihres Namens manuell aus- und einblendbar sein.
30. Einzelne bzw. mehrere Knoten müssen anhand ihres Namens gesucht werden können.
31. Für den ersten Treffer der Suche müssen die Zusatzinformationen in der PropertySheet von *Eclipse* angezeigt werden.
32. Es muss nach Verbindungswegen zwischen zwei Knoten gesucht werden können.
33. Das Markieren der erreichbaren Knoten ausgehend von einem bestimmten Knoten muss möglich sein.

3.2 Planung

3.2.1 Vorgehensmodell

Auf Wunsch der itestra GmbH wurde vereinbart das Softwareprodukt mit einem evolutionären Vorgehensmodell zu entwickeln. Der itestra GmbH war es wichtig:

- früh im Entwicklungsprozess ein ausführbares Programm zu haben und
- Anforderungen sukzessive nachreichen zu können.

In dieser Bachelorarbeit wird die Definition der evolutionären Software-Entwicklung von Ludewig und Lichter [JL09] verwendet, die wie folgt lautet:
"*Vorgehensweise, die eine Evolution der Software unter dem Einfluss ihrer praktischen Erprobung einschließt. Neue und veränderte Anforderungen werden dadurch berücksichtigt, dass die Software in sequenziellen Evolutionsstufen entwickelt werden.*"

Es wurde mit dem Kunden vereinbart drei dieser *Evolutionsstufen* zu entwickeln: Evolutionsstufe 1, Evolutionsstufe 2 und die finale Version. Nach jeder Evolutionsstufe gibt es eine Evaluierung durch die Mitarbeiter der itestra GmbH, die das Werkzeug testen und ein konstruktives Feedback geben. Dieses Feedback enthält: gefundene Fehler, geänderte Anforderungen und neue Anforderungen. Durch den praktischen Einsatz und dem darauffolgenden Feedback wird gewährleistet, dass ein Produkt entwickelt wird, das später wirklich von den Mitarbeitern der itestra GmbH eingesetzt werden kann.

3.2.2 Evolutionsstufe 1

Geplante umzusetzende Funktionen

Die erste Evolutionsstufe sollte eine rudimentäre Visualisierung des Graphen enthalten. Des Weiteren sollten so viele Funktionen wie möglich implementiert werden. Eine bestimmte Menge an Funktionen war dabei nicht definiert. Vielmehr ging es in der ersten Evolutionsstufe darum, zu überprüfen, dass die Entwicklung in die richtige Richtung geht und Missverständnisse aufzudecken.

Umgesetzte Funktionen

- Anzeige des Graphen in beliebiger Vergrößerung mit verschiebbarem Ausschnitt
- Manuelles Aus- und Einblenden einzelner Knoten per Maus oder durch Eingabe ihres Namens
- Suchen nach Knoten anhand ihres Namens
- Anzeige von Zusatzinformationen
- Markieren der erreichbaren Knoten ausgehend von einem bestimmten Knoten

Feedback

Das Feedback der Mitarbeiter der itestra GmbH fiel grundsätzlich positiv aus. Sie hatten dennoch einige Punkte anzumerken:

- Die Anordnung diverser Menüpunkte war nicht optimal und sollte verbessert werden.
- Das Aktualisieren des Layouts im Graphen sollte in einem eigenen Prozess ausgeführt werden, damit lang andauernde Berechnungen die Oberfläche nicht unnötig blockierten.
- Die *View* DependencyLoader sollte ausgebaut und in die Anforderungen übernommen werden. Dabei sollte die Historie der eingegebene Parameterwerte gespeichert werden. Fehler in den Parameterwerten sollten abgefangen und ausgegeben werden.
- Die vorhandene *Perspective* erschien nicht sinnvoll und sollte überarbeitet werden.
- Die Visualisierung sollte dahingehend erweitert werden, dass mehrere Graphen gleichzeitig geöffnet sein können.

Probleme während der Entwicklung

- **Integration des Analysewerkzeugs von der itestra GmbH in die Eclipse IDE**
  Um das geforderte Plug-in zu entwickeln, war ein Zugriff auf das Analysewerkzeug notwendig. Dieses musste erst in mehrere Plug-ins portiert werden, das aufgrund von fehlenden Abhängigkeiten der verschiedenen Plug-ins untereinander länger dauerte als geplant.

3.2.3 Evolutionsstufe 2

Geplante umzusetzende Funktionen

Die zweite Evolutionsstufe sollte bereits alle geforderten Funktionen enthalten, damit zur finalen Version nur noch Fehler behoben werden müssen.

Umgesetzte / überarbeitete Funktionen

- Anzeige des Graphen in belieber Vergrößerung mit verschiebbarem Ausschnitt
- Anzeige von Zusatzinformationen per Mouse-Over und PropertySheet
- Fokussieren einzelner Knoten im Graphen und Ausblenden aller Knoten, die weiter als n Koten entfernt sind.
- Manuelles Aus- und Einblenden einzelner Knoten per Maus oder durch Eingabe ihres Namens
- Suche nach Knoten anhand ihres Namens
- Suche nach Verbindungswegen zwischen Knoten
- Markieren der erreichbaren Knoten ausgehend von einem bestimmten Knoten
- Layouts: ExtendedTreeLayout, ExtendedRadialLayout
- Log Stream des DependencyAnalyzers wird im DependendyLoader angezeigt

Bekannte Probleme des Werkzeugs

Folgende zwei Probleme traten während der Entwicklung der zweiten Evolutionsstufe auf und wurden erst während der Entwicklung der finalen Version gelöst.

- Das Filtern des Graphen war bei großen Graphen (ca. 6000 Knoten und 15.000 Kanten) unbrauchbar, weil es zu langsam war.
- Das Schließen eines ähnlich großen Graphen war nicht möglich, da *Eclipse* dabei abstürzte.

Feedback

Das Feedback der Mitarbeiter der itestra GmbH bestätigte die Korrektheit und Fortschritte der Entwicklung. Folgende Anmerkungen kamen von den Mitarbeitern der itestra GmbH:

- Die Anwendung muss noch stabiler werden. Diverse Exceptions wurden beobachtet.
- Die Anzeige der *View* Actions ist bei mehreren geöffneten Graphen nicht konsistent.
- Diverse Verbesserungen an der Usability der Anwendung.

Probleme während der Evolutionsstufe

- **Erweiterung der Layout Algorithmen aus Zest**
  Die Layout Algorithmen aus *Zest* sind unter der Eclipse Public License (EPL) veröffentlicht. Eine Änderung dieser Algorithmen erfordert eine Lizenzierung der eigenen Software unter EPL. Da diese Lizenz aber unter Umständen mit den Anforderungen der itestra GmbH in Konflikt steht, wurde beschlossen die angepassten Layout Algorithmen in ein eigenes Plug-in zu packen, unter EPL zu veröffentlichen und dieses wiederum unverändert in der eigenen Software zu verwenden. Die Änderung der vorhandenen Standardalgorithmen war notwendig, da die Algorithmen das Layout auf die gegebene Fenstergröße anpassen und dies vom Kunde nicht gewünscht ist. Des Weiteren sollen die Layouts vorberechnet werden können, ohne sie direkt anzuwenden.
- **Umstieg von View auf Editor**
  Zu Beginn der Entwicklung war es vorgesehen den Graph in einer eigenen *View* (siehe Kapitel 2.1.1) anzuzeigen. Damit wäre es nicht möglich gewesen mehrere Graphen parallel zu öffnen, da eine *View* genau einmal geöffnet sein kann. Die Anforderung, dass mehrere Graphen gleichzeitig geöffnet sein können, wurde mit dem ersten Prototyp erkannt. Deshalb wurde die *View* auf einen *Editor* umgestellt. Diese Änderung war mit größerem Aufwand verbunden:
  - Ein *Editor* benötigt immer eine Eingabe, die durch eine eigene Datenklasse realisiert werden muss.
  - Ein *Editor* hat im Gegensatz zu einer *View* keine Toolbar, daher mussten die vorhandenen Funktionen in der Toolbar der alten *View* in die Gesamtanwendung integriert werden.
  - Da es nun mehrere Graphen gleichzeitig geben kann, muss überwacht werden, welcher Graph aktuell durch den Benutzer ausgewählt ist. Das wurde durch ein Observer-Pattern mit der Klasse `GraphEditorMonitor` gelöst (siehe Kapitel 3.3).
- **DependencyLoader vom Debugging-Tool zur Anforderung**
  Ursprünglich war die *View* DependencyLoader, mit der man das Analysewerkzeug der itestra GmbH starten kann, nur für Debugging-Zwecke für den Entwickler geplant. Nach dem ersten Prototyp, in dem diese *View* enthalten war, erhob der Kunde allerdings die Anforderung, diese *View* im Produkt zu belassen.

- **Details View ersetzt durch PropertySheet von Eclipse**
  Im ersten Prototyp war eine *View* enthalten, die Zusatzinformationen zu einem markierten Knoten anzeigt. Während der Entwicklung des zweiten Prototyps wurde klar, dass eine eigene *View* dafür unnötig ist, da es in *Eclipse* schon die *View* PropertySheet gibt, in der man genau diese Informationen darstellen kann. Die Umstellung auf die *View* PropertySheet erwies sich dennoch schwieriger als gedacht, da dazu das Interface `IAdaptable` implementiert werden musste und die Anzeige nicht direkt gefüllt werden kann, sondern über die Selection Events im Graphen gesteuert wird.
- **Umstellung auf eigenes Datenmodell**
  Anfangs baute die Visualisierung des Graphen auf Basis des schon existierenden Datenmodells der itestra GmbH auf. Da aber das Visualisierungswerkzeug unabhängig von der itestra GmbH veröffentlicht werden soll, musste ein eigenes Datenmodell modelliert und implementiert werden. Damit trotzdem die nahtlose Anbindung zwischen dem vorhandenen Analysewerkzeug und der zu entwickelten Visualisierung sichergestellt werden kann, musste ein Konverter für die Datenmodelle implementiert werden. Es wurde das Interface `IModelAdapter` erstellt, damit weitere Konverter nach dieser Vorlage implementiert werden können.

3.2.4 Finale Version

Geplante umzusetzende Funktionen

Die finale Version enthält alle Funktionen und kann damit von den Mitarbeitern der itestra GmbH abschließend evaluiert werden.

Realisierte Funktionen

Alle Anforderungen wurden in der finalen Version umgesetzt.

Feedback

Das Feedback der Mitarbeiter der itestra GmbH wurde für die finale Version in einer separaten Evaluation erfasst (siehe Kapitel 4).

Probleme während der Evolutionsstufe

- **Verschieben des Graphen**
  Um den sichtbaren Ausschnitt des Graphen intuitiv verschieben zu können, mussten manuell die Scrollbars manipuliert werden. Eigentlich gibt es dazu eine Funktion im anfangs verwendeten GraphViewer von *JFace*, aber diese Funktion hat in diesem

Kontext nicht das gewünschte Ergebnis geliefert. Die Implementierung, um den Graph zu verschieben, war demnach schwerer als gedacht.

- **Zoomen des Graphen**
  Es gibt die Möglichkeit das Zoomen eines Graphen durch Menü-Einträge oder Kontextmenü-Einträge zu steuern. Das ist aber umständlich und verschwendet viel Platz in den entsprechenden Menüs. Besser ist daher das Zoomen mit Kombination der STRG-Taste und des Mausrads, da diese Variante den meisten Benutzer durch andere Anwendungen schon bekannt ist. Um die Zoomstufe festzulegen gibt es auch eine Funktion im anfangs verwendeten GraphViewer von *JFace*, aber diese Funktion hat in diesem Kontext nicht das gewünschte Ergebnis geliefert. Die Implementierung, um im Graph zu zoomen, war demnach schwerer als gedacht.

- **Performance Probleme beim Filtern**
  Das Filtern von Knoten im Graphen wurde im ersten und zweiten Prototyp über den GraphViewer von *JFace* geregelt. Es stellte sich aber dann heraus, dass bei großen zu filternden Mengen, diese Funktion nicht effizient genug war und dazu führte, dass die gesamte *Eclipse* Anwendung abstürzte. Um das Filtern für große Knotenmengen dennoch anbieten zu können, wurde das Filtern auf Basis des eigenen Datenmodells neu entwickelt. Im folgenden wurde der GraphViewer von *JFace* durch die direkte Anzeige des *Zest* Graphen ersetzt, da der GraphViewer keine erkennbaren Vorteile mehr lieferte.

- **Probleme beim Schließen von großen Graphen**
  Bei der Entwicklung des zweiten Prototyps wurde festgestellt, dass sich der *Editor* mit einem großen Graphen mit mehreren Subgraphen nicht mehr schließen lässt. Das genaue Problem war dabei lange nicht klar. Zur Lösung des Problems wurde ein StackOverflow Eintrag[2] angelegt, der allerdings zu keiner Lösung führte. Nach einiger Zeit der Fehlersuche wurde klar, dass der Fehler im Framework von *Zest* zu finden ist. Schließlich wurde der Fehler gefunden und ein *Eclipse* Bug Report[3] geschrieben. Der Grund für das Problem war eine while-Schleife, deren Weitermachbedingung nie als falsch ausgewertet wurde und damit zu einer Endlosschleife führte. Da der Fehler nur bei bestimmten Grapheigenschaften auftrat, wurde er anfangs nicht bemerkt. Insgesamt verzögerte dieser Fehler den Fortschritt der Bachelorarbeit um ca. eine Woche.

- **Wechsel von JGraphT zu JUNG**
  Um die Anforderung 32 (siehe Kapitel 3.1) zu erfüllen, wurde eine Bibliothek benötigt, die einen Verbindungsweg zwischen zwei Knoten im Graphen berechnet. Zu Beginn wurde dazu JGraphT gewählt, da diese Bibliothek weit verbreitet und gut dokumentiert ist. OpenSource *Eclipse* Plug-ins sollten unter der EPL veröffentlicht werden. JGraphT wurde unter LGPL veröffentlicht und EPL ist mit LGPL nicht kompatibel. Deshalb wurde die Berechnung des kürzesten Weges auf die Bibliothek JUNG umgestellt,

[2]http://stackoverflow.com/questions/17369676/cant-close-editor-part-with-big-zest-graph
[3]https://bugs.eclipse.org/bugs/show_ bug.cgi?id=412446

welche unter der Lizenz BSD veröffentlicht wurde, die wiederum kompatibel mit der EPL ist.

3.2.5 Zeitplanung

Die Zeitplanung dieser Bachelorarbeit lässt sich grob in vier Phasen gliedern: die Einarbeitungsphase, die Entwicklung der ersten Evolutionsstufe, die Entwicklung der zweiten Evolutionsstufe und die Entwicklung der finalen Version (siehe Abbildung 3.1).

Die Einarbeitungsphase beginnt mit einer allgemeinen Recherche über *Eclipse* Plug-ins. Ziel ist es herausfinden, was Plug-ins genau sind und wie man sie entwickelt. Nach dieser Recherche folgt die Entwicklung eines Labormusters (vgl. [JL09]), um herauszufinden, wie ein Graph mit einem Plug-in visualisiert werden kann. Gleichzeitig erfolgt die Literaturrecherche über schon existierende Werkzeuge, die sich mit der gleichen Aufgabenstellung beschäftigen, und über Bibliotheken, welche die Entwicklung erleichtern können. Während der gesamten Einarbeitungsphase wird versucht, das bestehende Analysewerkzeug der itestra GmbH in die eigene Entwicklungsumgebung einzubauen und die Schnittstellen zu verstehen.

