Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Integration verschiedenartiger Modelltransformationen mittels einer Workflow-DSL

Bachelorarbeit

im Studiengang Informatik

von

Lars Krumwiede

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Wolfgang Nejdl
Betreuer: M.Sc. Sebastian Meyer

Hannover, 7. August 2011

Inhaltsverzeichnis

Zusammenfassung

Am Fachgebiet Software Engineering ist eine Sammlung von mehreren Tools für die Erstellung und Verarbeitung von Modellinstanzen auf Basis des Eclipse Modeling Frameworks (EMF) entstanden. Dies sind Tools für Modell-zu-Modell-Transformationen, zum Laden und Speichern von Modellinstanzen und zum Visualisieren beliebiger EMF-basierter Klassenmodelle. Diese Sammlung soll in Zukunft noch erweitert werden.

In dieser Arbeit wird eine domänenspezifische Sprache entwickelt mit der es möglich ist, diese Tools zu orchestrieren. Dabei wird untersucht, welche Informationen implizit in der DSL versteckt werden können, um möglichst einfach und komfortabel Workflows erstellen zu können in denen die Tools zum Einsatz kommen. Hierbei werden in der DSL für bestimmte Aufgaben eigenständig Tools ausgewählt und es können auch mehrere Tools automatisch verknüpft werden um zu einem gewünschten Ergebnis zu kommen. Außerdem wird für die DSL eine eigene Eclipse-IDE mithilfe des Frameworks Xtext erstellt.

Abstract

The Fachgebiet Software Enigneering has a collection of tools for creating and manipulating model instances based on the Eclipse Modeling Framework (EMF). These are tools for model-to-model-transformation, for the loading and saving of model instances and for visualizing any EMF-based class models. This collection is supposed to be extended in the future.

In this thesis a domain specific language will be developed which allows one to orchestrate these tools. It will be examined which information can be implicitly hidden inside the DSL to make it as easy and as comfortable as possible to create workflows utilising the tools. Therefore the automated selection of appropriate tools for specific tasks will be realized, even if that means that multiple tools have to be linked up with each other in order to get to the desired result. On top of that an Eclipse-IDE for the DSL will be build using the Xtext framework.

1 Einleitung

1.1 Motivation

Am Fachgebiet Software Engineering (SE) ist eine Sammlung von mehreren Tools für die Erstellung und Verarbeitung von Modellinstanzen auf Basis des Eclipse Modeling Frameworks (EMF) entstanden.

Hierbei handelt es sich vor allem um Tools die Word-Dokumente in Instanzen eines allgemeinen Dokumentenmodells umsetzen. Weiterhin existieren Tools, um diese Modellinstanzen in ein anderes Zielmodell zu transformieren. Letztlich können auch Wiki-Markups oder Word-Dokumente auf Basis der verwendeten Modelle generiert werden. Zusätzlich existiert ein Tool, um beliebige EMF-basierte Klassenmodelle zu visualisieren und so einen schnellen Überblick zu bekommen. Zukünftig soll diese Sammlung noch um weitere Tools wie z.B. neue Ein- und Ausgabe-Formate oder Modelltransformationsschritte erweitert werden.

Jedes dieser bisher existierenden Tools liegt jedoch als eigenständige Java-Applikation vor, die nicht direkt miteinander benutzt werden können. Insbesondere stellt sich das schnelle Ausprobieren einer neuen Idee auf Basis existierender Tools als sehr aufwändig heraus, da jedes Mal ein neues Java-Projekt erstellt werden muss, welches die Tools miteinander verknüpft.

Was alle Tools gemeinsam haben ist das verwendete Metamodell. Im EMF wird das sogenannte Ecore-Metamodell verwendet um Modelle zu beschreiben und auch die vorhandenen Tools arbeiten mit auf Ecore basierenden Modellen. Das Problem ist, dass die Schnittstellen zwischen den Tools variieren und es keine klar definierten Übergabepunkte für Modellinstanzen zwischen den Tools gibt.

1.2 Aufgabenstellung

Im Rahmen dieser Arbeit soll zunächst eine Konzeption für einen Workflow erstellt werden, der es erlaubt, die bereits existierenden Tools miteinander zu verknüpfen. Um analysieren zu können, welche Anforderungen an einen solchen Workflow gestellt werden, soll ein schon real existierender, in Java realisierter, Workflow untersucht werden. Dieser muss mittels des erstellten Konzepts modellier- und ausführbar sein.

Als Beispiel wird die Umsetzung eines in Word geschriebenen Protokolls in eine Wiki-Seite benutzt. Hierzu muss zuerst das Word-Dokument in ein Modell umgesetzt werden. Dieses allgemeine Modell eines strukturierten Dokuments wird dann mittels einer Modell-zu-Modell-Transformation in ein semantisches Protokollmodell überführt. Anschließend wird aus diesem Protokollmodell das Wiki-Markup der Ausgabe generiert.

Um nachzuweisen, dass das entwickelte Konzept funktioniert, ist eine textuelle DSL zu erstellen, die die Integration der einzelnen Tool-Schritte zu einem gesamten Workflow in einer einheitlichen Weise ermöglicht.

1.3 Gliederung der Arbeit

In Kapitel 2 werden grundlegende Begriffe erläutert, die für den Rest der Arbeit wichtig sind.

Anschließend werden in Kapitel 3 die Anforderungen an die zu erstellende Sprache ermittelt. Hierfür wird ein Beispiel aus der Realität vorgestellt, auf das sich auch im weiteren Verlauf bezogen wird und das aufzeigt, was mit der DSL erreicht werden soll. Außerdem wird eine Abgrenzung zu anderen, bereits existierenden Sprachen vorgenommen, die ähnliche Aufgaben verfolgen.

Nachdem die Anforderungen definiert sind werden in Kapitel 4 die Konzepte entwickelt, die von der Sprache umgesetzt werden sollen.

Im Implementierungsteil in Kapitel 5 wird in Ausschnitten die Umsetzung dieser Konzepte gezeigt.

Die Ergebnisse der Arbeit werden in Kapitel 6 beurteilt und es wird ein Vergleich zur Ausgangssituation ohne die DSL gezogen, um zu verdeutlichen, wo die Vorteile durch die neue Sprache liegen.

Abschließend wird in Kapitel 7 die Arbeit zusammengefasst und es wird ein Ausblick auf mögliche Erweiterungen gegeben.

2 Grundlagen

In diesem Kapitel werden einige für diese Arbeit wichtige Begriffe erklärt, auf denen in den nachfolgenden Kapiteln aufgebaut wird.

2.1 Modellgetriebene Softwareentwicklung

Bei der Entwicklung von Software kommt es häufig vor, dass sich bestimmte Teile des Codes auf gleiche oder ähnliche Art wiederholen. Um sich hierbei doppelte Arbeit zu sparen setzt der Ansatz der modellgetriebenen Softwareentwicklung auf das automatische Generieren von Quellcode aus Modellen. Hierzu wird das zu entwickelnde System zunächst durch ein Modell, beispielsweise in UML, beschrieben. Diese Beschreibung muss nicht das gesamte System umfassen, sondern kann von einer Grobarchitektur bis zur Modellierung von Details reichen.

2.1.1 Der Modellbegriff

Ein Modell kann allgemein als Abbildung eines Ausschnitts der Realität bezeichnet werden. Dabei gibt es ein zugehöriges Original mit verschiedenen Eigenschaften. Aus diesen müssen für die Modellbildung die relevanten Eigenschaften ausgewählt werden, dafür sind der Zweck und die Zielgruppe des Modells wichtig. Ein Modell ist also immer für eine bestimmte Aufgabe spezifisch und so kann es von einem Original mehrere sich unterscheidende Modelle geben, deren Güte nur im Zusammenhang mit dem Zweck des Modells beurteilt werden kann.

Im Zusammenhang der modellgetriebenen Softwareentwicklung wird der Begriff Modell wie folgt definiert: Ein Modell ist eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems [1].

In der Praxis werden Modelle von Softwaresystemen oder Teilen davon erstellt, aus denen durch geeignete Tools (zum Beispiel die des EMF) automatisch Code generiert werden, wodurch wiederkehrende, die Infrastruktur eines Programms betreffende Programmieraufgaben wegfallen, die sowohl zeitaufwändig als auch fehleranfällig sind.

2.1.2 Modellierungsebenen

Um ein besseres Verständnis dafür zu bekommen was Modelle sind und wie man sie beschreiben und verwenden kann sind zunächst die Begriffe Modellinstanz, Modell, Metamodell und Meta-Metamodell relevant. Durch sie werden die vier Ebenen der Modellierung beschrieben, die von der Object Management Group (OMG) im Standard Meta-Object Facility (MOF) festgelegt wurden. Zur Erläuterung dieser Begriffe werden sie auf die Programmiersprache Java angewendet.

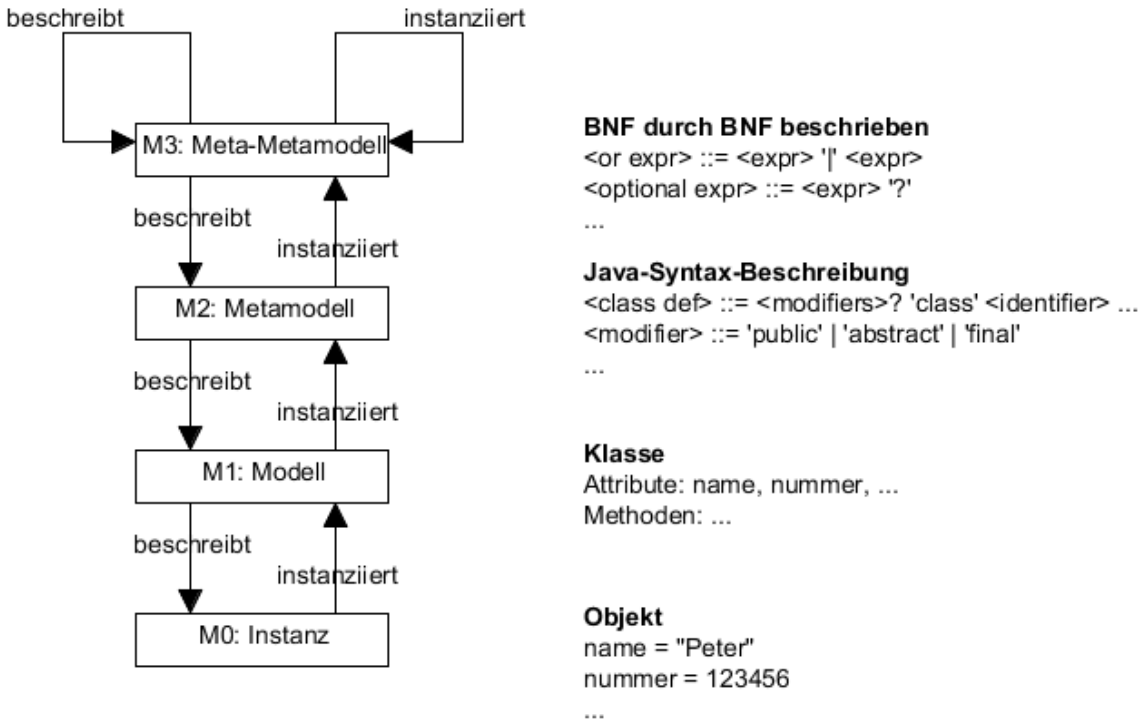


Abbildung 1: Ebenen der Modellierung nach MOF, nach [1]

Betrachtet man eine Klasse in Java, so kann man sie als ein Modell für die zugehörigen Objekte auffassen, denn die Klasse ist die Repräsentation der Struktur und Funktion all ihrer Instanzen. Demnach ist jedes Objekt eine Modellinstanz ihrer Klasse und nach dessen Struktur aufgebaut. Wenn die Klasse zum Beispiel vorgibt, dass ein Attribut *nummer* existiert, so hat jedes Objekt genau ein Attribut mit diesem Namen. Der Wert dieses Attributs kann aber von Instanz zu Instanz unterschiedlich sein. Die Modellinstanz ist also die konkreteste Ebene der Beschreibung, wie in Abbildung 1 zu sehen ist. Diese Ebene wird als M0 bezeichnet und steht an unterster Stelle, da hier noch keine Abstraktion vorgenommen wurde, sondern hier Attribute mit konkreten Werten belegt werden.

Die darüber liegende Ebene M1 enthält im Beispiel eine Beschreibung der Klasse, dies geschieht in der Programmiersprache Java, was Java zu einer Modellierungssprache macht, mit der Probleme der realen Welt modelliert werden können.

Die sich anschließende Frage ist, wie man das Modell selber beschreiben kann, das heißt man möchte ein Modell für Modelle festlegen und dies geschieht mit dem sogenannten Metamodell. Java lässt sich beschreiben durch die Backus-Naur-Form (BNF), einer Sprache mit der kontextfreie Grammatiken dargestellt werden können und die auch als Metasprache bezeichnet wird, weil durch diese Sprache andere Sprachen definiert werden können und sie so auf einer höheren Ebene steht.

Diese Kette der Metaisierung ließe sich theoretisch unendlich weiter führen, allerdings ist das für die Praxis nicht geeignet, deshalb definiert die MOF als höchste Ebene M3 die Ebene der Meta-Metamodelle. Zum einen lassen sie durch sie Metamodelle beschreiben, im Beispiel wird dadurch also die Struktur der BNF festgelegt, zum anderen lassen sich Meta-Metamodelle selber durch

Meta-Metamodelle beschreiben. Die BNF lässt sich selber durch eine kontextfreie Grammatik ausdrücken und daher ist die BNF geeignet sich selber zu beschreiben. Das macht sie neben dem Metamodell für andere Sprachen zu einem Meta-Metamodell für sich selbst.

Wie in Abschnitt 1.1 erwähnt arbeitet das EMF mit dem Ecore-Metamodell zur Beschreibung all seiner Modelle und auch die Tools die von der zu entwickelnden DSL orchestriert werden arbeiten auf Modellen denen das Ecore-Metamodell zugrunde liegt.

2.1.3 Modelltransformationen

Ein weiterer Aspekt, der hier von Bedeutung ist sind die Modelltransformationen, also die Umwandlung eines Modells in eine andere Form. Modelltransformationen können nach verschiedenen Kriterien unterschieden werden.

Eine Möglichkeit der Unterscheidung ist die Anzahl von Eingabe- und Ausgabemodellen. Es ist möglich, dass eine Transformation mehr als ein Modell als Eingabe und auch mehr als ein Modell als Ausgabe hat. Solche Fälle sind allerdings für diese Arbeit unwichtig, da hier ausschließlich mit Tools gearbeitet wird, die höchstens ein Modell als Eingabe und eins als Ausgabe verwenden.

Es kann auch zwischen Modell-zu-Modell-, Modell-zu-Text- und Text-zu-Modell-Transformationen unterschieden werden. Bei Modell-zu-Modell-Transformationen wird ein Modell in ein anderes umgewandelt, die Transformation legt also Regeln fest, wie die Umwandlung einer Instanz des Eingabemodells in eine Instanz des Ausgabemodells umgesetzt wird. Hierbei kann das Zielmodell auch mit dem Quellmodell identisch sein. In diesem Fall spricht man von einer endogenen Modelltransformation (auch Modellmodifikation oder engl. in-place transformation). Als Beispiel kann man sich ein Modell für Dokumente vorstellen, in dem alle Überschriften fett geschrieben werden sollen. Hierfür wird als Eingabe eine Instanz des Dokumentenmodells verwendet und innerhalb dieser Instanz werden die entsprechenden Änderungen vorgenommen. Das Ausgabemodell ist dann ebenfalls das Dokumentenmodell. Unterscheiden sich Ein- und Ausgabemodell so spricht man von einer exogenen Modell-zu-Modell-Transformation. Zum Beispiel könnte das Dokumentenmodell in ein Modell für mathematische Berechnungen umgewandelt werden, wenn zum Beispiel in dem Dokument Formeln enthalten sind. Diese Modelle unterscheiden sich voneinander und jedes verfolgt einen anderen Zweck.

Bei der exogenen Modelltransformation können sich auch die Metamodelle unterscheiden, dies ist aber für diese Arbeit nicht von Bedeutung, da die hier verwendeten Tools auf Modellen arbeiten, die alle ein gemeinsames Metamodell, das Ecore-Metamodell, teilen.

Bei der Modell-zu-Text-Transformation wird aus einem Modell Text erstellt. Ein häufiger Anwendungsfall hierfür ist das automatische Erzeugen von Quellcode aus einem Modell. Hierfür gibt es viele Tools, die zum Beispiel aus einem UML-Modell Code für verschiedene Programmiersprachen erzeugen können. Auch das Speichern eines Modells in eine Datei kann als Modell-zu-Text-Transformation gesehen werden, da hier eine Art Text erzeugt wird, auch wenn dieser für Menschen eventuell nicht lesbar ist.

Die Gegenrichtung ist die Text-zu-Modell-Transformation, bei der aus einem Text eine Modellinstanz erzeugt wird. Dies kann zum Beispiel beim Lesen eines Quellcodes durch einen Parser geschehen. Der Quelltext beschreibt in strukturierter Weise ein Programm und diese Struktur lässt sich in einem Modell darstellen. Die entstehende Modellinstanz repräsentiert das Programm

und kann zum Beispiel in der semantischen Analyse weiterverarbeitet werden und weitere Transformationen können angewendet werden.

Die Tools, die in dieser Arbeit durch eine DSL orchestriert werden sollen, führen verschiedene Modelltransformationen durch. So gibt es Tools zum Laden einer Modellinstanz aus einer Datei, also Text-zu-Modell-Transformationen als auch zum Speichern einer Datei und auch Modell-zu-Modell-Transformationen werden von einigen Tools durchgeführt.

2.2 Domänenspezifische Sprachen

Eine domänenspezifische Sprache (DSL) ist eine Programmiersprache, die auf einen bestimmten Problembereich, ihre Domäne, zugeschnitten ist. Im Gegensatz dazu stehen die General Purpose Languages (GPL), also universell einsetzbare Programmiersprachen, mit denen alle im Computer darstellbaren Probleme modelliert werden können. Der Vorteil von DSLs liegt in genau dieser eingeschränkten Ausdruckskraft, da durch die Konzentration auf lediglich einen Bereich eine wesentlich kompaktere, intuitivere und für den Domänenexperten verständlichere Syntax entstehen kann als es in einer GPL möglich wäre. Dabei wird typischerweise versucht, die Terminologie aus dem abzubildenden Problembereich in die Syntax der DSL einfließen zu lassen, sodass Domänenexperten, die im Normalfall keine Programmierer sind, einen leichten Einstieg in die DSL bekommen und mit ihr arbeiten können, ohne die dahinter stehende Implementierung zu kennen. (nach [3])

Ein Beispiel für eine DSL ist die Programmiersprache der frühen Versionen der Software *Mathematica*, mit der mathematische Probleme beschrieben werden können. Dabei wird eine speziell an die Bedürfnisse dieses Bereichs angepasste Syntax verwendet, die zum Beispiel das aus der Mathematik bekannte Symbol ':=' für eine Definition oder den Ausdruck 'x!' für die Berechnung der Fakultät von x enthält. Außerdem werden Dinge wie das Rechnen mit komplexen Zahlen und Matrizenrechnung unterstützt. Solche Elemente helfen Mathematikern, die vorher noch keine Programmiersprache verwendet haben, in *Mathematica* einen leichteren Einstieg zu finden als zum Beispiel in C oder C++.

