Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 66

Integration der Analyse von Abhängigkeitsgraphen in eine Reengineering-IDE

Fabian Müller

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer:	Dipl.-Inf. Ivan Bogicevic
	Dipl.-Inf. Jonathan Streit
Beginn am:	01. Mai 2013
Beendet am:	07. August 2013
CR-Nummer:	D.2.6, K.6.3

Kurzfassung

Diese Bachelorarbeit beschäftigt sich mit der Planung und Durchführung eines Softwareentwicklungsprozesses. Dabei wird über die Zeitplanung, die Planung der Umsetzung, die Implementierung und die abschließende Evaluation berichtet. Am Ende dieses Dokuments befindet sich das Handbuch des entwickelten Programms.
Ziel des Softwareprojektes war die Integration des Kommandozeilenprogramms "DependencyAnalyzer" in die Eclipse-IDE. Der DependencyAnalyzer ist ein von der itestra GmbH entwickeltes Werkzeug mit umfangreichen Funktionalitäten zur Analyse von Quellcodes und wird zur Planung und Durchführung von Restrukturierungsmaßnahmen verwendet. Die Integration soll die wichtigsten Funktionen in der Eclipse-IDE zur Verfügung stellen, um den Einsatz einem breiteren Publikum zu ermöglichen. So wurde beispielsweise eine View implementiert, die verfügbare Kommandozeilenargumente aus dem DependencyAnalyzer extrahiert und eine einfache und komfortable Parametrisierung und Ausführung ermöglicht. Des Weiteren wurde für die Visualisierung der Ergebnisse des DependencyAnalyzers eine eigene Marker-View entwickelt und kontextabhängige Aufrufmöglichkeiten in den Navigator und in einen COBOL-Editor integriert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

1 Einleitung

1.1 Gliederung

Diese Arbeit ist in folgende Kapitel unterteilt:

Kapitel 1 - Einleitung: In diesem Kapitel wird das Thema der Bachelorarbeit präsentiert sowie das Ziel für den Industriepartner und das Institut für Softwaretechnologie an der Universität Stuttgart vorgestellt.

Kapitel 2 - Literaturrecherche: In diesem Kapitel werden Programme vorgestellt, die sich mit einem ähnlichen Thema beschäftigt haben und der Frage, warum deren Funktionalität nicht zufriedenstellend ist.

Kapitel 3 - Entwicklung In diesem Kapitel wird der gesamte Entwicklungsprozess beschrieben. Dabei wird insbesondere auf die Planung der Umsetzung, den Schwierigkeiten und die Umsetzung selbst eingegangen.

Kapitel 4 - Evaluation In diesem Kapitel wird die Durchführung und das Ergebnis der Abschlussevaluation beim Kunden beschrieben.

Kapitel 5 - Handbuch In diesem Kapitel werden alle Funktionen des DependencyUtilizers sowie deren Benutzung erläutert.

1.2 Vorstellung des Industriepartners

Die Firma itestra GmbH ist ein 2004 gegründeter Software-Dienstleister im Bereich Solution Engineering, Software Governance und Produktentwicklung mit Sitz in München [ite07]. Ein Großteil der durchgeführten Projekte beschäftigt sich mit der Renovierung und Restrukturierung vorhandener Softwaresysteme. Ziel dabei ist es, durch Effizienzsteigerungen dem Kunden im IT-Bereich zu Einsparungen zu verhelfen. Des Weiteren bietet die itestra GmbH ihren Kunden selbst entwickelte Tools, wie beispielsweise Kernbankensysteme und Logistiksteuerungen an, die je nach Kundenwunsch angepasst werden können.

1.3 Aufgabenstellung

Für die Durchführung und Planung von Restrukturierungsprojekten setzt die itestra GmbH ein selbst entwickeltes Programm, den DependencyAnalyzer, ein. Dieses Programm bietet umfangreiche Funktionalitäten zur Analyse von Quellcodes in verschiedenen Programmiersprachen. Es lassen sich beispielsweise Abhängigkeitsgraphen erstellen oder nicht genutzte Codesegmente identifizieren. Die Visualisierungsmöglichkeiten des DependencyAnalyzers sind beschränkt, werden aber durch die Integration in die Eclipse-IDE in einer parallel stattfindenden Bachelorarbeit verbessert.

Ziel dieser Bachelorarbeit ist es, den DependencyAnalyzer in die Eclipse-IDE zu integrieren, dass eine direkte Verbindung zwischen dem zu analysierenden Quellcode und der Analyse des DependencyAnalyzers hergestellt werden kann. Die einzelnen Anforderungen lauten wie folgt [1]:

- Aufruf des Werkzeugs zur Erstellung eines Abhängigkeitsgraphen aus der Eclipse-IDE heraus mit der Möglichkeit, den zu analysierenden Quellcode in Eclipse auszuwählen und die Optionen für die Analyse festzulegen
- Kontextabhängige Aufrufmöglichkeiten in der Eclipse-IDE, z. B. Abhängigkeitsgraph mit der aktuell im Editor ausgewählten Prozedur oder der Variable als Ausgangspunkt
- Anzeige von Zusatzinformationen im Abhängigkeitsgraphen, z. B. aus welcher Quelldatei stammt dieser Knoten des Graphs?
- Sprungmöglichkeiten zwischen Quellcode und Knoten im Abhängigkeitsgraph
- Markierung von Knoten, die in der Analyse als ungenutzt/unerreichbar erkannt wurden, im Quellcode (Warning-Marker)

Des Weiteren soll die Implementierung in Zusammenarbeit mit Mitarbeitern der itestra GmbH evaluiert werden, um gegebenenfalls weitere nützliche Funktionalitäten zu identifizieren.

1.4 Motivation

Zur Zeit wird der DependencyAnalyzer nur von wenigen Mitarbeitern der itestra GmbH verwendet. Dies liegt hauptsächlich an der unkomfortablen und aufwendigen Bedienung über die Kommandozeile. Die Integration in die Eclipse-IDE soll die Bedienung und Konfiguration deutlich vereinfachen und beschleunigen, um das Programm einem größeren firmeninternen Nutzerkreis zugänglich zu machen und es eventuell an Kunden herauszugeben. Zudem soll der DependencyAnalyzer in zukünftigen Projekten eine größere Rolle spielen.
Das Interesse des Instituts für Softwaretechnologie besteht in der Umsetzung der Einbindung

[1] Aus der offiziellen Aufgabenstellung entnommen

des DependencyAnalyzers. In einem zukünftigen Projekt wird es eine ähnliche Problemstellung geben, bei der ein Kommandozeilenprogramm durch eine graphische Benutzeroberfläche erweitert werden soll. Die in dieser Bachelorarbeit entstandene Implementierung soll als Grundlage bzw. Lösungsvorschlag herangezogen werden.

2 Literaturrecherche

In diesem Kapitel werden Tools vorgestellt, die bereits Lösungen für das in dieser Bachelorarbeit zu bearbeitende Problem anbieten. Es werden dabei nur Tools in Betracht gezogen, die eine Erweiterung der Eclipse-IDE darstellen und den Quellcode als Graph visualisieren. Des Weiteren wird darauf eingegangen, weshalb diese Tools für den Industriepartner nicht zufriedenstellend sein können.

2.1 Metrics

"'Metrics" ist ein von Lance Walton entwickeltes Plugin für Eclipse, welches während des Build-Vorgangs Metriken von Java-Projekten erhebt [met] und eine Visualisierung der Abhängigkeiten erstellt. Im folgenden wird die Version 1.3.6 betrachtet, die am 24.04.2013 erschien. "Metrics" wurde unter der Lizenz "'Common Public License 1.0" veröffentlicht.

2.1.1 Funktionen

"Metrics" erhebt und bewertet während des Build-Vorgangs von Java-Projekten umfangreiche Metriken des Quellcodes. Dazu gehören unter anderem die Anzahl der Codezeilen, Vererbungstiefe von Klassen, durchschnittliche Anzahl der Parameter pro Methode und die Tiefe von verschachtelten Anweisungsblöcken. Zudem besteht die Möglichkeit, den Abhängigkeitsgraph eines Java-Projekts zu erstellen (siehe Abb. 2.1).

2.1.2 Probleme

Die von "Metrics" erfassten Daten sind umfangreich und bieten einen Überblick über die Codequalität. Diese sind für Restrukturierungsmaßnahmen aber nicht von unmittelbarer Relevanz. Auch die Visualisierung der Abhängigkeiten ist dafür nicht geeignet. Es können nur die Abhängigkeiten von Paketen angezeigt werden. Der Graph ist durch die vielen Überschneidungen nur schwer lesbar. Zudem ist die Analyse von anderen Programmiersprachen nicht möglich.

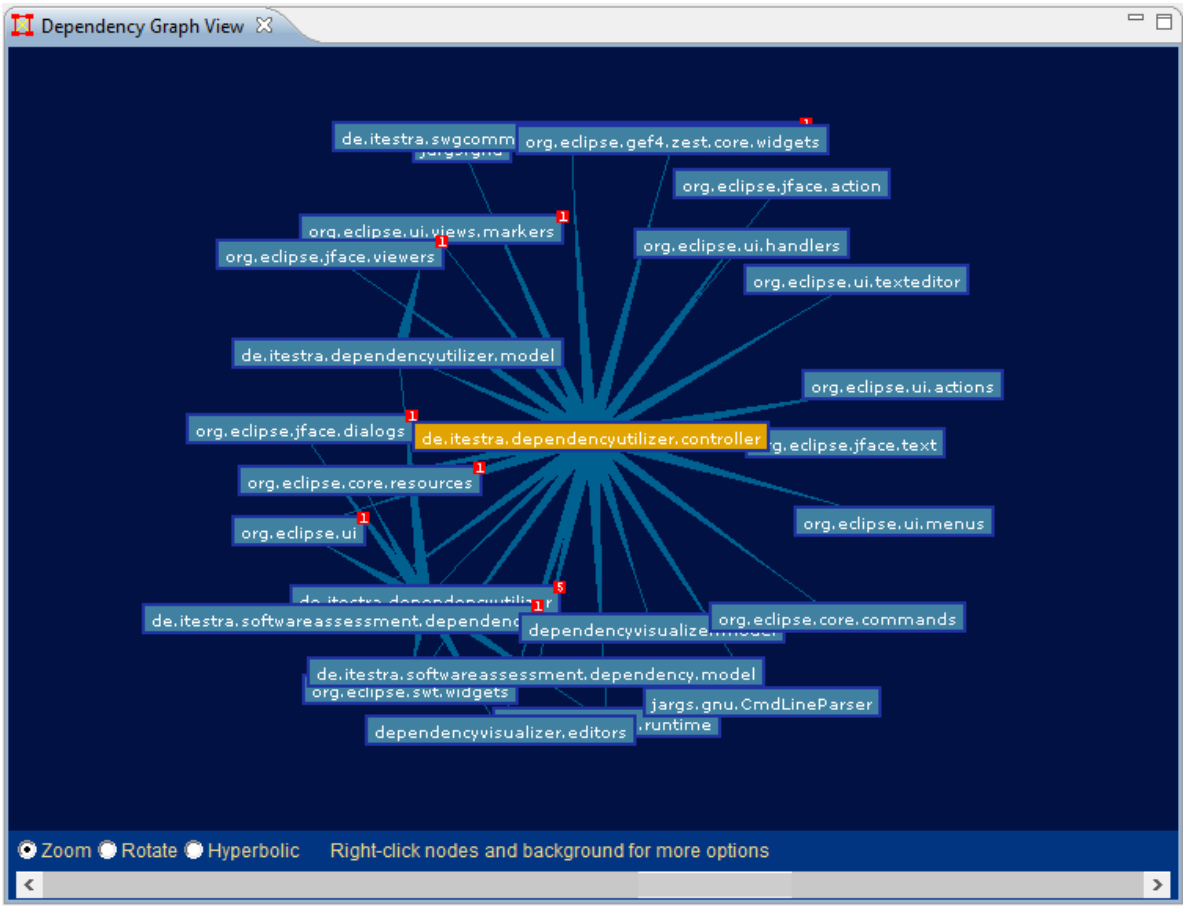

Abbildung 2.1: Visualisierung des Controllers des DependencyUtilizers mit “Metrics“

2.2 X-Ray

Bei X-Ray handelt es sich um ein Eclipse-Plugin zu Visualisierung von Softwarestrukturen. Dieses Projekt wird von Jacopo Malnati entwickelt [xra13]. Es entstand im Rahmen einer Bachelorarbeit und wurde anschließend als Forschungsprojekt weiter geführt [xra08]. Diese Begutachtung bezieht sich auf die Version 1.0.4.1, welche am 13. Dezember 2007 veröffentlicht wurde. Um das Programm nutzen zu können, muss zusätzlich das Graphikframework “GEF“ als Plugin installiert werden. X-Ray wurde als Open Source Projekt veröffentlicht, genauere Lizenzangeben werden jedoch nicht gemacht.