Gegen Ende der Einarbeitungsphase beginnt die Planung der Architektur. Diese muss mit einer anderen Bachelorarbeit abgestimmt werden, welche die Aufrufe des Analysewerkzeugs der itestra GmbH mithilfe eines separaten Plug-ins vereinfacht und auch die Anbindung an die hier entwickelte Graphvisualisierung bieten soll.

Die Entwicklung der ersten Evolutionsstufe soll eine erste ausführbare Visualisierung liefern (siehe Kapitel 3.2.2). Anschließend findet eine erste Evaluation durch die Mitarbeiter der itestra GmbH statt.

Die zweite Evolutionsstufe soll das Feedback der ersten Evaluation umsetzen und bereits alle zu implementierenden Funktionen enthalten (siehe Kapitel 3.2.3). Anschließend findet die zweite Evaluation durch die Mitarbeiter der itestra GmbH statt.

Spätestens nach der zweiten Evolutionsstufe soll die Zwischenpräsentation der Bachelorarbeit stattfinden, damit das dortige Feedback noch eingebaut werden kann. In der Zwischenpräsentation soll das Thema, der aktuelle Stand und die noch ausstehenden Aufgaben der Bachelorarbeit vorgestellt werden.

Die finale Version soll das Feedback der zweiten Evaluation umsetzen und gegebenenfalls noch bestimmte Bedienkonzepte verbessern (siehe Kapitel 3.2.4). Anschließend findet die abschließende Evaluation des Werkzeugs statt (siehe Kapitel 4).

Am Ende der Bachelorarbeit soll eine Abschlusspräsentation stattfinden, die das Ergebnis vorstellt. Während der gesamten Zeit soll an dem eigentlichen Dokument der Bachelorarbeit gearbeitet werden.

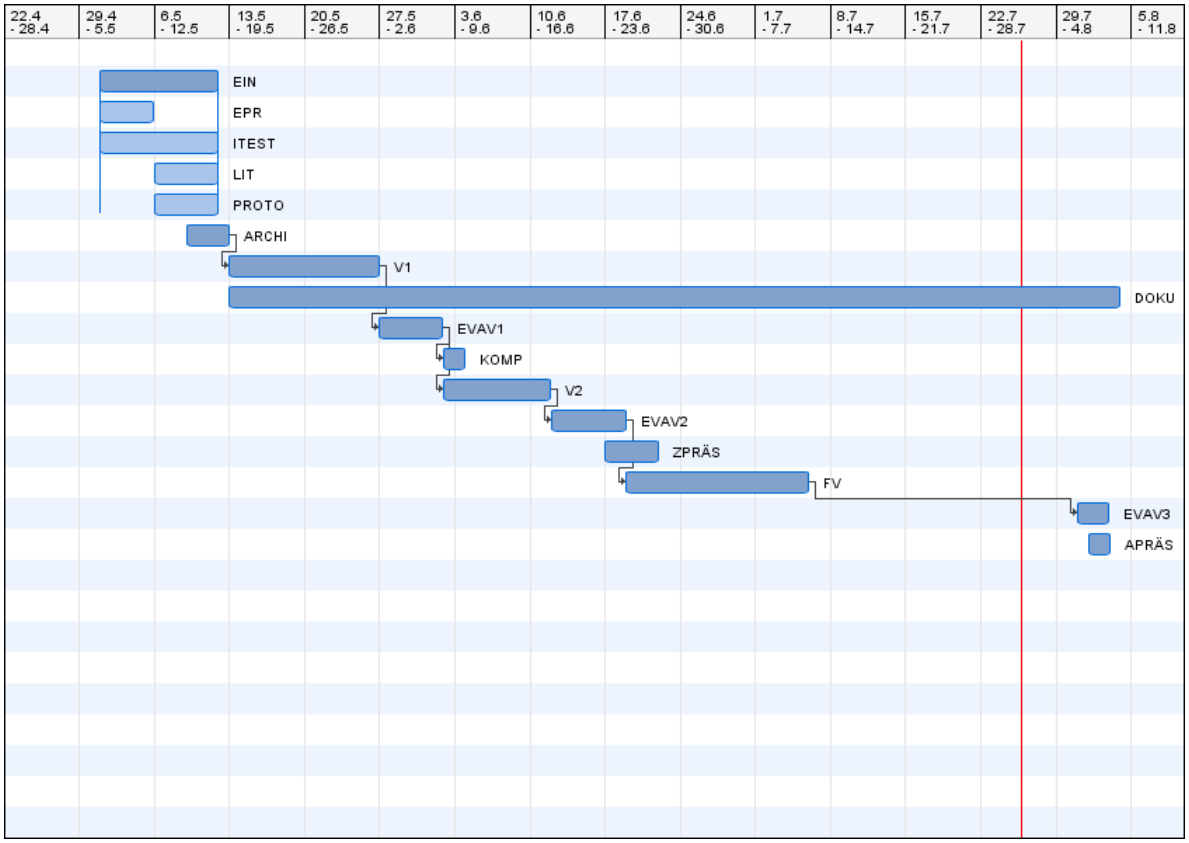

Abbildung 3.1: Gantt-Diagramm

Legende zum Gantt-Diagram

- **EIN**: Einarbeitung in das Thema
  - **EPR**: *Eclipse* Plug-in Recherche
  - **ITEST**: Einbindung des bestehenden Analysewerkzeugs der itestra GmbH in eigene Entwicklungsumgebung
  - **LIT**: Literaturrecherche
  - **PROTO**: Erstellung eines Labormusters nach [JL09]
- **ARCHI**: Architekturentwurf
- **V1**: Entwicklung der ersten Evolutionsstufe
- **DOKU**: Dokument der Bachelorarbeit
- **EVAV1**: Evaluation der ersten Evolutionsstufe

- **KOMP**: Planung der weiteren Komponenten
- **V2**: Entwicklung der zweiten Evolutionsstufe
- **EVAV2**: Evaluation der zweiten Evolutionsstufe
- **ZPRÄS**: Zwischenpräsentation
- **FV**: Entwicklung der finalen Version
- **EVAV3**: Evaluation der finalen Version
- **APRÄS**: Abschlusspräsentation

3.2.6 Retrospektive der Zeitplanung

Während der Bachelorarbeit ergaben sich insgesamt vier Verschiebungen / Änderungen in der Zeitplanung, die auch im Termin-Drift-Diagramm (siehe Abbildung 3.2) zu sehen sind.

1. **Verschiebung der Zwischenpräsentation**
   Die Zwischenpräsentation musste noch am Anfang der Bachelorarbeit zu einem späteren Zeitpunkt verschoben werden, weil kein früherer Termin für alle Beteiligten gefunden werden konnte. Dies hatte aber keinen Einfluss auf den weiteren Verlauf der Bachelorarbeit.
2. **Verzögerung der finalen Version**
   Während der Entwicklung der finalen Version trat ein Problem mit der verwendeten *Eclipse* Bibliothek, für das Zeichnen des Graphen, auf (siehe Abschnitt *Probleme während der Evolutionsstufe* in Kapitel 3.2.4). Dadurch verzögerte sich die Fertigstellung der finalen Version um ca. eine Woche.
3. **Verschiebung der Abschlusspräsentation und Abgabe des Dokuments**
   Aufgrund der Verzögerung bei der Entwicklung der finalen Version, musste die weitere Planung angepasst werden. Dadurch verschob sich der geplante Termin der Abschlusspräsentation und das geplante Ende der Bachelorarbeit.
4. **Verspätete Planung der abschließenden Evaluation**
   Erst bei Fertigstellung der finalen Version wurde bemerkt, dass keine Evaluation für diese Version in der Zeitplanung berücksichtigt wurde. Diese Evaluation war zwar vorgesehen, wurde aber nicht in der Zeitplanung festgehalten.

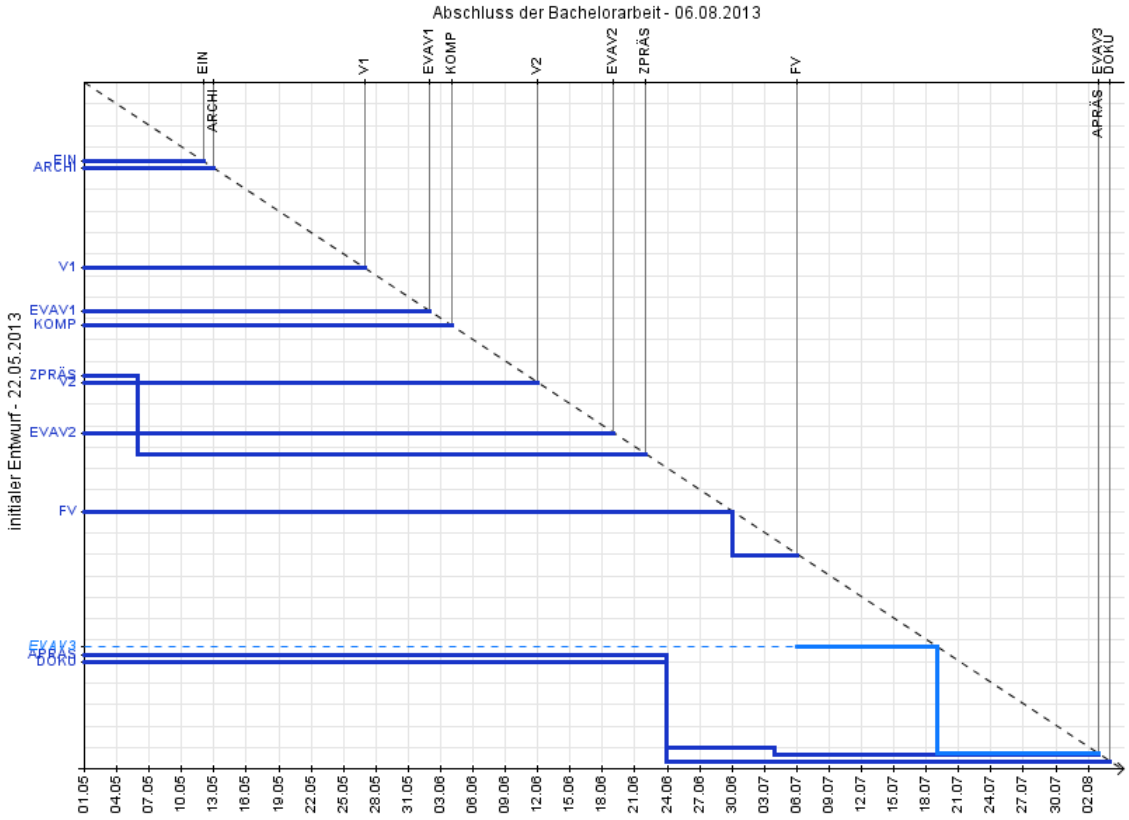

Abbildung 3.2: Termin-Drift-Diagramm

(Das Gantt-Diagramm und das Termin-Drift-Diagramm wurde mit dem Werkzug GTD-Manager[4] von der Universität Stuttgart erstellt.)

[4]http://www.iste.uni-stuttgart.de/en/se-old/werkzeuge/gtd-manager.html

Zeiterfassung

Das unten stehende Diagramm (siehe Abbildung 3.3) zeigt eine grobe Verteilung des Aufwands in Prozent auf folgende Tätigkeiten:

- **Einarbeitung**: *Eclipse* Plug-in Recherche, Einbindung des bestehenden Analysewerkzeugs der itestra GmbH, Literaturrecherche, Labormuster
- **Entwicklung**: Architekturentwurf, Evolutionsstufe 1, Evolutionsstufe 2, finale Version, Entwicklertests
- **Evaluation**: Vorbereitung, Durchführung und Dokumentation der Evaluation mit den Mitarbeitern der itestra GmbH
- **Dokument**: Erstellen des Dokuments der Bachelorarbeit
- **Präsentation**: Vorbereitung und Durchführung der Zwischen- und Abschlusspräsentation
- **Meetings**: Meetings mit den Betreuern

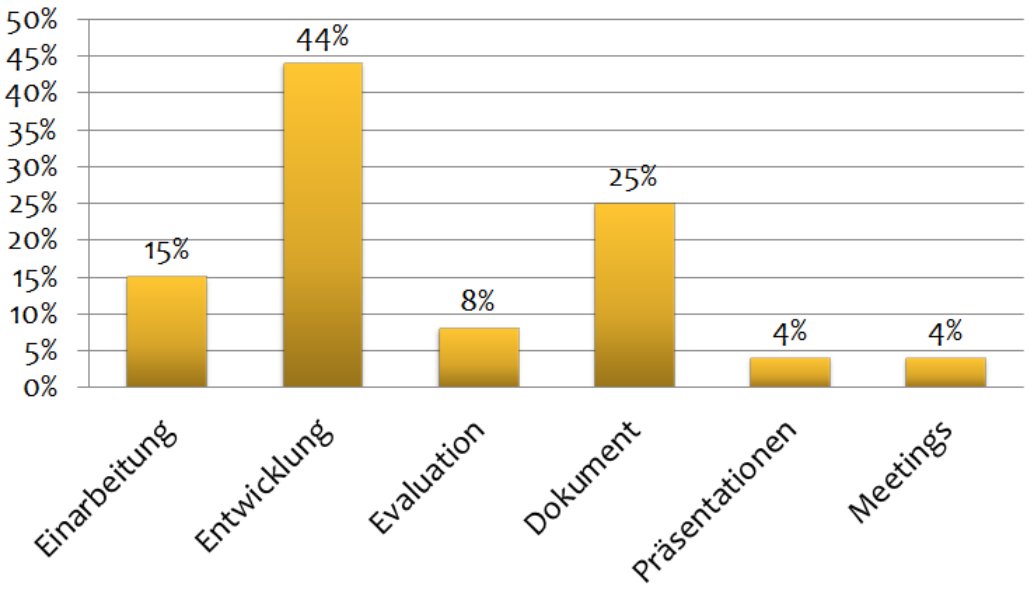

Abbildung 3.3: Zeiterfassung für Phasen

(Die Zeit wurde mit dem Werkzeug Toggl[5] erfasst und das Diagramm wurden mit Microsoft Excel 2010[6] erstellt.)