Zu diesem Beispiel ist noch anzumerken, dass die Sprache von *Mathematica* stark weiterentwickelt wurde und in den jüngeren Versionen auch Konstrukte enthält, die nicht mehr auf die ursprüngliche Domäne beschränkt sind. So werden zum Beispiel prozedurale, funktionale, regelbasierte und objektorientierte Programmierparadigmen unterstützt [2]. Durch die vielen Erweiterungen hat diese Sprache ihren domänenspezifischen Charakter verloren und kann heute eher als eine GPL aufgefasst werden.

Unterteilt werden können DSLs zum einen in interne (auch eingebettete) und zum anderen in externe DSLs. Bei internen DSLs werden Syntaxkonstrukte der zugrunde liegenden Programmiersprache verwendet, das heißt die DSL ist eine Untermenge ihrer Implementierungssprache. Ein Beispiel hierfür ist Rake, eine Build Management Sprache für Ruby, die selber die Syntax von Ruby verwendet. Externe DSLs dagegen haben eine ganz eigene Syntax und können entsprechend freier an die Problemdomain angepasst werden. Als Gegenstück zu Rake sei hier beispielhaft Make erwähnt, ein Build Management Tool für C und C++, das eine eigene Syntax definiert.

Eine weitere Art DSLs zu unterteilen sind textuelle und grafische DSLs. Bei textuellen DSLs geschieht die Eingabe eines Programms in Form von strukturiertem Text. Daneben gibt es aber

auch Sprachen, die grafische Elemente zur Problembeschreibung verwenden. Das wohl bekannteste Beispiel hierfür ist die Unified Modeling Language (UML). Diese Sprache verwendet Diagramme und spezielle grafische Elemente zur Modellierung von Systemen. Dabei ist jedem Symbol eine genaue Bedeutung zugeordnet und es gibt feste Regeln wie diese Symbole verwendet werden.

Für einen Problembereich können auch sowohl eine textuelle als auch eine grafische DSL existieren. Ein Beispiel hierfür ist die Software Graphviz zu der auf der einen Seite die textuelle Graphbeschreibungssprache DOT gehört, die aber auch ein Tool enthält um Graphen über eine grafische Oberfläche zu erstellen und zu bearbeiten. Die Graphen können dann, zum Beispiel beim Speichern, in die entsprechende Repräsentation in DOT transformiert werden und beim Lesen der Datei wieder zurück in ein grafisches Modell transformiert werden.

Für die Einsatzgebiete von DSLs gibt es viele Beispiele, wobei der Grad der Spezifität und auch die Nähe zur zugrunde liegenden Programmiersprache variieren. Ein Beispiel für eine sehr umfangreiche Sprache ist SQL, eine externe DSL für die Arbeit mit Datenbanken. Dieser Bereich ist sehr vielseitig und die Sprache entsprechend mächtig, es können zum Beispiel auch mathematische Operationen durchgeführt werden, unabhängig von jeder Datenbank, das heißt, dass hier Elemente einer GPL auftauchen. Ein Gegenbeispiel sind reguläre Ausdrücke, eine Sprache um Mengen von Zeichenketten zu beschreiben. Diese Domäne ist sehr klein und entsprechend hat die Sprache einen geringen Umfang und deckt keine Probleme außerhalb dieses Bereichs ab.

Zur Verarbeitung des Quellcodes einer DSL gibt es zwei Möglichkeiten: das Interpretieren und das Kompilieren. Beim Interpretieren wird der Code durchlaufen und die einzelnen Statements dabei von einem Interpreter direkt ausgeführt, während beim Kompilieren zunächst ein anderer Code aus der DSL erzeugt wird. Dieser Zwischencode kann zum Beispiel eine allgemeine Programmiersprache wie Java sein. Vorteil des Interpretierens ist, dass die Resultate ohne einen Zwischenschritt direkt sichtbar werden, allerdings zum Preis einer geringeren Ausführungsgeschwindigkeit als beim Kompilieren.

In dieser Arbeit wird eine textuelle, externe DSL entwickelt, das heißt es wird auf eine grafische Oberfläche verzichtet und die Syntax die entsteht ist unabhängig von der zugrunde liegenden Programmiersprache. Des Weiteren wird die Sprache interpretiert werden, da der zusätzliche Zwischenschritt des Kompilierens zugunsten der leichten und schnellen Benutzbarkeit vermieden werden soll.

2.3 DSLs entwickeln mit Xtext

Um DSLs zu entwickeln gibt es verschiedene Möglichkeiten, die sich in der Implementierungssprache und auch in der Unterstützung durch vorhandene Tools unterscheiden. Im Folgenden soll das Java-Framework Xtext betrachtet werden, mit dem in dieser Arbeit die DSL entwickelt wird.

2.3.1 Was ist Xtext?

Xtext ist ein Java-Framework zur Entwicklung von Programmiersprachen und DSLs, das auf dem Eclipse Modeling Framework (EMF) basiert, einem Framework zur Unterstützung bei der modellgetriebenen Softwareentwicklung. Zum Erstellen einer neuen DSL gibt man zunächst eine Grammatik in einer erweiterten Backus-Naur-Form (EBNF) an. Zu erwähnen ist hier, dass die

EBNF nicht der ISO-Norm (ISO/IEC 14977:1996(E)) entspricht, sondern eine vereinfachte Form einer BNF ist, die allerdings im Weiteren auch als EBNF bezeichnet wird, da es neben der ISO-Norm üblich ist auch andere Abwandlungen der BNF als EBNF zu bezeichnen.

Aus der Grammatik generiert das Framework automatisch einen Parser und gleichzeitig wird ein Klassenmodell für die abstrakte Syntax der Sprache generiert. Quelltexte der DSL werden vom generierten Parser in Instanzen dieses Modells umgewandelt, auf denen dann alle weiteren Schritte, wie die Fehleranalyse oder der Interpreter, arbeiten.

2.3.2 Integration mit Eclipse

Neben dem Parser wird aus der Grammatik auch die Infrastruktur generiert, die notwendig ist, um eine voll funktionsfähige Eclipse IDE für die eigene DSL zu implementieren.

Von vornherein ist schon ein einfaches Syntaxhighlighting implementiert, das Schlüsselwörter, Stringliterale, Zahlen, Kommentare und andere Syntaxelemente farbig voneinander abhebt. Das Highlighting kann noch weiter an die eigenen Bedürfnisse angepasst werden, doch für kleine DSLs ist diese Standardeinstellung bereits ausreichend.

Außerdem wird eine statische Fehleranalyse in der IDE durchgeführt, die Syntaxfehler erkennt und, wie in Eclipse üblich, rot unterstreicht und eine entsprechende Fehlermeldung anzeigt. Dadurch wird sichergestellt, dass der Quelltext zur Grammatik konform ist. Zusätzlich wird auch eine Autovervollständigung von Schlüsselwörtern der Grammatik abgeleitet, was eine einfache Korrektur solcher syntaktischer Fehler erlaubt.

Xtext bietet auch eine Schnittstelle zum Implementieren einer semantische Analyse an, die auf dem generierten Klassenmodell arbeitet. Es können hier sowohl Warnungen als auch Fehler ausgegeben werden, wobei das bekannte gelbe bzw. rote Unterstreichen der entsprechenden Codeteile unterstützt wird. Diese semantische Analyse muss vom Programmierer manuell erstellt werden und es können hier Fehler wie zum Beispiel fehlende Variablen oder Typinkompatibilitäten umgesetzt werden. Des Weiteren lassen sich für solche Fehler leicht Quick-Fixes erstellen, also Vorschläge wie der Fehler beheben werden kann.

Neben diesen grundlegenden Funktionen bietet Xtext noch Unterstützung für viele der anderen bekannten Funktionen von Eclipse, wie Code-Formatierung, Code-Folding, Quellcode-Navigation usw., welche ebenfalls anpassbar sind.

Dank dieser Features und einer flachen Lernkurve ist Xtext ein geeignetes Tool um eine in kurzer Zeit eine modernen Standards entsprechende IDE zu erstellen.

2.3.3 Alternativen zu Xtext

Neben Xtext gibt es viele weitere Frameworks, die das Erstellen von Programmiersprachen unterstützen. Da die hier verwendete Programmiersprache Java ist und eine Anforderung darin besteht, die DSL in Eclipse einzubinden, wird hier ein weiteres Framework angesprochen, das diesen Anforderungen genügt, die IDE Meta-Tooling Platform (IMP).

Wie auch Xtext ist IMP ein Framework mit dem eine Eclipse-IDE realisiert werden kann. Im

Gegensatz zu Xtext ist IMP mehr darauf ausgerichtet, vorhandene Parser wiederverwenden zu können (siehe „Design Goals“ [4]). Das heißt, wenn man für eine bereits vorhandene Sprache eine Eclipse-IDE erstellen möchte, kann man im Framework einen existierenden Parser einbinden und muss die Ausgabe nur noch an die von IMP verwendeten Interfaces anpassen. Da in dieser Arbeit eine neue Sprache von Grund auf erstellt wird, ist das in diesem Fall allerdings kein ausschlaggebender Vorteil für die Verwendung von IMP.

Auch IMP verwendet eine EBNF, allerdings ist diese umfangreicher als die von Xtext und damit zwar flexibler aber auch schwerer zu erlernen. Da schon zu Beginn der Arbeit abzusehen war, dass die zu entwickelnde Sprache in ihrer Syntax nicht sehr komplex werden würde, ist also hier Xtext zu präferieren, da die Grammatik schneller und einfacher erstellt werden kann.

Die Features, die IMP bietet sind weitgehend identisch mit denen von Xtext und auch der Mechanismus, über den sie implementiert werden, ist ähnlich. Auch bieten beide Frameworks sinnvolle Standardeinstellungen für diese Features, sodass hier kein Framework gegenüber dem andern deutlich im Vorteil ist.

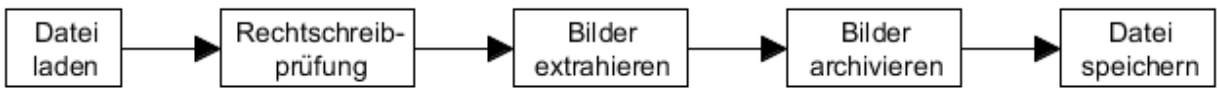
Insgesamt kann man sagen, dass sowohl Xtext als auch IMP gut geeignet sind, um eine DSL und eine zugehörige Eclipse-IDE zu erstellen, allerdings ist die Lernkurve von Xtext vermutlich etwas flacher, weshalb es in dieser Arbeit verwendet wird.

Eine andere Methode eine DSL zu entwickeln ist, auf ein Framework wie die oben genannten ganz zu verzichten und zum Beispiel nur mit einem Parsergenerator zu arbeiten, wodurch man nicht an Vorgaben des Frameworks gebunden ist und eine größere Flexibilität in der Entwicklung hat, allerdings ist das für diese Arbeit nicht sinnvoll, da es zu aufwändig ist und die zusätzliche Flexibilität hier nicht benötigt wird.

2.4 Workflows

Ein Workflow, übersetzbar mit “Arbeitsfluss”, ist ganz allgemein ein aus sukzessiven Teilschritten bestehender Arbeitsablauf (siehe Definition nach [5]).

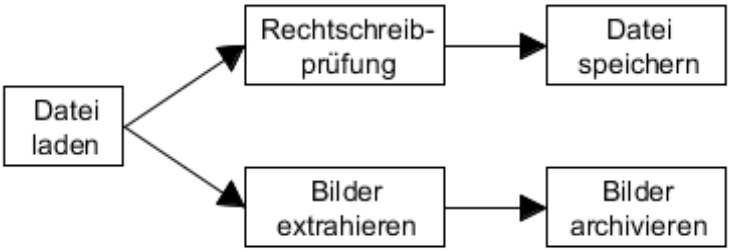
Wenn in dieser Arbeit von einem Workflow gesprochen wird, so bezieht sich dies immer auf einen Programmablauf. Als einfaches Beispiel soll hier die Verarbeitung eines Dokuments betrachtet werden. Dabei soll ein Dokument aus einer Datei gelesen werden, es soll eine Rechtsschreibprüfung stattfinden und es sollen alle im Dokument enthaltenen Bilder extrahiert und in ein gemeinsames Archiv gespeichert werden, bevor die Datei korrigiert und ohne Bilder wieder gespeichert wird. Betrachtet man dieses Beispiel, stellt man, dass es sich in mehrere Teilschritte aufspalten lässt und ein möglicher Ablauf des Workflows könnte wie folgt aussehen:



sequentieller Workflow

Dieser Ablauf arbeitet die Teilschritte sequentiell ab. Allerdings ist es auch möglich, um die

Bearbeitung zu beschleunigen, Schrittsequenzen parallel auszuführen. Schaut man sich den sequentiellen Ablauf an, stellt man fest, dass die Rechtschreibprüfung auch nach dem Extrahieren der Bilder oder nach dem Archivieren der Bilder geschehen könnte, da hier keine Abhängigkeit zur Rechtschreibprüfung besteht. Durch eine Parallelisierung könnte die Abarbeitung auch wie folgt aussehen:



paralleler Workflow

In diesem Fall lassen sich die Rechtschreibprüfung und das Extrahieren der Bilder parallel ausführen, da sie keine gemeinsamen Ressourcen verwenden. Das Speichern kann allerdings erst nach der Rechtschreibprüfung und das Archivieren der Bilder erst nach dem Extrahieren erfolgen.

Es ist also in machen Fällen eine Parallelität innerhalb eines Workflows möglich, wodurch die Abarbeitung beschleunigt werden kann. Die Realisierung im Beispielprogramm könnte durch mehrere Threads erfolgen, in diesem Fall einen für die obere und eine für die untere Teilsequenz. Problematisch wird das allerdings, wenn die Ergebnisse aus zwei parallelen Pfaden wieder zusammengeführt werden müssen, wenn also im Beispiel das korrigierte Dokument und das Bilderarchiv als E-Mail verschickt werden sollen, so entsteht ein zusätzlicher Aufwand bei der Umsetzung, da zuerst beide parallele Zweige abgeschlossen sein müssen, bevor der nächste Schritt erfolgen kann. Das ist in diesem einfachen Beispiel noch relativ leicht umsetzbar, aber bei komplexeren Workflows kann die Umsetzung wesentlich komplizierter ausfallen.

In dieser Arbeit wird eine DSL erstellt, die solche Workflows beschreibt und die dann ausgeführt werden. Um hierbei nicht die angesprochen Probleme der parallelen Verarbeitung zu haben, wird auf Parallelität verzichtet. Ein Workflow wie er hier vorkommt ist also stets als eine Sequenz aufzufassen, bei der die Teilschritte in der Reihenfolge abgearbeitet werden in der sie angegeben sind.

Ein weiterer wichtiger Punkt bei einem Workflow ist die Kommunikation zwischen den Schritten. Das Ergebnis des Ladens im Beispiel muss an den nächsten Arbeitsschritt weitergereicht werden. Bei einer sequentiellen Bearbeitung ist dies allerdings nicht weiter problematisch, da ein Zwischenergebnis immer nur zwischen zwei benachbarten Schritten ausgetauscht wird und zu jedem Zeitpunkt nur von einem Schritt bearbeitet wird.

3 Anforderungen

In diesem Kapitel werden die Anforderungen ermittelt, die an die zu entwickelnde DSL gestellt werden. Hierzu wird zunächst ein reales Beispiel eines vorhandenen Workflows vorgestellt an dem die existierenden Java-Tools und ihre Zusammenarbeit gezeigt und erläutert werden. Anschließend werden hierauf aufbauend die Anforderungen an die zu erstellende DSL ermittelt. Von diesen ausgehend werden dann existierende DSLs mit ähnlichen Zwecken analysiert und abgegrenzt, welche Ideen und Konzepte hieraus relevant sind und welche nicht. Die Beschreibung der Realisierung der Anforderungen erfolgt dann in Kapitel 4.

3.1 Beispiel eines realen Workflows

Am Fachgebiet SE sind Tools entstanden, die dazu dienen, Word-Dokumente auf verschiedene Arten zu verarbeiten. So können zum Beispiel beliebige Word-Dateien gelesen werden und ihr Inhalt kann verwendet werden, um nach geeigneter Verarbeitung, zum Beispiel als Wiki-Markup, gespeichert zu werden. Es wird nun beschrieben welche Tools es gibt und was ihre Aufgaben sind.

Ein konkreter Anwendungsfall: In einem Unternehmen werden zu verschiedenen Themen Meetings abgehalten. Diese Meetings werden alle nach dem gleichen Muster protokolliert und diese Protokolle als Word-Datei gespeichert.

Um die Ergebnisse eines Meetings dem Rest des Unternehmens zugänglich zu machen, werden diese Protokolle zunächst manuell in eine Wiki-Seite umgewandelt, welche dann im firmeninternen Netzwerk zur Verfügung gestellt wird.

Die bisher manuelle Umwandlung von Word-Dokumenten in Wiki-Seiten ist mühsam und fehleranfällig, verursacht also unnötige Kosten. Es ist klar, dass hier Raum für Verbesserung besteht und man möchte diesen Vorgang so weit wie möglich automatisieren.

Die naive Herangehensweise an das gegebene Problem wäre es, ein monolithisches Programm zu schreiben, das die Aufgabe in einem Schritt erledigt, also die Datei einliest, die Protokollinformationen extrahiert und als Wiki in eine andere Datei speichert. Das ist allerdings nicht sehr flexibel, denn wenn zum Beispiel die Datei nicht mehr als Word-Datei, sondern im PDF Format gespeichert würde oder sich die Protokollvorlage änderte, dann müsste das gesamte Programm angepasst werden.

Aus diesem Grund wurden für trennbare Teilaufgaben jeweils einzelne Tools, jede als eigene Java-Klasse, entwickelt, die verknüpft werden können, um so die Gesamtaufgabe durchzuführen. Änderungen im Ablauf der Verarbeitung werden dadurch lokal gehalten und weiterhin können einzelne Tools auch unabhängig voneinander in anderen Kontexten verwenden kann.