2.2.1 Funktionen

X-Ray enthält zwei unterschiedliche Diagrammtypen zur Visualisierung von Quellcodestrukturen, die sogenannte “System Complexity View“ und die “Class Dependency View and

Package Dependency View“ [xra13].
Die “System Complexity View“ visualisiert die Komplexität von Klassen und ihren Vererbungen (siehe Abb. 2.2). Klassen werden dazu als Rechtecke modelliert und in einer hierarchischen Baumstruktur angeordnet. Die Anzahl der Codezeilen in einer Klasse wird dabei auf die Höhe des Rechtecks gemappt, die Anzahl der Methoden auf die Breite. Eine Vererbungsbeziehung wird durch eine Kante ausgedrückt.
Die ”’Class Dependency View and Package Dependency View“ visualisiert die Abhängigkeit von Klassen oder Paketen. Die Klassen oder Pakete werden dabei kreisförmig als Knoten angeordnet. Eine Kante zwischen zwei Knoten repräsentiert dabei einen Methodenaufruf. Der Grad der Abhängigkeit wird auf die Kantendicke gemappt.

2.2.2 Probleme

X-Ray bietet nur die Möglichkeit, Java-Projekte zu analysieren [xra13]. Auch eine Parametrisierung der Analyse ist nicht möglich. So kann zum Beispiel die Graphtiefe nicht festgelegt oder eine Analyse auf Variablenebene nicht durchgeführt werden. Die Visualisierung ist zum Teil auch nur durch Interaktion mit den Graphen zu verstehen. Beispielsweise sind in Abb. 2.2 die Klassennamen nur über einen Tooltip sichtbar.
Da der Kunde hauptsächlich COBOL-Projekte analysieren will und eine umfangreiche Parametrisierung der Analyse benötigt, ist X-Ray für den Einsatz ungeeignet.

2.3 Zusammenfassung

Fast alle für die Eclipse-IDE verfügbaren Plugins zur Analyse von Quellcodes beschränken sich auf die Programmiersprache Java. Insbesondere für die Analyse von COBOL sind keine Lösungen vorhanden. Der Funktionsumfang dieser Plugins zur Planung und Durchführung von Restrukturierungsmaßnahmen ist nicht ausreichend und die Visualisierungen sind oft nicht befriedigend. Aus diesen Gründen ist die Integration des DependencyAnalyzers in die Eclipse-IDE sinnvoll.

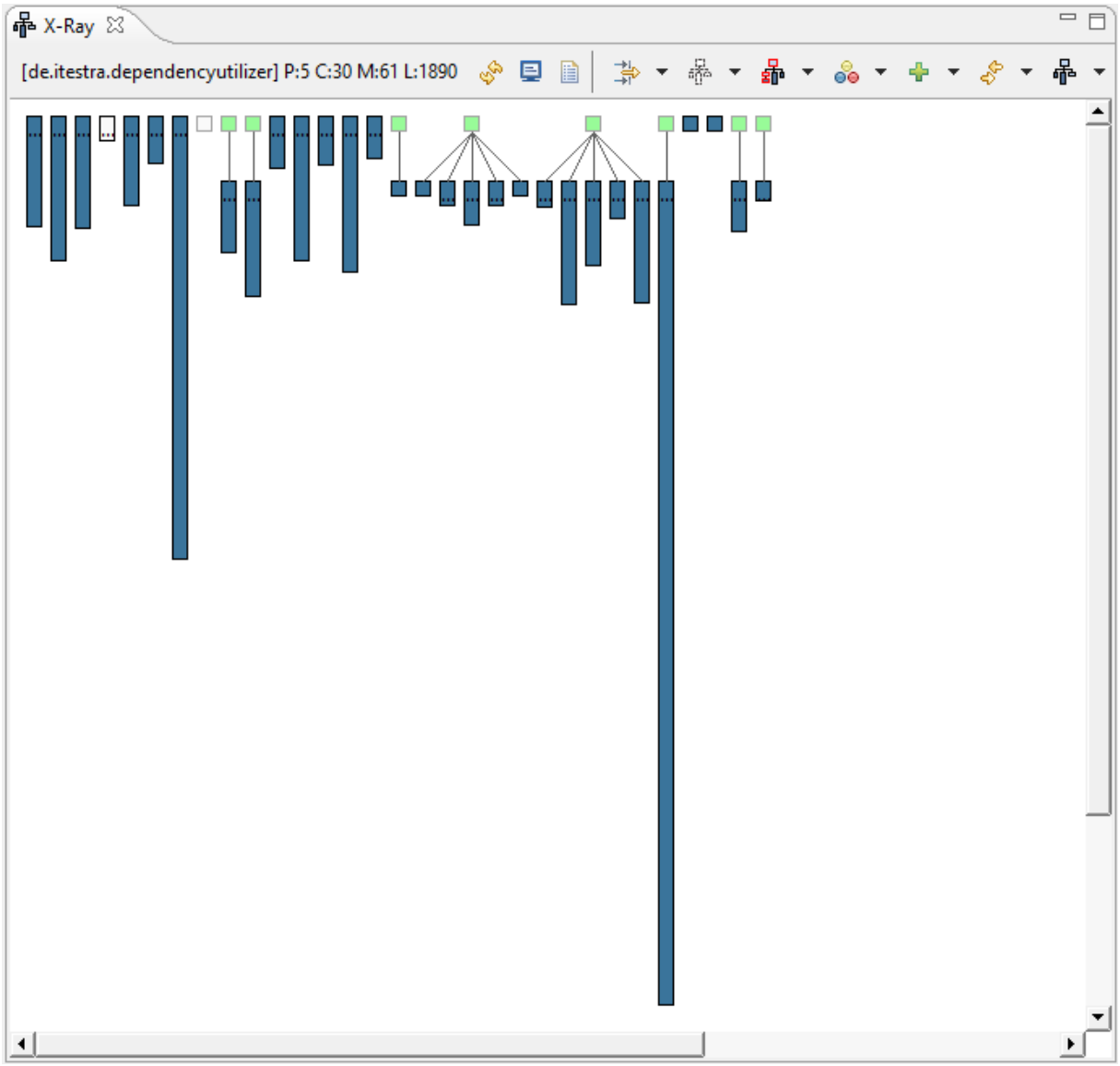

Abbildung 2.2: Visualisierung des DependencyUtilizers in der “System Complexity View“ von X-Ray

3 Entwicklung

3.1 Planung

3.1.1 Vorgehensmodell

Dem Kunde war es wichtig, die Zwischenstände der Implementierung zu testen und zu bewerten. Auf dieser Grundlage sollte das Vorgehen für den nächsten Entwicklungszyklus entwickelt werden. Eine externe Dokumentation des Tools wurde nicht gefordert. Aus diesen Gründen wurde von dem Industriepartner das Vorgehensmodell der evolutionären Entwicklung vorgeschlagen (vgl. [JL10]). Im Laufe der Entwicklung wurden somit drei Evolutionsstufen entwickelt und dem Kunden vorgestellt. Dieser konnte daraufhin die Funktionalitäten testen, Änderungswünsche äußern und die verbleibenden zu implementierenden Funktionen priorisieren. Wie in [JL10] erwähnt, besteht bei diesem Vorgehensmodell allerdings die Gefahr, dass durch die fehlende Planung die Qualität des entstandenen Quellcodes nicht zufriedenstellend ist. Insbesondere durch die fehlende externe Dokumentation könnte dadurch die Wartbarkeit beim Kunden beeinträchtigt werden. Aus diesem Grund wurde bei der Planung der letzten Implementierungphase genug Zeit eingeplant, um den Quellcode zu refaktorieren und die Kommentare zu überarbeiten.

3.1.2 Zeitplanung

Die Zeitplanung wurde mit Hilfe des GTD-Managers visualisiert[1] (siehe Abb. 3.1).

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

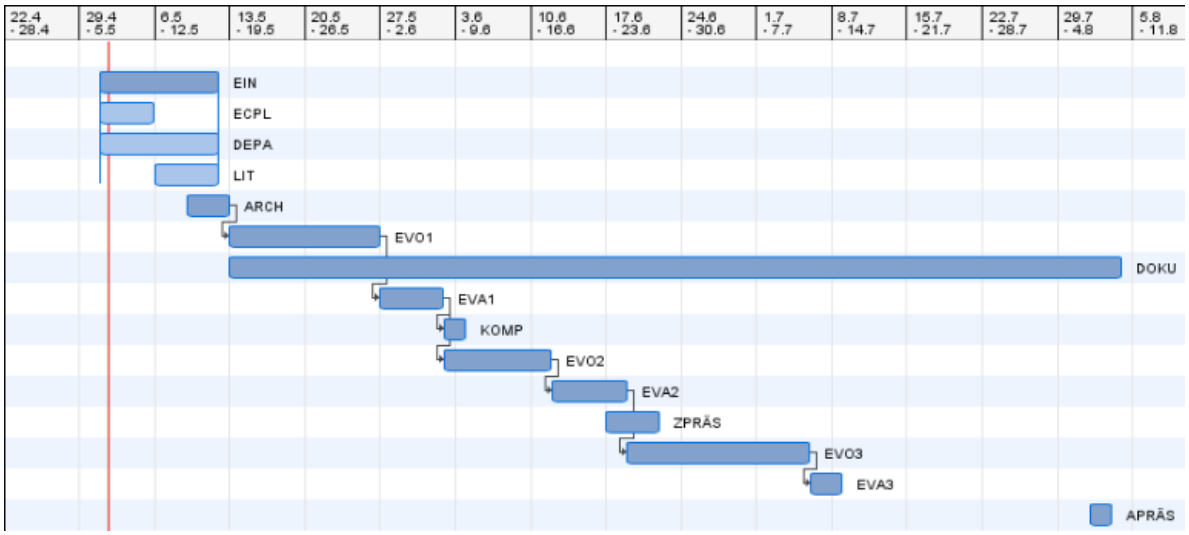

Abbildung 3.1: Zeitplanung der Bachelorarbeit als Gantt-Diagramm, erstellt mit dem GTD-Manager

Bedeutung der Abkürzungen:

EIN Einarbeitungsphase

ECPL Einarbeitung in die Entwicklung von Eclipse-Plugins

DEPA Einbindung des DependencyAnalyzers in ein Eclipse-Plugin-Testprojekt

LIT Durchführung der Literaturrecherche

ARCH Erstellung der Architektur

EVO1 Implementierung der 1. Evolutionsstufe

DOKU Anfertigung der Ausarbeitung

EVA1 Durchführung der Evaluation der 1. Evolutionsstufe

KOMP Identifizierung der zu implementierenden Komponenten in der 2. Evolutionsstufe

EVO2 Implementierung der 2. Evolutionsstufe

EVA2 Durchführung der Evaluation der 2. Evolutionsstufe

ZPRÄS Durchführung der Zwischenpräsentation

EVO3 Implementierung der 3. Evolutionsstufe

EVA3 Durchführung der Evaluation der 3. Evolutionsstufe

APRÄS Durchführung der Abschlusspräsentation

3.1.3 Planung der Umsetzung

Im Rahmen der Bachelorarbeit soll ein Eclipse-IDE-Plugin mit dem Namen "DependencyUtilizer" entstehen. Der DependencyUtilizer soll die benötigten Funktionen des DependencyAnalyzers an unterschiedlichen Punkten der Eclipse-IDE durch Erweiterungen zur Verfügung stellen. Folgende Komponenten der Eclipse-IDE sollen dabei erweitert werden.

- der Navigator, für die Auswahl des zu analysierenden Codes
- die Views, für die Anzeige einer Parametrisierungsoberfläche
- der Cobol-Editor "'NetCOBOL" [2], für die Auswahl der zu analysierenden Sections und Variablen
- der Grapheditor des DependencyVisualizers für die Anzeige von Zusatzinformationen und Sprungmöglichkeit von einem Knoten im Graph zu seiner Deklaration im Quellcode

Als Entwicklungsplattform soll die IDE "Eclipse for RCP and RAP Developers" in der Version 3.7 verwendet werden. Um eine mit der Eclipse-IDE konsistente Oberfläche zu erhalten, soll dafür das Oberflächenframework "SWT"[3] zum Einsatz kommen.

3.2 Einführung in den DependencyAnalyzer

In diesem Kapitel werden einige Grundlagen im Umgang des DependencyAnalyzers gelegt, die für die folgenden Kapitel von Relevanz sind.
Der DependencyAnalyzer ist ein Kommandozeilenprogramm, mit dessen Hilfe Quellcode in verschiedenen Sprachen analysiert werden kann. Zwei der Hauptaufgaben sind dabei die Erstellung von Abhängigkeitsgraphen und die Erkennung von nicht erreichbaren Quellcodes. Für die Analyse der unterschiedlichen Sprachen sind im Dependency Analyzer die Frontends zuständig. Für die Integration in Eclipse werden dabei hauptsächlich folgende Cobol-Frontends benötigt:

- `cob`, für die Analyse von COBOL-Sections
- `cob-var`, für die Analyse von Variablen in COBOL

Jedes Frontend besitzt eine Vielzahl an weiteren Argumenten, die bestimmten Gruppen zugeordnet sind. Die Angabe des Arguments `--input-dir`, welches den Eingabepfad festlegt, ist dabei verpflichtend. Alle Dateien in diesem Pfad werden für die Analyse herangezogen. Das Ergebnis einer Analyse wird zunächst in einem internen Model gespeichert. Anschließend wird daraus die gewünschte Ausgabe erstellt, wie ein mit der Beschreibungssprache "'DOT"' generierter Graph oder eine Adjazenzmatrix als CSV-Datei.