[5]https://www.toggl.com/
[6]http://office.microsoft.com/en-001/excel-help/getting-started-with-excel-2010-HA010370218.aspx

3.3 Architektur

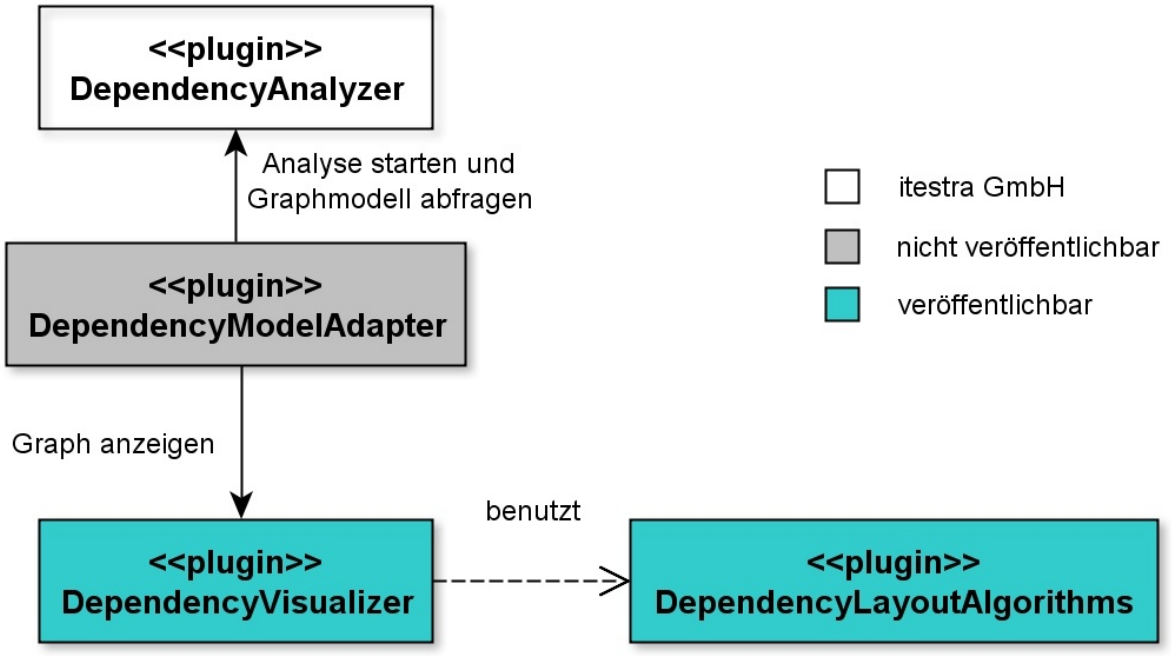

Abbildung 3.4: Plug-in Übersicht

Das entwickelte Werkzeug setzt sich aus drei Plug-ins zusammen: *DependenyVisualizer*, *DependencyLayoutAlgorithms* und *DependencyModelAdapter* (siehe Abbildung 3.4). Die eigentliche Visualisierung ist in dem Plug-in *DependenyVisualizer* implementiert. Dieses Plug-in verwendet das Plug-in *DependencyLayoutAlgorithms*, in dem die modifizierten Layout-Algorithmen von *Zest* zur Verfügung stehen. Diese beiden Plug-ins können veröffentlicht werden. Das dritte Plugin, *DependencyModelAdapter*, ist speziell für die itestra GmbH. Dort gibt es ein Konverter, der das Datenmodell des *DependencyAnalyzer* in das eigene Graphmodell umwandelt und eine *View*, mit der man die Analyse starten und das Datenmodell abfragen kann. Dieses Plug-in kann nicht veröffentlicht werden, da Komponenten des Analysewerkzeugs der itestra GmbH verwendet werden. Im folgenden werden die Architektur der einzelnen Plug-ins genauer erläutert.

3.3.1 DependencyVisualizer

Das Plug-in *DependencyVisualizer* enthält die eigentliche Visualisierung des Graphen und damit den größten Programmieraufwand der Bachelorarbeit. Das Plug-in ist in folgende Module aufgeteilt: editors, views, perspectives, handlers, model, util und res (siehe Abbildung 3.5).

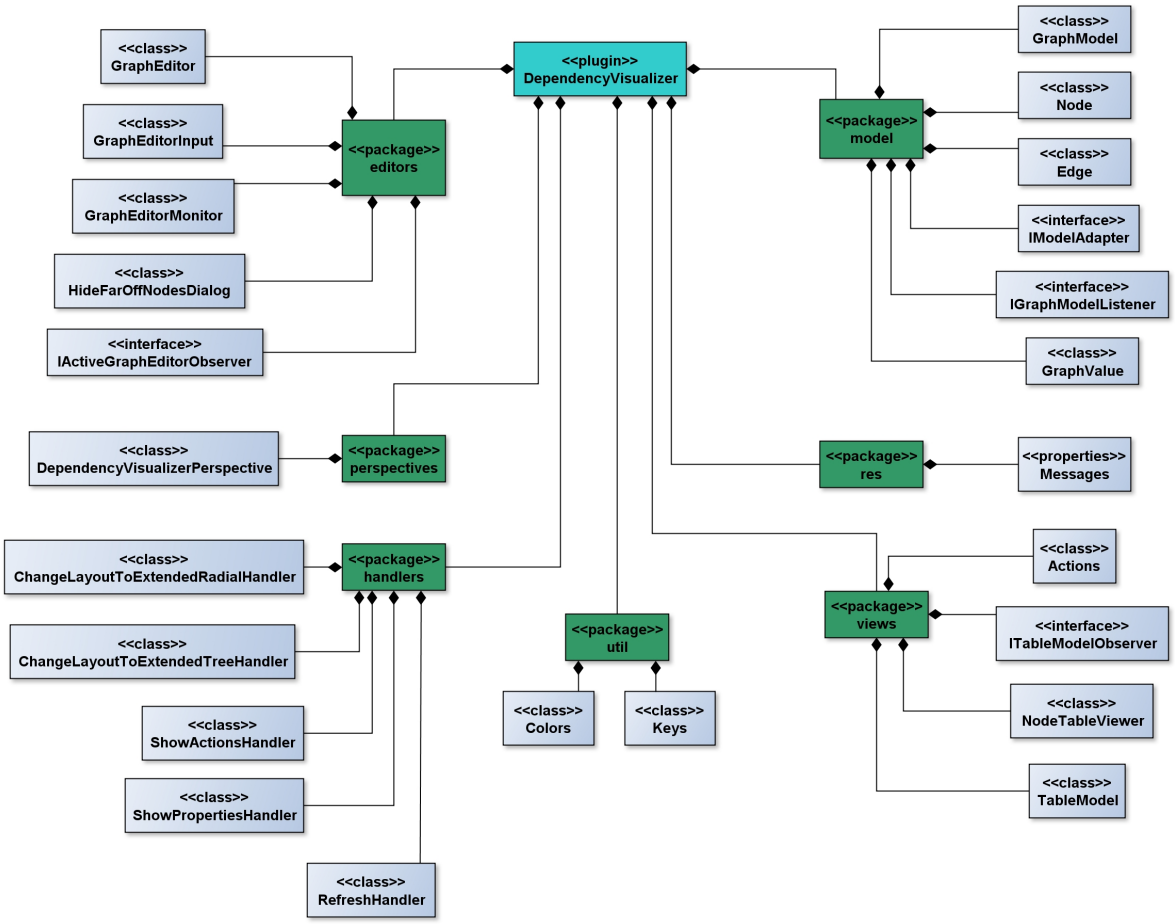

Abbildung 3.5: Komponentendiagramm DependencyVisualizer

Modul “editors“

Hauptbestandteil dieses Moduls ist die Klasse `GraphEditor`, die den *Eclipse Editor* implementiert, mit dem ein Graph visualisiert werden kann.

Ein *Editor* benötigt eine spezielle Input-Klasse (siehe Kapitel 2.1.1). Diese wird durch die Klasse `GraphEditorInput` realisiert. Dort ist hauptsächlich das Graphmodell gespeichert, aber auch Metadaten zu dem Graphen können dort hinterlegt werden.

Für die Erfassung und Überwachung des aktiven Grapheditors steht die Klasse `GraphEditorMonitor` über ein Singleton-Objekt zur Verfügung. Klassen, die das Interface `IActiveGraphEditorObserver` implementieren, können sich bei dem `GraphEditorMonitor` als *Observer* registrieren und werden dann benachrichtigt, falls der aktive Grapheditor wechselt oder sich ändert.

Die Klasse `HideFarOffNodesDialog` implementiert ein Dialogfenster, das für die Funktion "Ausblenden aller Knoten, die weiter als n Knoten entfernt sind" (siehe Anforderung 27 in Kapitel 3.1) benötigt wird. Dort kann der Benutzer angeben, wie viele Knoten ein Knoten entfernt sein muss, damit er ausgeblendet wird.

Modul "views"

Dieses Modul enthält die Klasse `Actions`, die eine *Eclipse View* implementiert. Diese *View* stellt alle Graphtransformationen zur Verfügung, die eine Eingabemaske benötigen:

- das Filtern von Knoten,
- das Suchen von Verbindungswegen zwischen zwei Knoten und
- das Suchen von Knoten anhand ihres Namens.

Alle *Views* sollten in diesem Modul implementiert werden, allerdings sind für die Bachelorarbeit keine weiteren geplant.

Modul "perspectives"

Dieses Modul enthält die Klasse `DependencyVisualizerPerspective`, die eine *Perspective* implementiert. Diese *Perspective* stellt die Standardanordnung der *Views* bereit, die in diesem Plug-in verwendet werden: die *View* Actions und die in *Eclipse* integrierte *View* PropertySheet.

Modul "handlers"

In diesem Modul sind die Handler-Klassen für die verschiedenen DependencyVisualizer Menüeinträge implementiert:

- **ChangeLayoutToExtendedRadialHandler**: aktualisiert den Graphen mit dem ExtendedRadialLayout
- **ChangeLayoutToExtendedTreeHandler**: aktualisiert den Graphen mit dem ExtendedTreeLayout
- **RefreshHandler**: aktualisiert den Graphen mit dem aktuellen Layout
- **ShowActionsHandler**: öffnet die *View* Actions
- **ShowActionsHandler**: öffnet die *View* PropertySheet

Modul “model“

Dieses Modul enthält alle Datenklassen für das eigene Graphmodell.

Die Klasse `GraphModel` repräsentiert das gesamte Modell für einen Graphen mit allen Knoten und Kanten. Es wird in der Klasse `GraphEditorInput` an den Grapheditoren übergeben. In dieser Klasse wird das Filtern von Knoten verwaltet, indem die gefilterten Knoten in separaten Listen gespeichert werden. Klassen, die das Interface `IGraphModelListener` implementieren, werden bei Änderung im Modell entsprechend benachrichtigt. Die Klasse `GraphEditor` implementiert dieses Interface.

Die Klasse `Node` repräsentiert einen Knoten im Graphen. Jeder Knoten hat eine ID, einen Namen und ein boolean-Wert, der ihn als Hauptknoten identifiziert. Hauptknoten werden im Graph hervorgehoben dargestellt und können zum Beispiel die main-Methoden von Programmen sein. Des Weiteren kann ein Knoten noch beliebige Metadaten enthalten.

Um die Zusatzinformationen eines Knoten in der *View* PropertySheet anzuzeigen, muss die Datenklasse des Knoten das Interface `IAdaptable` implementieren. Dazu wird die Klasse `NodePropertySource` benötigt. Sie implementiert das Interface `IPropertySource` und formatiert die Informationen des Knoten entsprechend für die *View* PropertySheet.

Die Klasse `Edge` repräsentiert eine Kante im Graphen. Jede Kante hat eine ID, eine Beschriftung und speichert den zugehörigen Ursprungs- und Zielknoten. Zusätzlich können noch beliebige Metadaten gespeichert werden.

Das Interface `IModelAdapter` soll von allen Konvertern implementiert werden, die ein externes Datenmodell in das eigene Graphmodell umwandeln. Zweck ist es, eine einheitliche Schnittstelle zu gewährleisten.

Die Klasse `GraphValue` repräsentiert ein einfaches Tupel, das eine Liste von Knoten und eine Liste von Kanten enthält. Diese Datenklasse wird genutzt, um Änderungen im Graphmodell zu kommunizieren, ohne das gesamte Graphmodell zu verschicken.

Modul “util“

Dieses Modul enthält verschiedene Hilfsklassen und Konstanten.

Die Klasse `NodeTableViewer` erweitert die Klasse `TableViewer` von *JFace*, um in einer Tabelle Knotenobjekte darzustellen. Unter anderem wird die Spaltenbreite automatisch an den Namen des Knoten angepasst. Als Datenmodell für die Tabelle wird ein Objekt der Klasse `TableModel` verwendet. Um bei Änderungen im Datenmodell benachrichtigt zu werden implementiert die Klasse `NodeTableViewer` das Interface `ITableModelObserver`.

Die Klasse `Colors` enthält vordefinierte Objekte der Klasse `Color`, die in der ganzen Anwendung verwendet werden können.

Die Klasse `Keys` enthält die IDs von bestimmten Tasten der Tastatur.

Modul “res“

Dieses Modul enthält Icons für verschiedene Funktionen des Werkzeugs und die Properties-Datei `Messages` für die Internationalisierung der Oberfläche. Aktuell wird nur die Sprache Englisch unterstützt.

3.3.2 DependencyLayoutAlgorithms

Das Plug-in *DependencyLayoutAlgorithms* enthält die verschiedenen Layoutalgorithmen, mit denen die Knoten des Graphen automatisch angeordnet werden können (siehe Abbildung 3.6).

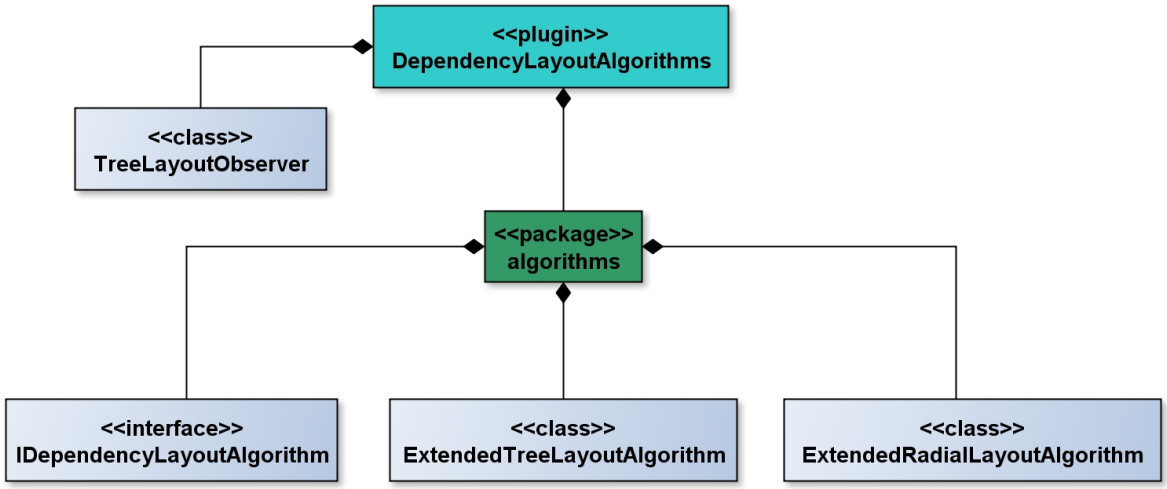

Abbildung 3.6: Komponentendiagramm DependencyLayoutAlgorithms

Jeder Layoutalgorithmus muss das Interface `IDependencyLayoutAlgorithm` implementieren. Damit wird sichergestellt, dass das Layout im Vorhinein berechnet werden kann und erst per expliziten Aufruf auf den Graphen angewendet wird. Alle Layoutalgorithmen basieren auf denen von *Zest*, skalieren den Graphen jedoch nicht auf das aktuelle Fenster.

Aktuell sind zwei Layoutalgorithmen implementiert:

- **ExtendedRadialLayoutAlgorithm** versucht die Knoten des Graphen radial anzuordnen und ist daher nicht für Multigraphen geeignet.
- **ExtendedTreeLayoutAlgorithm** ist der standardmäßige Layoutalgorithmus und versucht die Knoten des Graphen in einer hierarchischen Baumstruktur anzuordnen und ist auch für Multigraphen geeignet.

Zusätzlich enthält das Plug-in die Klasse `TreeLayoutObserver`. Sie ist eigentlich Teil der *Zest* Bibliothek und musste wegen eines Codierfehlers verändert werden (siehe Abschnitt *Probleme während der Evolutionsstufe* in Kapitel 3.2.4). Sobald der Fehler offiziell von *Eclipse* behoben wurde, kann diese Klasse entfernt und in der Klasse ExtendedTreeLayoutAlgorithm wieder die Klasse `TreeLayoutObserver` der *Zest* Bibliothek verwendet werden.