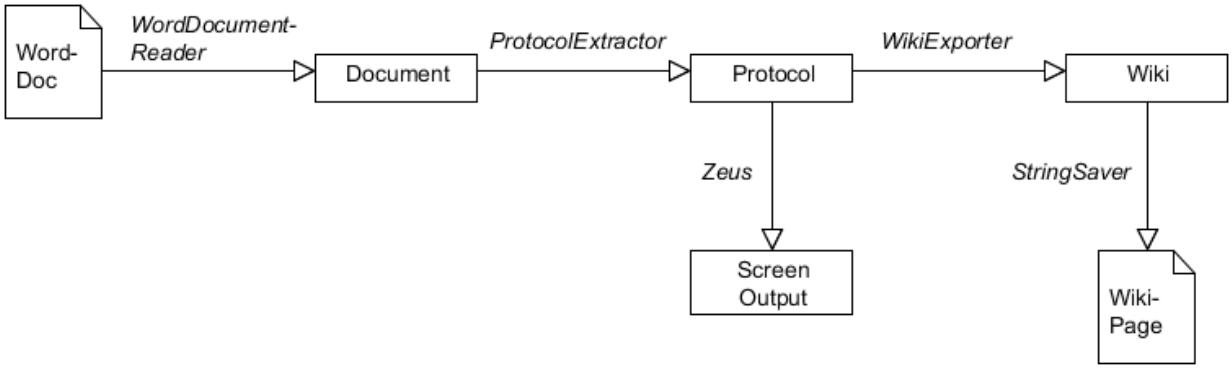


Abbildung 2: Beispielhafter Workflow

In Abbildung 2 ist der so entstehende Workflow visualisiert. Die Pfeile bedeuten, dass hier das Eingangselement in das Ausgangselement transformiert wird. An den Pfeilen steht kursiv der Name des für die Transformation verwendeten Tools.

Zur Erklärung der Elemente: das *Word-Doc* stellt die Datei dar, die das Protokoll enthält.

Die Elemente *Document*, *Protocol* und *Wiki* sind Modelle, auf denen die Tools arbeiten. Das *Document*-Modell kann allgemeine Dokumente darstellen, es enthält Elemente wie Textteile, Absätze, Tabellen usw. und ist von Dateiformaten unabhängig. Das *Protocol*-Modell ist ein Modell für Protokolle, die eine spezielle Form haben. Es enthält Informationen wie Thema, Agenda, Teilnehmer usw. Das Wiki-Modell ist aktuell lediglich ein String, also ein Text, der Zeichen zur Formatierung als Wiki-Markup enthält. Es ist also kein Modell im eigentlichen Sinn. Allerdings ist es an dieser Stelle durchaus sinnvoll, es als eine Art Modell zu sehen, da die Semantik des Strings hier bekannt ist und es außerdem möglich ist, dass in Zukunft für Wikis ein eigenes Modell erstellt wird.

Die *Wiki-Page* ist wiederum eine Datei, die die aus der Word-Datei gewonnene Wiki-Seite beinhaltet.

Der *Screen Output* steht für die Ausgabe auf den Bildschirm, typischerweise in einem eigenen Fenster. Dazu ist zu bemerken, dass man die Möglichkeit haben möchte, die Zwischenergebnisse der Tools zu betrachten, um deren Korrektheit überprüfen zu können. Dafür kann eine Modellinstanz zum Beispiel als Graph visualisiert werden.

3.1.1 Realisierung des Workflows in Java

Hier soll das angesprochene Beispiel in der Umsetzung als Java-Applikation gezeigt werden, um daran die Verwendung der Tools zu zeigen und später einen Vergleich mit der Umsetzung in der DSL ziehen zu können. Es werden hier nur essenzielle Codeteile gezeigt, der vollständige Quellcode befindet sich im Anhang in Abschnitt 8.1.

Zu Beginn stehen, wie in Java üblich, die Package-Bezeichnung und die benötigten Imports. Es folgt die Deklaration der Klasse, in der die main-Methode enthalten ist. In ihr laufen die Schritte des Workflows ab, die hier gezeigt werden sollen.

```
WordDocumentReader reader = new WordDocumentReader();
reader.setInputStream(new FileInputStream(filename));
Document document = reader.getDocument();
```

Word-Dokument laden

Das Tool um Word-Dateien zu lesen wird instantiiert und die zu lesende Datei angegeben (der Bezeichner *filename* wurde vorher deklariert). Als Eingabe erwartet dieses Tool einen *InputStream*.

```
if (document != null) {
    ProtocolExtractor extractor = new ProtocolExtractor(document);
    Protocol protocol = extractor.getProtocol();
```

Dokument aus Protokoll erstellen

Hier sieht man ein für Java typisches Vorgehen: eine Referenz wird zunächst auf *null* geprüft um sicherzustellen, dass sie gültig ist und keine *NullPointerException* verursacht. Als Eingabe für das Tool *ProtocolExtractor* wird das vorher geladene *document* verwendet.

```
if (protocol != null) {
    WikiExporter exporter = new WikiExporter();
    exporter.setDialect(WikiDialect.CONFLUENCE);
    exporter.setCommunicationFilename(communicationFile);
    exporter.setProtocol(protocol);
    String wikiMarkup = exporter.getWikiMarkup();
```

Wiki-Text aus Protokoll erstellen

Um einen Wiki-Text zu generieren benötigt das Tool *WikiExporter* neben dem Protokoll noch den Wiki-Dialekt dem das Wiki-Markup entsprechen soll, hier als Teil einer Enumeration übergeben, und eine Datei in der sich weitere Informationen über die in der Firma laufenden Projekte und ihre Mitarbeiter befinden. Mit diesen Informationen kann dann eine aussagekräftige Wiki-Seite erstellt werden. Das Resultat dieses Tools ist ein Java-String im gewählten Wiki-Format.

```
ZeusActivator viewer = new ZeusActivator();
viewer.showModelInstance(protocol, protocolModelFilename,
    ZeusActivator.DisplayType.Graph, "Protokoll");
```

Protokoll als Graph visualisieren

Zum Visualisieren einer beliebiger EMF-basierter Modellinstanzen existiert das Programm *Zeus*. Ein Problem bei der Verwendung dieses Programms in einer anderen Java-Applikation ist, dass *Zeus* als eigenständige Applikation konzipiert wurde, bei der vom Benutzer nach Starten des Programms über Dialoge Dateinamen angegeben werden, einmal von der Modellinstanz und vom zugehörigen Modell. Diese werden dann geladen und angezeigt. Die Benutzung über eine API in Java war dabei nicht vorgesehen.

Um das Programm nun direkt aus Java heraus aufrufen zu können wurde ein Wrapper geschrieben, der die interne Beschaffenheit von Zeus kennt und es ermöglicht das Programm zu starten und darin neue Tabs mit verschiedenen Visualisierungen zu erzeugen. Auf die Funktionsweise dieses Wrappers soll hier nicht näher eingegangen werden, da es nicht die Hauptaufgabe dieser Arbeit ist die Tools benutzbar zu machen. Der ZeusActivator ist nur als Nebenprodukt entstanden und soll im weiteren Verlauf als eines der gegebenen Tools betrachtet werden. Für seine Benutzung werden die Modellinstanz, hier *protocol*, der Name der Ecore-Datei, die die Definition des zugehörigen Modells enthält, die Art der Visualisierung und ein Titel für das erscheinende Fenster angegeben.

```
StringSaver saver = new StringSaver();
saver.save(wikiMarkup, "examples/wiki.txt");
```

Wiki-Text in eine Datei speichern

Der im Speicher liegende Wiki-String muss nun noch in einer Datei gespeichert werden. Auch hierfür wurde ein neues Tool geschrieben, das auf einfache Art einen String in eine Datei schreibt.

Bei dem hier gezeigten Workflow soll hervorgehoben werden, dass die Tools jeweils unterschiedliche Interfaces haben. So verlangt der *WordDocumentReader* zum Beispiel einen *InputStream* als Eingabe, während andere Tools mit Dateinamen arbeiten. Außerdem können unterschiedliche Parameter benötigt werden, wie der Wiki-Dialekt oder zusätzliche Dateinamen. Im Fall des Wiki-Dialekts ist es aber auch denkbar, dass das Tool einen Standard gesetzt hat, der es überflüssig machen kann, den Parameter anzugeben, das heißt, es können auch optionale Parameter existieren.

Zusammenfassend sind in Tabelle 1 noch einmal die derzeit existierenden Tools mit ihren Ein- und Ausgaben aufgelistet. In der Liste befinden sich auch Tools, die nicht im Beispiel vorkommen, die allerdings ähnlich arbeiten.

Tool	Input	Output
WordDocumentReader	Word-Datei	Instanz von Document
ExcelReader	Excel-Datei	Instanz von Document
ProtocolExtractor	Instanz von Document	Instanz von Protocol
WikiExporter	Instanz von Protocol Wiki-Dialekt Datei für zusätzliche Informationen	String im Wiki-Format
DocumentHTMLAdapter	Instanz von Document	String im HTML-Format
StringSaver	String	Text-Datei
DocxWriter	Instanz von Document	Word-Datei
ZeusActivator	Modellinstanz Modell-Datei	Ausgabe als Graph, Text oder HTML

Tabelle 1: Die vorhandenen Tools

3.1.2 Kategorisierung der Tools

Schaut man sich die Tools und ihre Aufgaben an, so lassen sie sich zunächst in drei Gruppen einteilen.

Die erste Gruppe besteht aus Tools, die eine Datei einlesen. Damit stehen sie am Anfang eines Workflows, da über sie die zu verarbeitenden Daten in den Workflow hineinkommen. Konkret sind das hier der *WordDocumentReader* und der *ExcelReader*, die jeweils eine Instanz des Modells *Document* als Ausgabe liefern.

Die zweite Gruppe beinhaltet Tools, die eine Modellinstanz in eine Instanz eines anderen Modells umwandeln, also eine Modell-zu-Modell-Transformation durchführen. Darunter fällt der *ProtocolExtractor*. Außerdem sollen hier auch der *WikiExporter* und der *DocumentHTMLAdapter*

dazu gezählt werden, denn obwohl sie als Ausgabe jeweils einen String liefern, also kein Modell im eigentlichen Sinn, können, wie in Abschnitt 3.1 erwähnt, diese Strings als eine Art Modell betrachtet werden, da sie implizit durch ihr Format Informationen enthalten, die wie ein Modell eine Struktur vorgeben. Für den Benutzer der Tools besteht semantisch bei diesen Strings ein Unterschied, da er ihre Bedeutung kennt.

Die dritte Gruppe von Tools sind diejenigen, die das Ende der Verarbeitung einer Modellinstanz bilden. Dies sind die Tools zum Speichern einer Instanz in eine Datei (*StringSaver* und *DocxWriter*) aber auch der *ZeusActivator* zum Visualisieren von Modellinstanzen. Hier lässt sich allerdings argumentieren, dass zwischen dem Speichern und visualisieren ein Unterschied besteht. Wenn der Benutzer der Tools eine Visualisierung einer Modellinstanz erstellt, so ist diese flüchtig und wird meist dazu verwendet um während der Entwicklung eines Tools einen schnellen Überblick darüber zu erhalten, ob es korrekt funktioniert, das heißt, es werden mehrfach Visualisierungen durchgeführt, deren Resultate aber später (nachdem das Fenster geschlossen wurde) nicht mehr benötigt werden. Auf der anderen steht das Speichern einer Modellinstanz in einer Datei. Die Datei bleibt bestehen, bis sie gelöscht wird und normalerweise bildet sie ein gewünschtes Resultat eines Workflows das später weiterverarbeitet werden soll. Eine Unterscheidung in Tools zum Speichern und Tools zum Visualisieren von Modellinstanzen ist also durchaus sinnvoll und wird deshalb vorgenommen.

Insgesamt werden hier also vier Kategorien von Tools unterschieden:

- Tools zum Laden einer Modellinstanz aus einer Datei
- Tools für Modell-zu-Modell-Transformationen
- Tools zum Speichern einer Modellinstanz in eine Datei
- Tools zum Visualisieren einer Modellinstanz

Diese Unterscheidung muss sich auch in der DSL widerspiegeln.

Es ist noch zu erwähnen, dass diese Unterteilung keinen Anspruch auf Vollständigkeit erhebt, sondern aus den praktischen Anforderungen der existierenden Tools entstanden ist. Es ist theoretisch möglich, dass in Zukunft Tools entstehen, die nicht eindeutig einer der Kategorien zugeordnet werden können, allerdings ist das vom aktuellen Standpunkt aus nicht abzusehen und eine Eigenschaft von DSLs ist es, dass sie problemspezifisch sind, also den Problembereich abdecken ohne darüber hinausgehende Probleme lösen zu wollen. Aus diesem Grund werden hier diese Kategorien als eine passende Lösung für das gegebene Problem festgelegt.

3.2 Realisierung als Eclipse-Plugin

Um die DSL möglichst gut benutzbar zu gestalten soll sie durch eine eigene Eclipse-IDE unterstützt werden. Wie in Abschnitt 2.5 erwähnt ist dies über das verwendete Xtext-Framework möglich und es können die von Eclipse bekannten Features auch für die eigene Sprache implementiert werden. Durch diese Features erhält der Benutzer der DSL nicht erst nach Ausführen eines Workflows eine Rückmeldung über dessen Korrektheit, sondern bereits beim Erstellen. Außerdem können für Fehler auf diese Weise Lösungsvorschläge (Quick-Fixes) gemacht werden, zum Beispiel durch das ermitteln passender Tools für eine bestimmte Aufgabe.

Die Unterstützung durch die IDE ist von großer Bedeutung, da ein Ziel der DSL das schnelle Ausprobieren von neuen Ideen ist. Durch eine kontextabhängige Autovervollständigung, die zum Beispiel passende Tools für eine Aufgabe vorschlägt, muss der Benutzer nicht zu jeder Zeit über alle Tools Bescheid wissen. Durch die permanente syntaktische und semantische Prüfung des Codes bekommt der Benutzer bereits zur Zeit der Erstellung eines Workflows eine Rückmeldung über die Korrektheit und nicht erst beim Ausführen.

Durch diese und andere Features wird das Erstellen von Workflows in der DSL stark vereinfacht.

3.3 Ausführen der Sprache durch einen Interpreter

Ein in der DSL geschriebener Workflow soll direkt ausführbar sein, das heißt es soll ein Interpreter implementiert werden. Im Gegensatz dazu wäre es denkbar aus dem Code der DSL zunächst einen Zwischencode in Java zu erstellen und diesen dann in einem zweiten Schritt auszuführen. Das ist allerdings nicht gewünscht, da dieser zusätzliche Schritt auch zusätzlichen Aufwand bedeutet. Zunächst kostet das Generieren des Zwischencodes Zeit und dann muss dieser, unter Umständen per Hand, ausgeführt werden. Das ist unkomfortabel und soll daher durch eine Interpreter vermieden werden. Außerdem ist ein Ziel der DSL, dass nicht für jeden Workflow ein eigenes Java-Projekt erstellt werden muss, daher ist es nicht sinnvoll, aus einem Workflow wieder Java-Code zu machen, da dieser nur dem Zweck dienen würde, unverändert ausgeführt zu werden und keine weitere Verwendung hätte.

3.4 Anbindung von Tools an die Sprache

Eine Anforderung besteht darin, dass die Sprache auf einfache und konsistente Weise um neue Tools erweitert werden kann. Wie in Abschnitt 3.1.2 angesprochen werden die Tools in vier Kategorien eingeteilt, das heißt, ein neues Tool muss zunächst einer dieser Kategorien zugeordnet und dann als solches der Sprache bekannt gemacht werden.

3.4.1 Zuordnung der Tools zu einer Kategorie

Die erste Frage ist also die Zuordnung des Tools zu einer der Kategorien. Hierfür bietet es sich an für jede der vier Tool-Kategorien jeweils ein eigenes, verbindliches Interface zu erstellen. Im Java-Beispiel war zu sehen, dass sich die Benutzung der Tools in manchen Details unterscheidet, sie sind also in der vorliegenden Form in ihrer Benutzung inkonsistent. Das ist bei der Erweiterung problematisch, kann aber durch die gemeinsamen Schnittstellen verhindert werden. Das heißt, die Sprache kann alle Tools einer Kategorie gleichartig ansprechen, wodurch Erweiterungen problemlos verwendet werden können. Wie die Interfaces im Detail aussehen wird in Abschnitt 5.1 besprochen.

3.4.2 Erweiterung um neue Tools

Die sich anschließende Frage ist, über welchen Mechanismus ein neues Tool der Sprache bekannt gemacht wird. Hierzu werden zwei Möglichkeiten mit ihren Vor- und Nachteilen besprochen, die bei der Umsetzung ausprobiert wurden.

Als erstes wird geklärt, welche Informationen für die Verwendung eines Tools gebraucht werden. Dies ist einmal die Klasse, die das Tool enthält, also eine Java-Klasse, die eines der vorgegebenen Interfaces implementiert. Außerdem möchte man ein Tool innerhalb der Sprache eindeutig identifizieren können, das heißt, jedes Tool soll einen eigenen, eindeutigen Bezeichner bekommen. Hierfür wurden zwei Mechanismen ausprobiert.

3.4.2.1 Definition innerhalb der DSL

Die erste Möglichkeit dies zu erreichen ist, die Definitionen der Tools als Teil der Sprache zu realisieren. Dafür wird ein Syntaxkonstrukt vorgesehen, über das Tools deklariert werden. Anzugeben sind dabei der Bezeichner, über den das Tool im DSL-Code angesprochen werden kann und die Klasse in der es liegt. Dies kann über den voll qualifizierten Klassennamen geschehen. Eine einfache Syntax hierfür könnte wie folgt aussehen:

```
define <Identifier>: <fully-qualified-class-name>
```

Der *Identifier* wird im weiteren Code der DSL verwendet, um das Tool anzusprechen.

Bei der Implementierung eines Interpreters würde per Reflection eine Instanz der Tool-Klasse (*fully-qualified-class-name*) erstellt, überprüft welches der Tool-Interfaces sie implementiert und die und die entsprechende Aktion würde ausgeführt.

Vorteile dieser Herangehensweise sind zum einen, dass der Benutzer der DSL direkt im Code sehen kann, welche Tools ihm zur Verfügung stehen. Er weiß, dass er alle Tools verwenden kann, die er deklariert hat. Dies ist aber auch gleichzeitig ein Nachteil, denn bei vielen Tools muss für jeden neuen Workflow in der DSL zunächst eine Reihe Deklarationen erstellt werden, die für die Aussage des Workflow selber keine große Bedeutung haben. Im Vordergrund stehen eigentlich die Verarbeitungsschritte, also die Verwendung der Tools. Die Workflows werden sich vermutlich kaum in den Deklarationen unterscheiden, da sich die Tool-Klassen selten ändern. Es entsteht also viel mehrfacher Code, was vermieden werden sollte, denn wenn sich ein Klassenname eines Tools ändert, so muss er in jedem Quelltext der DSL angepasst werden, was den Wartungsaufwand der Workflows erhöht.

Als ein Vorteil dieser Methode kann gesehen werden, dass der Interpreter auf diese Art nicht an die Realisierung der DSL als Eclipse-Plugin gebunden ist, sondern auch in anderen Umgebungen eingesetzt werden kann. Da die Realisierung als Eclipse-Plugin aber eine der Anforderungen ist, ist dieser Vorteil hier nicht von Bedeutung.

Ein Problem der Deklaration von Klassen innerhalb der DSL ist die Verwendung der Java-Reflection für die Instantiierung der Klassen. Beim Ausprobieren dieser Methode hat sich herausgestellt, dass der Zugriff auf Klassen in anderen Projekten des Eclipse-Workspaces nicht ohne Weiteres möglich ist. Es gibt hierbei ein Problem mit dem *ClassLoader*-Mechanismus über den in Java Klassen geladen werden. Nur über eine komplizierte Implementierung eines eigenen *ClassLoader*s war es möglich Klassen aus anderen Projekten verwenden zu können, allerdings dauerte dies oft mehrere Sekunden und in manchen Fällen traten Fehler auf, zum Beispiel wenn Projekte geschlossen wurden. Ohne auf die Details dieser Problematik weiter einzugehen sei hier festgestellt, dass diese Methode für die Aufgabe ungeeignet ist.

3.4.2.2 Definition außerhalb der DSL

Die zweite Möglichkeit der DSL Tools bekannt zu machen ist, die Definitionen der Tools über Extensions anzubinden. Da feststeht, dass die DSL als Eclipse-Plugin realisiert wird, steht auch der von Eclipse angebotene ExtensionPoint-Mechanismus zur Verfügung. Bei diesem Mechanismus stellt ein Plugin eine definierte Schnittstelle, den ExtensionPoint, zur Verfügung, über den andere Plugins mithilfe geeigneter Extensions angebunden werden können. Dabei bleibt ein hoher Grad der Entkopplung zwischen den Plugins bestehen, denn die einzige Verbindung besteht über diese klar definierte Schnittstelle. Solch eine Entkopplung ist auch für die DSL und die Tools wünschenswert, da dadurch die Erweiterbarkeit um neue Tools vereinfacht wird.

Das DSL-Plugin kann also einen ExtensionPoint anbieten, über den ein anderes Plugin, welches die Tools enthält, in einer entsprechenden Extension sowohl die Tool-Klassen als auch ihre Bezeichner festlegt. Das DSL-Plugin ruft dann alle Extensions ab und kennt damit die vorhandenen Tools.

Im Vergleich zur ersten Möglichkeit sind hierbei für den Benutzer der DSL die vorhandenen Tools nicht direkt sichtbar, was als Nachteil gesehen werden kann, da der Benutzer nicht mehr direkt in der DSL sehen kann, welche Tools ihm zur Verfügung stehen. Dieser Nachteil lässt sich aber durch die von Eclipse angebotene IDE ausgleichen, denn es können durch die Autovervollständigung die zur Verfügung stehenden Tools vorgeschlagen werden und es können nicht vorhandene Bezeichner schon während des Erstellens als Fehler markiert werden.

Außerdem hat man hier den Vorteil, dass der DSL-Code kompakter wird. Die Deklarationen geschehen nun nicht mehr in jedem Quelltext sondern sind an einer zentralen Stelle gebündelt, was den sich wiederholenden Code für die Deklarationen überflüssig macht. Man kann sich also bei der Erstellung eines Workflows auf dessen Aufgabe konzentrieren. Außerdem sind Änderungen der Tool-Klassen einfacher zu propagieren, da nur die Klasse in der Extension geändert werden muss während alle sich darauf beziehenden Quelltexte ihre Gültigkeit behalten.

In der Umsetzung hat sich diese Möglichkeit als geeignet erwiesen. Das Problem der Reflection, dass Klassen nicht gefunden werden wird dadurch umgangen, dass eine in einer Extension angegebene Klasse über die Eclipse-API instantiiert werden kann. Diese Methode ist schnell und weniger fehleranfällig als die im vorigen Abschnitt beschriebene.

3.5 Analyse existierender DSLs für Workflows

In diesem Abschnitt werden DSLs untersucht, die ähnliche Aufgabe verfolgen, wie die der zu erstellenden Sprache. Ziel ist es, verwendbare Konzepte zu finden und auch eine Abgrenzung zu anderen DSLs zu vorzunehmen.

3.5.1 make

make ist eine Tool, das ursprünglich entwickelt wurde um Projekte in der Programmiersprache C, die viele Quellcodedateien enthalten, zu einem lauffähigen Programm zu machen. Hierfür sind verschiedene Arbeitsschritte nötig, die durch *make* gesteuert werden können. Darunter fallen unter anderem das Kompilieren, das Linken und Dateioperationen, wie Dateien und Verzeichnisse erstellen, kopieren und löschen. Um diese Arbeitsschritte zu beschreiben gibt es eine eigene Sprache in der die so genannten Makefiles geschrieben werden. Die Makefiles werden dann von

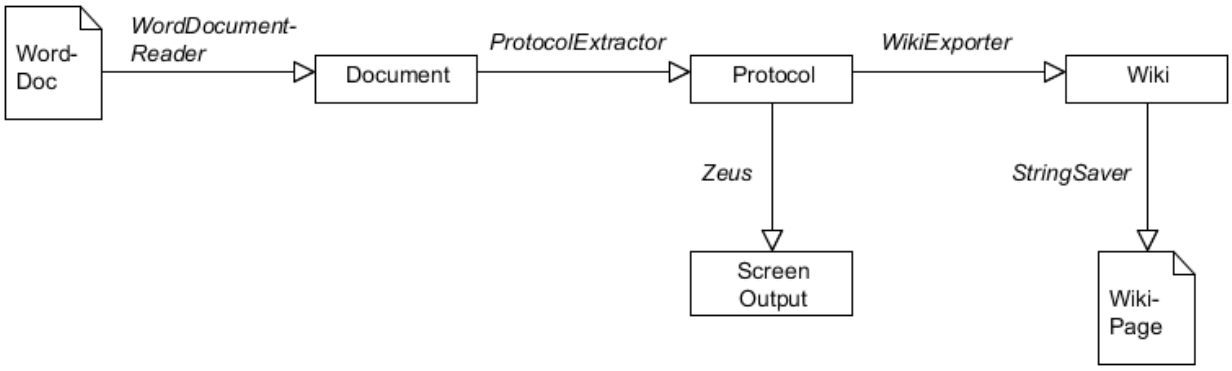
make gelesen und wie bei einem Interpreter direkt ausgeführt.

Ein Makefile hat typischerweise ein Ziel (Target), das erstellt wird und das aus weiteren Targets bestehen kann die wiederum aus weiteren Targets bestehen können usw. Die Syntax stellt dies wie folgt dar:

```
target : source1 source2 ...
        command
```

Auf der linken Seite steht zunächst das Ziel das erstellt wird und durch einen Doppelpunkt getrennt folgen alle Unterziele von denen dieses Ziel abhängt, die also zuerst erstellt werden müssen. Die Art wie das *target* zu erstellen ist wird in der nächste Zeile bzw. in den nächsten Zeilen beschrieben. Hier können beliebige Kommandozeilen-Befehle angegeben werden, die dann ausgeführt werden. Das bedeutet, man hat bei *make* auch die Möglichkeit Tools zu verwenden, die nichts mit der Erstellung von Projekten zu tun haben. Die Steuerung der Programme entspricht dem Aufruf auf der Kommandozeile, das heißt, es wird zunächst der Name eines ausführbaren Programms angegeben und darauf folgen mögliche Parameter als Strings.

Die Art wie *make* verwendet wird entspricht nicht mehr den heutigen Standards, was nicht weiter überraschend ist, da es in den späten 1970er Jahren entstanden ist. Deshalb ist die Frage, ob ein Workflow wie der reale Beispiel-Workflow aus Abschnitt 3.1 in *make* realisierbar ist. Hier noch einmal der Workflow:



Es ist möglich, die zu erstellende Wiki-Seite als Ziel anzugeben und sie abhängig von den benötigten Lade- und Transformationsschritten zu machen. Dafür müssten allerdings alle Tools als eigenständige, ausführbare Applikation vorliegen, damit sie von *make* über die Kommandozeile angesprochen werden können.

Problematisch ist auch die Kommunikation zwischen *make* und dem Benutzer. Typischerweise werden alle Meldungen auf der Konsole angezeigt und das erst nach Ausführen eines Makefiles. Es gibt keine Möglichkeit während des Erstellens des Workflows eine Rückmeldung zu erhalten.

Das Programm *make* bietet aufgrund der veralteten Handhabung keine gute Möglichkeit die hier benötigten Workflows umzusetzen. Es ist hier allerdings erwähnt worden, da es eines der ersten Tools mit einer eigenen Sprache zum Orchestrieren anderer Tools ist und auch heute noch, wenn

auch meist durch IDEs versteckt, oft Verwendung findet.

Die wichtigste Idee ist hierbei die der Abhängigkeiten der verschiedenen Schritte. In den vorgestellten Workflows findet sich dieses Element wieder und muss daher auch von der DSL umgesetzt werden.

3.5.2 Ant

Ein modernerer Ansatz mit einem ähnlichen Ziel wie *make* ist *Apache Ant*. Dieses Tool basiert auf der Programmiersprache Java und übernimmt für diese die Aufgabe wie *make* für C und C++. Die Workflows werden hier allerdings nicht in einer eigenen Sprache verfasst, sondern als XML-Datei realisiert, die sogenannte Build-Datei. Wie auch ein Makefile enthält eine solche Build-Datei Targets und diese können von anderen Targets abhängen, sodass ebenfalls eine Art Baumstruktur von Abhängigkeiten entsteht. Das Erstellen eines Targets in *Ant* geschieht über sogenannte Tasks, von denen viele, häufig gebrauchte, schon standardmäßig in *Ant* enthalten sind. Beispiele sind das Kompilieren von Java-Code oder Dateioperationen.

Auch eigene Tasks lassen sich in *Ant* erstellen. Dies geschieht über eine entsprechende Task-Definition in der Build-Datei mit der Angabe einer Java-Klasse die den Task repräsentiert und ein dafür vorgesehenes Interface implementiert. Über diesen Mechanismus ist es theoretisch möglich, die oben vorgestellten Tools mit *Ant* zu verwenden.

Nachteil an dieser Methode ist allerdings, dass XML keine kompakte Sprache ist, sondern eine, die dafür ausgelegt ist, möglichst leicht von Parsern gelesen werden zu können. Das heißt, dass das manuelle Erstellen und Warten von XML-Dateien per Hand umständlich und fehleranfällig ist und so erstellte Workflows schlecht lesbar wären.

Ein Vorteil von *Ant* ist, dass es von Eclipse unterstützt wird und *Ant*-Dateien auch direkt aus Eclipse heraus gestartet werden können. Außerdem bietet der XML-Editor von Eclipse eine gewisse Hilfestellung beim Erstellen von XML-Dateien. Syntaxhighlighting und automatische Einrückung machen die Texte lesbarer und auch eine einfache Autovervollständigung existiert, die die in *Ant* bekannten Schlüsselwörter und Tasks vorschlägt. Dadurch ist das manuelle Erstellen von Build-Dateien komfortabler möglich, als das Erstellen von Makefiles.

Wirklich geeignet ist *Ant* allerdings für die hier beschriebenen Workflows nicht, da die Fehlerbehandlung nicht ausreichend ist. Wie bei *make* werden auch hier Fehler erst nach dem Ausführen eines Workflows sichtbar, da *Ant* keine Informationen über die Arbeitsweise der Tools bzw. der eigenen Tasks hat und daher vor der Ausführung keinerlei semantische Prüfung durchführen kann.

3.5.3 Die Modeling Workflow Engine

Einen direkt auf Workflows zugeschnittenen Ansatz bietet die Modeling Workflow Engine (MWE). Diese Engine ist als Teil des Projekts openArchitectureWare entstanden und wurde entwickelt, um die im Projekt vorhandenen Modellierungstools, wie zum Beispiel Transformations- und Validierungstools oder Codegeneratoren, zu orchestrieren. Die verschiedenen Tools werden hierbei als Workflow-Komponenten bezeichnet und aus ihnen lässt sich ein Workflow zusammensetzen. Neben den schon existierenden, mit ausgelieferten Tools hat man die Möglichkeit, seine eigenen

Workflow-Komponenten zu erstellen und zu verwenden. Hierfür muss man eine Java-Klasse bereitstellen, die ein entsprechendes Interface implementiert.

Die Art der Workflow-Beschreibung geschah in der ersten Version von MWE noch in XML, was wie schon angesprochen nicht sehr gut handhabbar ist. In der aktuellen Version, MWE2, wurde allerdings eine eigene, auf die Anforderungen der Engine zugeschnittene DSL entwickelt, für die es auch ein Eclipse-Plugin mit IDE-Unterstützung gibt. Auf diese Weise ist das Erstellen der Workflows benutzerfreundlicher, da hier die typischen Eclipse-Features umgesetzt wurden und zum Beispiel eine Autovervollständigung für die in der Engine benutzbaren Workflow-Komponenten existiert.

Es folgt nun eine kurze Einführung in die Benutzung der MWE2 und es wird untersucht, wie die Workflows, die in dieser Arbeit umgesetzt werden sollen, in ihr realisiert werden können. Anschließend sollen Vor- und Nachteile dieses Ansatzes abgewägt und so die für die eigene DSL verwertbaren Konzepte extrahiert werden.

In jeder MWE2-Datei wird typischerweise ein einzelner Workflow beschrieben der aus mehreren Workflow-Komponenten besteht. Dies sieht in der Syntax wie folgt aus:

```
Workflow {
        component = MyComponent1 {
                ...
        }
        component = MyComponent2 {
                ...
        }
        ...
}
```

wobei *MyComponent1*, *MyComponent2* usw. Namen von Klassen sind, die das Interface *IWorkflowComponent* implementieren, die also gültige Workflow-Komponenten sind. Innerhalb der geschweiften Klammern einer Komponente können nun Attribute der Klasse gesetzt werden.

```
component = MyComponent {
        varBoolean = true
        varString = "abc"
}
```

Hierbei wird erwartet, dass die Klasse einen Setter bereit stellt, der eine bestimmte Form hat, dadurch kann die Engine erkennen, ob dieses Attribut existiert und kann es entsprechend setzen. Hierbei noch zu bemerken ist, dass in der aktuellen Version schon während des Erstellens eines Workflows die Klasse auf dieses Attribut überprüft wird und im Eclipse-Editor eine Fehlermeldung erscheint, falls es nicht vorhanden ist. Dadurch erhält der Benutzer bereits vor dem Ausführen des Workflows eine Rückmeldung über dessen Korrektheit. Über das Setzen der Attribute können also Parameter an die Tools übergeben werden.

Eine weitere Frage ist, wie die einzelnen Workflow-Komponenten miteinander kommunizieren können. So muss zum Beispiel eine geladene Modellinstanz vom Ladetool an ein Transformationstool weitergegeben werden und dessen Ergebnis dann wiederum von einem anderen Tool visualisiert werden. Für diese Aufgabe gibt es in der MWE sogenannte Slots. Ein Slot

ist nichts anderes als ein Eintrag in einer Map bzw. einem Dictionary, also einer Datenstruktur, die ein Mapping von Strings auf Objekte realisiert. Ein Workflow hat hierfür einen Kontext, der solch eine Map enthält und die allen Workflow-Komponenten bekannt gemacht wird. Wenn also ein Tool einen bestimmten Input eines anderen Tools benötigt, so kann entweder schon beim Programmieren der entsprechenden Komponenten ein fester Slot bestimmt werden, über den die Kommunikation stattfindet, oder es wird ein entsprechendes Attribut implementiert, dessen Wert dann über den oben beschriebenen Mechanismus im Workflow definiert wird. Das bedeutet, dass der Kommunikationsslot mit im Workflow festgelegt wird, was natürlich mehr Flexibilität bietet und wodurch eventuelle Überschneidungen in der Verwendung von Slots vermieden werden können. Ein Beispiel:

```
component = LoaderComponent {
          filename = "path/file.ext"
          outputSlot = "slot1"
}
component = TransformationComponent {
          inputSlot = "slot1"
          outputSlot = "slot2"
}
component = VisualizationComponent {
          inputSlot = "slot2"
}
```

Dem Ladetool (*LoaderComponent*) wird sowohl der Name der zu ladenden Datei (*filename*) übergeben als auch der Name des Slots in dem das Ergebnis des Ladevorgangs abgelegt werden soll (*outputSlot*). Derselbe Slot wird dann als Input des Transformationstools gesetzt und das gleiche passiert mit einem anderen Slot im nächsten Schritt für die Visualisierung.

Das sind die essenziellen Elemente der MWE mit denen auf einfache und kompakte Art und Weise Workflows umgesetzt werden können. Die Abarbeitung geschieht dabei sequentiell in der Reihenfolge wie die Komponenten im Workflow verwendet werden.

Die Frage ist nun, was hieraus für die zu entwickelnde DSL wichtig ist und welche Nachteile es hier gibt, die in der DSL vermieden werden sollen.

Ein entscheidender Punkt ist die Erweiterbarkeit der MWE um neue Workflow-Komponenten. Dies soll auch in der DSL möglich sein und der Mechanismus, über den dies in der MWE geschieht, ist die Java-Reflection, über die die Struktur der Komponenten erfragt wird. Das bietet die Möglichkeit beliebige Arten von Workflow-Komponenten zu unterstützt. Diese große Flexibilität ist allerdings für die zu entwickelnde DSL nicht notwendig, da die Tools und ihre Funktionsweisen bekannt sind. Dadurch können spezielle Interfaces festgelegt werden, die von vornherein bekannt sind und während die MWE nicht weiß, was die Komponenten für eine Aufgabe haben und wie sie zusammenarbeiten, ist das im Fall der DSL bekannt und kann genutzt werden, um das Erstellen eines Workflows für den Benutzer komfortabler zu gestalten. So ist die Verwendung von Slots zwar flexibel, da beliebige Objekte in einem Slot gespeichert werden können, im vorliegenden Fall ist aber bekannt, dass nur Modellinstanzen verarbeitet werden sollen und damit kann eine Variable neben der Modellinstanz noch zusätzliche Informationen enthalten, zum Beispiel das zugehörige Modell der Instanz. Durch dieses Zusatzwissen kann innerhalb des Workflows eine weitergehende Fehleranalyse durchgeführt werden, zum Beispiel kann geprüft werden, ob der Output eines Tools geeignet ist für die Verwendung als Input eines anderen Tools. Solche zusätzlichen, impliziten Informationen, die die DSL enthalten kann, müssen in der MWE explizit angegeben werden. Das

heißt, dass hier die Verbindung zwischen Modellinstanz und Modell nicht implizit geschieht, sondern unter Umständen immer mehrere Variablen verwendet werden müssen.

Solche und andere semantischen Aspekte lassen sich in der MWE wegen der benötigten Flexibilität nur sehr bedingt umsetzen. Es wäre denkbar die Zusatzinformationen explizit im Workflow, in Form von Variablen, anzugeben und die Tools vor der Ausführung diese Informationen prüfen zu lassen. Allerdings muss diese Prüfung dann vom Programmierer zur Verfügung gestellt werden.

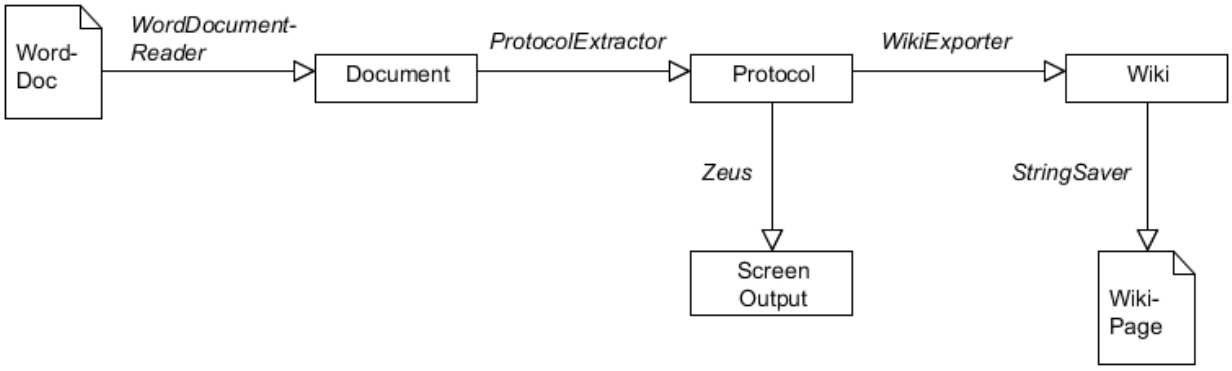
Insgesamt ist die MWE geeignet, Workflows, wie sie in dieser Arbeit vorkommen, umzusetzen. Allerdings ist sie nicht auf das konkrete Problem bezogen, sondern bietet Unterstützung für verschiedenste Arten von Workflows, deshalb kann das spezielle Wissen über die Aufgabe der Tools nicht von der Sprache verwendet werden. Um eine optimale Benutzbarkeit zu gewährleisten können also in einer eigenen DSL weitere Features angeboten werden, die in der MWE nicht umsetzbar sind.

4 Sprachentwurf

In diesem Kapitel wird erörtert, welche Konzepte in die Sprache eingehen, um die Anforderungen zu erfüllen und wie diese Konzepte in der Syntax der Sprache umgesetzt werden.

4.1 Konzepte der DSL

Hier werden die Aufgaben und Fähigkeiten der DSL in ihren Grundideen beschrieben, bevor es weiter unten um die konkrete Umsetzung dieser Ideen geht. Dafür wird wieder der Beispiel-Workflow aus Abschnitt 3.1 verwendet:



Die Unterteilung der Tools in die Kategorien

- Tools zum Laden einer Modellinstanz aus einer Datei
- Tools für Modell-zu-Modell-Transformationen
- Tools zum Speichern einer Modellinstanz in eine Datei
- Tools zum Visualisieren einer Modellinstanz

soll sich in der Sprache wiederfinden.

4.1.1 Ausführen eines Tools unter Verwendung seines Bezeichners

Eine offensichtlicher Anwendungsfall ist der, dass der Benutzer der DSL ein spezielles Tool auswählen und benutzen möchte um eine Aufgabe zu erfüllen. Zum Beispiel möchte er angeben, dass mithilfe des *WordDocumentReader*s eine Datei geladen werden soll. Dazu muss es möglich sein, den Bezeichner des Tools anzugeben und die Art der Aktion, die ausgeführt wird. Damit ist gemeint, dass es unterschiedliche Befehle für die jeweiligen Tool-Kategorien gibt. Dafür werden hier aus den Interfaces der Tools die Informationen ermittelt, die alle Tools einer Kategorie gemeinsam haben, um so die Anforderungen an den jeweiligen Befehl zu ermitteln.

4.1.1.1 Laden einer Modellinstanz aus einer Datei

Beim Laden ist auf jeden Fall der Name der Datei anzugeben. Aus dieser wird dann eine Modellinstanz gelesen, die im weiteren Verlauf des Workflows verarbeitet werden soll. Dafür muss sie gespeichert werden, um für andere Tools zur Verfügung zu stehen. Hierfür werden in der DSL Variablen verwendet, die während des Ladebefehls erstellt werden. Das heißt, es ist auch eine Variable anzugeben, in der die Modellinstanz gespeichert wird. Mit diesen Informationen und dem Bezeichner des Tools kann das Laden ausgeführt werden.

4.1.1.2 Transformation einer Modellinstanz

Für das Transformieren einer Modellinstanz in eine Instanz eines anderen Modells reicht es aus, das Tool und die Eingabeinstanz anzugeben. Das Tool weiß, wie diese zu verarbeiten ist, kann also diesen Schritt durchführen. Da die Modellinstanzen in Variablen liegen, muss hierfür ein Variablenname angegeben werden. Das Ergebnis kann dann diese Variable überschreiben, das heißt, der alte Inhalt der Variable ist dann nicht mehr zugänglich. Das ist nicht immer gewünscht, also kann optional noch eine weitere Variable angegeben werden, in der das Ergebnis der Transformation gespeichert wird.

4.1.1.3 Speichern einer Modellinstanz in einer Datei

Zum Speichern einer Modellinstanz sind deren Variable und die Zieldatei notwendig. Dabei kann, im Gegensatz zum Laden und Transformieren, das Ergebnis hier nicht in einer Variablen abgelegt werden.

4.1.1.4 Visualisieren einer Modellinstanz

Zum Visualisieren wird neben dem Tool nur die Variable benötigt in der die Instanz liegt, die gezeigt werden soll. Das Ergebnis der Visualisierung kann nicht weiter im Workflow verarbeitet werden.

4.1.1.5 zusätzliche Parameter

Neben den genannten Informationen bieten einige Tools zusätzliche Konfigurationsmöglichkeiten, die individuell unterschiedlich einzustellen sind. Obwohl in der DSL großer Wert auf die Konsistenz in der Benutzung der Tools gelegt wird, soll es die Möglichkeit geben, solche Zusatzeinstellungen für bestimmte Tools vorzunehmen. Aus diesem Grund wird für alle Tools die Möglichkeit eingeführt, zusätzliche Parameter in Form von Strings anzugeben. Das heißt, alle beschriebenen Befehle können zusätzlich noch Parameter beinhalten.

4.1.2 Eigenständiges Auswählen eines geeigneten Tools für eine bestimmte Aufgabe

Die Tools arbeiten auf Modellinstanzen und jedes Tool kennt die Ein- und Ausgabemodelle, mit denen es arbeitet. Diese Informationen können der DSL zur Verfügung gestellt werden, wodurch es

möglich wird, anstelle eines konkreten Tools, wie in Abschnitt 4.1.1 beschrieben, nur das Modell anzugeben von dem man eine Instanz als Resultat bekommen möchte. Im Workflow-Beispiel wird zunächst eine Datei geladen und dabei in eine Instanz des Modells *Document* umgewandelt. Nun kann der Benutzer das Tool, hier den *WordDocumentReader*, explizit benennen, doch ist es auch möglich und in manchen Situationen ausdrucksstärker, das Zielmodell anzugeben, also in diesem Fall zu sagen, dass die Datei als *Document* geladen werden soll. Der DSL sind alle vorhandenen Tools bekannt und sie kennt die Ein- und Ausgaben und kann daher das passende Tool automatisch auswählen.

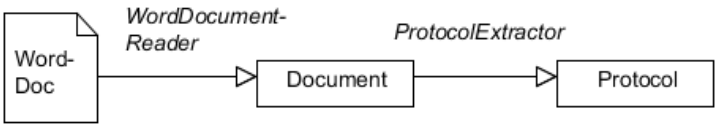
Dabei ist zu bemerken, dass es möglicherweise mehrere Tools gibt, deren Zielmodell gleich ist. Die beiden Tools *WordDocumentReader* und *ExcelReader* lesen beide eine Datei ein und haben als Ausgabe eine Instanz des Modells *Document*. Werden sie im selben Workflow verwendet und als Ziel des Ladens das Modell *Document* angegeben, kann die DSL nicht ohne zusätzliche Informationen entscheiden welches Tool verwendet werden soll.

In so einem Fall gibt es zwei Möglichkeiten, dieses Problem zu lösen: Entweder verwendet die DSL dann ein beliebiges passendes Tool, was allerdings zu Undurchsichtigkeit für den Benutzer führt, da er in einem solchen Fall nicht vorhersehen kann, wie sich die Sprache verhalten wird. Die zweite und hier gewählte Möglichkeit ist, dass solch eine Situation zu einem Fehler führt, der dem Benutzer mitgeteilt wird. Durch die Unterstützung durch die Eclipse-IDE kann das direkt nach Erstellen des Befehls in der DSL geschehen und es können Vorschläge gemacht werden, welche Tools an dieser Stelle eingesetzt werden können, sodass der Benutzer einen schnellen Überblick über die Alternativen erhält. Das bedeutet, dass in diesem Fall die Tools *WordDocumentReader* und *ExcelReader* als mögliche Alternativen vorgeschlagen werden.

Die automatische Erkennung eines Tools lässt sich sinnvoll für Lade- und Transformationsschritte anwenden, da hier das Ergebnis eine Modellinstanz ist. Beim Speichern und Visualisieren ist dies allerdings nicht anwendbar, da das Ergebnis eine Datei bzw. der Start eines Visualisierungsprogramms ist und damit kein Zielmodell existiert. Allerdings lässt sich dadurch, dass die Tools angeben, welche Modelle sie verarbeiten eine andere Vereinfachung umsetzten: Das verwendete Tool muss nicht mehr explizit angegeben werden, es reicht die Variable mit der Modellinstanz, die gespeichert bzw. visualisiert werden soll. Zu dieser ist auch das Modell bekannt und somit kann automatisch ein passendes Tool zum Speichern bzw. Visualisieren ermittelt werden. Auch hierbei können Fehler entstehen, zum Beispiel wenn es zwei Speichertools gibt, die das gleiche Modell in verschiedenen Weisen speichern. In diesen Fällen muss wiederum der Benutzer ein Tool auswählen.

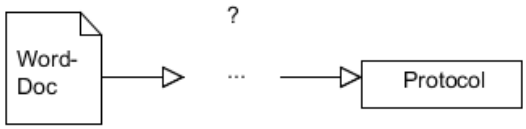
4.1.3 Eigenständige Verknüpfung mehrerer Tools für eine bestimmte Aufgabe

Betrachtet man den Beispiel-Workflow, so stellt man fest, dass die Datei, die geladen wird, zuerst als Instanz des Modells *Document* vorliegt, nur um von da aus in eine Instanz des Modells *Protocol* umgewandelt zu werden. Das heißt, dass der erste Schritt für die Korrektheit des Workflows zwar notwendig ist, aber das Zwischenergebnis nicht weiter gebraucht wird.



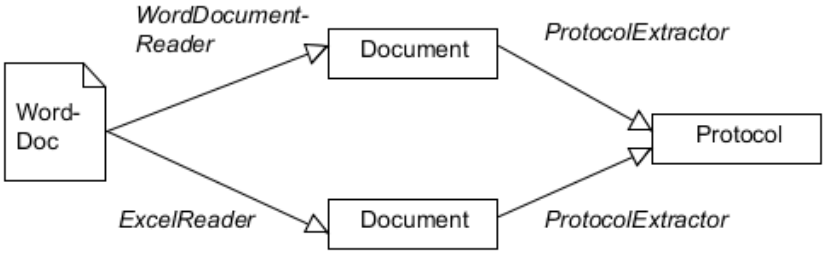
Das *Document* wird hier nicht weiter benötigt.

Aus diesem Grund soll es in der DSL möglich sein, in diesem Fall anzugeben, dass die Datei als Instanz von *Protocol* geladen werden soll, wodurch der Zwischenschritt des expliziten Ladens als *Document* in der Workflow-Beschreibung ausbleiben kann. Was dann im Interpreter passiert ist, dass versucht wird, aus den vorhandenen Tools eine Hintereinanderausführung derart vorzunehmen, dass das gewünschte Resultat produziert wird.



Der Benutzer gibt also an, dass eine Datei als *Protocol* geladen werden soll. Daher wird zunächst nach einem Tool gesucht, das in der Lage ist, eine Datei direkt als *Protocol* zu laden. Erst wenn diese Suche fehlgeschlagen ist wird versucht über Umwege auf das gewünschte Ergebnis zu kommen. Dazu würde im Beispiel zunächst das Laden als *Document* als möglicher Anfang gefunden und durch den Transformationsschritt vom *Document* zum *Protocol* mithilfe des *ProtocolExtractor* ist eine Lösung gefunden. Natürlich kann es auch noch andere Wege geben und es können beliebig viele weitere Transformationstools angehängt werden, bis der gewünschte Output entsteht, wobei eine doppelte Verwendung desselben Tools ausgeschlossen wird, da man davon ausgehen kann, dass eine mehrfache Anwendung einer Transformation auf die gleiche Instanz keinen Mehrwert bringt.

Probleme können hierbei entstehen, wenn es für die Umformung in ein bestimmtes Modell mehrere Möglichkeiten gibt. Werden im Beispiel-Workflow zum Beispiel sowohl *WordDocumentReader* als auch *ExcelReader* verwendet, die beide ein *Document* liefern, kommt es zu einem Problem.



Zunächst wird kein Tool gefunden, das in einem Schritt aus einer Datei ein *Protocol* machen kann. Deshalb wird versucht eine Kette von Tools zu finden, mit der dies möglich ist. Allerdings gibt es hier zwei Möglichkeiten, zum einen über den *WordDocumentLoader* und dann den *ProtocolExtractor* und zum anderen über den *ExcelReader* und dann den *ProtocolExtractor*. In diesem Fall kann nicht automatisch entschieden werden, was zu tun ist, also wird auch hier, wie im Beispiel des vorigen Abschnitts, eine Fehlermeldung an den Benutzer ausgegeben und dieser muss darüber entscheiden welche Kombination von Tools verwendet werden soll. Dabei können ihm die ermittelten Kombinationen vorgeschlagen werden.

Damit wird ein Konzept für die Verkettung von Tools benötigt. Um solch eine Verkettung oder Hintereinanderausführung festlegen zu können, wird ein eigenes Konstrukt in der Sprache eingeführt, durch das ein neues Tool erstellt werden kann, das einen eigenen Bezeichner bekommt und sich aus einer Untermenge vorhandener Tools zusammensetzt, die in der Reihenfolge in der sie angewendet werden sollen angegeben werden.

Hier schließen sich zwei Fragen an. Die erste Frage ist, wie ein solches Tool Parameter verarbeitet. Solch eine Kette besteht aus mehreren Tools, die jeweils feste und optionale Parameter besitzen können. Parameter werden an eine Tool-Kette als eine Liste von Strings übergeben, wie auch bei anderen Tools. Die Zuordnung der Parameter zu den einzelnen Tools der Kette ist nicht möglich, wenn es optionale Parameter gibt, da in so einem Fall ein Parameter unter Umständen mehreren Tools zugeordnet werden kann, je nachdem ob man ihn als optionalen Parameter des einen Tools oder als festen Parameter des nachfolgenden Tools betrachtet. Aus diesem Grund wird festgelegt, dass bei der Verwendung einer Tool-Kette keine optionalen Parameter möglich sind. Das heißt, dass alle Parameter so auf die einzelnen Tools der Kette verteilt werden, dass jedes Tool seine feste Parameterzahl erhält. Dabei wird die Reihenfolge eingehalten, in der die Tools in der Kette verwendet werden. Der Komfort, eine Tool-Kette benutzen zu können, wird also gegen die Möglichkeit optionaler Parameter eingetauscht.

Die zweite Frage bei Tool-Ketten ist, welche Kombinationen sinnvoll sind und daher unterstützt werden sollten und welche nicht. Nach den bisher ermittelten Anforderungen gibt es vier verschiedene Befehle in der DSL mit denen die vier Tool-Arten verwendet werden können. Erstellt man nun ein neues Tool als Kombination aus den vorhandenen, so muss auch dieses neue Tool einer der Kategorien eindeutig zuzuordnen sein, damit eine Verwendung mit dem entsprechenden Befehl möglich ist. Das schließt zum Beispiel eine Kette aus, die aus Laden, Transformieren und Speichern besteht, da das entstehende Tool sowohl als Ladetool als auch als Speichertool gesehen werden kann. Es werden damit sowohl eine Datei zum Lesen als auch eine Datei zum Speichern benötigt und für derartige Fälle müssten neue Befehle eingeführt werden, die mehrere Aufgaben vermischen. Auf solche Befehle soll aber verzichtet werden, um die verschiedenen Aufgaben der Tools auch in der Sprache klar voneinander zu trennen. Außerdem ist fraglich wie groß der Nutzen durch beliebige Kombination von Tools im Vergleich zum Aufwand beim Erstellen der DSL ist. Die Annahme ist hier, dass solche Kombinationen sehr selten verwendet würden und darum wird auf die Implementierung verzichtet.

Es werden nun die als sinnvoll erachteten Kombinationen von Tools und mögliche Anwendungen vorgestellt.

4.1.3.1 Laden einer Modellinstanz aus einer Datei

Beim Laden einer Datei wird deren Inhalt von einem Tool direkt in eine Instanz eines bestimmten

Modells umgewandelt. Im Beispiel werden sowohl Word-Dateien als auch Excel-Dateien zunächst als *Document* geladen und es gibt hier kein Tool zum Laden einer Datei das ein anderes Modell als Ergebnis hat. Möchte man nun ein oder mehrere Protokolle laden, also aus Word-Dokumenten Instanzen des Modells *Protocol* erstellen, so muss man immer zuerst den Schritt über das Modell *Document* machen, obwohl man die dabei entstehende Instanz nie verwendet. Hierfür ist es also sinnvoll Kombinationen von Tools zu erlauben, an deren Anfang ein Ladetool steht und das gefolgt wird von einer beliebigen Anzahl Transformationstools. Die entstehende Tool-Kette wird dann wie ein Ladetool behandelt.

4.1.3.2 Transformieren einer Modellinstanz

Im Beispiel wird ein *Document* in ein *Protocol* und dann in ein *Wiki* umgewandelt. Hier werden also zwei Transformationsschritte hintereinander ausgeführt, was man auch wie in einer Black-Box in einem Transformationsschritt von *Document* nach *Wiki* verstecken könnte. Zu diesem Zweck können neue Transformationen eingeführt werden, die sich aus einer beliebigen Anzahl anderer Transformationen zusammensetzen.

4.1.3.3 Speichern und Visualisieren von Modellinstanzen

Auch beim Speichern und Visualisieren können Transformationsschritte vorgeschaltet sein, die nicht explizit angegeben werden müssen. So möchte man vielleicht mehrere Instanzen von *Document* als PDF-Datei speichern. Angenommen, dass dafür ein zusätzlicher Transformationsschritt in ein PDF-Modell nötig ist, da es kein Tool gibt, das *Document* speichern kann, so kann man auf das explizite Umwandeln der Dokumente verzichten. Es wird dann versucht, geeignete Transformationen durchzuführen, in diesem Fall in das PDF-Modell, bis ein Modell entsteht, das sich mit einem Tool speichern lässt. Dafür wird dann in der Sprache eine entsprechende Kette aus Transformationstools und Speichertool erstellt und verwendet. Deshalb werden auch Kombinationen aus beliebig vielen Transformationen und abschließend einem Speicher- oder Visualisierungstool unterstützt.

4.1.4 Definition von Stringliteralen

Ein letzter Punkt, der kein wirkliches Konzept, sondern mehr eine Art Vereinfachung in der DSL darstellt ist, die Definition von Variablen, die Strings enthalten. In den Workflows wird mit Dateinamen gearbeitet und bei der Angabe von absoluten Pfaden können lange Strings entstehen, die die Lesbarkeit des DSL-Codes beeinträchtigen können. Daher soll es möglich sein, stattdessen Variablen mit kurzen und prägnanten Namen definieren zu können, die anstelle von Dateinamen und Parametern eingesetzt werden können.

4.2 Syntax

In diesem Abschnitt wird die Syntax der Sprache beschrieben. Dafür werden zunächst die vier Befehle gezeigt, mit denen die verschiedenen Tool-Arten ausgeführt werden können. Anschließend wird das Erstellen einer Tool-Kette beschrieben.

4.2.1 Laden einer Datei

Das Laden einer Modellinstanz aus einer Datei geschieht über den Befehl *load*.

```
load "filename" using LoaderID into variable;
```

Beim Laden wird der Name der Datei angegeben und es folgt nach dem Schlüsselword *using* das Tool, das benutzt werden soll. Hierbei wird der in der Konfiguration festgelegte Bezeichner des Tools verwendet. Der letzte Teil des Befehls gibt an in welcher Variable das Ergebnis des Ladens abgelegt wird.

Anstatt über *using* das konkrete Tool anzugeben kann hier auch das Zielmodell festgelegt werden:

```
load "filename" as ModelID into variable;
```

Hier wird automatisch ein Tool oder eine Kette von Tools für das Laden ausgewählt, bzw. eine Fehlermeldung erzeugt wenn es mehr als eine Möglichkeit gibt das Modell zu laden. Die Konstrukte *using LoaderID* und *as ModelID* können nicht beide im selben Befehl verwendet werden, da durch die Angabe eines Tool-Bezeichners auch das Zielmodell implizit festgelegt wird.

Jedes Tool kann zusätzliche Parameter benötigen. Diese werden am Ende des Befehls nach dem Schlüsselwort *params* eingefügt:

```
load "filename" as ModelID into variable params "parameter 1" "parameter 2";
```

Die Angabe der Parameter ist nur notwendig wenn das Tool mindestens einen Parameter verlangt, ansonsten kann sie weggelassen werden.

Der Befehl *load* hat einige Gemeinsamkeiten mit den anderen Befehlen, die hier einmal angesprochen werden und für alle Befehle gelten. Zunächst ist die Wahlmöglichkeit zwischen der Angabe eines Tools oder des Zielmodells überall gleich. Die Angabe des Tool-Bezeichners erfolgt jeweils nach dem Schlüsselwort *using* und die des Modells jeweils nach *as*, ausgenommen beim Transformieren (siehe unten).

Die nächste Gemeinsamkeit ist die Angabe der Parameter. Sie erfolgt immer am Ende des Befehls und kann wegfallen, wenn das gewählte Tool keine Parameter benötigt.

4.2.2 Transformieren einer Modellinstanz

Der nächste Befehl in der DSL ist *transform*, über den eine Modellinstanz transformiert werden kann:

```
transform variable to ModelID;
```

Hier wird anstatt *as* das Schlüsselwort *to* verwendet, weil es sprachlich besser passt, da eine Instanz nicht *als* ein Modell transformiert wird, sondern *in* ein Modell. Auch hier ist die Angabe eines Tools oder zusätzlicher Parameter möglich. Außerdem ist es möglich eine Zielvariable anzugeben, in der dann das Ergebnis der Transformation gespeichert wird:

```
transform variable to ModelID into otherVar;
```

Wird keine Zielvariable angegeben, so wird die Quellvariable überschrieben.

4.2.3 Speicher einer Datei

Der Befehl zum Speichern einer Modellinstanz in eine Datei ist *save*.

```
save variable to "filename";
```

Im Gegensatz zu *load* und *transform*, bei denen entweder das Tool oder das Zielmodell benötigt wird, kann beim Speichern neben der Variablen und dem Dateinamen auf weitere Angaben verzichtet werden. Das Modell ist implizit durch die Variable bekannt, deshalb kann ein passendes Tool ausgewählt werden.

Falls kein Tool gefunden wird, das die Variable in einem Schritt speichern kann, wird versucht über eine Kette von Transformationen die Instanz auf ein Modell zu bringen, für das es ein Speichertool gibt. Dies passiert mit der gleichen Syntax, also ohne Angabe eines expliziten Tools, und wird automatisch umgesetzt. Zu bemerken ist hierbei, dass der Inhalt der Variable sich durch die Transformationen nicht ändert, das heißt, die Variable behält auch nach diesem Schritt das gleiche Modell. Die Transformationen werden nur intern ausgeführt, die resultierende Modellinstanz in der Datei gespeichert, anschließend wird die Instanz verworfen.

Natürlich kann auch beim Speichern ein explizites Tool angegeben werden:

```
save variable using SaverID to "filename";
```

4.2.4 Visualisieren einer Modellinstanz

Ähnlich wie beim Speichern wird auch beim Visualisieren, was über den Befehl *show* möglich ist, kein explizites Modell benötigt. Dadurch ist der folgende Befehl völlig ausreichend um eine Modellinstanz mit einem parameterlosen Visualisierungstool anzuzeigen:

```
show variable;
```

Neben dem Tool lässt sich auch hier, analog zum Speichern, ein explizites Tool angeben oder es werden gegebenenfalls eigenständig Tool-Ketten gebildet.

4.2.5 Tool-Ketten definieren

Für das Definieren einer Kette von Tools gibt es das Schlüsselwort *toolchain*.

```
toolchain WikiLoader: DocLoader, DocToProtocol, ProtocolToWiki;
```

Hier wird ein Loader definiert, der das aus dem Beispiel-Workflow bekannte Problem löst, dass eine Datei als Dokument geladen, dann in ein Protokoll und dann in ein Wiki-Markup transformiert werden soll. Das neu entstandene Tool bekommt den Bezeichner *WikiLoader*, der im Workflow mit dem Schlüsselwort *using* verwendet werden kann. Danach folgen, in der Reihenfolge in der sie angewendet werden sollen, die Bezeichner der Tools, aus denen die Kette besteht.

4.2.6 Definition von Stringliteralen

Der letzte Teil der Syntax, der gezeigt wird ist die Definition von Stringliteralen:

```
docFile = "examples/Protokoll.doc";
```

Hier wird eine Variable *filename* definiert, in der ein String gespeichert wird. Diese Variable kann im weiteren Verlauf des Workflows anstelle von Dateinamen und Parametern eingesetzt werden, zum Beispiel wie folgt:

```
load docFile as Protocol into protocol;
```

Das erhöht die Lesbarkeit des DSL-Codes.

4.2.7 Die Grammatik der Sprache

Für einen Einblick in die Art, wie in Xtext eine Grammatik beschrieben wird, folgt ein Ausschnitt aus der Grammatikdefinition der DSL. Die gesamte Definition befindet sich im Anhang in Abschnitt 8.2.

```
Action:
          (LoadAction | TransformAction | ShowAction | SaveAction) ('params'
          params+=Parameter (params+=Parameter)*)? ';';
```

Hier sieht man die Definition der Regel für die vier Befehle zum Ausführen von Tools. Eine *Action* beinhaltet eine der in den Klammern stehenden Aktionen und anschließend folgt die Angabe der Parameter. Hierbei steht diese in Klammern gefolgt von einem Fragezeichen. Das bedeutet, dass der Ausdruck in Klammern genau einmal oder gar nicht auftauchen kann, womit die Angabe der Parameter optional ist. Zu sehen ist weiterhin, dass Schlüsselwörter wie *params* von Apostrophen oder wahlweise von Anführungszeichen umschlossen sind. Der Ausdruck *params+=Parameter* bedeutet, dass im Klassenmodell für die Syntax ein Attribut *params* erstellt wird, das einen

Container für beliebig viele Objekte des Typs *Parameter* darstellt. Dabei ist *Parameter* in der Grammatik wie folgt definiert:

```
Parameter:
          paramString=STRING | paramID=ID;
```

Hier sieht man zwei standardmäßig von Xtext zur Verfügung gestellte Terminale, also Zeichen, die nicht weiter in Parserregeln zerlegt werden, *STRING* und *ID*. *STRING* entspricht dabei einer Zeichenkette, die in Apostrophen oder Anführungszeichen steht und *ID* beschreibt eine Zeichenfolge für Bezeichner, wie sie auch in Java erlaubt sind. Diese Terminale werden hier an die Attribute *paramString* bzw. *paramID* gebunden, die im Modell der Syntax als Strings realisiert werden. Das bedeutet, dass als Parameter zum Beispiel *„parameter“*, *'parameter'* oder *parameter* eingesetzt werden kann.

Man sieht an diesen Auszügen aus der Grammatik, dass die in Xtext verwendete EBNF sich deutlich von der normalen BNF unterscheidet. So sind beispielsweise die Bezeichner nicht in spitzen Klammern gesetzt und Definitionen von Nichtterminalen werden durch : anstatt ::= eingeleitet. Ein entscheidender Unterschied ist die Möglichkeit, Attribute, die im Klassenmodell der Syntax erstellt werden sollen direkt in der Grammatik anzugeben und dabei zwischen einfachen Attributen und Containern unterscheiden zu können. Dadurch bekommt man schon bei der Grammatikdefinition Kontrolle darüber, wie das vom Parser erzeugte Ergebnis aussehen wird.

5 Implementierung

5.1 Interfaces der Tools

An dieser Stelle werden die Interfaces beschrieben, die sich aus den Anforderungen für die Tools und den Sprachkonstrukten ergeben. Dazu folgt eine Übersicht über die vier Interfaces *ILoader*, *ITransformer*, *ISaver* und *IViewer* die sich auf die vier Tool-Arten beziehen:

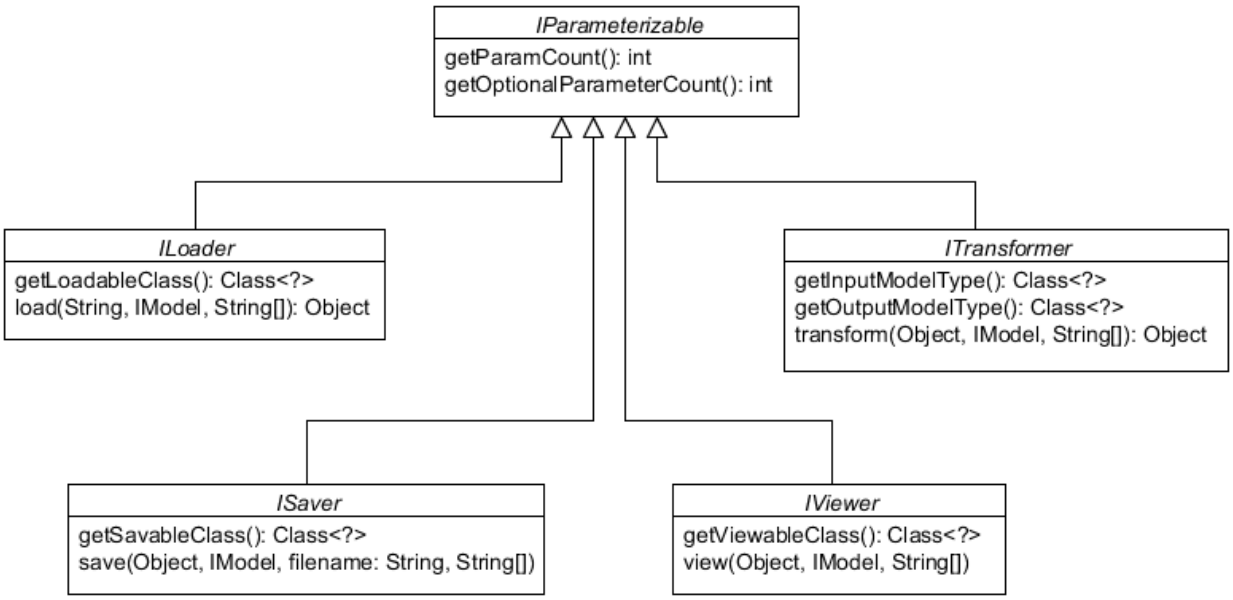


Abbildung 3: Interfaces für die Tools

In Abbildung 3 sieht man, dass alle Tools vom gemeinsame Interface *IParameterizable* erben. Über dieses Interface werden die Anzahl der zwingend erforderlichen und der optionalen Parameter angegeben, die ein Tool zur Verarbeitung benötigt. Die Übergabe der Parameter geschieht jeweils in der Methode zum Ausführen eines Tools, in Form eines String-Arrays.

Bevor die einzelnen Interfaces genauer betrachtet werden, wird das Interface *IModel*, das sich in jedem Tool-Interface als Parameter findet, erläutert.

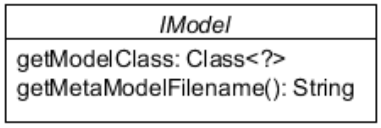


Abbildung 4: Interface für Modelle

Das Interface *IModel*, zu sehen in Abbildung 4, ist dafür da, der DSL Modelle bekannt zu machen und eventuelle Zusatzinformationen über diese Modelle bereitzustellen. Zum einen wird die Klasse

angegeben von der die Instanzen des Modells sind und zusätzlich ist es möglich einen Dateinamen anzugeben. In dieser Datei befindet sich das Modell in serialisierter Form. Im Beispiel-Workflow gibt es das Modell *Document* und dieses kann mithilfe des EMFs in eine Datei gespeichert werden. Im EMF wird die Datei als Metamodell bezeichnet, daher der Methodenname *getMetaModelFilename*, wobei es nach den in Abschnitt 2.1.2 beschrieben Modellierungsebenen als eine Form des Modells, also der Ebene M1, gesehen werden kann über der auf Ebene M2 das Ecore-Metamodell steht. Das ist allerdings eine Eigenheit der Terminologie innerhalb des EMF und ein Detail auf das hier nicht weiter eingegangen werden soll.

Neben dem Dateinamen ist es denkbar, dass in *IModel* noch andere Informationen untergebracht werden. Aktuell ist allerdings nur der Dateiname enthalten, da das konkret vorliegende Visualisierungsprogramm *Zeus* diese Datei benötigt. Da möglicherweise in Zukunft Tools entstehen, die diese Information ebenfalls benötigen, wird schon jetzt beim Ausführen eines Tools jeweils das zugehörige Modell übergeben.

Nun werden die einzelnen Tool-Interfaces betrachtet, beginnend mit *ILoader*:

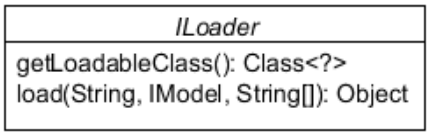


Abbildung 5: Interface für Tools zum Laden von Dateien

In Abbildung 5 ist das Interface für Tools zu sehen mit denen Dateien geladen werden können. Um die Kompatibilität der Tools in der DSL prüfen zu können, wird hier über die Methode *getLoadableClass* die Klasse angegeben zu der die Instanzen gehören, die geladen werden. Dadurch ist es zum Beispiel möglich, zu prüfen, ob das Ergebnis des Ladens als Eingabe eines Transformationsschrittes verwendet werden kann, indem über die Java-Methode *Class.isAssignable* getestet wird, ob der Eingabeklasse die Ausgabe des vorherigen Schritts zugewiesen werden kann.

Das eigentliche Laden wird über die Methode *load* durchgeführt, deren erster Parameter ein String ist, der den Namen der zu ladenden Datei enthält. Der zweite Parameter ist das angesprochen *IModel* über das eventuell benötigte Zusatzinformationen über das Modell abgerufen werden können. Der dritte Parameter ist ein String-Array in dem die in der DSL angegebenen Parameter enthalten sind. Der Rückgabewert ist die geladene Modellinstanz. Des Weiteren kann beim Laden ein Fehler auftreten, weshalb die Methode möglicherweise eine *Exception* wirft.

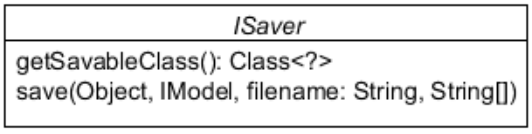


Abbildung 6: Interface für Tools zum Speichern von Dateien

In Abbildung 6 ist das Interface für das Speichern von Dateien zu sehen. Dies geschieht analog zum

Laden. Dazu wird hier, über *getSavableClass*, die Klasse angegeben, die als Eingabe akzeptiert wird. In der Methode *save* wird als erstes die zu speichernde Modellinstanz übergeben, anschließend folgt auch hier das Modell. Der dritte Parameter ist der Name der Datei die gespeichert wird und am Ende folgen die Parameter.

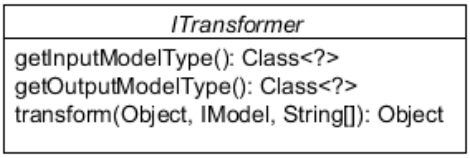


Abbildung 7: Interface für Transformationstools

Abbildung 7 zeigt die Schnittstelle für Transformationsschritte. Hierbei gibt es ein Eingabe- und ein Ausgabemodell, deren Klassen abgefragt werden können. Die Methode *transform* erhält die zu transformierende Modellinstanz, das Modell und die Parameter und gibt die nach der Transformation entstehende Modellinstanz zurück.

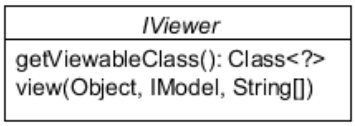


Abbildung 8: Interface für Visualisierungstools

Visualisierungen geschehen, wie in Abbildung 8 zu sehen, analog wie das Speichern, mit dem Unterschied, dass hier keine Zieldatei angegeben werden muss.

Neben diesen Schnittstellen für Modelle und Tools gibt es die Möglichkeit, dass ein Tool zusätzlich das Interface *IParamChecker* implementiert. Hierdurch ist es möglich, die Parameter, die in der DSL als Strings übergeben werden, vor dem Ausführen des Workflows durch die Tools prüfen zu lassen, wodurch in der Eclipse-IDE eine direkte Rückmeldung an den Benutzer gemacht werden kann, falls ein Parameter fehlerhaft ist.

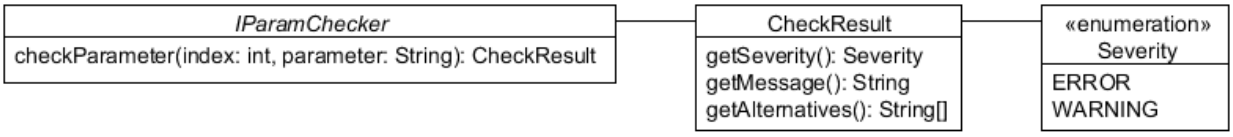


Abbildung 9: Zusätzliches Interface für Parameterprüfung

In Abbildung 9 ist das Interface *IParamChecker* gezeigt. Es hat nur eine Methode, die für jeden Parameter einzeln aufgerufen wird. Hierbei werden der Index des Parameters und der Parameter selbst angegeben und, falls dieser fehlerhaft ist, gibt die Methode ein entsprechendes *CheckResult* zurück. Hierin enthalten ist die Schwere des Fehlers (*severity*), die angibt, ob ein echter Fehler oder

lediglich eine Warnung erzeugt werden soll. Außerdem gibt es eine Nachricht als String, die den Fehler näher beschreibt. Wenn es Alternativen zum fehlerhaften Parameter gibt so können diese im *CheckResult* angegeben werden und in der IDE werden dadurch Quick-Fixes angeboten um den Parameter auszutauschen. Die Alternativen müssen nicht zwingend angegeben werden, da in manchen Fällen nicht alle möglichen Parameter bekannt sind, oder es einfach zu viele um sie alle anzugeben, das heißt das Attribut *alternatives* muss nicht zwingend ausgefüllt werden.

Wird das Interface *IParamChecker* von einem Tool implementiert, wird dies zur Laufzeit erkannt und die Parameter werden überprüft, ansonsten wird keine Prüfung vorgenommen.

5.2 Interpreter

Im Interpreter wird der Workflow durchlaufen und die Arbeitsschritte sequentiell ausgeführt. Dabei werden die jeweiligen Tool-Klassen mit den im vorigen Abschnitt beschrieben Interfaces aufgerufen und die Ergebnisse intern gespeichert. Die Syntax der DSL und die Interfaces sind aufeinander abgestimmt sodass bei der Angabe eines Tools durch seinen Bezeichner der Aufruf ohne großen Aufwand an das Tool weitergeleitet werden kann.

Hervorzuheben ist die Art, wie automatisch Tool-Ketten erkannt und verwendet werden, wenn ein Zielmodell angegeben wird, für das kein einfaches Tool existiert. Hierzu wird bei zu Beginn der Interpretation eine Liste mit allen möglichen Kombinationen von Tools erstellt. Gültige Kombinationen sind dabei, wie in Abschnitt 4.1.3 erläutert:

- ein Loader gefolgt von beliebig vielen Transformationen
- beliebig viele Transformationen
- beliebig viele Transformationen gefolgt von einem Saver
- beliebig viele Transformationen gefolgt von einem Viewer

Da hierbei jeweils eine beliebige Anzahl Transformationen enthalten ist, werden diese zuerst ermittelt. Dafür wird für jeden bekannten Transformer von diesem ausgehend jeder mögliche Nachfolger ermittelt, wobei die Kompatibilität von Eingabe- und Ausgabemodellen überprüft wird, anschließend wird für jedes entstandene neue Ende der Kette der Vorgang wiederholt. Dabei wird in jedem Schritt der aktuelle Transformer aus den möglichen Nachfolgern entfernt, damit keine unendlich langen Ketten entstehen, in denen immer wieder die gleichen Transformationen durchgeführt werden. Eine zweimal in der gleichen Kette ausgeführte Transformation wird damit ausgeschlossen, da davon ausgegangen werden kann, dass die selbe Transformation angewendet auf die selbe Modellinstanz beim zweiten Mal keinen Mehrwert bringt.

Eine Transformationskette wird im Interpreter als ein einziger Transformationsschritt betrachtet, wobei als Eingabemodell das des ersten Transformer und als Ausgabemodell das des letzten Transformer der Kette verwendet wird.

Sind alle Transformationsketten ermittelt so lassen sich die anderen Tool-Ketten leicht finden, indem versucht wird, vor jede Transformationskette jeden Loader bzw. hinter jede Transformationskette jeden Saver bzw. Viewer anzuhängen.

Nachdem die Ketten am Anfang einmal erstellt wurden, wird bei der Angabe eines Zielmodells, für das kein Tool gefunden wurde, das die Aufgabe in einem Schritt erledigt, versucht aus der Menge der Tool-Ketten eine passende zu finden.

5.3 Unterstützung durch die IDE

Zur bestmöglichen Unterstützung der Entwicklung eines Workflows in der DSL wird ein schrittweises Interpretieren schon bei der Erstellung durchgeführt. Dabei werden die Tools noch nicht ausgeführt, doch es werden die Bedingungen geprüft, die gelten müssen, um eine korrekte Arbeitsweise des Workflows sicherzustellen, soweit das zur Erstellungszeit möglich ist. Darunter fällt das Prüfen der Kompatibilität von Tools und das Vorschlagen geeigneter Tools für eine bestimmte Aufgabe. Soll zum Beispiel eine Modellinstanz transformiert werden, so lässt sich aufgrund der von den Tools zu Verfügung gestellten Informationen über Ein- und Ausgabemodelle feststellen, ob das gewühlte Tool geeignet ist um eine bestimmte Variable zu transformieren. Ist dies nicht der Fall wird eine Fehlermeldung angezeigt und es werden passende Tools vorgeschlagen die ersatzweise verwendet werden können.

Werden bei der Angaben eines Zielmodells mehrere mögliche Tools gefunden so kann der Interpreter nicht entscheiden, welches davon zu wählen ist und auch dann werden die gefundenen Möglichkeiten per Quick-Fix angeboten. Gibt es kein eindeutiges Tool, dafür aber mehrere mögliche Ketten von Tools, die automatisch ermittelt wurden, so bekommt der Benutzer Vorschläge eine der Ketten automatisch erstellen zu lassen. Diese wird dann in der Sprache durch den Aufruf des Quick-Fix definiert und an der entsprechenden Stelle eingesetzt verwendet.

Weiterhin werden Dateinamen beim Laden überprüft und eine Warnung ausgegeben, wenn es sie nicht gibt. Analog erscheint eine Warnung, wenn beim Speichern eine Datei überschrieben wird.

Durch das Interface *IParamChecker* hat der Programmierer der Tools die Möglichkeit, fehlerhafte Parameter mithilfe von Quick-Fixes zu ersetzen.

Durch diese Features wird die Erstellung von Workflows für den Benutzer komfortabler und einfacher.

6 Beurteilung der Sprache

6.1 Vergleich der DSL mit der Realisierung als Java Applikation

Hier wird der reale Beispiel-Workflow in Java direkt mit dem entsprechenden Code der DSL verglichen. Dazu wird zunächst der Java-Code aufgelistet und dann der DSL-Code vorgestellt und erläutert um anschließend Vergleiche ziehen zu können und zu diskutieren welche Vorteile die DSL im Vergleich zur Umsetzung in Java bietet.

```
import java.io.FileInputStream;

import olymp.wikiexporter.IWikiSyntaxFactory.WikiDialect;
import de.unihannover.se.eperformance.protocolextractor.ProtocolExtractor;
import de.unihannover.se.eperformance.protocolextractor.exporter.WikiExporter;
import de.unihannover.se.eperformance.protocolextractor.protocolmodel.Protocol;
import de.unihannover.se.km.armageddon.model.Document;
import de.unihannover.se.km.armageddon.reader.word.WordDocumentReader;

public class ExampleWorkflow {
    public static void main(String[] args) {
        try {
            String filename = "examples/Protokoll.doc";
            String communicationFile = "examples/Kommunikationsliste.organization";
            String protocolModelFilename = "examples/ProtocolModel.ecore";

            WordDocumentReader reader = new WordDocumentReader();
            reader.setInputStream(new FileInputStream(filename));
            Document document = reader.getDocument();

            if (document != null) {
                ProtocolExtractor extractor = new ProtocolExtractor(document);
                Protocol protocol = extractor.getProtocol();

                if (protocol != null) {
                    WikiExporter exporter = new WikiExporter();
                    exporter.setDialect(WikiDialect.CONFLUENCE);
                    exporter.setCommunicationFilename(communicationFile);
                    exporter.setProtocol(protocol);
                    String wikiMarkup = exporter.getWikiMarkup();

                    ZeusActivator viewer = new ZeusActivator();
                    viewer.showModelInstance(protocol, protocolModelFilename,
                        ZeusActivator.DisplayType.Graph, "Protokoll");

                    StringSaver saver = new StringSaver();
                    saver.save(wikiMarkup, "examples/wiki.txt");
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Umsetzung des Workflows in Java

Für die Erläuterung dieses Codes siehe Abschnitt 3.1.1.

Es folgt der vollständige, lauffähige DSL-Code der den gleichen Beispiel-Workflow beschreibt.

```
filename = "examples/Protokoll.doc";
communicationFile = "examples/Kommunikationsliste.organization";

load filename as Protocol into protocol;

transform protocol using ProtocolToWiki into wiki params communicationFile
"CONFLUENCE";

show protocol params "Graph";

save wiki to "examples/wiki.txt";
```

Umsetzung des Beispiels in der DSL

Zur Erläuterung des DSL-Codes: Es werden zunächst, wie auch im Java-Beispiel, die verwendeten Dateinamen definiert um den Rest des Codes lesbarer zu gestalten.

Der Befehl *load* wird verwendet um die Datei einzulesen, in der sich das Protokoll befindet. Sie wird dabei direkt als Instanz von *Protocol* gelesen, wobei implizit nach dem Laden als *Document* eine Transformation zum *Protocol* durchgeführt wird. Die geladene Modellinstanz wird in der Variablen *protocol* gespeichert.

Anschließend wird die Modellinstanz unter Angabe des entsprechenden Tools in ein Wiki-Markup umgewandelt. Hierbei werden die Parameter, die in Java über Setter gesetzt wurden, als Strings übergeben. Das beinhaltet eine zusätzlich benötigte Datei und die Art des zu erstellenden Wikis (hier wird ein Confluence-Wiki erstellt). Das Ergebnis wird in der Variable *wiki* gespeichert, in *protocol* ist also weiterhin das Protokoll enthalten.

Das Visualisieren des Protokolls geschieht ohne explizite Angabe eines Tools, dieses wird automatisch aus aus dem implizit enthaltenen Modell der Variable *protocol* ermittelt. Die Arten der Visualisierung werden hier als Parameter übergeben, in diesem Fall wird nur eine Ansicht als Graph erstellt.

Im letzten Schritt wird der Wiki-Text in eine Datei geschrieben. Auch hier kann auf die Angabe eines Tools verzichtet werden.

6.1.1 Codelänge

Der Java-Code des Beispiels ist 44 Zeilen lang, während der entsprechende DSL-Code nur 10 Zeilen lang ist. Die Frage ist nun, wieso dies so ist und ob sich dadurch echte Vorteile ergeben, denn nicht immer bedeutet kürzerer Code auch besseren Code.

Zunächst ist ein großer Teil des Java-Codes, der der Infrastruktur eines Java-Programms zugeschrieben werden kann, in der DSL überflüssig. Das sind die anfänglichen Imports und die package-Bezeichnung (im Beispiel nicht vorhanden) und auch die Klasse, die die main-Methode enthält. Diese Dinge bilden in Java eine Art Rahmen, ohne der Code nicht lauffähig ist, in der DSL kann aber darauf verzichtet werden. Das ist durchaus vorteilhaft, da das Aufbauen dieses Rahmens nicht direkt der eigentlichen Aufgabe dient. Man möchte einen Workflow beschreiben und kann sich in der DSL auf genau dieses Problem alleine konzentrieren.

Weiterhin ist zu bemerken, dass in Java der eigentliche Workflow von einem try-catch-Block umschlossen ist. Dies ist notwendig, da manche der Tools beim Verrichten ihrer Aufgabe eine *Exception* werfen können. Man hätte auch für jedes dieser Tools einen eigenen try-catch-Block einfügen können, oder mehrere catch-Blöcke, die unterschiedliche *Exception*s abfangen. Damit hätte man die Möglichkeit eine genauere Fehlerauswertung vorzunehmen oder nach dem Fehlschlagen eines Schrittes trotzdem andere Tools aufzurufen, allerdings ist das im Beispiel nicht sinnvoll, da die Tools jeweils die vorher erstellten Zwischenergebnisse benötigen.

In der DSL gibt es keine Konstrukte um Fehler abzufangen. Beim Abarbeiten eines Workflows werden Fehler einzelner Arbeitsschritte separat abgefangen und dem Benutzer mitgeteilt. Hierbei ist es für die Tools ebenfalls möglich *Exception*s zu werfen und der Benutzer sieht dann die entsprechende Nachricht dieser *Exception*. Dadurch ist die Fehlerbehandlung implizit Teil der DSL und muss vom Benutzer nicht selber umgesetzt werden.

Im Java-Beispiel werden einige Abfragen von Variablen auf *null* gemacht um zu verhindern, dass *NullPointerException*s auftreten. Dieses Vorgehen ist für Java typisch, es ist allerdings auch nicht direkt für den Workflow wichtig. Deshalb werden in der DSL solche Abfragen automatisch durchgeführt und müssen nicht explizit angegeben werden.

Der letzte Unterschied zwischen Java und der DSL ist der Code der sich direkt auf den Workflow bezieht. Dies sind zunächst die Deklarationen der Variablen, in denen für eine bessere Lesbarkeit im restlichen Code die Dateinamen gespeichert werden. Hierfür gibt es sowohl in Java als auch der DSL eine einfache Möglichkeit diese Variablen zu erstellen.

Die Durchführung der Schritte des Workflows geschieht in Java jeweils in mehreren Zeilen. Dabei wird zuerst das jeweilige Tool instantiiert, dann werden gegebenenfalls über Setter Parameter gesetzt und im letzten Schritt die eigentliche Ausführung des Tools angestoßen. All diese Schritte werden in der DSL in einem Befehl zusammengefasst. Dabei können ebenfalls Parameter für einzelne Tools gesetzt werden, es ist also die benötige Flexibilität gegeben um die gleichen Aufgaben durchzuführen. Außerdem ist in der DSL die Verwendung der Tools konsistenter als in Java, zum Beispiel werden hier beim Laden und Speichern nur Dateinamen angegeben, während in Java im Ladeschritt ein *InputStream* erwartet wird.

Insgesamt werden also für die DSL sinnvolle Annahmen getroffen und Vorgänge wie die Fehlerbehandlung automatisch vorgenommen, wodurch der Code wesentlich kompakter und dadurch kürzer wird, ohne dabei die Ausdruckskraft in Bezug auf die Aufgabe zu verlieren.

6.1.2 Aufwand für die Erstellung eines Workflows

Als nächstes soll der Aufwand verglichen werden, der nötig ist, um einen Workflow von Grund auf neu zu erstellen. Dazu werden zunächst die nötigen Schritte in Java und dann in der DSL beschrieben und anschließend verglichen. Als Ausgangssituation wird angenommen, wie es auch bei dieser Arbeit der Fall war, dass die Tools, die orchestriert werden sollen, bereits vorhanden sind, aber sich ihre Interfaces durchaus unterscheiden können. Es wird außerdem von der Verwendung von Eclipse ausgegangen.

6.1.2.1 Workflow in Java erstellen

Für einen neuen Workflow wird zuerst ein neues Projekt in Eclipse angelegt und alle Projekte in denen die Tools liegen, die verwendet werden sollen, werden als Abhängigkeiten hinzugefügt. Dann wird eine neue Java-Klasse mit einer main-Methode erstellt und darin der Workflow unter Verwendung der speziellen Interfaces der Tools programmiert. Der resultierende Code kann dann direkt ausgeführt werden.

6.1.2.2 Workflow in der DSL erstellen

Für die Verwendung der Tools in der DSL müssen diese zunächst an die vorgegebenen Interfaces angepasst werden. Dafür werden typischerweise Wrapper geschrieben, damit schon bestehende Programme die die Tools verwenden nicht geändert werden müssen. Es wird also für jedes Tool ein solcher Wrapper geschrieben, der jeweils die Ausführung des Tools kapselt und noch einige zusätzliche Informationen bietet, wie die akzeptierten Ein- und Ausgaben der Tools. Außerdem muss jeweils ein Wrapper für alle verwendeten Modelle geschrieben werden um diese in der DSL eindeutig identifizieren zu können und zusätzliche Informationen über das Modell zu geben.

Sind die Wrapper erstellt, so müssen ihre Klassen der DSL bekannt gemacht werden. Dafür kann in einem Plugin-Projekt der von der DSL angebotene ExtensionPoint verwendet werden. Hier werden dann für alle Tools ein Bezeichner und die zugehörige Klasse angegeben.

Nun kann eine neue Eclipse-Instanz gestartet werden, in der dann die Extensions mit dem ExtensionPoint der DSL verknüpft sind, wodurch die Tools der DSL bekannt sind. Wird nun eine neue Datei mit der Endung der DSL (.tools) erstellt, wird der entsprechende DSL-Editor geöffnet und man kann seinen Workflow beschreiben und wird dabei durch die Features der IDE unterstützt.

6.1.2.3 Vergleich

Es ist festzustellen, dass der initiale Aufwand zum Erstellen eines Workflows in Java geringer ist als in der DSL, da hier die Anpassung an die Interfaces und das Erstellen einer Extension wegfallen. Daraus resultiert allerdings auch eine Inkonsistenz bei der Verwendung der Tools, wodurch wie in Abschnitt 6.1.1 beschrieben der Java-Code länger und eventuell schlechter lesbar wird.

Eine Möglichkeit, den Aufwand für die DSL zu verringern ist, die Tools bereits bei der Entwicklung an die verlangten Interfaces anzupassen. Geht man also von der Neuerstellung eines Tools aus, so kann hier der Aufwand der Anbindung an die Sprache verringert werden. Es bleibt dann lediglich das schreiben der Wrapper für die Modelle und das Erstellen der Extensions.

6.1.3 Wartung existierender Workflows

Ist bereits ein Workflow vorhanden, für den die nötigen Tools schon an die Sprache angebunden sind und möchte man diesen bearbeiten so gibt es verschiedene Aufgaben, die durchgeführt werden können.

Möchte man ein Tool in einem Workflow gegen ein anderes austauschen, so kann man das in der DSL tun, indem man einfach den Bezeichner austauscht und eventuell, sofern das alternative Tool

noch nicht in der DSL bekannt ist, dieses an die Sprache anbindet. Ist der Befehl, der das Tool aktiviert, über die Angabe eines Modells realisiert, so kann es auch sein, dass der Workflow gar nicht geändert werden muss. Das ist der Fall, wenn ein Tool gegen ein anderes ausgetauscht wird und beide dasselbe Modell verwenden. Wird also von außen das ursprünglich verwendete Tool von der Sprache abgekoppelt und das neue angekoppelt, bleibt der Workflow ohne Änderungen gültig. Ansonsten beschränkt sich die Änderung auf wenige Worte, nämlich den Bezeichner und eventuell die Parameter, sofern das neue Tool andere Parameter erwartet.

Beim Austausch eines Tools in Java ist es meist nicht mit dem Austausch der verwendeten Klasse getan. Da hier keine konsistenten Interfaces für die Tools existieren, müssen unter Umständen mehrere Zeilen Code geändert werden und der Programmierer muss das Interface des neuen Tools kennen.

Ein anderer Fall beim Warten eines Workflows ist das Einfügen eines neuen Teilschrittes, zum Beispiel einer neuen Transformation mit anschließendem Speichern. Diese Befehle können in der DSL auf einfache und konsistente Weise hinzugefügt werden, wobei durch die IDE eine Unterstützung gegeben ist, da passende Tools dem Benutzer vorgeschlagen bzw. automatisch verwendet werden.

In Java müssen wiederum mehrere Zeilen Code eingefügt werden, wobei die Tools explizit angegeben und jeweils ihr individuelles Interface verwendet werden muss. Auch können hier zusätzliche Abfragen auf *null* oder anderer Code zur Fehlerbehandlung, wie ein try-catch-Block, notwendig sein, was nochmal Code hinzufügt.

Insgesamt lässt sich feststellen, dass die Wartung eines vorhandenen Workflows in der DSL einfacher ist, da zum einen weniger Code geändert werden muss als in Java und zum anderen durch die Unterstützung der DSL durch die IDE passende Tools für eine Aufgabe vorgeschlagen werden und deren Verwendung konsistent ist.

6.2 Problembezogenheit

In der DSL sind implizite Informationen enthalten, die in Java explizit angegeben werden müssen. So enthält eine Variable in der DSL sowohl eine Modellinstanz als auch das zugehörige Modell, dargestellt über das Interface *IModel*. In diesem sind wiederum weitere Informationen enthalten, konkret ist das die Datei in der das serialisierte Modell liegt. Im Java-Beispiel sieht man wie in der Variablen *protocolModelFilename* diese Information explizit aufgeführt werden muss.

Diese Informationen stehen den Tools in der DSL implizit zur Verfügung und können verwendet werden, um dem Benutzer nicht nur eine syntaktische, sondern auch eine semantische Unterstützung beim Erstellen eines Workflows zu bieten. Zum Beispiel werden inkompatible Arbeitsschritte erkannt und dem Benutzer gemeldet und die Bildung von Tool-Ketten wird auf Gültigkeit geprüft.

Hierin besteht der domänenspezifische Aspekt dieser DSL. Java konzentriert sich nicht auf ein einziges Problem, sondern bietet genug Flexibilität für eine große Menge von Problemen, wodurch allerdings ein größerer Aufwand für deren Lösung entstehen kann als es in einer dafür ausgelegten DSL der Fall ist.

6.3 Unterstützung durch die IDE

Ein wichtiger Punkt bei der Beurteilung der DSL ist die Unterstützung durch die Eclipse-IDE, da sie die Sprache wesentlich benutzbarer macht als in einem normalen Texteditor der Fall wäre. Zum Beispiel werden die Bezeichner für Tools und Modelle per Autovervollständigung vorgeschlagen, das heißt, der Benutzer muss sich die Namen nicht merken oder in den entsprechenden Plugins danach suchen. Außerdem werden beim Erstellen von Tool-Ketten passende Tools vorgeschlagen und Inkompatibilitäten zwischen Tools führen direkt zu Fehlermeldungen.

Außerdem ist auch die von Xtext aus der Grammatik abgeleitete Autovervollständigung von Schlüsselwörtern hilfreich, da hierdurch die Sprache auch verwendet werden kann ohne die Grammatik im Detail zu kennen.

Die Features der IDE machen es komfortabel, mit der DSL zu arbeiten, sie beschleunigen das Erstellen und Bearbeiten von Workflows und helfen, Fehler zu vermeiden.

6.4 Fazit

Zusammenfassend ist die Umsetzung eines Workflows in der DSL kürzer und kompakter, aber auch mit mehr Aufwand beim ersten Erstellen verbunden. Das Anpassen von vorhandenen Tools an die vorgegebenen Interfaces kostet zwar Zeit, doch ist die Verwendung der Tools hierdurch konsistent, was in Java nicht unbedingt der Fall ist.

Durch den Bezug auf das konkrete Problem kann in der DSL eine inhaltliche Unterstützung angeboten werden, die beim Erstellen des Workflows auf implizite Informationen zugreift, die in Java explizit angegeben werden müssen. Durch die IDE werden diese Informationen genutzt um dem Benutzer dabei zu helfen, schnell und einfach Workflows zu erstellen und dabei Fehler zu vermeiden.

7 Ausblick und Zusammenfassung

In diesem Kapitel wird zunächst ein Ausblick auf mögliche Erweiterungen dieses Projekts gegeben und anschließend wird die Arbeit zusammengefasst.

7.1 Ausblick

7.1.1 Erweiterung der Sprache

Um die Funktionalität der DSL weiter auszubauen kann sie um neue Konzepte erweitert werden.

Ein Beispiel hierfür ist das Verarbeiten mehrerer Modellinstanzen in einem Arbeitsschritt. Als Anwendungsbeispiel ist es möglich, dass man alle Dateien aus einem bestimmten Verzeichnis lesen und verarbeiten möchte. Dafür muss man in der aktuellen Version der DSL für jede Datei einen eigenen Ladebefehl ausführen und die Instanzen jeweils einzeln verarbeiten. Für dieses Problem ist nun eine Art foreach-Schleifenkonstrukt wie in Java denkbar, mit dessen Hilfe über alle Inhalte eines Verzeichnisses iteriert werden kann. Diese Idee wurde nicht umgesetzt, da sie nicht konkret benötigt wurde und solch ein Konzept aufwändig umzusetzen ist. Es wird hierfür eine Art Iterator benötigt, der in jedem Schleifendurchgang angepasst wird und eventuell muss ein Mechanismus für Scoping eingeführt werden, da in einer Schleife typischerweise ein eigener Scope verwendet wird. Auch die mögliche Verschachtelung von Schleifen muss bei der Umsetzung berücksichtigt werden und kann zu Problemen führen. Aus diesen Gründen wurde auf die Implementierung eines solchen Mechanismus verzichtet.

7.1.2 Grafische DSL

Aufbauend auf der entstandenen textuellen DSL ist es möglich eine grafische DSL zu erstellen. Dafür könnte zum Beispiel das Graphical Editing Framework verwendet werden, mit dessen Hilfe sich grafische Editoren für Eclipse realisieren lassen. Man hätte dann die Möglichkeit, Workflows mithilfe von Symbolen zu beschreiben, dadurch könnte der in der Arbeit verwendete Beispiel-Workflow wie folgt aussehen:

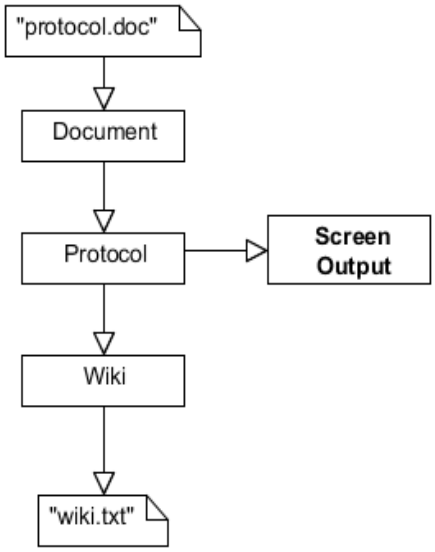


Abbildung 10: Workflow in einer grafischen DSL

Die Notation in Abbildung 10 ist sehr einfach gehalten und soll nur die Grundidee einer grafischen DSL verdeutlichen. Der Arbeitsfluss ist hier anhand der Pfeile leicht nachzuverfolgen und man sieht an welchen Stellen eine „Abzweigung“ entsteht, also das Ergebnis eines Schrittes als Eingabe für mehrere weitere Schritte verwendet wird. Im Beispiel wird das *Protocol* sowohl als *Screen Output* als auch für die Transformation in ein *Wiki* verwendet.

In dieser einfachen Darstellung werden keine eventuellen Parameter oder die Angabe eines konkret zu verwendeten Tools berücksichtigt. Solche in der DSL vorhandenen Möglichkeiten könnten auch in der grafischen DSL realisiert werden.

7.2 Zusammenfassung

In dieser Arbeit wurde eine domänenspezifische Sprache entwickelt, mit der es möglich ist, Workflows zu beschreiben, mit denen Tools zur Verarbeitung von Modellen orchestriert werden können. Für die DSL wurde mithilfe des Eclipse Modeling Frameworks und Xtext eine Eclipse-IDE realisiert in der die aus Eclipse bekannten Features wie Autovervollständigung und Fehleranalyse unterstützt werden.

Zunächst wurden die Grundlagen für die Arbeit beschrieben, um anschließend anhand eines realen Beispiels für einen Workflow in Java die Anforderungen an die DSL zu ermitteln. Darauf aufbauend wurden die Konzepte entwickelt, die in der Sprache umgesetzt wurden und einige interessante Aspekte der Implementierung wurden erläutert.

Abschließend wurde eine Einschätzung der Sprache und ihrer Vorteile vorgenommen und ein Ausblick auf mögliche Erweiterungen gegeben.

8 Anhang

8.1 Listing des Beispiel-Workflows in Java

```
import java.io.FileInputStream;

import olymp.wikiexporter.IWikiSyntaxFactory.WikiDialect;
import de.unihannover.se.eperformance.protocolextractor.ProtocolExtractor;
import de.unihannover.se.eperformance.protocolextractor.exporter.WikiExporter;
import de.unihannover.se.eperformance.protocolextractor.protocolmodel.Protocol;
import de.unihannover.se.km.armageddon.model.Document;
import de.unihannover.se.km.armageddon.reader.word.WordDocumentReader;

public class ExampleWorkflow {
    public static void main(String[] args) {
        try {
            String filename = "examples/Protokoll.doc";
            String communicationFile = "examples/Kommunikationsliste.organization";
            String protocolModelFilename = "examples/ProtocolModel.ecore";

            // read the Document from file using the WordDocumentReader
            WordDocumentReader reader = new WordDocumentReader();
            reader.setInputStream(new FileInputStream(filename));
            Document document = reader.getDocument();

            if (document != null) {
                // transform the Document to a Protocol
                ProtocolExtractor extractor = new ProtocolExtractor(document);
                Protocol protocol = extractor.getProtocol();

                if (protocol != null) {
                    // transform the Protocol to a wiki markup string
                    WikiExporter exporter = new WikiExporter();
                    exporter.setDialect(WikiDialect.CONFLUENCE);
                    exporter.setCommunicationFilename(communicationFile);
                    exporter.setProtocol(protocol);
                    String wikiMarkup = exporter.getWikiMarkup();

                    // visualize protocol as graph
                    ZeusActivator viewer = new ZeusActivator();
                    viewer.showModelInstance(protocol, protocolModelFilename,
                        ZeusActivator.DisplayType.Graph, "Protokoll");

                    // save the String to a file
                    StringSaver saver = new StringSaver();
                    saver.save(wikiMarkup, "examples/wiki.txt");
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

8.2 Grammatik in erweiterter Backus-Naur-Form

```
grammar de.unihannover.se.thesis.lkrumwiede.tools.Tools with
org.eclipse.xtext.common.Terminals

generate tools "http://www.unihannover.de/se/thesis/lkrumwiede/tools/Tools"

Start:
	(definitions+=Definition)* (stringDefs+=StringDef)* (actions+=Action)*;

Definition:
	'toolchain' id=ID ':' toolIDs+=ID ',' toolIDs+=ID (',' toolIDs+=ID)* ';';

StringDef:
	var=ID '=' value=STRING ';';

Action:
	(LoadAction | TransformAction | ShowAction | SaveAction) ('params'
params+=Parameter
	(params+=Parameter)*)? ';';

Parameter:
	paramString=STRING | paramID=ID;

LoadAction:
	'load' (file=InFile | fileVar=ID) (('as' modelID=ID) | ('using'
usedTool=ID)) 'into' var=ID;

TransformAction:
	'transform' var=ID (('to' modelID=ID) | ('using' usedTool=ID)) ('into'
destVar=ID)?;

SaveAction:
	'save' var=ID ('using' usedTool=ID)? 'to' (file=OutFile | fileVar=ID);

ShowAction:
	'show' var=ID ('using' usedTool=ID)?;

InFile:
	filename=STRING;

OutFile:
	filename=STRING;
```

8.3 Anleitung zur Erweiterung der DSL um Tools

Hier wird an einem einfachen Beispiel das Erweitern der DSL erläutert. Hierfür werden ein neues Modell und ein neues Tool zum Laden hinzugefügt.

Voraussetzung ist, dass in der verwendeten Eclipse-Version Xtext 2.0.0 installiert ist, dies ist die Version, die für diese Arbeit verwendet wurde. Außerdem müssen die beiden Plugins

- de.unihannover.se.thesis.lkrumwiede.tools_1.0.0
- de.unihannover.se.thesis.lkrumwiede.tools.ui_1.0.0

installiert sein (dafür reicht das Einfügen der Plugins in den Ordner „dropins“ im Eclipse-Verzeichnis aus).

Als erstes wird ein Plugin-Projekt benötigt. Das kann das Projekt sein, in dem sich das Tool befindet oder ein beliebiges anderes Plugin-Projekt. In diesem Projekt muss das Plugin *de.unihannover.se.thesis.lkrumwiede.tools_1.0.0* als Abhängigkeit hinzugefügt werden.

Als nächstes werden die Wrapper erstellt, die das Modell und das Tool zu den jeweiligen Schnittstellen kompatibel machen. Hierfür werden zwei neue Klassen erstellt, *MyModel* und *MyLoader*. Dabei implementiert *MyModel* das Interface *IModel*, das sich im Package *de.unihannover.se.thesis.lkrumwiede.tools.interpretation* befindet. Nach dem automatischen Hinzufügen der abstrakten Methoden werden diese ausgefüllt, so dass der Code wie folgt aussieht:

```
import de.unihannover.se.thesis.lkrumwiede.tools.interpretation.IModel;

public class MyModel implements IModel {

	@Override
	public Class<?> getModelClass() {
		return Document.class;
	}

	@Override
	public String getMetaModelFilename() {
		return "models/DocumentModel.ecore";
	}

}
```

getModelClass gibt die Klasse zurück, die das Modell darstellt, im Beispiel die Klasse *Document*. Die Methode *getMetaModelFilename* gibt die Ecore-Datei an, in der das Modell in serialisierter Form vorliegt, im Beispiel „models/DocumentModel.ecore“.

Als nächstes wird der Wrapper für das Ladetool *MyLoader* auf die gleiche Weise erstellt. Das Interface *ILoader* liegt dabei ebenfalls im Package wie *IModel*.

```
import de.unihannover.se.km.armageddon.model.Document;
import de.unihannover.se.km.armageddon.reader.word.WordDocumentReader;
```

```
import de.unihannover.se.thesis.lkrumwiede.tools.interpretation.ILoader;
import de.unihannover.se.thesis.lkrumwiede.tools.interpretation.IModel;

public class MyLoader implements ILoader {

	@Override
	public int getParamCount() {
		return 0;
	}

	@Override
	public int getOptionalParameterCount() {
		return 0;
	}

	@Override
	public Class<?> getLoadableClass() {
		return Document.class;
	}

	@Override
	public Object load(String filename, IModel model, String[] params)
			throws Exception {
		WordDocumentReader reader = new WordDocumentReader();
		reader.setInputStream(new FileInputStream(filename));
		Document document = reader.getDocument();
		return document;
	}

}
```

Die Methoden *getParameterCount* und *getOptionalParameterCount* geben die Anzahl benötigter und optionaler Parameter zurück, im Beispiel werden keine Parameter akzeptiert. *getLoadableClass* spezifiziert die Klasse von der die geladenen Objekte sind. Das Laden selber findet in der Methode *load* statt. Im Beispiel wird das existierende Tool *WordDocumentReader* verwendet, um das Laden durchzuführen. Zurückgegeben wird die Modellinstanz die das Tool als Ergebnis liefert.

Nachdem die Wrapper erstellt sind, müssen sie der DSL bekannt gemacht werden. Hierfür wird in dem Plugin-Projekt das die Wrapper enthält die Datei „MANIFEST.MF“ geöffnet. Im Tab „Extensions“ klickt man nun auf „Add“. Im sich öffnenden Dialog wählt man unter „Extension Points“ aus der Liste „de.unihannover.se.thesis.lkrumwiede.tools.models“ aus (wenn der Eintrag nicht zu sehen ist, kann der Haken bei „Show only extension points from the required plug-ins“ entfernt und die Abhängigkeit automatisch hinzugefügt werden). Es wird mit „Finish“ bestätigt und nach dem Schließen des Dialoges taucht im Fenster „All Extensions“ die neue Extension auf. Nach einem Rechtsklick auf die Extension wählt man „New“ und dann „model“. Anschließend muss man die Klasse die den Wrapper des Modells enthält unter „class“ angeben. Das ist in diesem Fall „MyModel“. Im Feld „id“ wird der Bezeichner eingetragen über den das Modell innerhalb der DSL identifiziert wird. Zum Beispiel kann man hier „Document“ wählen.

Nach der gleichen Methode wird nun die Extension „de.unihannover.se.thesis.lkrumwiede.tools.loaders“ hinzugefügt und hierin ein neuer „loader“ erstellt. Auch hier sind die Klasse, im Beispiel „MyLoader“, und ein Bezeichner für den Loader in der DSL anzugeben.

Sind die Wrapper in der Manifest-Datei eingetragen, kann per Rechtsklick auf das Manifest unter „Run As“ mit „Eclipse Application“ eine neue Instanz von Eclipse gestartet werden, in der dann der ExtensionPoint der DSL mit den neu erstellten Extensions verknüpft ist und dadurch in der DSL das Modell und das Tool bekannt sind.

Im neuen Eclipse wird nun ein neues, leeres Projekt (General Project) angelegt. In diesem Projekt wird eine neue Datei mit der Endung „.tools“ erstellt, zum Beispiel „test.tools“. Daraufhin erscheint ein Dialog der fragt, ob die Xtext Nature zum Projekt hinzugefügt werden soll. Nach dem Bestätigen steht dann für alle .tools-Dateien der Editor der DSL zur Verfügung.

Nun kann ein Workflow erstellt werden, der sich über das Menü „Run As“ per „Tools File Launcher“ starten lässt und dann interpretiert wird.

Literaturverzeichnis

[1] Kurt Schneider. Folien zur Vorlesung: Moderne Software-Entwicklungsmethoden, Wintersemester 2010/11

[2] Homepage von Wolfram Mathematica, Aufrufdatum: 01.08.2011: http://www.wolfram.com/mathematica/compare-mathematica/

[3] Ghosh, Debasish: *DSLs in Action*. Manning Publications Co., Stamford 2011

[4] Homepage der IMP, Aufrufdatum: 06.08.2011: http://www.eclipse.org/imp/

[5] Homepage von SoftENGINE, Aufrufdateum: 06.08.2011: http://www.softengine.de/glossar.php#Workflow

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 07.08.2011

_____

Lars Krumwiede