[2] http://www.fujitsu.com/global/services/software/netcobol/
[3] http://www.eclipse.org/swt/

3.3 1. Evolutionsstufe

3.3.1 Planung

In der ersten Evolutionsstufe sollten zunächst die Funktionalitäten des DependencyAnalyzers in Eclipse anderen Plugins zur Verfügung gestellt werden. Um eine gute Modularisierung zu erreichen, war die Erstellung eines separaten Plugins aus dem DependencyAnalyzer naheliegend.
Als erste Aufrufsmöglichkeit in Eclipse sollte das Zeichnen eines Graphen aus dem Kontextmenü des Navigators ermöglicht werden. Da die Funktionalität des parallel entwickelten Plugins zur Visualisierung von Graphen, der DependencyVisualizer, noch nicht genutzt werden konnte, sollte auf die Exportfunktionalitäten des DependencyAnalyzers zurückgegriffen werden.
Die Ermittlung von totem Code sollte ebenfalls in den Navigator integriert werden. Anstatt der Generierung einer Datei sollten die benötigten Informationen direkt aus dem Model des DependencyAnalyzers ermittelt und als Marker in der "Problem"-View angezeigt werden.
Für die Einstellung aller verfügbaren Argumente des DependencyAnalyzers sollte eine View erstellt werden, die diese Argumente verständlich visualisiert und die einfache Manipulation der Parameter ermöglicht. Dafür sollte zunächst ein einfache Visualisierung ohne Funktion erstellt werden, um mit dem Kunde diese Idee diskutieren zu können.

3.3.2 Schwierigkeiten

Bei der ersten Evolutionsstufe gab es lediglich bei der Erstellung des Plugins des DependencyAnalyzer Probleme. Einige Abhängigkeiten wurden nicht hinzugefügt, so dass das DependencyAnalyzer-Plugin bei der Ausführung Fehler meldete. In Rücksprache mit dem Kunden konnten diese Abhängigkeiten identifiziert und nicht benötigter, fehlerverursachender Quellcode entfernt werden, so dass sich die dadurch entstandene Verzögerung in Grenzen hielt.

3.3.3 Umsetzung

Zunächst wurden aus dem DependencyAnalyzer selbst und den Bibliotheken, deren Funktionalitäten der DependencyAnalyzer benötigt, folgende Plugins erstellt:

- `de.itestra.conqat.dependency`, mit dem DependencyAnalyzer
- `de.itestra.lib3rd`, mit benötigten Bibliotheken von Drittanbietern
- `de.itestra.swgcommons`, mit benötigten Parsern und Utility-Klassen
- `org.conqat.lib.scanner`, mit benötigten Scannern

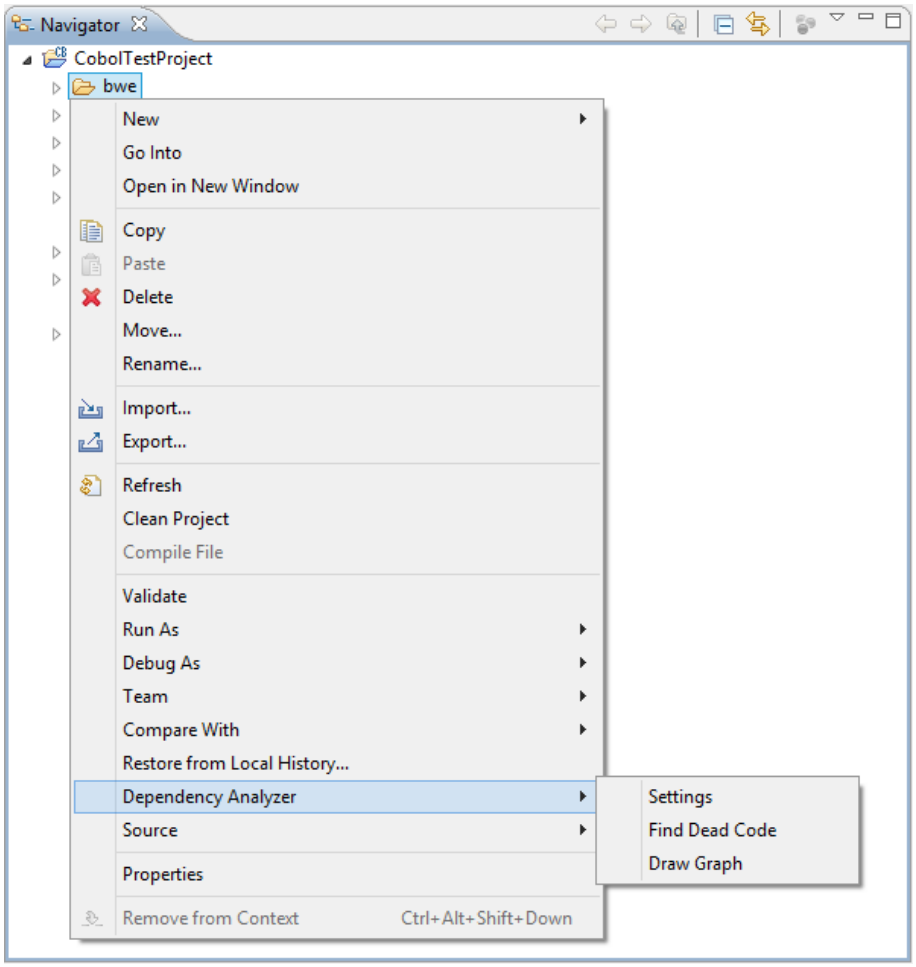

Abbildung 3.2: Funktionsumfang des Navigator-Kontextmenüs in der 1. Entwicklungsstufe

Als erster Schritt der Integration wurde das Kontextmenü des Navigators erweitert. Bei der Auswahl eines Ordners ist das Kommando "Quick Draw" und "Find Dead Code" verfügbar (siehe Abb. 3.2). Für die Generierung des Abhängigkeitsgraphen über das Kommando "Quick Draw" wurde folgende Parametriesierung des DependencyAnalyzers verwendet: `cob -input-dir EingabePfad -g AusgabePfad`. Das Wort "Eingabepfad" wird dabei durch den Pfad des selektierten Ordners im Navigator ersetzt, das Wort "AusgabePfad" durch die Eingabe des Benutzers in einem FileChooser. Der DependencyAnalyzer generiert darauf mit Hilfe der Sprache "DOT" eine PNG-Datei, die den Abhängigkeitsgraphen enthält.

Nach der Ausführung des Kommandos "Find Dead Code" wird der DependencyAnalyzer mit der Parametrisierung `cob -input-dir Pfad` gestartet. "Pfad" wird hier ebenfalls durch den Pfad des selektierten Ordners im Navigator ersetzt. Die nicht erreichbaren Knoten werden anschließend samt Zusatzinformationen direkt aus dem Model des Dependen-

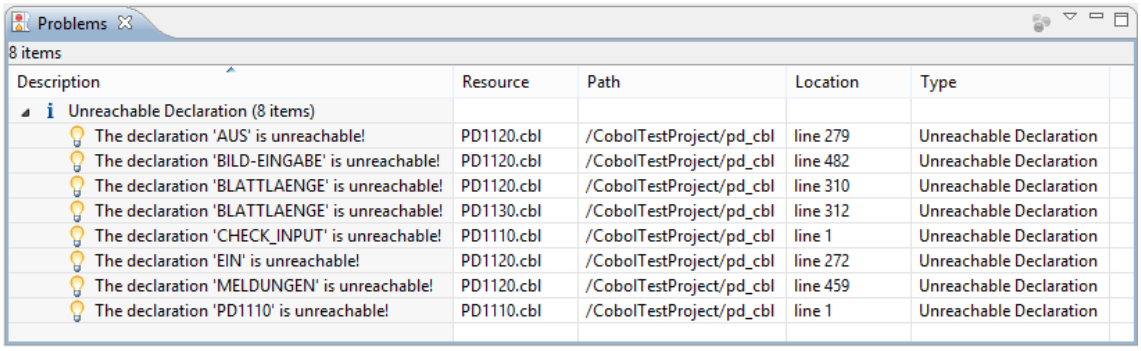

Abbildung 3.3: Visualisierung der nicht erreichbaren Knoten in der 1. Entwicklungsstufe

cyAnalyzers entnommen. Für jeden dieser Knoten wird anschließend in der in Eclipse enthaltenen "Problem"-View ein Marker angelegt (siehe Abb. 3.3). Jeder Marker enthält dabei die folgenden Informationen:

- eine Beschreibung
- die Datei, in der der nicht erreichbare Knoten enthalten ist
- den Pfad der Datei, in der der Knoten enthalten ist
- die Zeilennummer der Deklaration des Knoten
- die Typenbezeichnung

Die Marker zeigen bei Doppelklick das übliche Verhalten, es wird die in dem Marker hinterlegte Datei geöffnet und die entsprechende Zeile markiert.
Um mit dem Industriepartner die Umsetzung die Parametrisierung des DependencyAnalyzers über die Oberfläche diskutieren zu können, wurde ein Prototyp erstellt (siehe Abb 3.4. Diese Einstellungsoberfläche sollte ebenfalls über den Navigator aufgerufen werden können (siehe Abb. 3.2). Dabei sollte der selektierte Typ und der Pfad aus dem Navigator übernommen werden. Die einzelnen Argumente sollten aus dem DependencyAnalyzer extrahiert und in ExpandItems gruppiert werden. Es waren drei mögliche Ausgaben geplant: Die Generierung eines PNG-Bildes, die Markierung der nicht erreichbaren Knoten mit Marker in der "Problem"-View und die Graphvisualisierung mit Hilfe des DependencyVisualizers.

3.3.4 Evaluation

Nach der Erprobung der ersten Evolutionsstufe war dem Industriepartner eine enge Integration mit dem DependencyVisualizer sehr wichtig. Auf die Integration der vom DependencyAnalyzer angebotenen Exportfunktionen sollte weitgehend verzichtet werden. Lediglich die Anzeige von nicht erreichbarem Code als Marker und die Visualisierung des DependencyVisualizers seien relevant. In diesem Rahmen sollte auch der Zugriff auf die

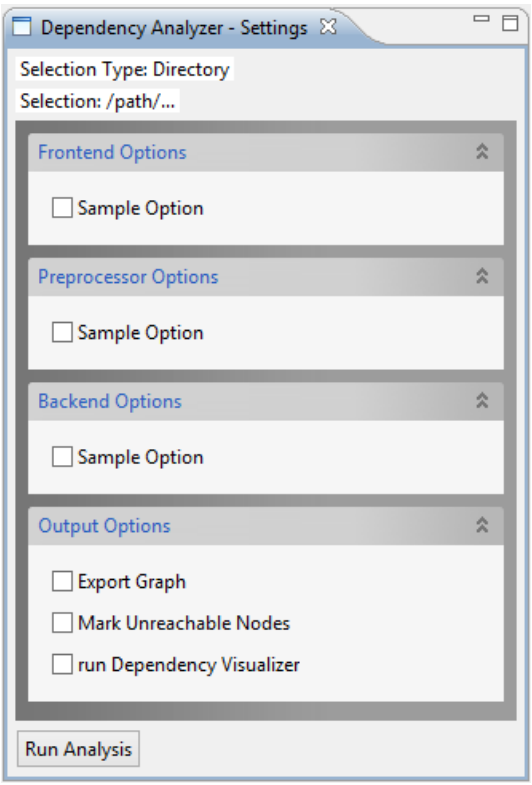

Abbildung 3.4: Prototyp der Oberfläche zur Parametrisierung des DependencyAnalyzers in der 1. Entwicklungsstufe

Einstellungsoberfläche der Argumente des DependencyAnalyzers in das Menü "Dependency Visualizer" in der Menüleiste verschoben werden. Die Priorisierung der verbleibenden Funktionen lautete wie folgt: Die Integration des DependencyAnalyzers sollte so generisch wie möglich gehalten werden. Eine Extraktion der verfügbaren Argumente aus dem DependencyAnalyzer wäre dabei ideal. Der nächste Schritt sollte die Umsetzung der Sprungmöglichkeiten zwischen dem Grapheditor des DependencyVisualizers und dem Quellcode sein. Als letztes sollte die Analyse von einer ausgewählten Variablen oder Section heraus gestartet werden können.