3.3.3 DependencyModelAdapter

Das Plug-in *DependencyVisualizer* enthält die Anbindung an das Analysewerkzeug der itestra GmbH. Das Plug-in ist in folgende Module aufgeteilt: main, views, handlers, und res (siehe Abbildung 3.7).

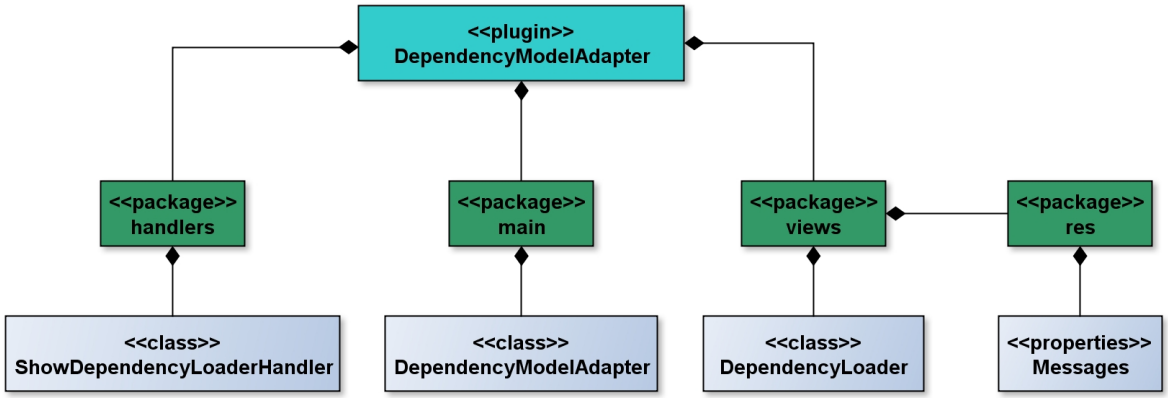

Abbildung 3.7: Komponentendiagramm DependencyModelAdapter

Modul “main“

Dieses Modul enthält die Klasse `DependencyModelAdapter`, welche die Umwandlung des Datenmodells des Analysewerkzeugs von der itestra GmbH in das eigene Graphmodell realisiert. Sie implementiert das Interface `IModelAdapter`. Die Klasse `Declaration` wird auf die Klasse `Node` und die Klasse `Reference` auf die Klasse `Edge` abgebildet.

Modul “views“

Dieses Modul enthält die Klasse `DependencyLoader`, die eine *View* implementiert, mit der das Analysewerkzeug gestartet werden kann und anschließend der resultierende Graph mit der Klasse `GraphEditor` dargestellt wird. Zum Start des Analysewerkzeugs müssen die entsprechenden Kommandozeilenparameter angegeben werden. Die bisher angegebenen Kommandozeilenparameter werden persistent im Plug-in gespeichert.

Modul “handlers“

In diesem Modul ist die Handler-Klasse für die Erweiterung im DependencyVisualizer Menü implementiert: `ShowDependencyLoaderHandler`. Sie öffnet die *View* DependencyLoader.

Modul “res“

Dieses Modul enthält die Properties-Datei `Messages` für die Internationalisierung der Oberfläche. Aktuell wird nur die Sprache Englisch unterstützt.

(Alle Komponentendiagramme wurden mit dem Werkzeug yEd[7] erstellt.)

[7]http://www.yworks.com/en/products_ yed_ about.html

4 Evaluation

In diesem Kapitel wird die Vorbereitung, Durchführung und Ergebnisse der abschließenden Evaluation beschrieben. Ziel dieser abschließenden Evaluation war es:

- die bestehenden Funktionen zu evaluieren,
- die Usability des entwickelnden Werkzeugs zu überprüfen und
- fehlende oder wünschenswerte Funktionen zu identifizieren.

4.1 Fragenkatalog

In der Vorbereitung auf die abschließende Evaluation wurden folgende Fragen identifiziert, mit Hilfe denen die oben genannten Ziele erreicht werden sollten. Die Fragen sind durchgehend nummeriert, damit die spätere Zuordnung einfacher ist.

Allgemeine Fragen zum Werkzeug

1. Bringt das Werkzeug einen Vorteil für die tägliche Arbeit?
2. Werden Sie das Werkzeug regelmäßig einsetzen?
3. Werden Sie das Werkzeug Ihren Kollegen weiterempfehlen?
4. Welche Funktionen fehlen Ihnen? Was würden Sie noch hilfreich finden?

Fragen zur Erzeugung des Graphen

5. Ist das Erzeugen des Abhängigkeitsgraphen über die *View* DependencyLoader eine Funktion, die Sie in Ihrer täglichen Arbeit benötigen?
6. Ist die Bedienung der *View* DependencyLoader verständlich und nachvollziehbar?
7. Sind die Oberflächenelemente der *View* DependencyLoader verständlich angeordnet?
8. Sind Sie insgesamt mit dieser Funktion zufrieden? Wenn nein, warum nicht?

Fragen zur Darstellung des Graphen

9. Finden Sie sich im Graphen durch die Standardanordnung der Knoten und Kanten zurecht?
10. Lässt sich der Graph wie erwartet verschieben?
11. Lässt sich der Graph wie erwartet in verschiedenen Vergrößerungen anzeigen?
12. Ist die Anzeige der Zusatzinformationen zu Knoten leicht zu finden? Haben Sie beide Möglichkeiten gefunden?
13. Sind die Zusatzinformationen an der richtigen Position?
14. Sind Sie insgesamt mit der Anzeige der Zusatzinformationen zufrieden? Wenn nein, warum nicht?
15. Funktioniert das Neuanordnen des Graphen mit einem anderen Layout-Algorithmus wie erwartet?
16. Funktioniert die Anzeige von mehreren Graphen und verhält sich die Oberfläche konsistent?

Fragen zum Markieren von erreichbaren Knoten

17. Ist das Markieren von erreichbaren Knoten eine Funktion, die Sie in Ihrer täglichen Arbeit benötigen?
18. Wie häufig wird diese Funktion verwendet werden?
19. Ist die Bedienung des Markierens von erreichbaren Knoten verständlich und nachvollziehbar?
20. Trifft das Resultat des Markierens Ihre Erwartungen? Wenn nein, warum nicht?
21. Sind Sie insgesamt mit dieser Funktion zufrieden? Wenn nein, warum nicht?

Fragen zum Suchen im Graphen

22. Ist die Suche im Graphen eine Funktion, die Sie in Ihrer täglichen Arbeit benötigen?
23. Wie häufig wird diese Funktion verwendet werden?
24. Ist die Bedienung der Suche verständlich und nachvollziehbar?
25. Trifft das Resultat der Suche Ihre Erwartungen? Wenn nein, warum nicht?
26. Ist das Verhalten der Suche nachvollziehbar, wenn es mehrere Suchtreffer gibt?
27. Sind Sie insgesamt mit der Such-Funktion zufrieden? Wenn nein, warum nicht?

Fragen zum Filtern von Knoten

28. Ist das Filtern von Knoten eine Funktion, die Sie in Ihrer täglichen Arbeit benötigen?
29. Wie häufig wird diese Funktion verwendet werden?
30. Ist die Bedienung des Filtern verständlich und nachvollziehbar?
31. Es gibt zwei Möglichkeiten Knoten zu filtern. Haben Sie beide gefunden?
32. Trifft das Resultat des Filterns Ihre Erwartung? Wenn nein, warum nicht?
33. Sind Sie insgesamt mit der Filter-Funktion zufrieden? Wenn nein, warum nicht?

Fragen zum Finden von Verbindungswegen zwischen Knoten

34. Ist das Finden von Verbindungswegen zwischen Knoten eine Funktion, die Sie in Ihrer täglichen Arbeit benötigen?
35. Wie häufig wird diese Funktion verwendet werden?
36. Ist die Bedienung dieser Funktion verständlich und nachvollziehbar?
37. Trifft das Resultat dieser Funktion Ihre Erwartung? Wenn nein, warum nicht?
38. Sind Sie insgesamt mit dieser Funktion zufrieden? Wenn nein, warum nicht?

4.2 Durchführung

Zur Durchführung der abschließenden Evaluation standen drei Mitarbeiter der itestra GmbH zu Verfügung, die nacheinander befragt wurden. Die entwickelnden Plug-ins waren schon in den entsprechenden Entwicklungsumgebungen installiert und einsatzbereit.

Zu Beginn wurde die allgemeine Aufteilung der einzelnen Oberflächenelemente erläutert. Im Weiteren wurde immer eine Funktion des Werkzeugs kurz erklärt, wobei die Erklärung abhängig von den schon bestehenden Erfahrungen des Probanden mit dem Werkzeug unterschiedlich viel Zeit beanspruchte. Nach der Erklärung versuchten die Probanden die Funktion praktisch einzusetzen, indem sie mit realen Datensätzen das Werkzeug ausprobierten. Währenddessen wurden die zu der Funktion passenden, vorbereiteten Fragen gestellt und über die Umsetzung diskutiert.

Insgesamt dauerte die Befragung der Mitarbeiter der itestra GmbH etwas mehr als drei Stunden. Die Ergebnisse wurden schriftlich notiert und im nächsten Kapitel zusammengefasst.

4.3 Ergebnisse

Die Ergebnisse der abschließenden Evaluation werden in diesem Kapitel nach den Funktionen gruppiert und zusammengefasst dargestellt.

Werkzeug allgemein

Alle Probanden sehen in dem Werkzeug einen Vorteil für ihre tägliche Arbeit und werden es ihren Kollegen weiterempfehlen. Im Vergleich zu dem Grapheditor yEd (siehe Kapitel 2.2.5) bietet es zwar nicht so viele Möglichkeiten, da es aber in der Entwicklungsumgebung *Eclipse* integriert ist, können Analysen schneller und effizienter durchgeführt werden. Die Probanden können sich gut eine selektive Benutzung vorstellen: für den schnellen Überblick nehmen sie das hier entwickelte Werkzeug, für sehr detaillierte Analysen würden sie den Grapheditor yEd verwenden. Wenn dann alle neuen Anforderungen eingearbeitet würden, wäre auch die alleinige Verwendung des Werkzeugs möglich.

Erzeugung des Graphen

Das Erzeugen des Graphen über die *View* DependencyLoader wird als nützliche Funktion angesehen. Die Bedienung und Anordnung der Oberflächenelemente ist übersichtlich und entspricht den Erwartungen. Besonders die Historie der eingegebenen Kommandozeilenparameter und die Anzeige von Fehlermeldungen ist positiv. Es wurden aber noch sinnvolle Erweiterungen des DependencyLoaders erwähnt. Ein Mitarbeiter führt die aktuellen Analysen über ein vorgefertigtes Batch-Skript aus und würde gerne die Aufrufe direkt in das Textfeld des DependecyLoaders übernehmen. Dazu wäre es nötig ein mehrzeiliges Textfeld zu verwenden und Platzhalter im Text durch Umgebungsvariablen zu ersetzen. Des Weiteren wurde ein Oberflächenelement zum Abbrechen der gestarteten Analyse gewünscht. Grundsätzlich würde eine Oberfläche bevorzugt werden, mit der man sich die Kommandozeilenaparameter des Analysewerkzeugs zusammenstellen kann. Das aber ist Teil einer anderen Bachelorarbeit.

Darstellung des Graphen

Mit dem standardmäßigen Layout des Graphen waren die Probanden zufrieden und dieses Layout wird als einzig wirklich sinnvolles und am natürlichsten angesehen. Das radiale Layout wurde unterschiedlich bewertet: ein Proband empfand dieses Layout als besser navigierbar, die anderen würden nur das Standardlayout verwenden. Zum Ändern des Layouts würden sich die Probanden eine Toolbar in *Eclipse* wünschen, da die Bedienung über das Menü ihnen zu umständlich ist. Das Verschieben und Vergrößern des Graphen wurde als intuitiv empfunden. Mit der Anzeige der Zusatzinformationen über Mouse-Over und der *View* PropertySheet waren die Probanden sehr zufrieden, wobei sie noch Ideen für weitere Informationen hatten, die dort hilfreich wären.

Markieren von erreichbaren Knoten

Die Funktion zum Markieren von erreichbaren Knoten wurde unterschiedlich bewertet. Zum einen wird sie als sehr nützlich bewertet, zum anderen würde man eher die nicht erreichbaren Konten filtern wollen. Generell ist aber klar was die Funktion liefert und wie sie funktioniert. Es wäre wünschenswert, dass die markierten Knoten direkt selektiert wären und auch das Markieren der Rückrichtung zur Verfügung stehen würde.

Suchen im Graphen

Das Suchen im Graphen wurde als die am häufigsten verwendete Funktion beschrieben. Besonders das Springen zum gesuchten Knoten im Graphen wurde als positiv angesehen. Allerdings wäre es wünschenswert, wenn man nicht den vollständigen Namen eingeben müsste, sondern ein Matching mit Trefferliste vorhanden wäre.

Filtern von Knoten

Das Filtern von Knoten ist für die Mitarbeiter der itestra GmbH eine sehr wichtige Funktion, die ebenfalls oft verwendet werden wird. Die Geschwindigkeit der Funktion wurde gelobt. Die Bedienung ist einfach und klar. Allerdings wäre auch hier bei der Eingabe des Knotennamens ein Matching mit Trefferliste wünschenswert. Bei dem Filtern weit entfernter Knoten ausgehend von einem bestimmten Knoten wäre eine optionale Einstellung wünschenswert, mit der man einfach alle nicht erreichbaren Knoten filtern könnte.

Suchen von Verbindungswegen zwischen Knoten

Das Suchen von Verbindungswegen zwischen Knoten ist eine sehr spezifische Funktion und wird nicht von allen Mitarbeitern der itestra GmbH gleich häufig benutzt. Diese Funktion ist vor allem für die Fehlerbehebung sehr wichtig und ist für die Mitarbeiter, die in diesem Bereich arbeiten, sehr nützlich. Für den erfolgreichen Einsatz dieser Funktion, wäre es aber wichtig, dass nicht nur der kürzeste Pfad angezeigt wird, sondern alle existierenden Pfade abgefragt werden könnten.

5 Benutzerhandbuch

5.1 Installationsanleitung

5.1.1 Einbinden der Plugins

Um den DependencyVisualizer in *Eclipse* verwenden zu können, müssen folgende zwei Plug-ins in *Eclipse* eingebunden werden:

- de.nolleryc.dependencyvisualizer
- de.nolleryc.dependencylayoutalgorithms

Dazu müssen die Dateien `de.nolleryc.dependencyvisualizer_1.0.0.jar` und `de.nolleryc.dependencylayoutalgorithms_1.0.0.jar` in den Ordner "eclipse/plugins" kopiert und *Eclipse* neu gestartet werden.

5.1.2 Abhängigkeiten zu anderen Plug-ins

Um beide Plug-ins benutzen zu können, muss das Plug-in *Graphical Editing Framework Zest Visualization Toolkit SDK* eingebunden werden. Dieses Plug-in kann über folgende Update-Site bezogen werden:

`https://hudson.eclipse.org/hudson/job/gef4-master/lastSuccessfulBuild/artifact/update-site/`

In *Eclipse* installiert man neue Software über das Menü Help -> Install New Software... (siehe Abbildung 5.1).

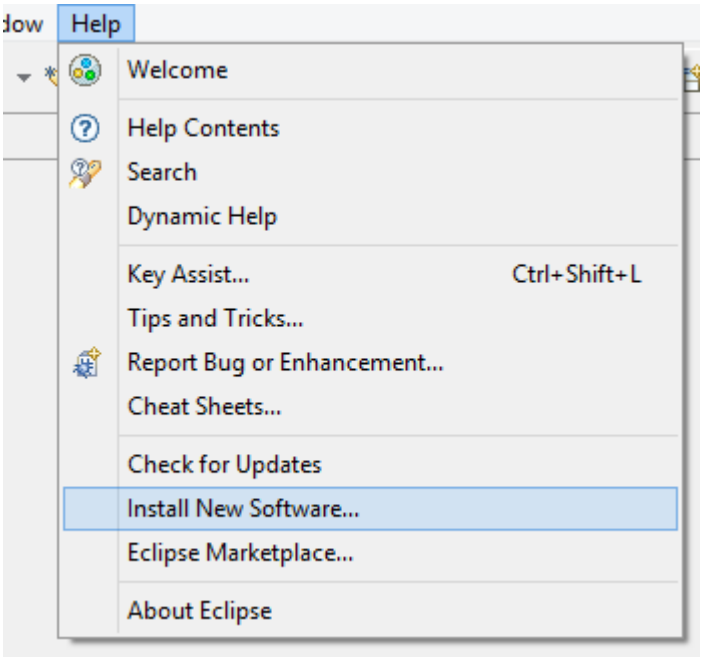

Abbildung 5.1: Eclipse Plug-in installieren

Anschließend muss der oben stehende Link unter “Work with“ eingegeben werden. In der erscheinenden Liste wählt man das Plug-in *Graphical Editing Framework Zest Visualization Toolkit SDK* aus und drückt den “Next“-Button (siehe Abbildung 5.2). Danach folgt man den Installationsanweisungen von *Eclipse*.

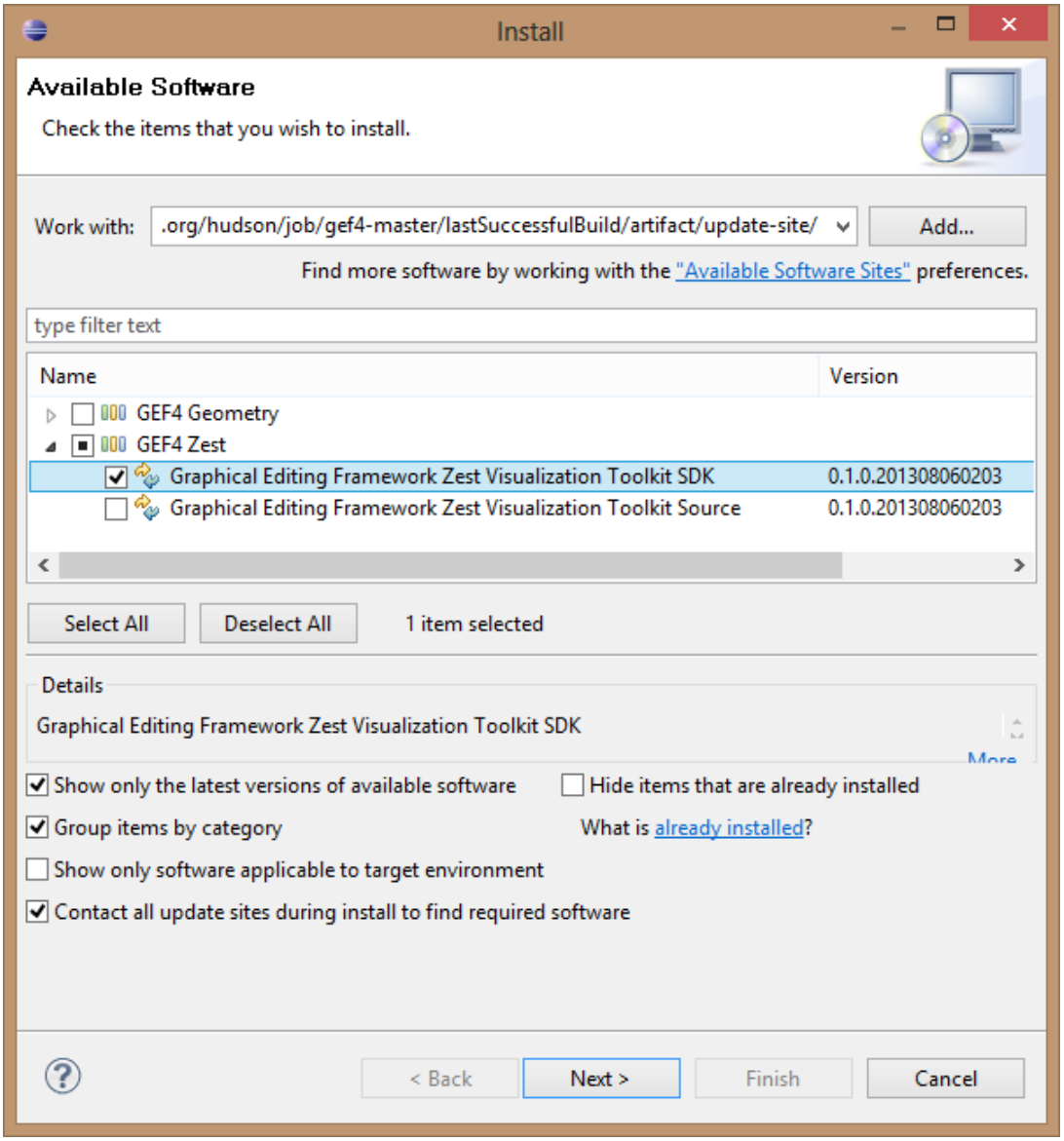

Abbildung 5.2: Zest Plug-in installieren

Des Weiteren ist das Werkzeug von folgenden *Eclipse* Plug-ins abhängig, die allerdings schon in *Eclipse* eingebunden sind:

- org.eclipse.ui
- org.eclipse.core.runtime
- org.apache.log4j
- org.eclipse.draw2d

5.2 Laden der Perspective DependencyVisualizer

Das Plug-in DependencyVisualizer bietet eine eigene *Perspective* an, damit die *Views* passend zur Anzeige des Graphen angeordnet werden können. Um diese *Persepective* zu laden, navigiert man zuerst zum Menü Window -> Open Perspective -> Other... (siehe Abbildung 5.3). Anschließend öffnet sich ein Fenster mit allen verfügbaren *Perspectives*. Dort wählt man die *Perspective* DependencyVisualizer (siehe Abbildung 5.4).

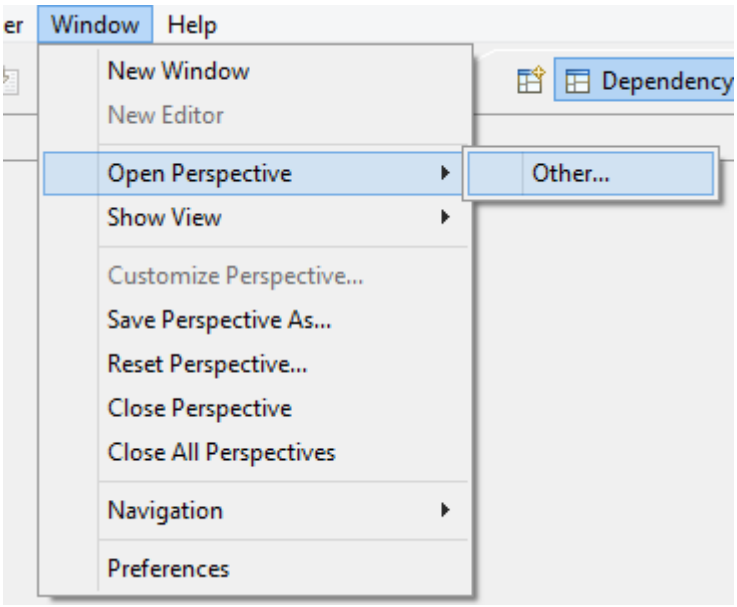

Abbildung 5.3: Laden der Perspective DependencyVisualizer (1)

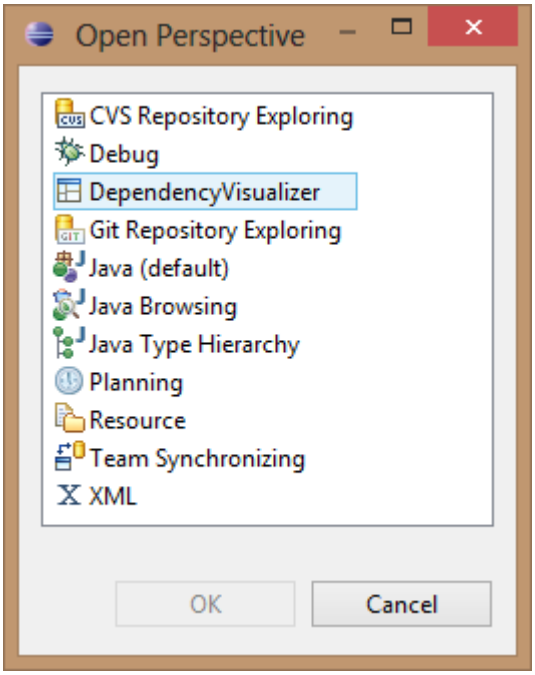

Abbildung 5.4: Laden der Perspective DependencyVisualizer (2)

5.3 Laden eines Graphen

Um einen Graphen zu laden, muss in *Eclipse* ein neuer *Editor* geöffnet werden. Dazu bietet die Klasse `IWorkbenchPage` die Methode

```
openEditor(IEditorInput input, String editorId)
```

mit den folgenden Parametern:

- **input**: Objekt der Klasse `GraphEditorInput`
- **editorId**: "de.nolleryc.dependencyvisualizer.editors.grapheditor" oder `GraphEditor.ID`

Das übergebene Objekt der Klasse `GraphEditorInput` enthält das Graphmodell, den Namen der Visualisierung und benutzerdefinierbare Metadaten.

5.4 Graph verschieben

Um den Graphen im *Editor* zu verschieben, gibt es zum einen die horizontale und vertikale Scrollbar. Zum anderen kann man auch auf eine leere Fläche im *Editor* klicken und der Mauszeiger verändert sich zu einem Kreuz (siehe Abbildung 5.5). Solange die Maustaste gedrückt bleibt, kann man mit der Mausbewegung den Graphen verschieben.

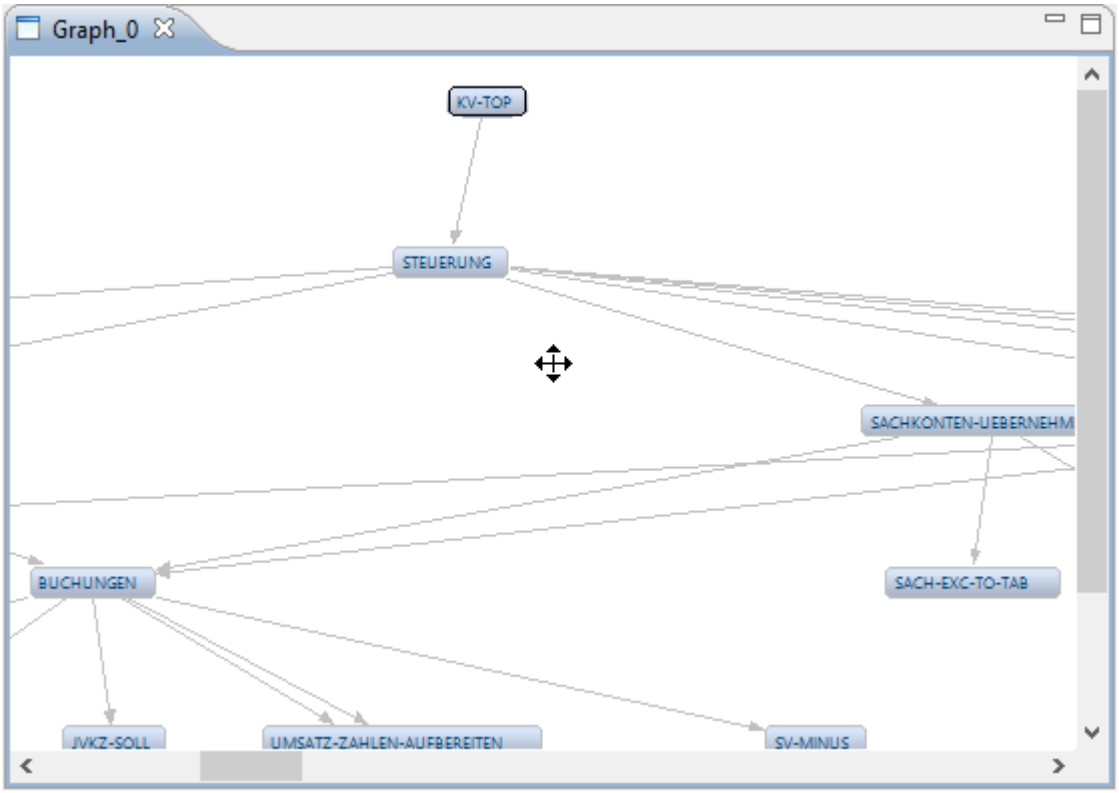

Abbildung 5.5: Graph verschieben

5.5 Graph vergrößern

Um den Graphen in verschiedenen Vergrößerungen anzuzeigen, kann die Kombination von der STRG-Taste und dem Mausrad verwendet werden. Der Benutzer hat die Möglichkeit zwischen acht Vergrößerungsstufen zu wählen (siehe Abbildung 5.6).