3.4 2. Evolutionsstufe

3.4.1 Planung

Ziel der zweiten Evolutionsstufe war zunächst die Aufarbeitung der Ergebnisse der Evaluation der ersten Evolutionsstufe. In diesem Rahmen wurde das Kommando "Show Settings View" in den Menüpunkt "Dependency Visualizer" in der Menüleiste verschoben und die

Exportmöglichkeit von Graphen als Bilder entfernt.
Als nächstes sollte nach einer Möglichkeit gesucht werden, alle verfügbaren Frontends und ihre zugehörigen Argumente aus dem DependencyAnalyzer zu extrahieren, um sie später in der Parametrisierungsoberfläche verwenden zu können.
Für die Umsetzung der Sprungmöglichkeiten von einer Variablen oder Section zu einem Knoten im Grapheditor und der Start der Analyse - ausgehend von einer Variablen oder Section - sollte ermittelt werden, wie Variable und Sections im COBOL-Quellcode als solche erkannt werden können. Einen Ansatz bot der NetCOBOL-Editor. Diese Informationen wurden von ihm bereits in der "Outline"-View zur Verfügung gestellt. Zudem sollte die Sprungmöglichkeit von einem Knoten im Grapheditor zu seiner Deklaration im Quellcode implementiert werden und Zusatzinformationen für Knoten im Grapheditor als Tooltip angezeigt werden.

3.4.2 Schwierigkeiten

Die Extraktion der Frontends und zugehöriger Argumente gestaltete sich schwieriger als zunächst vermutet. Der DependencyAnalyzer stellte diese Informationen über keine Schnittstelle nach außen zur Verfügung. Aus diesem Grund musste der Quellcode des DependencyAnalyzers überarbeitet werden.
Die Ermittlung des selektierten Typs im NetCOBOL-Editor war ebenfalls nicht möglich. Die Implementierung eines eigenen Parsers auf Tokenebene erschien dabei als einfachste Lösung.

3.4.3 Umsetzung

Für die Extraktion der Kommandozeilenargumente wurde der DependencyAnalyzer folgendermaßen angepasst: Es wurde eine Schnittstelle hinzugefügt, die die Frontends für den DependencyUtilizer zur Verfügung stellt. Dafür wurde die Klasse "FrontendInformation" angelegt. (siehe Listing 3.1).

Für die Auswertung der Kommandozeilenargumente wird im DependencyAnalyzer die Bibliothek "JArgs"[4] verwendet. Diese Bibliothek verwaltet intern alle hinzugefügten Argumente, stellt einige dieser Informationen jedoch nicht wieder zur Verfügung. Insbesondere der Zugriff auf die erwarteten Parameter der Argumente ist nicht möglich. Aus diesen Gründen wurde von der itestra GmbH diese Klasse bereits in Form der Klasse "ExtendenCmdLineParser" so erweitert, dass die Verwaltung von Hilfetexten möglich war. In diesen "ExtendenCmdlLineParser" wurde nun die Funktionalität hinzugefügt, um alle benötigten Informationen zu ermitteln und sie ebenfalls über eine Schnittstelle zur Verfügung zu stellen (siehe Listig 3.2). Alle so ermittelten Argumente werden ihrer Kategoriezugehörigkeit nach zurückgegeben. Ein Objekt der Klasse "OptionHelp" enthält das Argument, den zugehörigen Hilfetext und den Typ des erwarteten Parameters.

[4] http://jargs.sourceforge.net

Listing 3.1 Beschreibung der Klasse "FrontendInformation" für den Austausch von Frontend-Informationen

```
 * This class is used by the DependencyRunner to provide
 * frontends to external plugins.
 *
 * @author Fabian Mueller
 *
 */
public class FrontendInformation
{
        /**
         * Contains the commandline-parameter
         */
        public String parameter;

        /**
         * Contains the frontend-description
         */
        public String description;

        /**
         * Contains all file extensions that are associated
         * with this frontend
         */
        public List<String> fileExtensions;
}
```

Listing 3.2 Signatur der Methode zur Ermittlung der Argumente eines Frontends

```
/**
* Aquires all available commandline options for the given frontend
* and general backend options.
*
* @param frontendName
* @return <Category, HelpOptions>
*/
public static SortedMap<String, TreeSet<OptionHelp>> getCmdLineOptionsByFrontend(String
    frontendName);
```

Für die Sprungmöglichkeit von einem Knoten im Grapheditor zu seiner Deklaration im Quellcode wurde zunächst das Kontextmenü des Grapheditors um das Kommando "Go to Deklaration" erweitert (siehe Abb. 3.5). Um den den entsprechenden Codeabschnitt zu Kennzeichnen, wurde ein Marker auf der im Knoten des Graphen hinterlegten Resource erstellt. Nach der Durchführung des Sprungs wird das Markerobjekt wieder entfernt.

Um zu ermitteln, ob es sich bei dem selektierten Token im NetCOBOL-Editor um eine Variable oder Section handelt, wurde ein Parser auf Tokenebene implementiert. Dieser Parser grenzt zunächst das selektierte Token ab. Die Leerzeichen vor und nach dem Token dienen dabei als Grenzpunkte. Anschließend wird aus dem Kontext abgeleitet, ob es sich um eine Variable oder Section handelt. In Rücksprache mit dem Industriepartner wurden dabei folgende Richtlinien festgelegt: Falls vor dem Token das Schlüsselwort "'PERFORM" steht

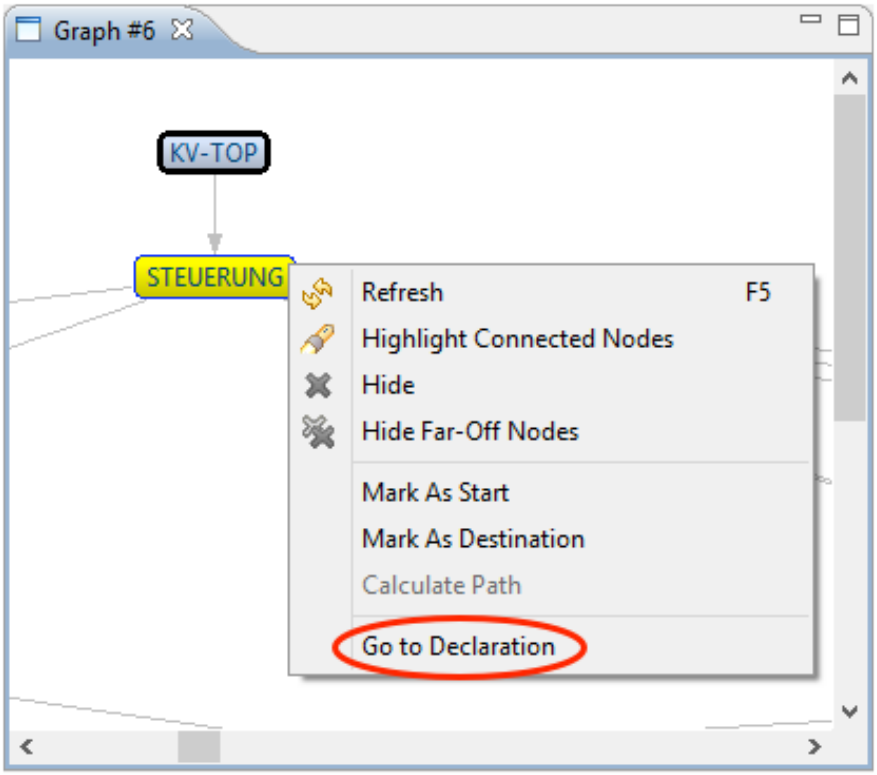

Abbildung 3.5: Kommando “Go to Declaration“ im Kontextmenü des Grapheditors

oder dem Token das Schlüsselwort “SECTION“ folgt, handelt es sich um eine Section. Alle weiteren Tokens werden als Variable interpretiert.

Um Zusatzinformationen zu den einzelnen Knoten im Grapheditor erhalten zu können, wurden entsprechende Tooltips hinzugefügt. Diese Tooltips enthalten die folgenden Informationen:

- die ID des Knotens
- den Name des Knotens
- die Zeilennummer, in der der Knoten deklariert wurde
- der Pfad der Datei, in der der Knoten deklariert wurde
- die Scope des Knotens

3.4.4 Evaluation

Dem Industriepartner hat die zweite Evolutionsstufe sehr gut gefallen. Alle Änderungen am DependencyAnalyzer wurden in die firmeninterne Versionsverwaltung eingecheckt. Für die Einblendung von Zusatzinformationen gab es einen Verbesserungswunsch. Diese sollten neben dem Tooltip auch in der bereits existenten "Properties"-View angezeigt werden.

3.5 3. Evolutionsstufe

3.5.1 Planung

In der letzten Evolutionsstufe sollte zunächst das Ergebnis der Evaluation aus der zweiten Evolutionsstufe umgesetzt werden. Dazu sollte neben den Tooltips auch die "SettingsView zur Darstellung von Zusatzinformationen im Grapheditor verwendet werden. Nachdem nun die Ermittlung von Tokens in dem NetCOBOL-Editor möglich war, sollte die Sprungmöglichkeit von Sections und Variablen zu einem Knoten im Grapheditor und die Analyse ausgehend von einer Variablen oder Section ermöglicht werden. Des Weiteren sollten die neuen Schnittstellen des DependencyAnalyzers genutzt werden, um die Argumente in der Einstellungsoberfläche auswählen und parametrisieren zu können.

3.5.2 Schwierigkeiten

Bei der Umsetzung der Sprungmöglichkeit von einer Variablen oder Section zu einem Grapheditor musste beachtet werden, dass eine Section oder Variable in mehreren Graphen enthalten sein kann. Für den Benutzer sollte es dabei möglich sein, den gewünschten Grapheditor auszuwählen.
Bei der Oberfläche zur Einstellung der Argumente war eine ansprechende Darstellung mit der ursprünglichen Architektur nicht zu erreichen. Da jedes Argument durch ein Composite mit einem eigenen LayoutManager repräsentiert wurde, könnten die einzelnen Oberflächenkomponenten nicht einheitlich ausgerichtet werden (siehe Abb. 3.6).
Für die automatische Generierung der Oberfläche aus den verfügbaren Argumenten sollte der Layoutmanager "RowLayout" [5] des SWT-Frameworks verwendet werden. Für jedes Argument sollte dazu ein eigenes Composite angelegt werden, welches entsprechend des zu erwartenden Parametertyps eine Eingabemöglichkeit zur Verfügung stellt. Im Konstruktor sollte den Objekten, die dieses Composite enthalten, das übergeordnete Composite übergeben werden, um das Composite hinzufügen zu können. Um die Ermittlung der Argumente und ihre Parameter zu vereinheitlichen, sollten diese Verwaltungsobjekte das Interface "IOptionComposite" (siehe Listig 3.3) implementieren.

[5] http://www.eclipse.org/articles/article.php?file=Article-Understanding-Layouts/index.html

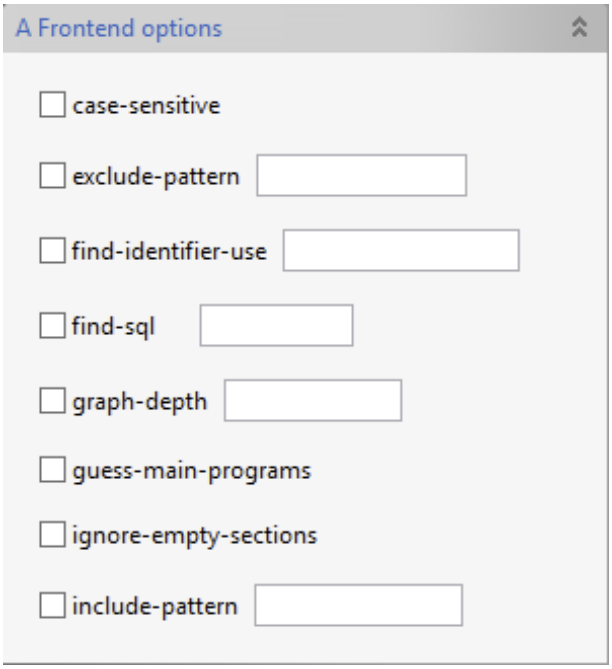

Abbildung 3.6: Probleme mit dem Layout bei der Repräsentation eines Arguments durch ein Composite in einem ExpandItem

Listing 3.3 Interface für die Verwaltung der datentypspezifischen Parametrisierung

```
/**
 * This interface is used for the consistent access to
 * arguments with different parameter-datatypes.
 *
 * @author Fabian Mueller
 *
 */
public interface IOptionComposite
{
	/**
	 * Generates the commandline-parameters of
	 * the represented argument.
	 * @return commandline argument with parameters
	 */
	public abstract List<String> getArgs();
}
```

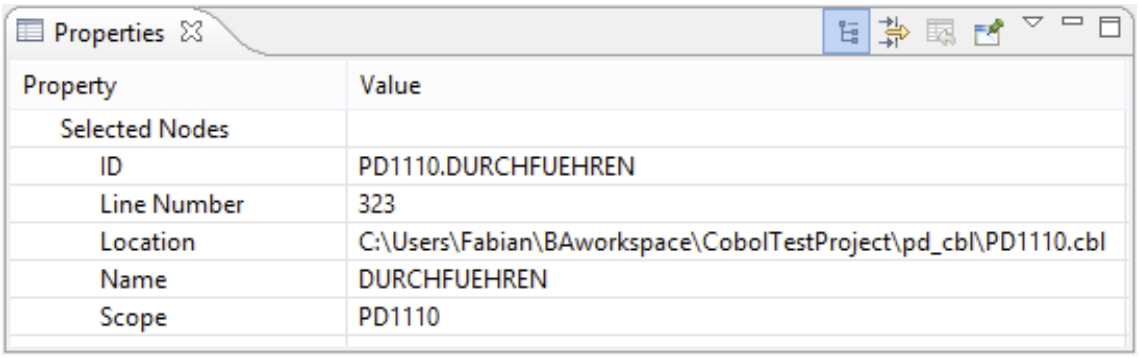

Abbildung 3.7: Anzeige von Zusatzinformationen eines Knoten in der "Properties"-View

Die Visualisierung des Ergebnisses der Analyse, ausgehend von einer Variablen oder Section, sollte als Marker erfolgen. Da die bisher verwendete "Problem"-View als Anzeige der Marker hierfür nicht passend ist, musste eine Alternative entwickelt werden.

3.5.3 Umsetzung

Zunächst wurden die Ergebnisse der Evaluation aus der vorherigen Evaluationsstufe umgesetzt. Für die Anzeige der Zusatzinformationen eines Knotens wird nun neben dem Tooltip die "Properties"-View verwendet (siehe Abb. 3.7).

Für die Umsetzung der Sprungmöglichkeit wird auf den in der zweiten Evolutionsstufe entwickelten Parser zurückgegriffen. Der Name des Tokens und der Dateiname der Datei, in der das Token enthalten ist, dienen dazu alle Grapheditoren zu ermitteln, in denen das Token als Knoten vorliegt. Im Kontextmenü des NetCOBOL-Editors werden diese Grapheditoren anschließend aufgelistet, so dass der Benutzer den gewünschten Editor auswählen kann (siehe Abb. 3.8).

Für den Aufruf des DependencyAnalyzers - ausgehend von einer Variablen oder Section - wird ebenfalls der Parser aus der vorherigen Evolutionsstufe verwendet. Nachdem ein Token ausgewählt wurde, stehen zwei verschiedene Kommandos im Kontextmenü des NetCOBOL-Editors bereit. Das Kommando "Reachable From Here" ermittelt alle Knoten, die von diesem Token aus erreichbar sind; das Kommando "Is Reached From" sucht diejenigen Knoten, von denen aus dieses Token erreichbar ist. Die Parametrisierung des DependencyAnalyzers lautet dabei wie folgt:

- bei gewählter Variable: `cob-var -input-dir Pfad`
- bei gewählter Section: `cob -input-dir Pfad`

"'Pfad" wird dabei durch den übergeordneten Ordner im Datensystem ersetzt. Die gewünschte Knotenmenge wird anschließend direkt aus dem Model des DependencyAnalyzers ermittelt. Dazu wird der Startknoten anhand des Tokennames und der Name der Datei,

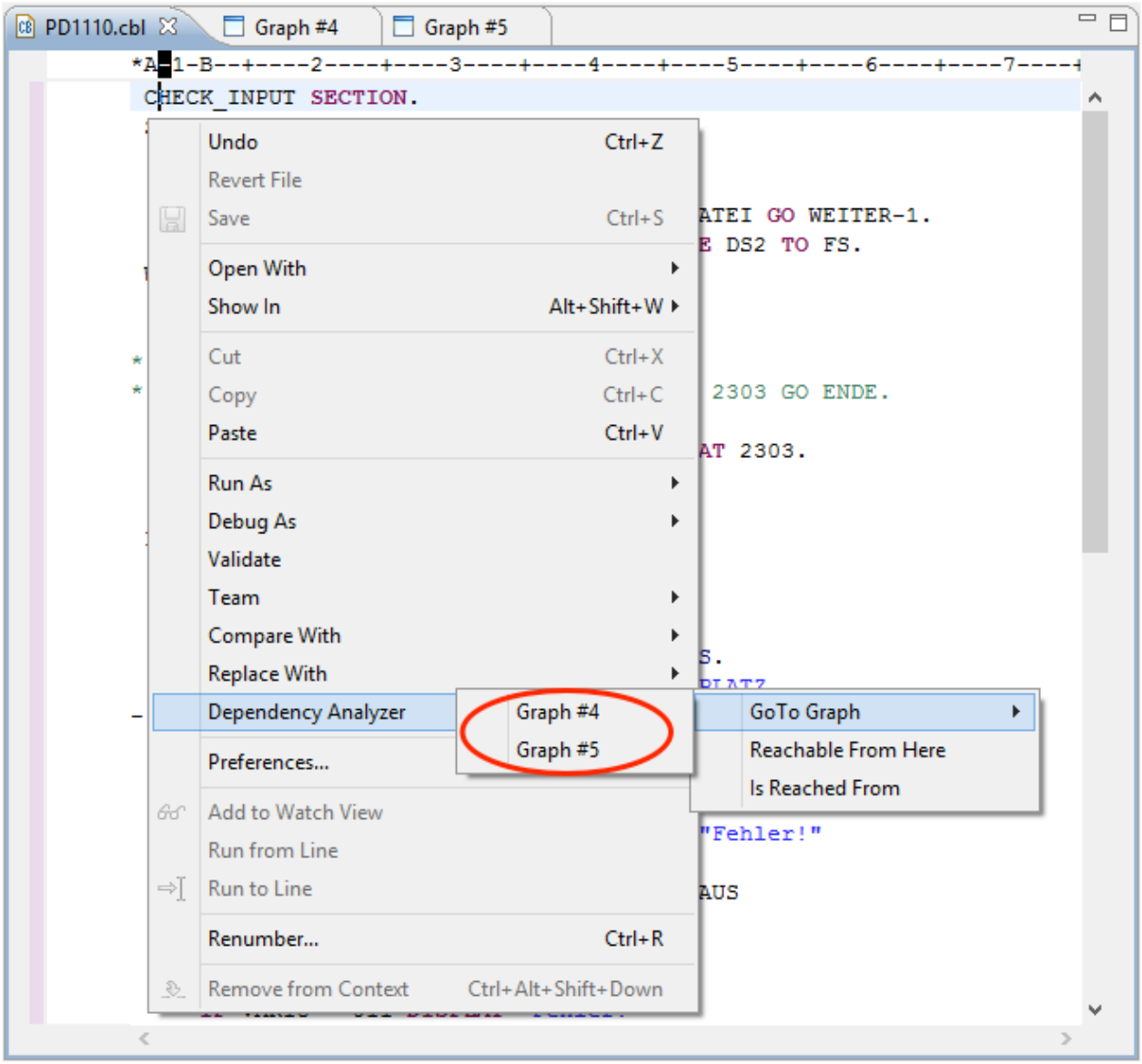

Abbildung 3.8: Kontextmenü des NetCOBOL-Editors mit Sprungmöglichkeit zu mehreren Graphen

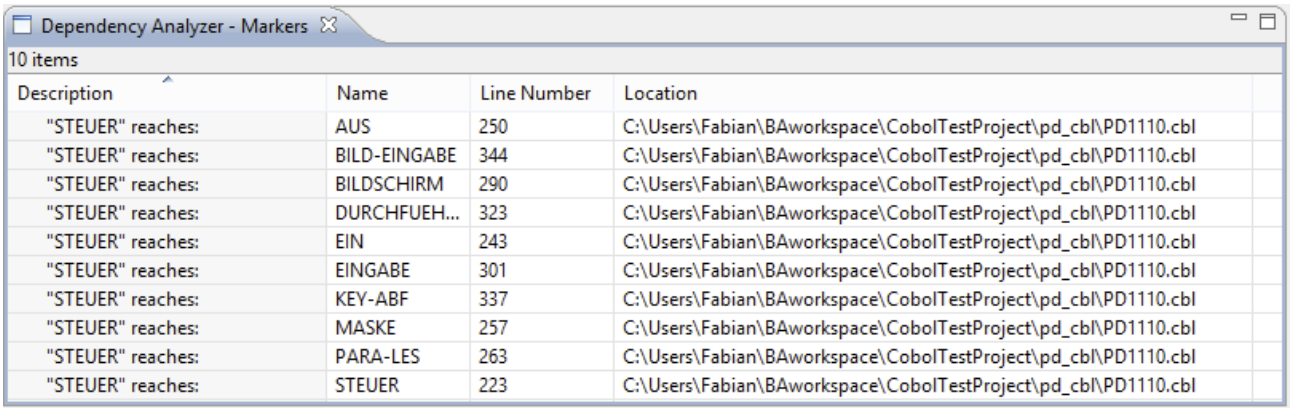

Abbildung 3.9: Visualisierung der erreichbaren Knoten in der “DependencyAnalyzer - Markers“ - View

in der sich das Token befindet, im Model identifiziert. Über entsprechende Methoden des DependencyAnalyzers wird nun, nach Übergabe des gefundenen Startknotes, das Ergebnis ermittelt. Für die Visualisierung der Ergebnisse wurde die neue Marker-View “Dependency Analyzer - Markers“ implementiert (siehe Abb 3.9. In dieser View werden nun alle vom DependencyUtilizer generierten Marker angezeigt.

Die Implementierung der Einstellungsoberfläche der Argumente konnte wie geplant umgesetzt werden. Für jeden unterschiedlichen Datentyp eines Parameters wurde ein Verwaltungsobjekt angelegt. Diese Verwaltungsobjekte enthielten zunächst ein Composite mit Oberflächenelementen, das in ExpandItems gruppiert visualisiert wurde. Jedes Composite verfügt dabei über eine Checkbox, nach dessen Aktivierung dieses Argument erzeugt werden kann. Boolsche Parameter haben keine weiteren Elemente, String- und Integerparameter verfügen zusätzlich über ein geeignetes Eingabefeld. Nach anfänglichen Problemen wurde der RowLayout-Manager durch den GridLayout-Manager ersetzt. Dadurch mussten allerdings die Verwaltungsobjekte der Argumente umstrukturiert werden. Die einzelnen Oberflächenelemente konnten nicht mehr auf einem intern verwalteten Composite angebracht werden, sondern wurden direkt auf dem ExpandItem platziert. So konnte das Layout ansprechender gestaltet werden (siehe Abb. 3.10).

Neben den automatisch generierten ExpandItems wurden zwei weitere hinzugefügt, “Special Options“ und “CmdLine Arguments“ (siehe Abb. 3.11). In den “Special Options“ wurden bestimmte Argumente aufgenommen, die ihre Parameter direkt aus der Eclipse-IDE beziehen. Dazu gehören die Argumente `-input-dir` und `-expand-includes-from`. Beide Argumente erhalten nach Aktivierung den Pfad der aktuellen Selektion im Navigator als Parameter. Im ExpandItem ”’CmdLine Arguments“ können alle Argumente generiert und manuell editiert werden. Zusätzlich wird die Kommandozeilenausgabe des DependencyAnalyzers angezeigt.

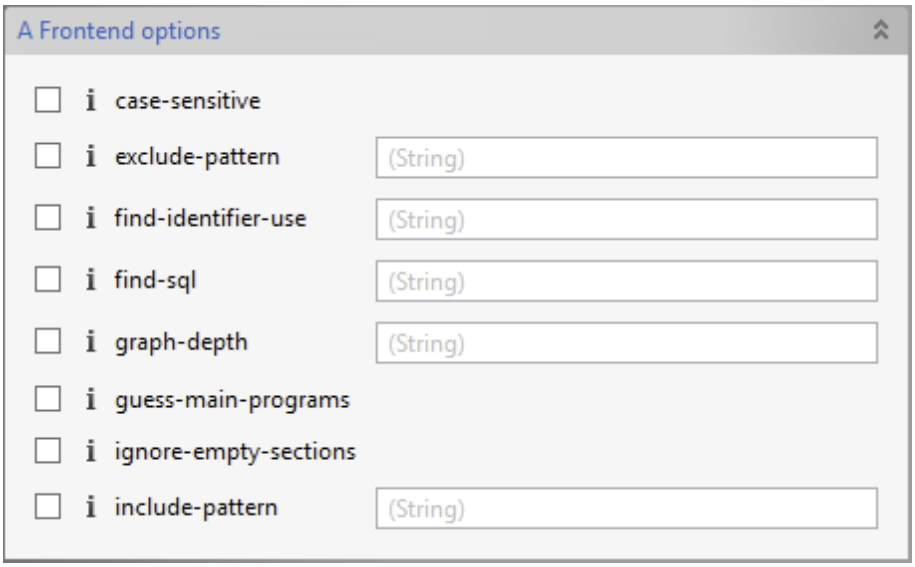

Abbildung 3.10: Repräsentation eines Arguments in einem ExpandItem

Die Optionen für die Anzeige können unterhalb der ExpandItems eingestellt werden (siehe Abb. 3.11). Es besteht die Möglichkeit, den nicht erreichbaren Code in der "DependencyAnalyzer - Makers"-View zu visualisieren oder den DependencyVisualizer zu starten.

3.6 Retrospektive

Die Entwicklung des DependencyUtilizers verlief weitgehend problemlos, so dass die ursprüngliche Zeitplanung eingehalten werden konnte. Aufgetretene Probleme des DependencyAnalyzers oder offene Fragen bezüglich der Entwicklung konnten immer schnell mit dem Industriepartner geklärt werden. Verzögerungen einzelner Meilensteine kamen somit kaum vor oder konnten durch eingeplante Zeitpuffer wieder relativiert werden.

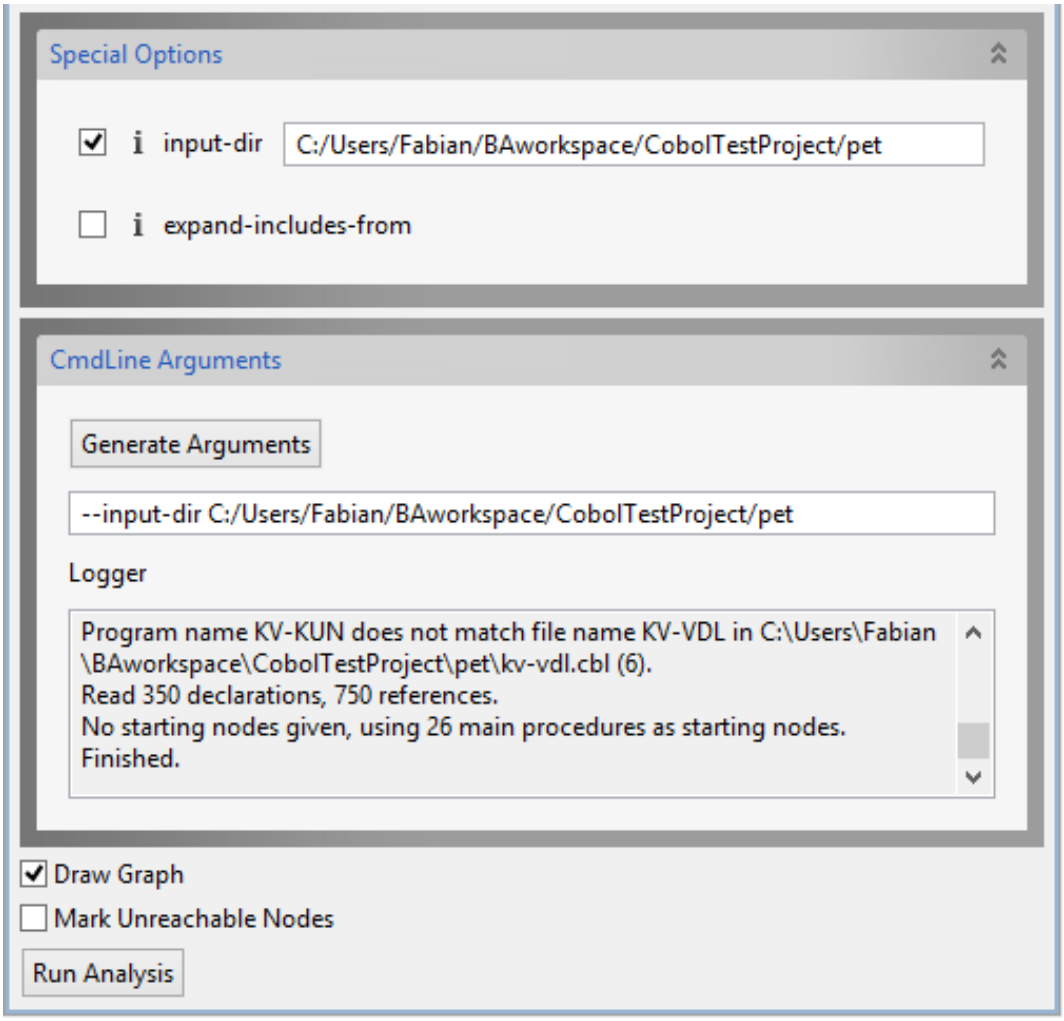

Abbildung 3.11: ExpandItems “Special Options“ und “CmdLine Arguments“ in der Oberfläche für die Parametrisierung des DependencyAnalyzers

4 Evaluation

Ziel der Abschlussevaluation des DependecyUtilizers war es, die entwickelten Funktionen auf ihre Einsatzfähigkeit und Usability zu testen sowie weitere Funktionen zu identifizieren, die für den Einsatz sinnvoll wären. Die Umsetzung der Ergebnisse ist allerdings nicht Teil dieser Bachelorarbeit. Sie dient dem Industriepartner als Hilfestellung für die Weiterentwicklung.

4.1 Durchführung

Die Evaluation wurde zusammen mit drei Mitarbeitern der itestra GmbH durchgeführt. In Einzelgesprächen wurde die Funktionsweise Schritt für Schritt erklärt, so dass die Mitarbeiter jede einzelne Funktion parallel an ihrem Arbeitsplatz testen konnten. Anschließend wurden Meinungen und Wünsche eingeholt, die am Ende dieses Kapitels zusammengefasst werden.

4.2 Fragenkatalog

Die in diesem Fragenkatalog aufgeführten Fragen dienten als Leitlinie für die Evaluation.

Navigator Schnellzugriff - Graph zeichnen

- Wird die Funktionalität benötigt?
- Wie häufig wird die Funktion verwendet werden?
- Ist die Funktion im richtigen Menü an der richtigen Stelle platziert?
- Ist diese Funktion insgesamt zufriedenstellend?

Navigator Schnellzugriff - Nicht erreichbaren Code anzeigen

- Wird die Funktionalität benötigt?
- Wie häufig wird die Funktion verwendet werden?
- Ist die Funktion im richtigen Menü an der richtigen Stelle platziert?
- Ist diese Funktion insgesamt zufriedenstellend?

Einstellungsoberfläche der Parameter

- Wird die Funktionalität benötigt?
- Wie häufig wird die Funktion verwendet werden?
- Befindet sich der Menüpunkt im richtigen Menü und an der richtigen Stelle?
- Ist die Konfiguration der Eingabeparameter verständlich?
- Wird die manuelle Manipulation der Eingabeparameter benötigt?
- Fehlen wichtige Parameter oder Frontends?
- Enthält das Expanditem “Special Options“ alle benötigten Features?
- Werden zwingend weitere Funktionalitäten benötigt?
- Ist diese Funktion insgesamt zufriedenstellend?

Eigenschaftenansicht des Grapheditors

- Wird diese Funktion benötigt?
- Wie häufig wird die Funktion verwendet werden?
- Sind die Zusatzinformationen in der Properties-View richtig aufgehoben?
- Werden die vorhandenen Zustzinformationen benötigt?
- Werden weitere Zusatzinformationen benötigt?
- Ist diese Funktion insgesamt zufriedenstellend?

Sprungmöglichkeiten von einem Codefragment zu einem Grapheditor

- Wird die Sprungmöglichkeit von einem Codefragment zum Graphen benötigt?
- Wie häufig wird die Sprungmöglichkeit verwendet werden?
- Befindet sich der Menüpunkt im richtigen Menü an der richtigen Stelle?
- Ist diese Funktion insgesamt zufriedenstellend?

Sprungmöglichkeiten von einem Knoten im Grapheditor zum Quellcode

- Wird die Sprungmöglichkeit von einem Knoten im Grapheditor zur entsprechenden Stelle im Quellcode benötigt?
- Wie häufig wird die Sprungmöglichkeit verwendet werden?
- Befindet sich der Menüpunkt im richtigen Menü an der richtigen Stelle?
- Ist diese Funktion insgesamt zufriedenstellend?

Suche von Knoten, die von gewählter Variable erreichbar sind

- Wird diese Aufrufmöglichkeit des DependencyAnalyzers benötigt?
- Wie häufig wird die Funktion verwendet werden?
- Befindet sich der Menüpunkt im richtigen Menü an der richtigen Stelle?
- Entspricht das Ergebnis der Analyse den Erwartungen?
- Ist diese Funktion insgesamt zufriedenstellend?

Suche von Knoten, von denen die gewählte Variable erreichbar ist

- Wird diese Aufrufmöglichkeit des DependencyAnalyzers benötigt?
- Wie häufig wird die Funktion verwendet werden?
- Befindet sich der Menüpunkt im richtigen Menü an der richtigen Stelle?
- Entspricht das Ergebnis der Analyse den Erwartungen?
- Ist diese Funktion insgesamt zufriedenstellend?

Suche von Knoten, die von gewählter Section erreichbar sind

- Wird diese Aufrufmöglichkeit des DependencyAnalyzers benötigt?
- Wie häufig wird die Funktion verwendet werden?
- Befindet sich der Menüpunkt im richtigen Menü an der richtigen Stelle?
- Entspricht das Ergebnis der Analyse den Erwartungen?
- Ist diese Funktion insgesamt zufriedenstellend?

Suche von Knoten, von denen die gewählte Section erreichbar ist

- Wird diese Aufrufmöglichkeit des DependencyAnalyzers benötigt?
- Wie häufig wird die Funktion verwendet werden?
- Befindet sich der Menüpunkt im richtigen Menü an der richtigen Stelle?
- Entspricht das Ergebnis der Analyse den Erwartungen?
- Ist diese Funktion insgesamt zufriedenstellend?

Anzeige von Markern

- Ist die Anzeige der Quellcodemarkern in der DependencyUtilizer-Marker-View verständlich?
- Sind alle benötigten Attribute der Marker vorhanden?
- Werden weitere Markerattribute benötigt?
- Werden weitere Funktionen in der Marker-View benötigt?
- Ist diese Funktion insgesamt zufriedenstellend?

Allgemein

- Fehlen wichtige Funktionen, die den Einsatz des DependencyUtilizers verhindern?
- Gibt es kritische Fehler, die den Einsatz des DependencyUtilizers verhindern?
- Ist der Einsatz des DependencyUtilizers in Zukunft vorstellbar?
- Welche weiteren Funktionen wären wünschenswert?
- Ist diese Funktion insgesamt zufriedenstellend?

4.3 Ergebnis

Der generelle Eindruck des DependencyUtilizers bei der Evaluation war durchweg sehr positiv. Alle befragten Mitarbeiter können sich vorstellen, den DependencyUtilizer in Zukunft für ihre Projekte einzusetzen und würden den Einsatz weiterempfehlen. Es folgt eine Auflistung genauerer Angaben zu den einzelnen Funktionen.

Schnellzugriffe im Navigator

Die Funktionen im Kontextmenü des Navigators wurden als sehr sinnvoll erachtet. Die Mitarbeiter sehen darin die folgenden Vorteile: Es lässt sich ohne Parameterkonfiguration ein Graph generieren und nicht erreichbarer Code identifizieren, um sich einen Überblick über die Quellcodestrukturen zu verschaffen. Die Auswahl des Ordners ersetzt schon zu großen Teilen die Filterung der Knoten im Graph. Dadurch versprechen sich die Mitarbeiter einen deutlichen Zeitgewinn.

Einstellungsoberfläche der Parameter

Die Einstellungsoberfläche der Parameter empfinden die Mitarbeiter als wichtige Funktion. Der Schritt von der Kommandozeilenkonfiguration hin zu einer leicht zu verstehenden graphischen Oberfläche soll insbesondere neuen Benutzern den Einstieg erleichtern.
Folgende Funktionen würden sich die Mitarbeiter wünschen, um die Bedienung weiter zu vereinfachen. Die Hilfetexte werden momentan als Tooltip eines jeden Kommandozeilenarguments eingeblendet. Die Anzeigedauer ist bei längeren Hilfetexten jedoch zu kurz, um ihn vollständig lesen zu können. In so einem Falle muss der Benutzer die Maus erneut über den Kommandozeilenschalter bewegen, um den Hilfetext erneut einzublenden. Aus diesem Grund wünschen sich die Mitarbeiter eine Anzeige der Hilfetexte ohne zeitlich begrenzte Anzeigedauer.
Für die Eingabe der Parameter von String-Argumente sind folgende Erweiterungen erwünscht: Da viele String-Parameter Pfade als Eingabe haben, könnte hinter jedem dieser Eingabefelder ein FileChooser angebracht werden um die Eingabe zu erleichtern. Des Weiteren könnten für die Konfiguration der Pfadangaben relativ zu dem aktuellen Workbench-Pfad Eclipse-Umgebungsvariablen verwendet werden. Beispielsweise könnte die Standardparametrisierung des Eingabepfads das zugehörige Projekt der aktuellen Datei im Editor sein. Dadurch ließe sich auch die Zuordnung des Eingabepfads zu den “Special Options“ vermeiden.
Für das ExpandItem “Special Options“ konnte eine weitere wichtige Funktion identifiziert werden. Der DependencyAnalyzer bietet die Möglichkeit, bestimmte Strings zu ersetzten. Dies geschieht über das Argumentpaar “replace“ und “by'", welches beliebig oft verwendet werden kann. Da aus dem DependencyAnalyzer diese Information nicht extrahiert werden kann, müsste diese Funktion separat implementiert werden.
Kleinere Verbesserungen an dem Layout wäre ebenfalls wünschenswert. Bei kleineren

Auflösungen kann es passieren, dass durch das Aufklappen zu vieler ExpandItems das ExpandItem “Special Options“ nicht mehr sichtbar ist.

Zusatzinformationen von Knoten im Grapheditor

Die Einblendung von Zusatzinformationen ist in bestimmten Situationen durchaus sinnvoll, die bei der Evaluation anwesenden Mitarbeiter werden diese Funktion aber seltener einsetzen. Die Anzeige dieser Informationen in der Properties-View ist der Einblendung über Tooltips aber auf jeden Fall vorzuziehen. Keinem der Mitarbeiter haben wichtige Informationen gefehlt und alle wichtigen Funktionen wurden angezeigt.

Sprungmöglichkeiten zwischen Code und Grapheditor

Die Sprungmöglichkeit ist für alle Mitarbeiter eine wichtige Funktion, die auch häufiger zum Einsatz kommen wird. Folgende Erweiterungen wurden dabei vorgeschlagen. Die Sprungfunktionen sind zur Zeit nur über die entsprechenden Kontextmenüs erreichbar. Eine Einführung von Hotkeys wäre an dieser Stelle angebracht.
Das Kontextmenü im Coboleditor enthält ein weiteres Menü, in welchem alle Grapheditoren aufgelistet sind, die den ausgewählten Knoten enthalten. Sofern der ausgewählte Knoten nur in einem Graphen vorhanden ist, sollte die Sprungmöglichkeit direkt unter dem Punkt “DependecyAnalyzer“ ohne Editorauswahl zur Verfügung stehen.

Erreichbarkeitsanalyse ausgehend von Variablen und Sections

Der Start des DependencyAnalyzers, ausgehend von einer Variable oder Section, ist eine sehr wichtige Funktion, die häufig zum Einsatz kommen wird. Die Integration in den NetCOBOL-Editor empfanden alle anwesenden Entwickler als richtig. Bei der Platzierung der Marker für die gefundenen Sections und Variablen gab es den Vorschlag, diese in die bereits vorhandene “Search“-View zu integrieren. Hier ließen sich die Ergebnisse von mehreren Suchanfragen zusammenfassen. Die Mitarbeiter wünschten sich zudem eine weitere Spalte, in der angegeben wird, ob der Knoten direkt oder transitiv erreichbar ist.
Die Analyse, ausgehend von einer Variablen, entspricht nicht den Erwartungen. Aktuell ist nur bei einer Redefinition eine Verbindung zwischen zwei Variablen vorhanden. Es wäre allerdings sinnvoller, nur eine Funktion anzubieten, die alle Codesegmente auflistet, die die selektierte Variable verwenden.

5 Handbuch

5.1 Einleitung

Dieses Handbuch bietet einen Einstieg in die Benutzung des DependencyUtilizers. Bei dem DependencyUtilizer handelt es sich um ein Eclipse-Plugin, welches die Funktionalitäten des DependencyAnalyzers der itestra GmbH und des DependencyVisualizers erweitert und komfortabel über entsprechende Menüs in der Eclipse-IDE anbietet. Dieses Handbuch enthält eine Installationsanleitung sowie eine vollständige Funktionsbeschreibung.

5.2 Installation

Um den DependencyUtilizer nutzen zu können, muss das Plugin in den Ordner `/plugins/` des Eclipse-Verzeichnisses kopiert werden. Gegebenenfalls müssen die folgenden Abhängigkeiten ebenfalls installiert werden.

- de.nolleryc.dependencyvisualizer
- de.nolleryc.dependencymodeladapter
- de.itestra.conqat.dependency
- de.itestra.lib3rd
- de.itestra.swgcommons
- NetCOBOL-Editor[1]

Für die Installation des DependencyVisualizers muss das "Graphical Editing Framework Zest Visualization Toolkit SDK" in Eclipse installiert werden. Anschließend stehen die Erweiterungen in der Eclipse-IDE zur Verfügung. Falls Eclipse zum Zeitpunkt der Installation geöffnet war, ist zuvor ein Neustart notwendig.

[1] http://www.fujitsu.com/global/services/software/netcobol/

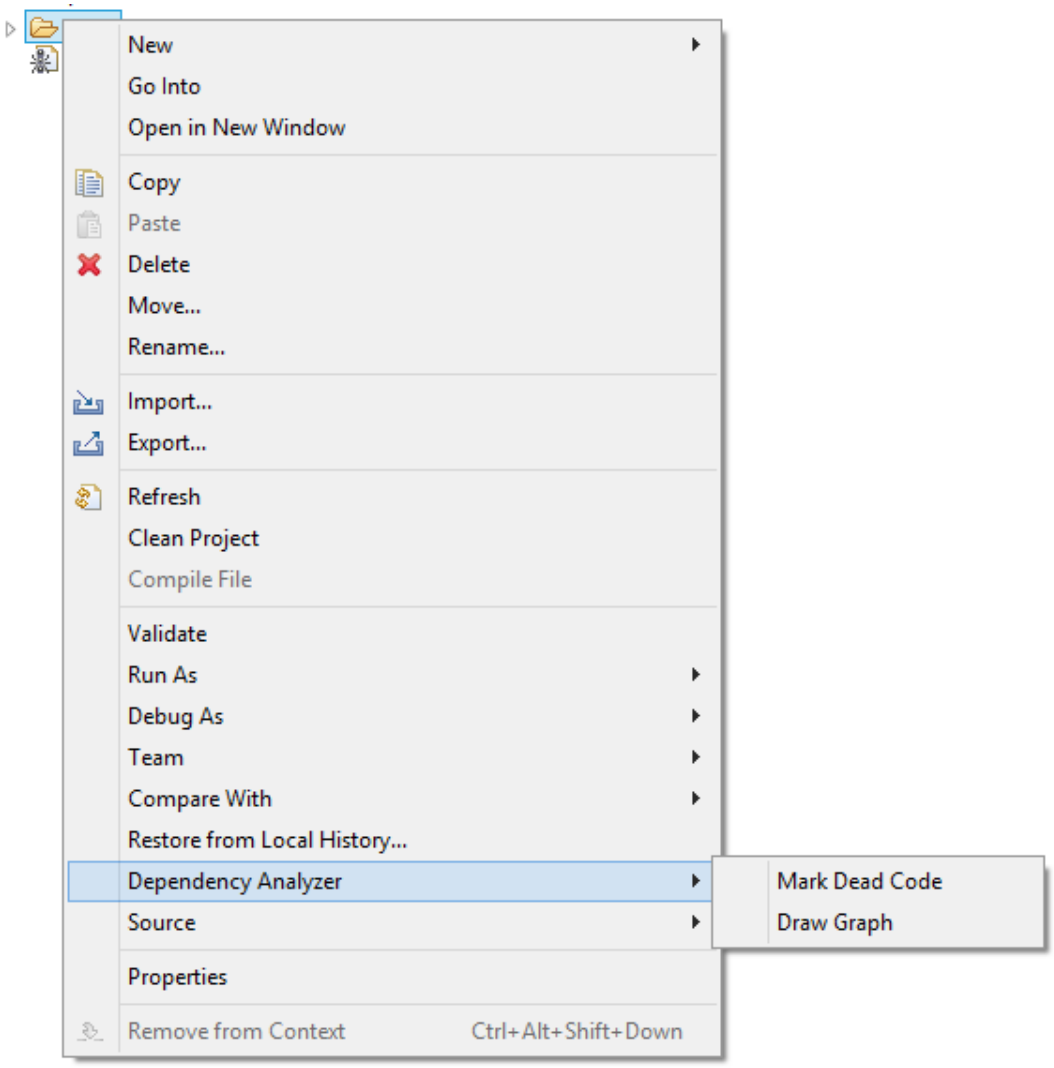

Abbildung 5.1: Erweiterungen des Kontextmenüs im Navigator

5.3 Funktionen

5.3.1 Navigator Schnellzugriff

Im Kontextmenü des Navigators stehen die Funktionen für das Zeichnen von Graphen und die Markierung von nicht erreichbaren Knoten zu Verfügung.

Diese Funktionen sind nach einem Rechtsklick auf einen Ordner im Menü "Dependency-Analyzer" zugänglich (siehe Abb. 5.1). Das Model des DependencyAnalyzers wird dabei mit folgendem Aufruf generiert:

```
cob -input-dir Pfad
```

Dependency Analyzer - Markers

8 items

Description	Name	Line Number	Location
Unreachable declaration:	AUS	279	C:\Users\Fabian\runtime-EclipseApplication\CobolTestProject\pd_cbl\PD1120.cbl
Unreachable declaration:	BILD-EINGABE	482	C:\Users\Fabian\runtime-EclipseApplication\CobolTestProject\pd_cbl\PD1120.cbl
Unreachable declaration:	BLATTLAENGE	310	C:\Users\Fabian\runtime-EclipseApplication\CobolTestProject\pd_cbl\PD1120.cbl
Unreachable declaration:	BLATTLAENGE	312	C:\Users\Fabian\runtime-EclipseApplication\CobolTestProject\pd_cbl\PD1130.cbl
Unreachable declaration:	CHECK_INPUT	1	C:\Users\Fabian\runtime-EclipseApplication\CobolTestProject\pd_cbl\PD1110.cbl
Unreachable declaration:	EIN	272	C:\Users\Fabian\runtime-EclipseApplication\CobolTestProject\pd_cbl\PD1120.cbl
Unreachable declaration:	MELDUNGEN	459	C:\Users\Fabian\runtime-EclipseApplication\CobolTestProject\pd_cbl\PD1120.cbl
Unreachable declaration:	PD1110	1	C:\Users\Fabian\runtime-EclipseApplication\CobolTestProject\pd_cbl\PD1110.cbl

Abbildung 5.2: Markeransicht des DependencyUtilizers

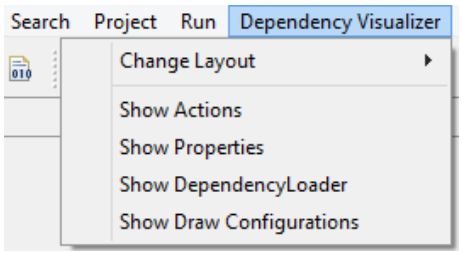

Abbildung 5.3: Aufruf der Einstellungsoberfläche der Parameter

"Pfad" wird dabei durch den Pfad des Ordners ersetzt, auf dem das Kontextmenü geöffnet wurde. Eine weitergehende Konfiguration der Optionen und Parameter ist hierbei nicht möglich.
Durch das Kommando "Mark Dead Code" werden alle Knoten, die in dem Model des Dependance Analyzers als "nicht erreichbar" markiert wurden, in der "Dependency Analyzer - Markers"-View als solche angezeigt (siehe Abb. 5.2)). Durch das Kommando "Draw Graph" wird der Grapheditor des DependencyVisualizers mit dem Model des DependencyAnalyzers gestartet.

5.3.2 Einstellungsoberfläche der Parameter

Um den Aufruf des DependencyAnalyzers zu parametrisieren, steht die "Dependency Analyzer - Settings"-View zur Verfügung. Diese ist über den Menüpunkt "Dependency Visualizer" im Menü zu erreichen (siehe Abb. 5.3).

Sobald diese View gestartet wurde, kann das gewünschte Frontend aus dem ExpandItem "Frontends" ausgewählt werden. Im Anschluss daran werden alle verfügbaren Argumente aus dem DependencyAnalyzer extrahiert und in ExpandItems gruppiert. Um ein Argument zu verwenden, muss die entsprechende CheckBox markiert werden. String- und Integeroperationen haben zusätzlich ein Textfeld, in das die benötigten Argumente eingetragen werden können.

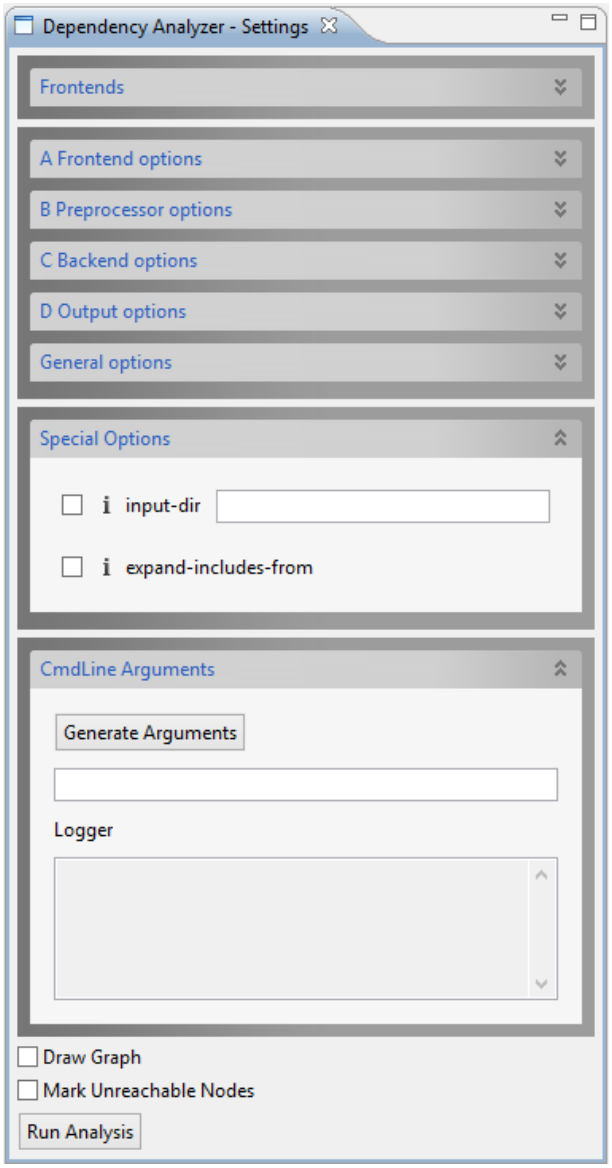

Abbildung 5.4: Einstellungsoberfläche der Parameter

Das ExpandItem “Special Options“ enthält zwei Argumente, die aus ihrer ursprünglichen Gruppierung entnommen wurden und spezielle Erweiterungen beinhalten. Nach Aktivierung des Arguments “input-dir“ wird der Pfad der aktuellen Selektion im Navigator als Parameter übernommen und kann gegebenenfalls editiert werden. Das Argument “expand-includes-from“ erhält nach Aktivierung ebenfalls diesen Pfad.
Das ExpandItem “CmdLine Arguments“ stellt weitere Optionen für die Parametrisierung zur Verfügung. Über den Button “Generale Arguments“ werden die Argumente entsprechend der Konfiguration generiert und können bearbeitet werden. Falls das Feld leer ist, wird diese Eingabe ignoriert, enthält es Kommandozeilenargumente, werden diese für die Analyse verwendet. Der Logger gibt dabei die Ausgabe des DependencyAnalyzers wieder.
Über die CheckBoxen “Draw Graph“ und “Mark Unreachable Nodes“, kann wie gewohnt der Grapheditor des DependencyVisualizers oder die “Dependency Analyzer - Markers“-View geöffnet werden. Der Button “Run Analysis“ startet die ausgewählte Operation.

5.3.3 Zusatzinformationen zu Knoten im Grapheditor

Im Grapheditor gibt es die Möglichkeit, Zusatzinformationen zu den einzelnen Knoten einzublenden. Folgende Details sind dabei abrufbar:

- ID
- Line Number
- Location
- Name
- Scope

Diese Informationen erscheinen als Tooltip, sobald der Mauszeiger über den Knoten bewegt wird, oder nach Klick auf einen Knoten in der Properties-View, eine in Eclipse bereits vorhandene Oberfläche zur Anzeige von Eigenschaften (siehe Abb. 5.5). Falls die Properties-View nicht sichtbar ist, kann dies über den Menüpunkt “Show View“ unter “Window“ eingestellt werden.

5.3.4 Sprungmöglichkeiten zwischen Section/Variable und Grapheditor

Um von einem Knoten im Grapheditor zu der entsprechenden Deklaration im Quellcode zu springen, steht im Kontextmenü eines jeden Knoten das Kommando “Go to Declaration“ zur Verfügung (siehe Abb. 5.6).

Dabei wird die Quelldatei in einem Coboleditor geöffnet und der gewünschte Knoten markiert. Dabei ist darauf zu achten, dass diese Datei in dem Workspace-Verzeichnis von Eclipse vorhanden ist.
Der Sprung von einer Variablen- oder Sectiondeklaration zu einem Grapheditor ist ebenfalls möglich. Dazu muss der Cursor auf dem gewünschten Token im Coboleditor aktiv sein. Im

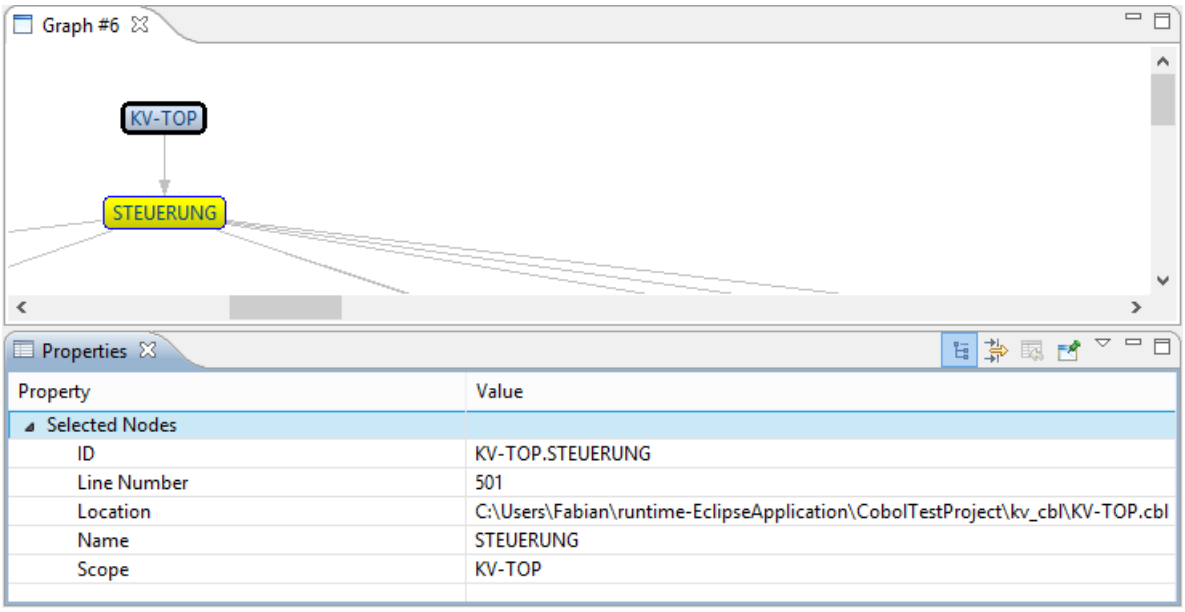

Abbildung 5.5: Eigenschaftsansicht eines Knotens

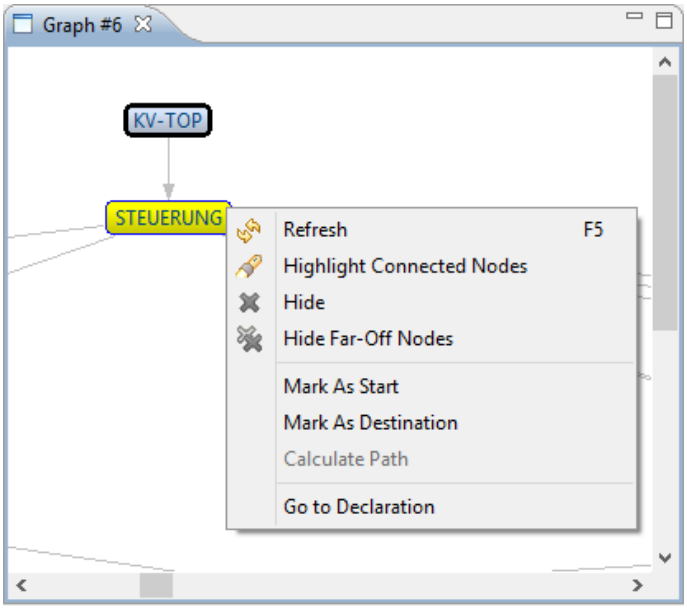

Abbildung 5.6: Kontextmenü des Grapheditors

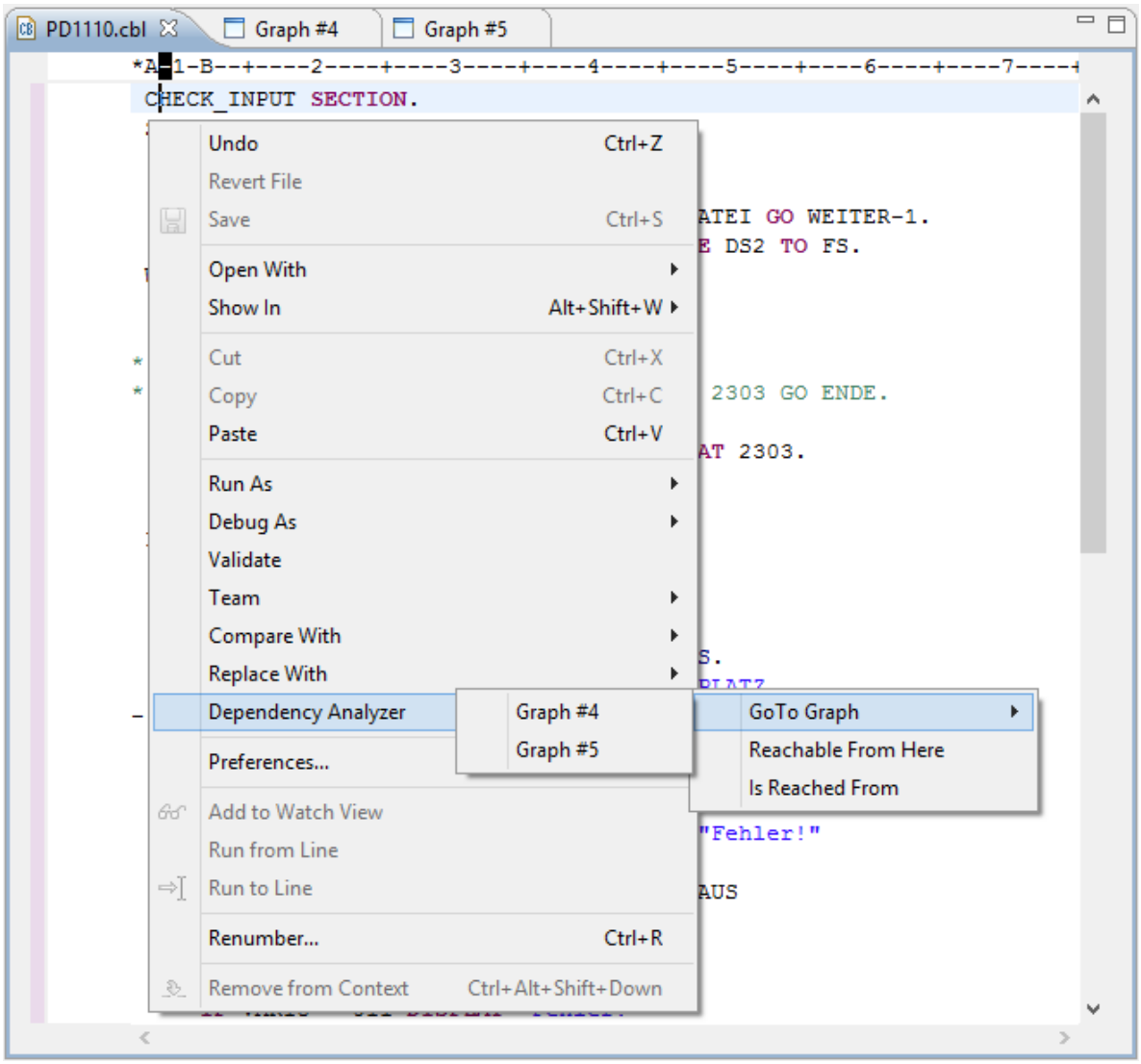

Abbildung 5.7: Kontextmenü des NetCOBOL-Editors

Kontextmenü “Dependency Analyzer“ befindet sich ein weiteres Menü mit dem Namen “Go To Graph“ (siehe Abb. 5.7).

Hier werden alle Grapheditoren aufgelistet, die das selektierte Token als Knoten enthalten. Mit Klick auf den gewünschten Graphen wird dieser geöffnet und der gesuchte Knoten fokussiert.

5.3.5 Erreichbarkeitsanalyse von Variablen und Sections

Die Funktionen zur Analyse der Erreichbarkeit von Knoten bezüglich einer Section oder Variable sind über das Kontextmenü des NetCOBOL-Editors unter dem Menüpunkt “Dependency Analyzer“ verfügbar. Es gibt zum einen die Möglichkeit, über das Kommando “Reachable From Here“ alle Knoten zu ermitteln, die von dem ausgewähltem Lexem erreichbar sind, zum anderen können über das Kommando “Is Reachable From Here“ alle Knoten ermittelt werden, von denen das ausgewählte Lexem erreichbar ist. Bei der Auswahl ist es wichtig, dass sich der Cursor in dem gewünschtem Lexem befindet. Die Parametrisierung des DependencyAnalyzers ist dabei wie folgt:

- Variable: `cob-var -input-dir Pfad`
- Section: `cob -input-dir Pfad`

“Pfad“ wird dabei durch den Pfad des Ordners ersetzt, in dem sich die Datei befindet. Die Ergebnisse der Analyse werden als Marker in der “Dependency Analyzer - Markers“-View angezeigt (siehe Abb. 5.2).

Literaturverzeichnis

[ite07] itestra GmbH, 2007. URL http://www.itestra.de. (Zitiert auf Seite 9)

[JL10] H. L. Jochen Ludewig. *Software Engineering*. dpunkt.verlag, 2010. (Zitiert auf Seite 17)

[met] Eclipse Metrics Plugin. URL http://eclipse-metrics.sourceforge.net. (Zitiert auf Seite 13)

[xra08] X-Ray 1.0.4.1, my Bachelor and Research Project, 2008. URL http://xray.inf.usi.ch/xray.php. (Zitiert auf Seite 14)

[xra13] X-Ray - software visualization 1.0.4.1, 2013. URL http://marketplace.eclipse.org/content/x-ray-software-visualization#.Uf9lVRZEcso. (Zitiert auf den Seiten 14 und 15)

Alle URLs wurden zuletzt am 05. 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