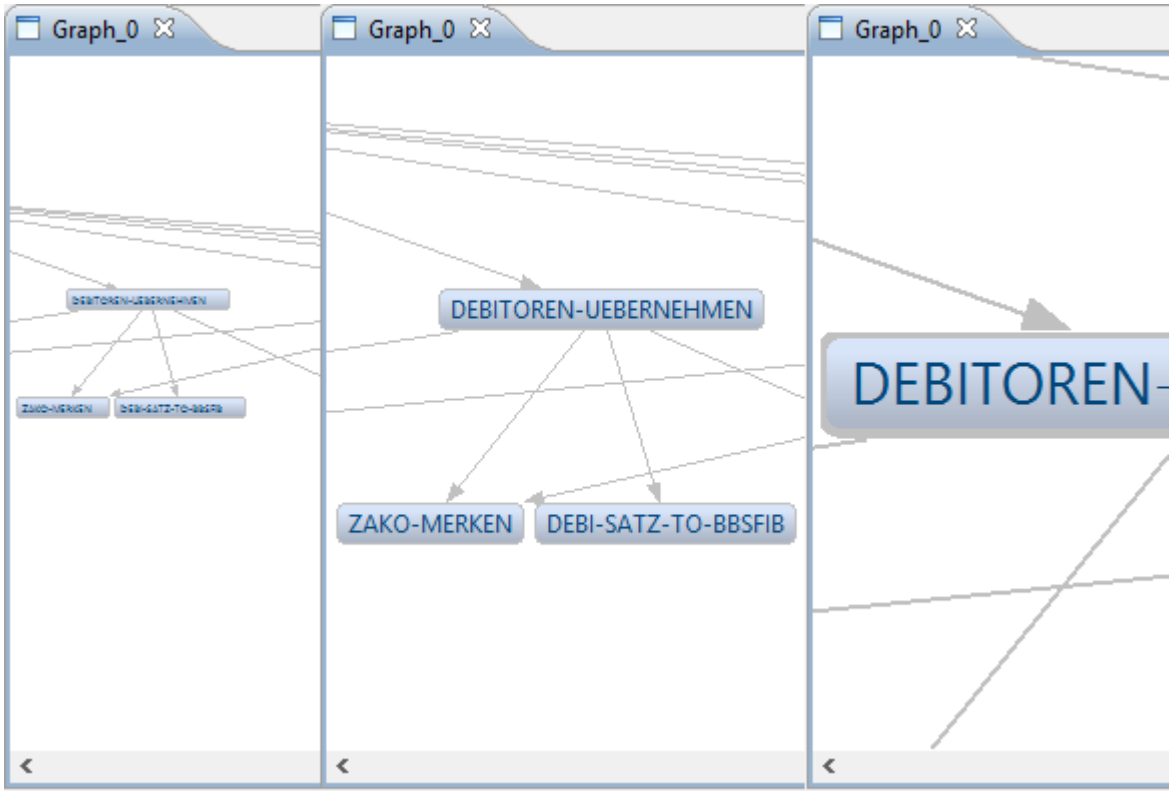

Abbildung 5.6: Graph vergrößern

5.6 Layout ändern

Das Layout des Graphen kann über das Menü DependencyVisualizer -> Change Layout verändert werden (siehe Abbildung 5.7). Es stehen zwei Layouts zu Verfügung:

- **ExtendedTreeLayout**: ordnet die Knoten in einer hierarchischen Baumstruktur an
- **ExtendedRadialLayout**: ordnet die Knoten radial zu den Hauptknoten an (nicht für Multigraphen geeignet)

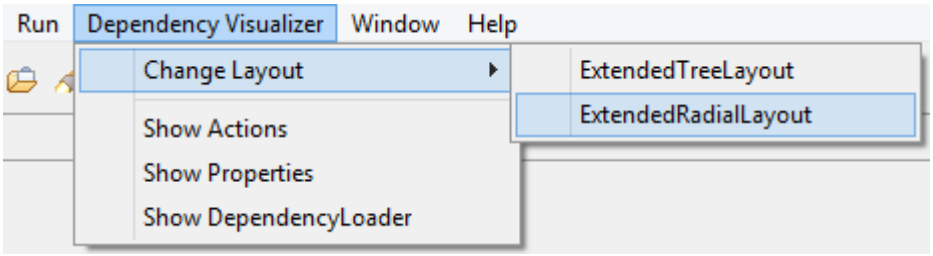

Abbildung 5.7: Layout ändern

5.7 Öffnen der View PropertySheet

Die *View* PropertySheet von *Eclipse* kann über das Menü DependencyVisualizer -> Show Properties geladen werden (siehe Abbildung 5.8).

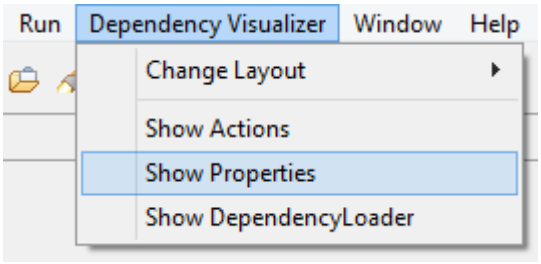

Abbildung 5.8: Öffnen der View PropertySheet

5.8 Anzeige von Zusatzinformationen

Zu jedem Knoten im Graphen sind Zusatzinformationen wie die ID, der Name, etc. gespeichert. Diese Informationen sind über zwei Wege verfügbar: per Mouse-Over und über die *View* PropertySheet (siehe Abbildung 5.9).

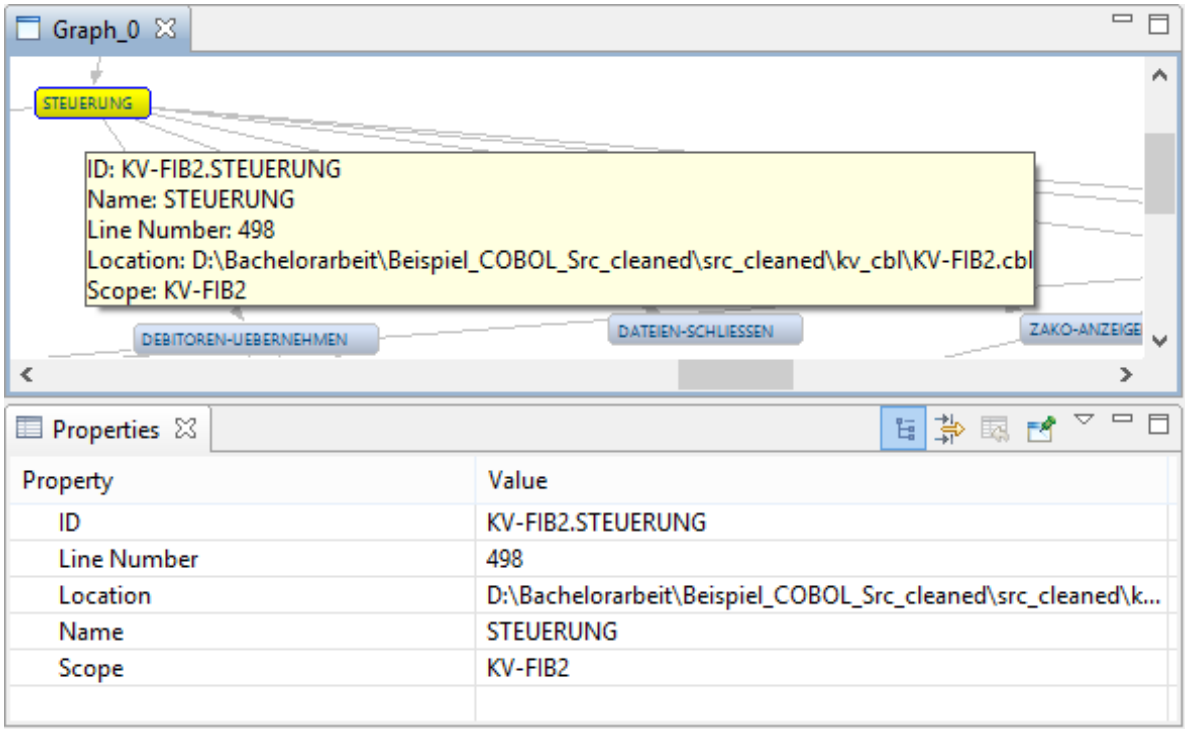

Abbildung 5.9: Anzeige von Zusatzinformationen

5.9 Markieren der erreichbaren Knoten ausgehend von einem bestimmten Knoten

Das Plug-in DependencyVisualizer bietet die Möglichkeit alle erreichbaren Knoten ausgehend von einem bestimmten Knoten zu markieren. Dazu klickt man mit der rechten Maustaste auf einen Knoten und wählt im Kontextmenü den Menüpunkt "Highlight Connected Nodes" (siehe Abbildung 5.10). Anschließend werden alle erreichbaren Knoten und die dazugehörigen Kanten farblich hervorgehoben (siehe Abbildung 5.11).

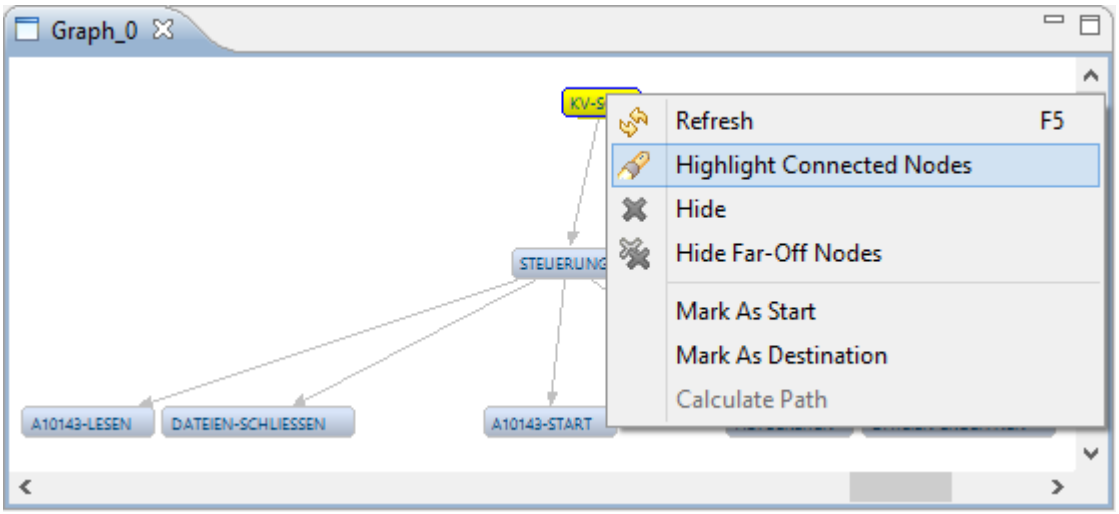

Abbildung 5.10: Markieren der erreichbaren Knoten

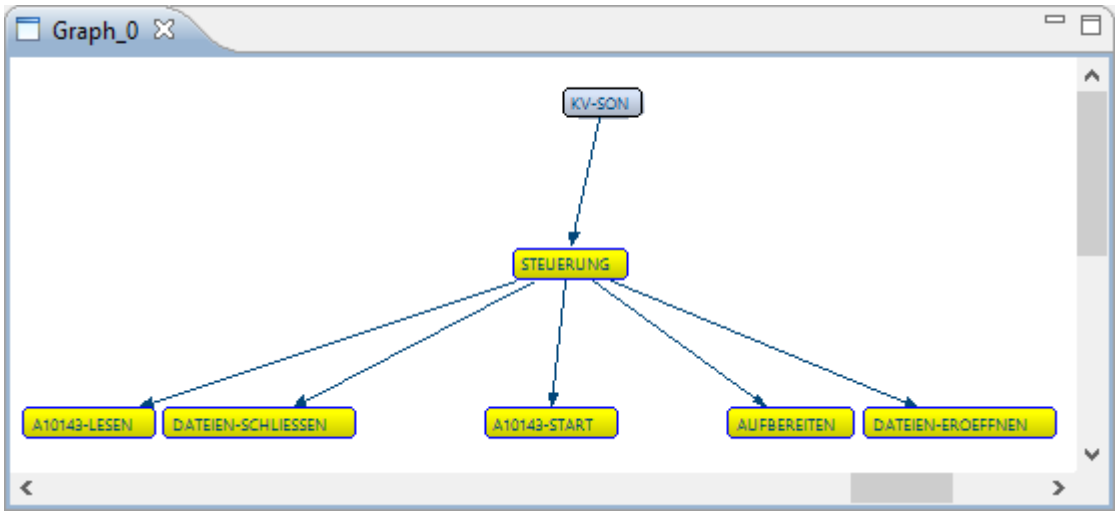

Abbildung 5.11: Markieren der erreichbaren Knoten (Resultat)

5.10 Öffnen der View Actions

Die *View* Actions kann über das Menü DependencyVisualizer -> Show Actions geladen werden (siehe Abbildung 5.12).

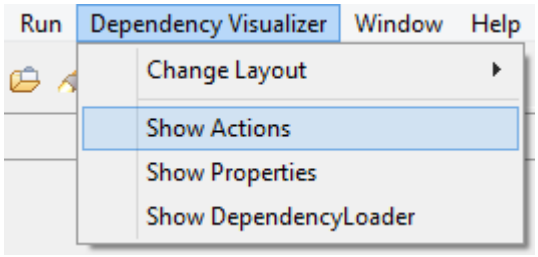

Abbildung 5.12: Öffnen der View Actions

5.11 Suchen von Knoten

Über die *View* Actions können Knoten über ihren Namen gesucht werden. Dazu muss die *View* geöffnet und die Rubrik "Search" expandiert sein. Anschließend kann der exakte Name in das entsprechende Textfeld eingegeben werden (siehe Abbildung 5.13). Falls es mehrere Suchtreffer gibt, kann man über die zwei Buttons durch das Ergebnis navigieren.

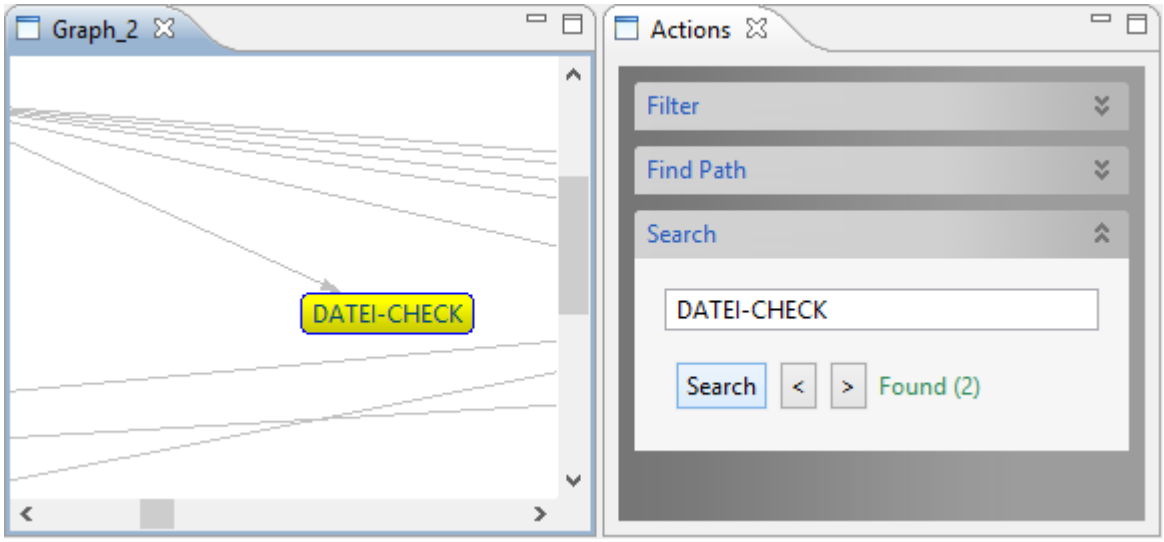

Abbildung 5.13: Suchen von Knoten

5.12 Filtern von Knoten

Filtern von Knoten bedeutet, dass sie aus dem aktuellen Graphen gelöscht werden. Alle dazugehörigen Kanten werden ebenfalls entfernt. Diese Knoten können dann über die Suche nicht mehr gefunden werden und werden beim Layout nicht mehr berücksichtigt. Alle gefilterten Knoten werden in einer Liste in der *View* Actions in der Rubrik "Filter" gespeichert (siehe Abbildung 5.14). Es gibt drei Möglichkeiten Knoten zu filtern:

- anhand ihres Namens,
- direkt im Graphen und
- über das "Hide Far-Off Nodes"-Dialogfenster.

5.12.1 Filtern von Knoten anhand ihres Namens

Beim Filtern anhand des Namens muss der exakte Name in das Textfeld in der *View* Actions unter der Rubrik "Filter" eingetragen werden (siehe Abbildung 5.14).

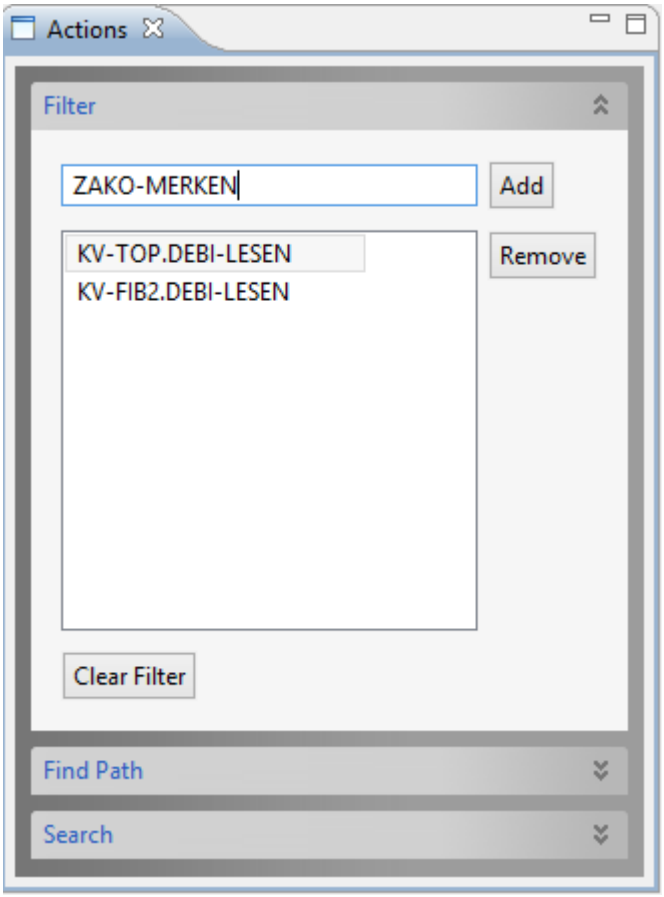

Abbildung 5.14: Filtern von Knoten anhand ihres Namens

5.12.2 Filtern von Knoten im Graphen

Um einen Knoten direkt im Graphen zu filtern, muss man mit der rechten Maustaste auf diesen Knoten klicken und anschließend im Kontextmenü den Menüpunkt “Hide“ auswählen (siehe Abbildung 5.15).

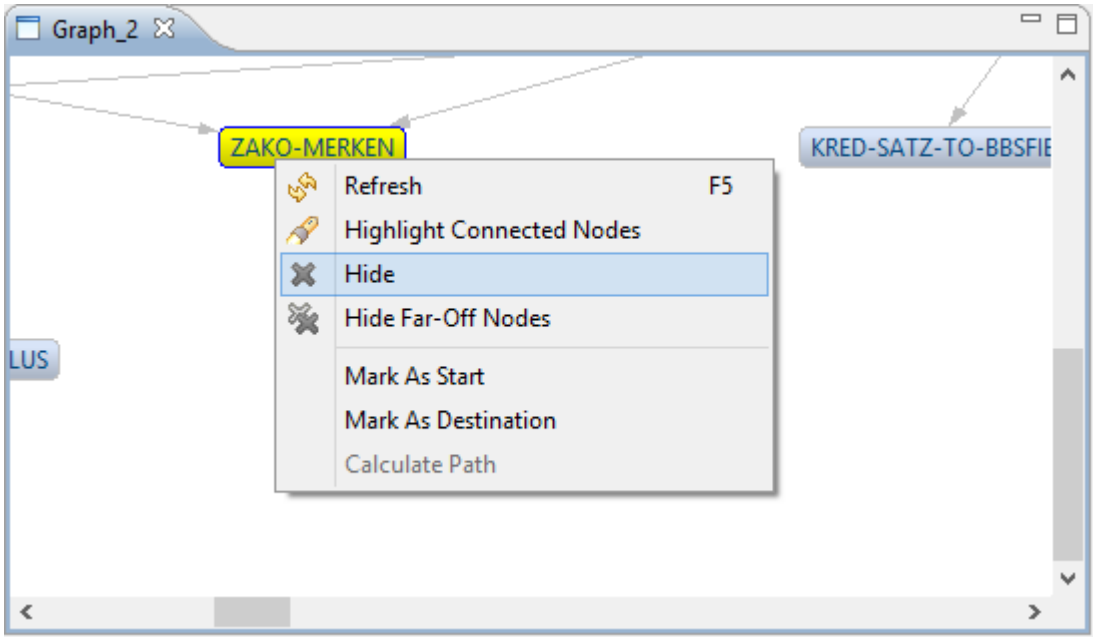

Abbildung 5.15: Filtern von Knoten im Graphen

5.12.3 Filtern von Knoten, die mindestens n Knoten von einem bestimmten Knoten entfernt sind

Es gibt die Möglichkeit alle Knoten zu filtern, die mindestens n Knoten von einem bestimmten Knoten entfernt sind. Dazu muss man mit der rechten Maustaste auf diesen bestimmten Knoten klicken und im Kontextmenü den Menüpunkt “Hide Far-Off Nodes“ auswählen (siehe Abbildung 5.16). Anschließend öffnet sich ein Dialogfenster, in dem man die Mindestentfernung (angegeben in Knoten) der zu filternden Knoten angeben muss (siehe Abbildung 5.17).

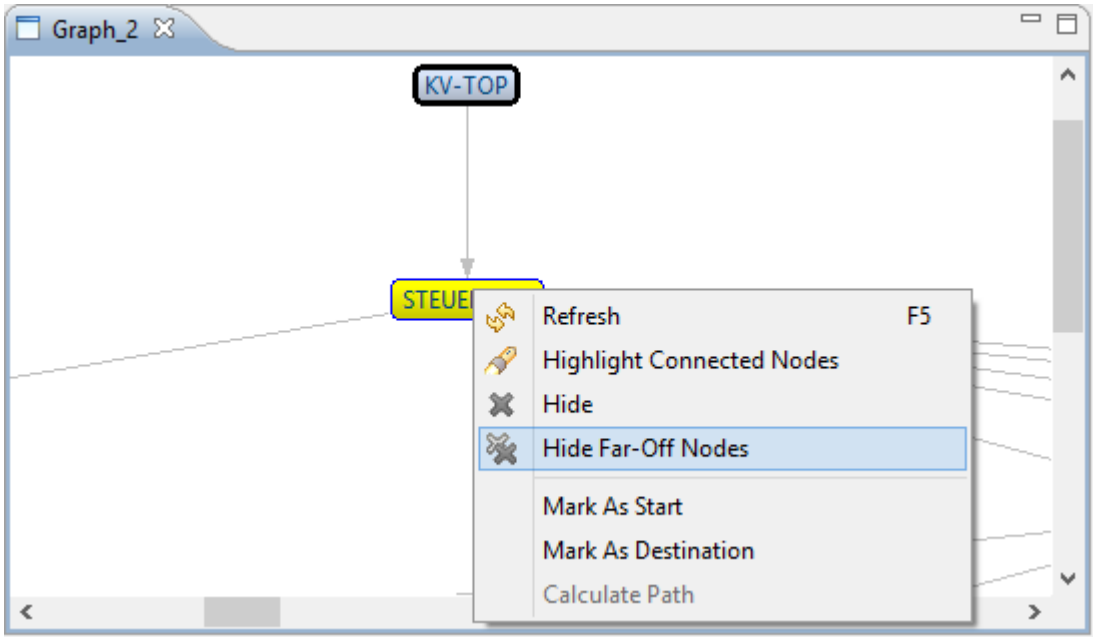

Abbildung 5.16: Filtern von Knoten, die mindestens n Knoten von einem bestimmten Knoten entfernt sind (1)

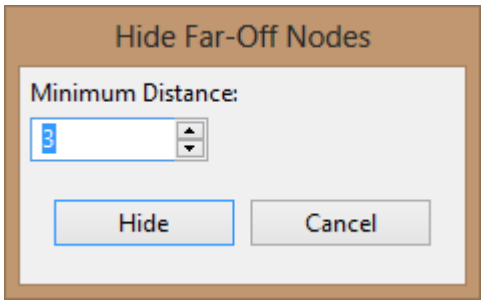

Abbildung 5.17: Filtern von Knoten, die mindestens n Knoten von einem bestimmten Knoten entfernt sind (2)

5.13 Suchen von Verbindungswegen zwischen zwei Knoten

Um einen Verbindungsweg zwischen zwei Knoten zu suchen, muss man zuerst den Start- und Zielknoten definieren. Dazu klickt mit der rechten Maustaste auf den Startknoten und wählt im Kontextmenü den Menüpunkt “Mark As Start“ aus (siehe Abbildung 5.18). Anschließend klickt man mit der rechten Maustaste auf den Zielknoten und wählt im Kontextmenü den Menüpunkt “Mark As Destination“ aus (siehe Abbildung 5.19).

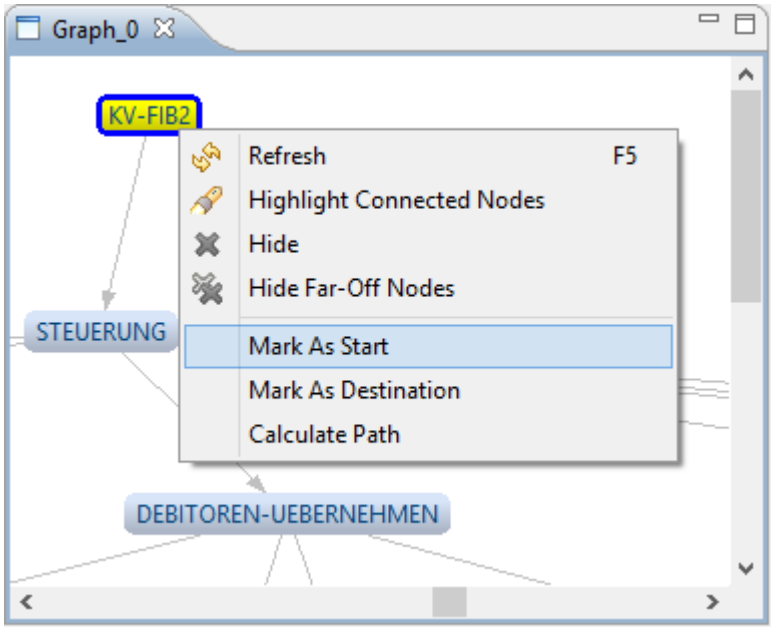

Abbildung 5.18: Markieren des Startknotens zur Suche von Verbindungswegen zwischen zwei Knoten

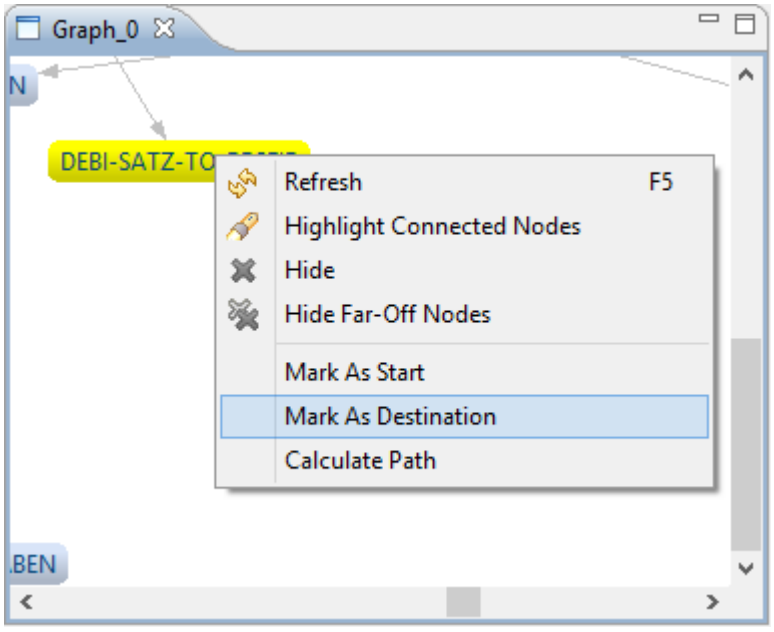

Abbildung 5.19: Markieren des Zielknotens zur Suche von Verbindungswegen zwischen zwei Knoten

Nun öffnet man in der *View* Actions die Rubrik “Find Path“. Dort kann man den Start- und Zielknoten noch einmal kontrollieren bzw. auch wieder löschen. Über den Button “Calculate Path“ wird die Suche gestartet (siehe Abbilung 5.20). Dies kann auch über das Kontextmenü im Graphen über den Menüpunkt “Calculate Path“ erfolgen. Der resultierende Pfad ist der kürzeste existierende Pfad der über den Dijkstra’s Shortest-Path-Algorithmus aus der JUNG Bibliothek (siehe Kapitel 2.3.2) gefunden wurde.

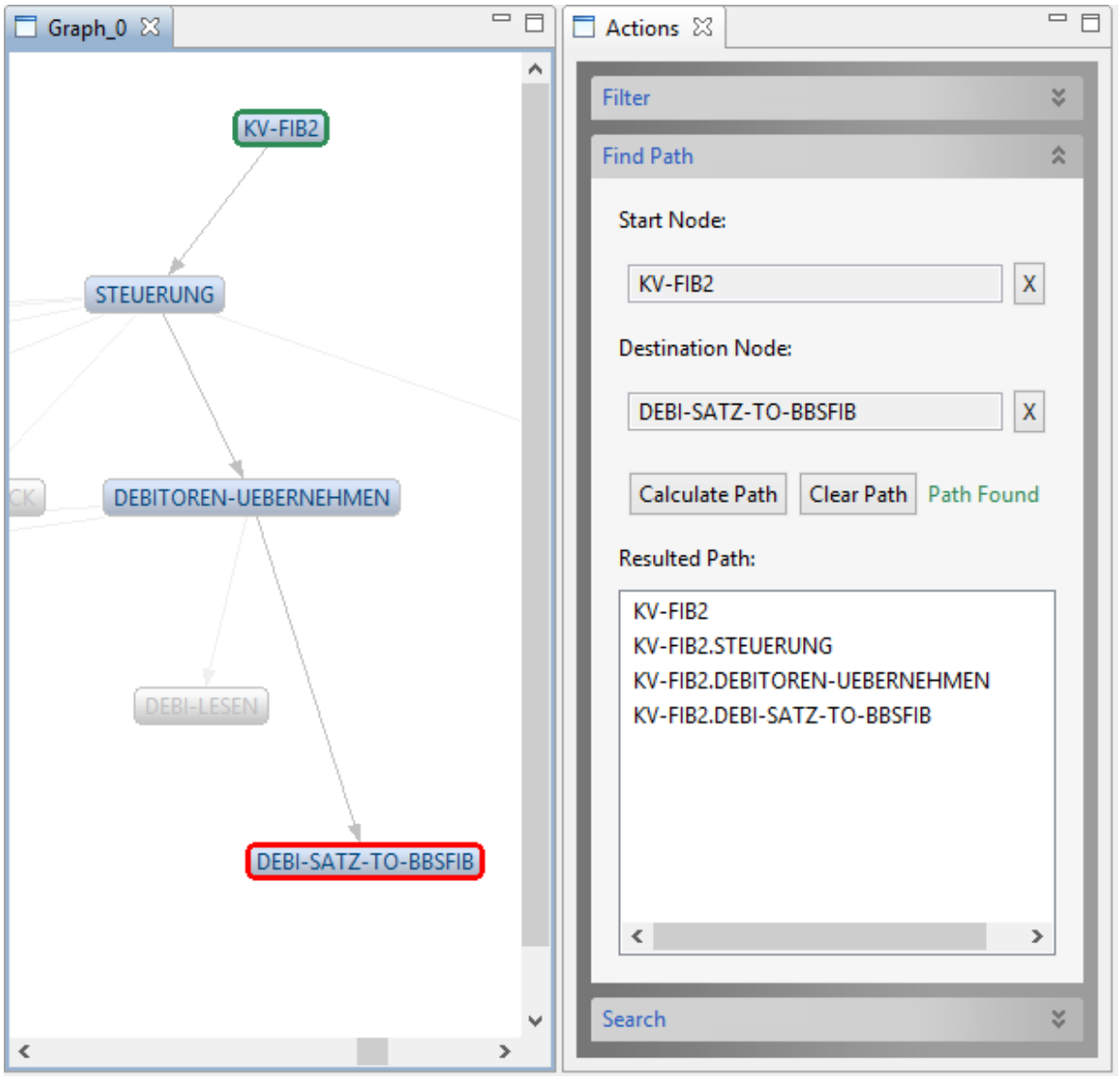

Abbildung 5.20: Suchen von Verbindungswegen zwischen zwei Knoten (Resultat)

5.14 Anbindung an das Analysewerkzeugs der itestra GmbH

5.14.1 Einbindung der notwendigen Eclipse Plug-ins

Für die Anbindung an den DependencyAnalyzer der itestra GmbH ist das Plug-in

de.nolleryc.dependencymodeladapter

notwendig. Die Datei `de.nolleryc.dependencymodeladapter_1.0.0.jar` muss in den Ordner "eclipse/plugins" kopiert und *Eclipse* neu gestartet werden.

Zusätzlich müssen folgende Plug-ins von der itestra GmbH in *Eclipse* eingebunden sein:

- de.itestra.conqat.dependency
- de.itestra.swgcommons
- de.itestra.lib3rd

Des Weiteren ist das Plug-in de.nolleryc.dependencymodeladapter von folgenden Plug-ins abhängig, die allerdings schon in *Eclipse* eingebunden sind:

- org.eclipse.ui
- org.eclipse.core.runtime

5.14.2 Öffnen der View DependencyLoader

Die *View* DependencyLoader kann über das Menü DependencyVisualizer -> Show DependencyLoader geladen werden (siehe Abbildung 5.21).

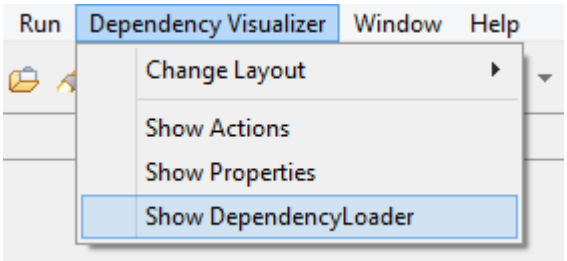

Abbildung 5.21: Öffnen der View DependencyLoader

5.14.3 Starten der Analyse

Mit der *View* DependencyLoader kann das Analysewerkzeug der itestra GmbH gestartet werden. Dazu müssen in dem Textfeld "Command Line Parameters" die Kommandozeilenparameter für das Werkzeug eingegeben werden. Mit dem "Load"-Button wird die Analyse gestartet. Alle Ausgaben des Analysewerkzeugs werden abgefangen und in dem Log Stream der *View* ausgegeben (siehe Abbildung 5.22).

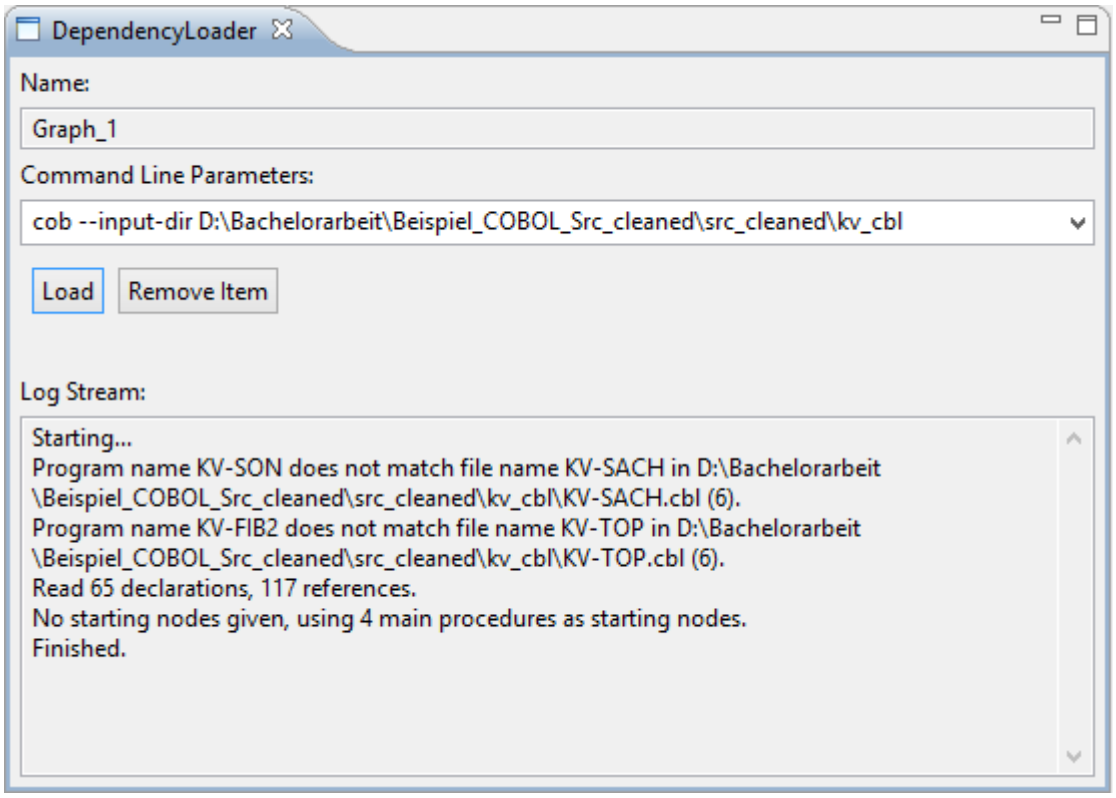

Abbildung 5.22: Starten der Analyse

Die bisher verwendeten Kommandozeilenparameter werden im Plug-in gespeichert und sind über die Auswahlbox im Textfeld der *View* verfügbar (siehe Abbildung 5.23). Die Kommandozeilenparameter können damit wiederverwendet werden.

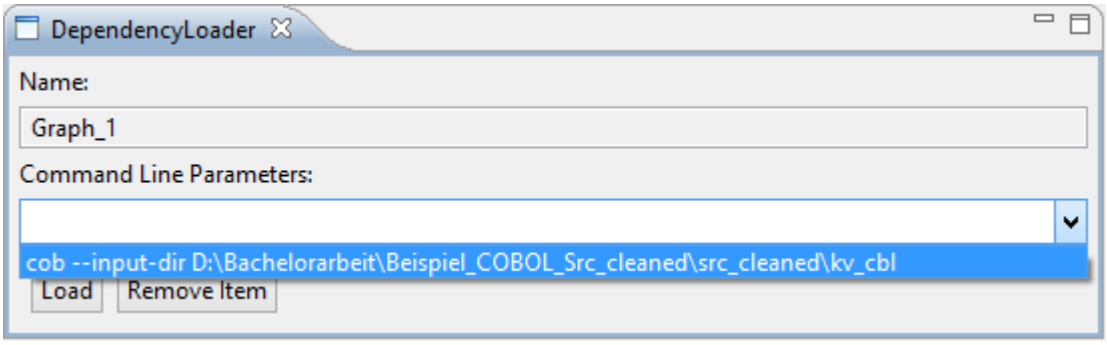

Abbildung 5.23: Starten der Analyse mit gespeicherten Parameter

5.15 Deinstallation von Eclipse Plug-ins

Je nachdem wie ein Plug-in installiert wurde, muss es deinstalliert werden:

- **Installation über Kopieren in den "Plugin"-Ordner**:
  Wurde ein Plug-in zum Installieren in den "Plugin"-Ordner von *Eclipse* kopiert, so muss dieses Plug-in lediglich von dort entfernt und *Eclipse* neu gestartet werden.
- **Installation über eine Update-Site**:
  Wenn ein Plug-in über die Installationsmaske von *Eclipse* installiert wurde, muss es auch darüber wieder deinstalliert werden. Dazu öffnet man das Menü Help -> Install New Software... (siehe Abbildung 5.1). Dort klickt man auf den Text "What is already installed?". In dem sich öffnenden Fenster werden alle installierten Plug-ins angezeigt. Man wählt den Eintrag für das zu deinstallierende Plug-in (z. B. *Zest*) aus und drückt den "Uninstall"-Button (siehe Abbildung 5.24). Danach folgt man den Anweisungen von *Eclipse*.

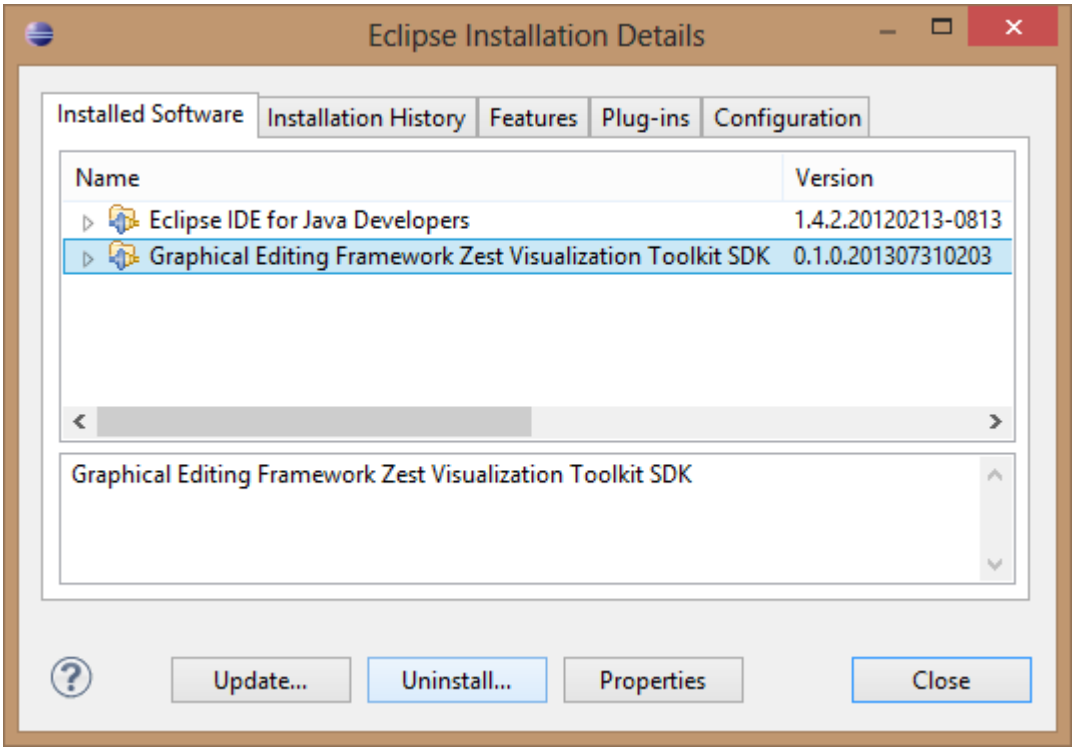

Abbildung 5.24: Zest Plug-in deinstallieren

6 Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurde in Kooperation mit der itestra GmbH eine *Eclipse* Plug-in zur Visualisierung von Abhängigkeitsgraphen entworfen und implementiert.

Die itestra GmbH ist ein Software-Dienstleister im Bereich unternehmenskritischer Prozesse, Systeme und Anwendungen. Zum Hauptarbeitsgebiet dieses Unternehmens gehören Beratungs- und Reengineeringsleistungen, wozu sie ein selbst entwickeltes Analysewerkzeug besitzen. Allerdings war die bisherige Visualisierung der resultierenden Abhängigkeitsgraphen unzureichend und soll in Zukunft durch das hier entwickelte *Eclipse* Plug-in ersetzt werden.

Die hier entwickelte Lösung soll den Abhängigkeitsgraphen dynamisch visualisieren und einige Transformationen auf dem Graphen anbieten.

Die Software wurde auf Wunsch der itestra GmbH mit der Vorgehensweise der evolutionären Software-Entwicklung entwickelt. Dazu wurden drei Evolutionsstufen implementiert, die jeweils durch eine Evaluation mit den Mitarbeiter der itestra GmbH abgeschlossen wurden. Die Evaluation der letzten Evolutionsstufen wurde in einer Art Interview mit den Mitarbeitern der itestra GmbH durchgeführt. Den Probanden wurden nacheinander Funktionen des Werkzeugs vorgestellt und anschließend Fragen zum praktischen Einsatz gestellt.

Aus der Entwicklung resultierten insgesamt drei *Eclipse* Plug-ins: DependencyVisualizer, DependencyLayoutAlgorithms und DependencyModelAdapter. Das Plug-in DependencyVisualizer sorgt für die eigentliche Visualisierung des Abhängigkeitsgraphen und nutzt dabei die Layout-Algorithmen aus dem Plug-in DependencyLayoutAlgorithms. Das Plug-in DependencyModelAdapter stellt die Anbindung an das bestehende Analysewerkzeug der itestra GmbH bereit.

Die Arbeit wurde mit einem Benutzerhandbuch zur Bedienung der entwickelten Software abgeschlossen.

Ausblick

Das Werkzeug DependencyVisualizer wird in Zukunft von den Mitarbeitern der itestra GmbH eingesetzt. Bei der abschließenden Evaluation wurden neue Anforderungen identifiziert, die über den Rahmen dieser Bachlorarbeit hinaus, eingebaut werden können. Weiterhin besteht die Möglichkeit das Werkzeug unter einer OpenSource-Lizenz wie EPL zu veröffentlichen.

Literaturverzeichnis

[AT 13] AT & T Research. GraphViz, 2013. URL http://www.graphviz.org/Home.php. (Zitiert auf Seite 14)

[Bar13] Barak Naveh and Contributors. JGraphT, 2013. URL http://jgrapht.org/. (Zitiert auf Seite 20)

[hel13] hello2morrow. Sotograph, 2013. URL https://www.hello2morrow.com/products/sotograph. (Zitiert auf den Seiten 7, 17 und 18)

[ite13] itestra GmbH. itestra GmbH, 2013. URL http://www.itestra.de/home/. (Zitiert auf Seite 9)

[JGr13] JGraph Ltd. Java Graph Visualization Library, 2013. URL http://www.jgraph.com/jgraph.html. (Zitiert auf den Seiten 7, 15 und 16)

[JLo9] H. L. Jochen Ludewig. *Software Engineerig*. dpunkt.verlag, 2009. (Zitiert auf den Seiten 25, 31 und 32)

[Jos13] Joshua O'Madadhain, Danyel Fisher, Tom Nelson. JUNG, 2013. URL http://jung.sourceforge.net/. (Zitiert auf Seite 21)

[The13a] The Eclipse Foundation. Eclipse Public License (EPL) Frequently Asked Questions, 2013. URL http://www.eclipse.org/legal/eplfaq.php. (Zitiert auf den Seiten 20 und 21)

[The13b] The Eclipse Foundation. GEF (Graphical Editing Framework), 2013. URL http://www.eclipse.org/gef/. (Zitiert auf Seite 13)

[The13c] The Eclipse Foundation. PDE Incubator Dependency Visualization, 2013. URL http://www.eclipse.org/pde/incubator/dependency-visualization/. (Zitiert auf den Seiten 7, 16 und 17)

[The13d] The Eclipse Foundation. Zest: The Eclipse Visualization Toolkit, 2013. URL http://www.eclipse.org/gef/zest/. (Zitiert auf Seite 13)

[yWo13a] yWorks. yEd, 2013. URL http://www.yworks.com/en/products_yed_about.html. (Zitiert auf Seite 18)

[yWo13b] yWorks. yFiles, 2013. URL http://www.yworks.com/en/products_yfiles_about.html. (Zitiert auf Seite 21)

Alle URLs wurden zuletzt am 06. 08. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift