TECHNISCHE UNIVERSITÄT ILMENAU

Fakultät für Informatik und Automatisierung

Institut für Technische Informatik und Ingenieurinformatik

Fachgebiet Softwaresysteme/Prozessinformatik

Diplomarbeit

zur Erlangung des akademischen Grades Diplom-Informatiker

Softwarearchitektur-Entwurf und Realisierung eines Repositories für Modell-übergreifende Traceability

vorgelegt von:

Steffen Lehnert

Informatik 2005

Betreuer:

Priv.-Doz. Dr.-Ing. habil. Matthias Riebisch

Dipl.-Inf. Stephan Bode

Inventarisierungsnummer: 2010-12-03/087/IN05/2232

Eingereicht am: 19.11.2010

I. Kurzfassung

Der moderne Software-Entwicklungsprozess wird durch eine Vielzahl von Entwurfswerkzeugen und Methoden unterstützt, wobei eine Reihe von phasen- und zweckspezifischen Entwurfsmodellen entstehen. Abhängigkeiten und Entwurfsentscheidungen die mehr als ein Modell umspannen, können dabei explizit mittels Traceability-Links konkretisiert werden, um die Entscheidungen zu dokumentieren und ihre spätere Nachvollziehbarkeit gewährleisten zu können. Das Wissen über solche Beziehungen ist dabei für viele Teilbereiche des modernen Software-Engineerings von enormer Bedeutung, so etwa für Change Impact Analysis, Reverse Engineering und Reengineering.

In der vorliegenden Arbeit werden Ansätze und Methoden entwickelt, die eine einheitliche Verwaltung aller Entwurfsmodelle und eine automatisierte Erstellung und Speicherung von Traceability-Beziehungen mittels vordefinierter Regeln ermöglichen. Dabei wird besonderen Wert auf die modellübergreifende Analysefähigkeit gelegt, d.h. die Verknüpfung von Anforderungen, Architekturentwurf und weiteren Beschreibungsmitteln, wodurch sich diese Arbeit von bestehenden Ansätzen unterscheidet. Im weiteren Gegensatz zu existierenden Arbeiten die genau ein Paradigma zur Findung von Traceability-Beziehungen verfolgen, wird in dieser Arbeit eine Kombination aus Traceability-Regeln und Methoden des Information Retrieval benutzt, um Traceability-Beziehungen zwischen Modellen zu identifizieren.

Hierfür werden Metamodelle für Traceability-Regeln und Traceability-Beziehungen entwickelt, die in ein Eclipse-basiertes Repository eingebunden werden, welches Modelle aufnehmen und verwalten kann. Desweiteren werden die Metamodelle der Unified Modeling Language (UML), User Requirements Notation (URN) und Web Ontology Language (OWL) in das Repository eingebunden und ein Katalog von Traceability-Regeln, sowie Traceability-Typen definiert. Die zu entwickelnde Analysekomponente des Repository wird anschließend bewertet und die Ergebnisse mit bestehenden Ansätzen verglichen.

I. Abstract

The modern software development process is supported by a multitude of tools and technologies, which results in a huge number of documents and design models that accompany the final software. Dependencies and decisions which are based upon more than one design model can be expressed and documented as traceability links, to ensure their future comprehension and availability.
The knowledge of such relations and dependencies can be vital for further measures aimed on maintaining or improving the software system, like Change Impact Analysis, Reverse Engineering or Reengineering.

This thesis provides means to achieve a unified versioning of design models and the automated generation of traceability links among those models, using predefined rules. Special emphasis has been put on the model-spanning analysis-capability, i.e. the establishment of links between requirements, design- and other conceptual models. A combination of traceability rules and Information Retrieval is used for identifying traceability relations between the models, in contrast to related works where only one method is used for this purpose.

Therefore metamodels for traceability rules and traceability relations have been developed and integrated into an Eclipse-based model repository, which is capable of storing and managing models. In addition, the metamodels of the Unified Modeling Language (UML), User Requirements Notation (URN) and Web Ontology Language (OWL), as well as a catalogue of traceability rules have been integrated into the repository. The automated generation of traceability links has been evaluated and the results have been compared to other works.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Der Hauptgrund für die Schaffung eines CASE-Werkzeuge-übergreifenden Repository für Entwurfsmodelle liegt in der stetig wachsenden Zahl von Entwurfswerkzeugen, Modelltypen und Modellelementen. Durch die mangelnde Erweiterbarkeit vieler Werkzeuge ist es somit schwierig, wenn gar unmöglich, Verbindungen zwischen einzelnen Entwurfsmodellen automatisiert zu erkennen. Gerade die Zusammenhänge zwischen diesen Modellen entscheiden jedoch häufig über die Güte der endgültigen Implementierung bzw. über den Verständnisgrad beim Reengineering.
Fehlerhafte Zusammenhänge oder nicht erkannte Abhängigkeiten können sich in fataler Weise auf das Design der Software und deren Umsetzung in Quellcode auswirken bzw. führen sie zu falschen Schlussfolgerungen beim Verstehen einer Architektur. Das Erkennen von Zusammenhängen und Abhängigkeiten zwischen einzelnen Artefakten ist zudem die Grundlage für weitere, tiefgreifendere Analysemethoden und Prüfmöglichkeiten.

Mit Fortschreiten des Entwicklungsprozesses werden Änderungen an vielen Modellen vorgenommen, die direkte Auswirkungen auf abhängige Modelle haben. Werden diese Änderungen nicht auf die abhängigen Modelle übertragen, entstehen Inkonsistenzen durch fehlerhafte Modellsynchronisation. Das Benutzen eines gemeinsamen Repository kann hierbei Abhilfe schaffen und sorgt zudem dafür, dass alle an dem Softwaresystem beteiligten Personen zu jeder Zeit mit der gleichen Version der Entwurfsmodelle arbeiten. Somit wird auch das Risiko minimiert, dass veraltete Anforderungen umgesetzt bzw. Änderungen schlicht vergessen werden. Ein weiterer Faktor der die Schaffung eines solchen Modell-Repository stärkt ist die Tatsache, dass aktuelle Versionierungssysteme nur mangelnde Unterstützung für grafische Modelle bietet. So lassen sich mit rein textbasierten Merge-Algorithmen Änderungen in Graphenstrukturen schlecht erfassen und maschinell auflösen. Auch das manuelle Auflösen von Konflikten in Entwurfsmodellen gestaltet sich mit bisheriger Versionierungssystemen eher schwierig, weshalb auch hier ein speziell auf grafische Entwurfsmodelle zugeschnittenes Repository von Vorteil wäre.

1.2. Aufgabenstellung

Die zwei Hauptziele die mit dem Repository verfolgt werden sollen, sind zum einem die Möglichkeit zur Einbindung neuer Modelltypen in das Repository selber und zum anderen das automatisierte Auffinden von Gemeinsamkeiten und Zusammenhängen zwischen einzelnen Entwurfsmodellen.

Das Einbinden neuer Modelle soll dabei ohne weitere Änderungen an bestehenden Komponenten vorgenommen werden können. Gleichzeitig soll sichergestellt werden, dass neue Modelle ohne Einschränkungen importiert werden können, d.h. es dürfen keine Informationen beim Import verloren gehen. Zugleich soll ein Export aus dem Repository in der Form möglich sein, dass die exportierten Modelle wieder in ihren ursprünglichen Entwurfswerkzeugen verwendbar sind.
Werden die Modelle in dem entsprechenden Entwurfswerkzeug erneut modifiziert, sollen sie in ihrer aktualisierten Version wieder in das Repository zurück überführt werden können.
Neue Modelltypen sollen dabei durch das Einbinden neuer Plug-ins in das Repository verfügbar gemacht werden, die zuvor mit Hilfe des Eclipse Modeling Frameworks (siehe Kapitel 2.4.) als Eclipse Plug-ins erstellt wurden.

Das Auffinden und Verknüpfen zusammenhängender Modelle bzw. Modellelemente durch Traceability-Links soll im Repository anschließend voll automatisiert erfolgen. Jedoch sollen Links auch manuell angelegt werden können. Gründe hierfür sind u.a. die Tatsache, dass eventuell nicht alle Zusammenhänge automatisiert gefunden werden können und der Entwickler eingreifen muss. Andererseits kann ein Entwickler Zusammenhänge aber auch als „noch unklar“ deklarieren und sie speziell kennzeichnen. Dies ist besonders beim Reengineering von Vorteil, da bestehende Beziehung mit fortschreitendem Verständnis der Architektur verfeinert und ergänzt werden können.

Darüber hinaus soll das Repository auch viele der grundlegendsten Versionierungsfunktionalitäten bieten. Das bedeutet insbesondere die Versionierung von importierten Modellen, das Mergen von Versionen des gleichen Modells, sowie die ständige Verfügbarkeit der Modell-History um schnell einen Überblick über die verschiedenen Revisionen eines Modells zu erhalten. Auch wünschenswert ist das parallele Vorhandensein verschiedener Versionen des gleichen Modells in verschiedenen Projekten, um Alternativen in der Entwicklung auf ihre Eignung vergleichen zu können.

Als eine der letzten wichtigen Hauptanforderungen ist die Plattformunabhängigkeit des Repository zu nennen. Realisiert werden soll dies durch die Verwendung von Eclipse Plug-ins und Java als Programmiersprache, um eine Ausführung auf jedem Betriebssystem zu ermöglichen.

Das Resultat dieser Diplomarbeit wird zum einem aus dem realisierten Modell-Repository bestehen, in das Modelle importiert, exportiert, sowie intern versioniert werden können. Zusätzlich werden die Metamodelle für die komplette UML, OWL, GRL und UCM soweit vorbereitet, dass deren Instanzmodelle in das Repository aufgenommen werden können. Gleichzeitig werden auf XSL-Technologie basierende Konvertierungs-Templates für alle Elemente der UML, OWL, GRL und UCM geschaffen, um die mit den CASE-Werkzeugen erstellten Instanzmodelle in das repository-interne Format konvertieren zu können, um einen anschließenden Import zu ermöglichen.

1.3. Einordnung des Themas

Durch die Vielzahl an CASE-Werkzeugen und Entwurfsmethoden gibt es für jeden Teilschritt im Softwareentwicklungsprozess bestimmte Modelle, die die Ideen und Gedanken der jeweiligen Phase in Diagrammen umsetzen. Um die phasenübergreifende Nachvollziehbarkeit von Entwurfsentscheidungen gewährleisten zu können, müssen Abhängigkeiten und Zusammenhänge zwischen einzelnen Modellen durch Traceability-Links ausgedrückt werden. Auch für das Reengineering nehmen die Beziehungen und Abhängigkeiten zwischen Entwurfsmodellen eine wichtige Rolle ein, da es zunächst am wichtigsten ist die Struktur der Software zu verstehen, was meistens nur mit abstrahierenden Modellen und Traceability-Links möglich ist. Erst wenn die Struktur verstanden wurde, können Maßnahmen zur Erneuerung und zum Umbau ergriffen werden.

Allen Anwendungen gemeinsam ist die Tatsache, dass isoliert betrachtete Entwurfsmodelle wenig hilfreich sind um den Kontext des Softwaresystems oder dessen innere Zusammenhänge zu erfassen. Durch das zusammenhängende Betrachten von Modellen, die alle einen anderen Teilaspekt einer Software ausdrücken, ist es möglich einen umfassenden und vor allem vollständigen Blick auf die Architektur des Systems zu werfen, oder im umgekehrten Fall Unvollständigkeiten und Inkonsistenzen frühzeitig aufzudecken.

Auch die Tatsache dass mittlerweile immer mehr Personen aus verschiedenen Umfeldern an der Entwicklung von Softwaresystemen beteiligt sind, stärkt die Forderungen nach einer umfassenden Darstellung des Systems und all seinen Zusammenhängen. Allerdings kann man nicht voraussetzen, dass jeder Stakeholder über die Werkzeuge und das Wissen ihrer Bedienung verfügt, sodass ein zusammenfassendes Repository mit einheitlicher Bedienung, einheitlicher Sicht und vor allem der Fähigkeit zur automatisierten Verknüpfung auch hier von Vorteil wäre.

1.4. Abgrenzung zu vorherigen Diplomarbeiten

Dieser Diplomarbeit gehen zwei Arbeiten anderen Studenten voraus, die sich ebenfalls mit der Entwicklung eines Repository für Entwurfsmodelle befasst haben, allerdings unter anderen Gesichtspunkten und mit anderen Anforderungen.

Die Arbeit von Herrn Dirk Michael [MIC05] befasste sich mit der Frage, wie man Modelle mit Hilfe eines Repository auf Konsistenz prüfen kann, d.h. das Aufspüren von Modelleigenschaften, die gewisse Konsistenzbedingungen, die an das Modell gestellt werden verletzen. Als Resultat der Arbeit ist ein Eclipse Plug-in entstanden, welches über grundlegende Repository-Funktionalitäten verfügt, wie etwa das Anlegen neuer Projekte, der Import von Modellen und natürlich die Konsistenzprüfungen anhand vordefinierter Regeln.

Einen etwas anderen Fokus hatte die Arbeit von Herrn Daniel Rohe [ROH06], der sich mit der Konsistenzprüfung von Modellen auf Konformität mit dem verwendeten Komponentenmodell befasste. Als Ergebnis entstand auch hierbei wieder ein Plug-in für Eclipse, mit dem die Konsistenzprüfungen durchgeführt werden konnten.

Beiden Arbeiten gemeinsam ist die Tatsache, dass nur wenig Modelltypen und Modellelemente unterstützt wurden, wohingegen diese Arbeit bereits die komplette UML, OWL und URN unterstützt. Für folgende Studienarbeiten ist zudem eine Erweiterung um BPMN, Themenkarten, Faktorentabellen und weitere Modelltypen geplant.
Der Hauptunterschied liegt jedoch darin, dass sich diese Arbeit hauptsächlich auf das Aufspüren von Zusammenhängen und Abhängigkeiten zwischen Entwurfsmodellen konzentriert, ein Aspekt der in den anderen 2 Arbeiten kaum berührt wurde. Natürlich ist es auch denkbar das Repository, welches als Ergebnis dieser Arbeit entstehen soll, zu Konsistenzprüfungen zu verwenden. Vorrangig steht jedoch die automatisierte Verknüpfung im Vordergrund.

1.5. Skizze des Lösungsweges

Im folgenden Abschnitt sollen die wichtigsten Ideen, Konzepte und Arbeitsschritte vorgestellt werden, die im Laufe dieser Arbeit entstehen bzw. auf die dabei zurückgegriffen wird.

Aufgrund der Vielzahl an Repository-Projekten die im Laufe der letzten 5 Jahre entstanden sind bietet es sich an, auf ein bereits bestehendes Projekt zurückzugreifen um sich stärker auf den Hauptteil dieser Arbeit, die Auffindung von Abhängigkeiten konzentrieren zu können. Sobald ein entsprechendes Repository gefunden ist, muss dessen verwendetes Metamodell analysiert werden, um Informationen über die Art und Weise des Modellimports, sowie die eventuell notwendige Modellkonvertierung zu erlangen. Falls eine Konvertierung der Modelle notwendig sein sollte, muss im nächsten Schritt nach einem geeigneten Verfahren für die Konvertierung gesucht werden und dieses anschließend für die entsprechenden Metamodelle umgesetzt werden.

Der Hauptteil der Arbeit befasst sich anschließend mit der Frage, wie man mithilfe von vordefinierten Regeln Traceability-Beziehungen zwischen Modellen und Modellelementen erkennen kann.
Hierfür müssen die dabei auftretenden Traceability-Beziehungen zunächst klassifiziert und anschließend verfeinert werden. Desweiteren muss für die Einbindung und Versionierung von Traceability-Links ein repository-konformes Metamodell geschaffen werden.
Sind diese Vorbereitungsschritte abgeschlossen, müssen die Regeln definiert werden, mit denen anschließend nach Traceability-Beziehungen zwischen Modellen gesucht werden soll. Neben der eigentlichen Suche nach passenden Modellen, dienen die Regeln vor allem auch zur Klassifizierung der dabei entstehenden Links, denn ohne die Information warum zwei Elemente in Beziehung zueinander stehen, ist die Kenntnis über eine Abhängigkeit nur von geringem Nutzen.

Für die Definition und Abarbeitung der Regeln soll dabei eine eigene Regelsprache und der dazugehörige Interpreter entwickelt werden. Im Gegensatz zu vorhandenen Anfragesprachen für XML-Dokumente, soll die dabei entstehende Sprache wesentlich benutzerfreundlicher gestaltet werden und bereits von Anfang an so ausgelegt werden, dass damit nicht nur die Generierung von Traceability-Links ermöglicht werden soll. Aufgrund der konzeptuellen und algorithmischen Verwandtschaft zwischen Suchanfragen, Traceability-Suche und Konsistenzprüfungen, soll die Regelsprache auch zur Überprüfung von Konsistenzregeln und zur Abarbeitung von Suchanfragen genutzt werden können. Da sich alle drei Probleme auf die Suche nach den entsprechenden Modellen bzw. Modellelementen reduzieren lassen, muss für jeden der drei Anwendungsfälle lediglich eine andere Ergebnisverarbeitung (d.h. Ausgabe der Suchergebnisse, Linkerstellung,

Ausgabe der Warnhinweise) durch die Regelsprache und den dazugehörigen Interpreter abgedeckt werden. Abbildung 1 soll dabei das Konzept des Repository, der Regelkataloge und Benutzerinteraktion verdeutlichen.

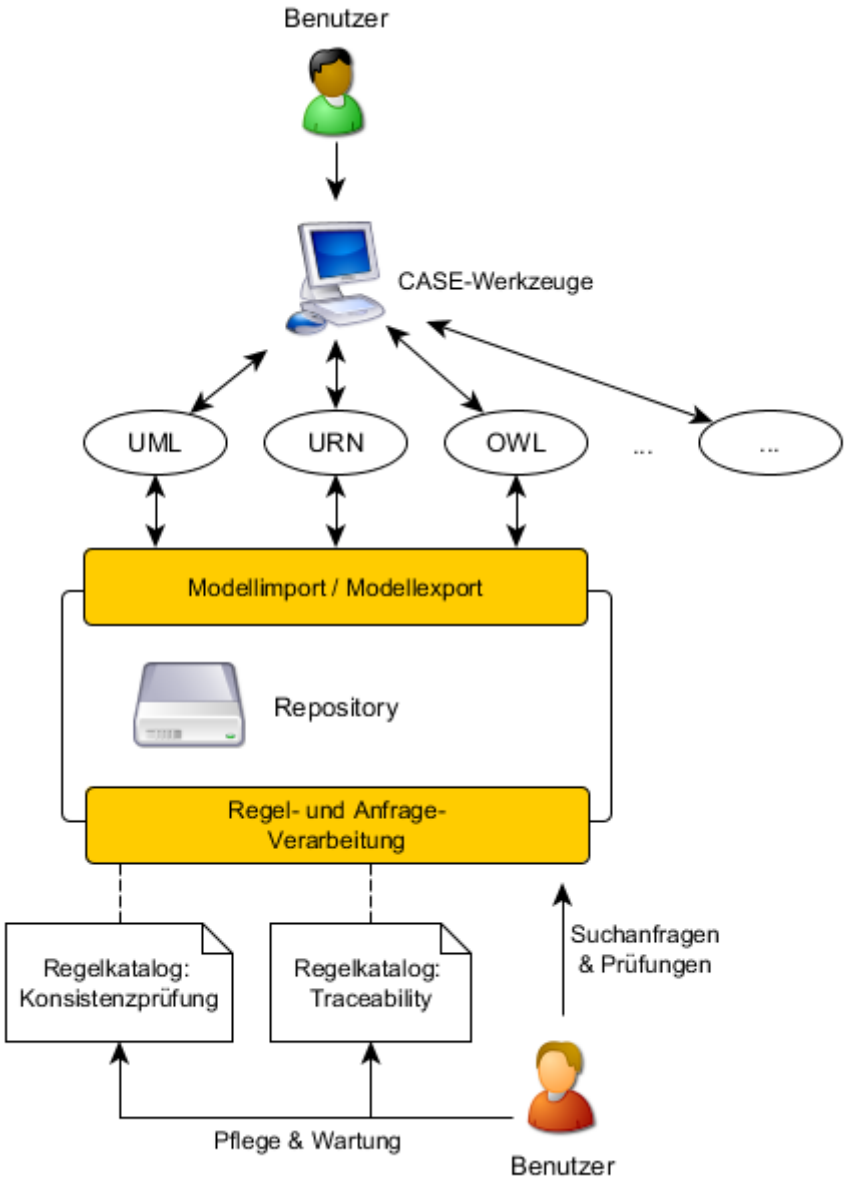


Abbildung 1: Das Konzept des Repository

Die offenen Fragen die dabei noch geklärt werden müssen, betreffen vor allem die Frage nach dem Zeitpunkt der Abhängigkeitsanalyse. Neben der sofortigen Ausführung beim Import bzw. Update eines Modells ist ein benutzergesteuerter Modus genauso vorstellbar, wie ein lastorientiertes Verfahren, welches bei geringer Arbeitslast im Repository die Traceability-Regeln anwendet und nach Abhängigkeiten sucht.

1.6. Aufbau der Arbeit

Diese Arbeit beginnt in ihrer Einleitung mit einer Einordnung des Themas in den Kontext der modellbasierten Softwareentwicklung, gefolgt von einer kurzen Abgrenzung der Ziele dieser Arbeit von vorhergehenden Diplomarbeiten. Nach kurzer Schilderung der Motivation und Aufgabenstellung erfolgt dann schließlich eine Grobskizzierung des in dieser Arbeit verfolgten Lösungsweges.

Im 2. Kapitel werden alle für das Verständnis dieser Arbeit erforderlichen Grundlagen kurz erläutert und auf weitere Informationen verwiesen, die nicht in den Umfang dieser Arbeit aufgenommen werden können. Darauf aufbauend werden im Kapitel „Stand der Technik“ aktuelle Technologien und Lösungsansätze hinsichtlich ihrer Eignung im Rahmen dieser Diplomarbeit beschrieben und bewertet. Dies schließt u.a. Verfahren zur Traceability-Generierung, Traceability-Metamodelle, Anfragesprachen und die Bewertung von bestehenden Modell-Repositories ein.

Im Hauptteil dieser Arbeit, dem Kapitel „Entwurf“, werden die Einzelschritte der Problemlösung detailliert dargestellt, d.h. es werden Lösungsansätzen ausgearbeitet, Architekturentwürfen und ausgewählte Algorithmen umfassend erläutert, die für die automatisierte Generierung von Traceability-Links in einem Modell-Repository erforderlich sind.

Im letzten Kapitel dieser Arbeit werden ausgewählte Teile und Vorgehensschritte der Implementierung vorgestellt und das Vorgehen bei Modellkonvertierung und Metamodell-Einbindung in das Repository erläutert. Zudem werden die Ergebnisse der Arbeit durch eine Evaluierung qualitativ bewertet und mit anderen Arbeiten verglichen.

Den Abschluss dieser Arbeit bildet eine kurze Zusammenfassung des in dieser Arbeit entwickelten Konzeptes zur automatisierten Abhängigkeitsanalysen zwischen Entwurfsmodellen und ein Ausblick auf mögliche Erweiterungen und offene Fragen.

2. Grundlagen

Im folgenden Kapitel sollen alle Grundlagen, Technologien und Methoden kurz erläutert werden, die das Grundgerüst dieser Diplomarbeit bilden.

2.1. Unified Modeling Language

Die Unified Modeling Language [UML], kurz UML, liegt momentan in der Version 2.3 vor und wird durch die ISO/IEC 19501 genormt. Sie ist die momentan am weitesten verbreitete Modellierungssprache, die zudem viele Aspekte des Softwareentwicklungsprozesses durch ihre Diagramme unterstützt. Es existieren zahlreiche CASE-Werkzeuge, die die Elemente der UML in verschiedenem Umfang abdecken. Eines der umfangreichsten und bekanntesten Open Source Projekt sind die für die Eclipse Entwicklungsumgebung geschaffenen UML2 Tools [UML2TOOLS], die als eine Sammlung von Eclipse Plug-ins nahtlos in die Entwicklungsumgebung integriert werden können. Für den Bereich der modellgetriebenen Softwareentwicklung stellt die UML die Basis für automatische Codegeneration aus Entwurfsmodellen dar. Anhand der Modelle lassen sich in kürzester Zeit die benötigten Klassenrümpfe, Interfaces, Methoden und Attribute generieren und später erweitern.

Der Abstraktionsgrad der von der UML bereitgestellten Modelle und Modellelemente variiert dabei von „Sehr abstrakt“ (z.B. Verteilungsdiagramm) bis „Sehr implementierungsnah“ (z.B. detailliertes Klassendiagramm) und ist somit geeignet, viele Aspekte einer Softwarearchitektur bzw. einer Implementierung darzustellen. Aufgrund der Vielzahl an UML-Diagrammen die ein Softwareprojekt typischerweise begleiten, erlangt das Erkennen und Aufspüren von Abhängigkeiten und Zusammenhängen zwischen einzelnen Modellen immer mehr an Bedeutung. Anhand dieser gefundenen Abhängigkeiten lassen sich Fehler frühzeitiger erkennen und umgehen, bzw. wird die Behebung des Fehlers vereinfacht. Auch die Frage nach einer effizienten Synchronisation der Entwurfsmodelle steht im direkten Zusammenhang mit der Vielfalt der angebotenen Modelle und Modellverknüpfungen.

2.2. Web Ontology Language

Bei der Web Ontology Language (OWL) handelt es sich um eine Spezifikation des World Wide Web Consortium (WC3) zur Darstellung, Manipulation und maschinellen Verarbeitung von Ontologien im Sinne des Semantic Web. Aktuell ist die Web Ontology Language [OWL] in der Version OWL 2

verfügbar, die im Oktober 2009 durch die WC3 veröffentlich wurde. Im Zuge dieser Arbeit wird die OWL 2 verwendet, was insbesondere durch die Nutzung des Werkzeuges Protégé [PROTEGE] begründet wird, welches u.a. einen Export der generierten Ontologien in einem OWL/XML-Format anbietet.

Da Ontologien hervorragend für die Repräsentation von Begriffen und denen zwischen ihnen herrschenden Beziehungen geeignet sind, bietet sich eine Verknüpfung mit anderen Entwurfsmodellen, beispielsweise aus der UML, an um dort vorhandene Beziehungen und Abhängigkeiten besser erkennen zu können.

2.3. User Requirements Notation

Die User Requirements Notation [URN], kurz URN, wurde von der ITU-T mit dem Standard Z.151 normiert und dient zur Darstellung und Modellierung von Anforderungen die an ein Softwaresystem gestellt werden. Dabei gliedert sich die URN in zwei Teilnotation: die Goal-oriented Requirements Language (GRL) für die Darstellung nichtfunktionaler Anforderungen und die Use Case Maps (UCM) zur Darstellung funktionaler Anforderungen.

Für die Darstellung und Modellierung von URN-Modellen kommt in dieser Arbeit das jUCMNav-Projekt [JUCMNAV, AKR06] zum Einsatz, welches als eigenständiges Eclipse Plug-in realisiert wurde und sowohl die GRL, als auch die UCM vollständig abdeckt, jedoch in gewissen Bereichen vom Standard Z.151 abweicht. jUCMNav lässt dabei auch einen Export [GAO10] und Import der Modelle nach genormten Z.151 Standard im XML-Format zu.

2.3.1. Goal-oriented Requirements Language

Die Teilnotation der URN für die Darstellung nichtfunktionaler Anforderungen bietet mit ihren Modellelementen Goal und Softgoal eine Abstufung von Anforderungen an, die durch Tasks (Wege zur Zielerfüllung) und Ressourcenverfügbarkeit als Zieleigenschaft ergänzt werden können. Elemente der GRL lassen sich wiederum durch die 4 Beziehungstypen Korrelation, Mitwirkung, Dekomposition und Zielerfüllung miteinander verbinden. Zusätzlich können weitere Eigenschaften unterstützend an eine Beziehung vergeben werden, die Auskunft über die Eignung eines Lösungsweges geben können.

2.3.2. Use Case Maps

Mithilfe der Use Case Maps lassen sich die funktionalen Anforderungen eines Softwaresystems in grafischen Modellen erfassen. Die Modelle haben dabei eine gewisse Ähnlichkeit zu Sequenzdiagrammen und Zustandsautomaten der UML und bieten Modellelemente wie Start-, End- und Warteplätze, sowie Verzweigungen und Pfade an. Aufgrund dieser Tatsache lassen sich mit einem UCM-Diagramm auch Kontrollflüsse und Aufgabenverteilungen realisieren und darstellen.

2.4. Eclipse Modeling Framework (EMF)

Das Eclipse Modeling Framework [EMF, EGM05] ist ein Open Source Projekt für modellbasierte Softwareentwicklung und Java-Codegenerierung. Das Framework ist in der Lage automatisch den passenden Java-Code zu Modellen zu generieren, sodass anschließend Instanzen dieses Modells in Java-Projekten verwendeten werden können. Als Grundlage für die Erstellung von Modellen dient dabei das Ecore-Metamodell, welches alle benötigten Elemente wie etwa EClass oder EAttribute enthält. Neben der Erstellung von Modellen können mithilfe des EMF Frameworks auch komplette Metamodelle generiert werden, deren Instanzmodelle anschließend wieder in Projekten verwendet werden können. Dieses Vorgehen wird im Laufe dieser Diplomarbeit mehrmals angewandt, um die Metamodelle verschiedener Modellierungsstandards wie etwa UML oder URN auf ein Ecore-basiertes Metamodell abbilden zu können.
Sobald das Ecore-Modell erstellt wurde, wird daraus automatisch ein Generatormodell (genmodel) erzeugt, mit dessen Hilfe sich der tatsächliche Java-Code generieren lässt. Neben dem eigentlichen Modell-Code lässt sich ein zusätzlicher Edit-Code erzeugen, der speziell für den Zugriff auf die Modellinstanzen und deren Attribute ausgelegt ist. Sobald dieser Code mit Hilfe des Generatormodells erzeugt wurde, kann zusätzlicher Code für Eclipse-basierte Editoren erzeugt werden. Dieser Editor-Code erlaubt die einfache und schnelle Erstellung von Editor-Plug-ins für die Eclipse Plattform, mit deren Hilfe sich die erzeugten Ecore-Modelle instanziieren und bearbeiten lassen. Zusätzlich kann aus dem Generatormodell Code für Testfälle und Testklassen generiert werden, die einen anschließenden Test des erstellten Modells erlauben. Neben der manuellen Erzeugung eines Ecore-Modells besteht auch die Möglichkeit, Modelle aus XSD-Schemadefinitionen oder bestehenden Rose Class Modellen und ähnlichen Vorlagen zu generieren. Dazu wird beim Import automatisch eine Validierung der verwendeten Vorlage vorgenommen, um eventuelle Fehler und Inkonsistenzen von Beginn an zu vermeiden.

Aufgrund der Vielseitigkeit von EMF und der guten Werkzeugunterstützung, ist EMF die Grundlage für viele Projekte aus den Bereichen modellbasierte Softwareentwicklung und modellbasierte Architekturentwicklung und bildet daher auch die Basis für diese Diplomarbeit.

2.5. Modellkonvertierung

Die folgenden 3 Abschnitte sollen einen Überblick über die Möglichkeiten zur Konvertierung von Entwurfsmodellen [CH03] bieten. Jedes der dabei vorgestellten Verfahren wird jeweils kurz beschrieben und aufgrund seiner Vor- und Nachteile auf seine Eignung im Rahmen dieser Diplomarbeit untersucht.

2.5.1. Atlas Transformation Language (ATL)

Die ATL [ATL] ist Teil einer auf Eclipse aufbauenden Entwicklungsumgebung für das Konvertieren von Entwurfsmodellen anhand von fest definierten Regeln. Konvertierungsregeln werden dabei in einer ATL-eigenen Sprache verfasst und lehnen sich syntaktisch an den Konstrukten bekannter Programmiersprachen wie etwa C++ oder Java an. Das Projekt wird stetig weiterentwickelt und liegt aktuell in Version 3.1.0 vor. Neben den eigentlichen Sprachkonstrukten und der Konvertierung von Modellen bietet ATL noch eine Reihe weiterer Funktionen, wie etwa Syntax Highlighting und einen eigenen Debugger, um die Arbeit mit ATL zu erleichtern.
Aufgrund der Tatsache dass man zum Arbeiten mit ATL zuallererst die Transformationssprache erlernen muss und Transformationen dann auch nur über Eclipse und die ATL IDE möglich sind, habe ich mich gegen die Nutzung von ATL entschieden. Der Hauptgrund hierfür ist die Tatsache, dass die Konvertierung direkt beim Import von Modellen in das Repository möglich sein soll, was jedoch durch die Tatsache unmöglich gemacht wird, dass Transformationen nur innerhalb der ATL IDE durchführbar sind.

2.5.2. Tiger EMF / EMF Henshin

Mit dem ehemaligen Tiger EMF Projekt [TIGEREMF] welches im Januar 2010 in das Eclipse Henshin Projekt [HENSHIN] integriert wurde, werden Modelle durch Graphen-Transformationen in andere Modelle überführt.
Für die Umwandlung eines Modells werden insgesamt 4 verschiedene Regeltypen [BEEKKT07] benötigt, die bei der Konvertierung ausgewertet und angewandt werden. Die sogenannten left-hand-side-Rules (LHS) bestimmen dabei die Struktur des Eingabemodells, während die right-hand-side-

Rules (RHS) die Struktur des Zielmodells definieren. Zur Konvertierung werden zusätzliche negative-application-conditions (NAC) benötigt, sowie die eigentlichen Abbildungsregeln von Eingangsstruktur zur Ausgangsstruktur. Aufgrund der Komplexität der zu verwendenden Metamodelle (z.B. UML: 199 verschiedene Elemente) ist eine Konvertierung durch die Definition von Graphen für Vor- und Nachbedingungen als eher schwierig und unübersichtlich einzustufen. Auch bei anderen Metamodellen wie das der URN mit „lediglich“ 79 Elementen geraten die dabei entstehenden Graphen zu unübersichtlich und groß, sodass das Auffinden von Fehlern sehr erschwert wird.

2.5.3. Extensible Stylesheet Language Transformation (XSLT)

Ein häufig genutztes Verfahren zur Konvertierung von XML-basierenden Dateien in andere XML-Formate stellt die Extensible Stylesheet Language (XSLT) dar [XSLT]. Da die meisten CASE-Werkzeuge Modelle in einem XMI-Format exportieren bzw. abspeichern, bietet es sich an diese Modelle mit Hilfe der XSLT in andere Formate zu überführen. Zur Konvertierung wird die logische Baumstruktur der XMI-Dateien ausgenutzt, die mithilfe von Templates und Umwandlungsregeln in das Zielformat überführt werden. Die eigentliche Konvertierung erfolgt mit Hilfe eines XSLT-Prozessors. Der Vorteil dieses Verfahrens besteht in der Verfügbarkeit vieler XSLT-Prozessoren, die u.a. auch in modernen Webbrowser wie Firefox und Opera integriert sind. Allein in der Standardversion der Eclipse Entwicklungsumgebung sind 2 verschiedene XSLT-Prozessoren integriert, darunter auch der XALAN Prozessor. Aufgrund der weiten Verbreitung dieses Verfahrens, der Verfügbarkeit vieler Prozessoren und der Tatsache, dass nahezu jedes CASE-Werkzeug einen Modellexport in XML-Format anbietet, erachte ich die XSLT-Transformation als das geeignetste Verfahren für die Konvertierung von Entwurfsmodellen für ein gemeinsames Repository.

3. Aktueller Stand der Technik

Aufbauend auf den im vorherigen Kapitel vorgestellten Grundlagen sollen nun aktuelle Projekte, Methoden und Forschungsergebnisse vorgestellt werden, die sich mit der Entwicklung eines Modell-Repository oder der Abhängigkeitsanalyse und Synchronisation von Entwurfsmodellen befassen. Insbesondere sollen dabei bestehende Repositories für Entwurfsmodelle untersucht und aktuelle Arbeiten zu den Themen Traceability-Modellierung, Traceability-Extraktion und Anfragesprachen für Traceability-Regeln bewertet werden.

3.1. Modell-Repositories

Unter Ausnutzung des Eclipse Modeling Frameworks entstanden im Laufe der letzten 3 Jahre mehrere Repository Projekte mit dem Ziel, ein einheitliches Repository für Entwurfsmodelle zu schaffen. Im Laufe der Recherchetätigkeiten zu dieser Arbeit musste ich jedoch feststellen, dass viele dieser Projekte nicht über ein frühes Anfangsstadium hinaus gekommen sind und häufig nach kurzer Zeit wieder eingestellt wurden, so z.B. AMOR[1]. Lediglich 3 der gefundenen Repositories werden aktuell noch gepflegt und weiterentwickelt und sollen daher im folgenden Abschnitt analysiert und auf ihre Eignung als Ausgangsbasis für das geplante Repository untersucht werden.

3.1.1. CDO Model Repository

Das frei verfügbare CDO Repository [CDO] wurde bis 2009 weiterentwickelt und wird aktuell von IBM betreut und erweitert. Der Hauptfokus dieses Repository liegt auf verschiedenen Versionierungsfunktionen und einem möglichst effizienten Laufzeitverhalten.

Das CDO Repository bietet dabei folgende Features:

- Anlegen und Speichern neuer Modelle
- Löschen von Modellen
- Versionierung von Modellen
- Mehrbenutzerbetrieb
- austauschbare Persistenzschicht

[1] http://www.model-repository.de/

Features die nicht durch das Repository abgedeckt werden:

- momentan kein Offline-Modus verfügbar
- kein Mergen von Modellrevisionen
- Modellsuche

Der Hauptnachteil dieses Repository ist jedoch die mangelnde bzw. praktisch nicht mehr vorhandene, öffentlich zugängliche Dokumentation, was u.a. durch den Einfluss von IBM auf dieses Projekt zu erklären ist. Sämtliche noch vorhandenen Dokumentationsseiten sind entweder komplett leer oder mit Default-Werten gefüllte Artikel-Stubs. Lediglich zur Laufzeit- und Speicherplatzoptimierung erscheinen kontinuierlich neue Publikationen, die jedoch wenig zum Verständnis der Repository-Architektur und dessen Schnittstellen beitragen. Aus diesem Grund habe ich mich gegen die Nutzung des CDO Repository entschieden, da eine Erweiterung des Repository ohne tiefere Kenntnis seines internen Aufbaus und der angebotenen Schnittstellen nur schwer umzusetzen ist.

3.1.2. Eclipse Model Repository

Das von der Universität Leipzig entwickelte Projekt [EMR] befindet sich momentan noch in einer recht frühen Entwicklungsphase, bietet aber schon umfangreiche Möglichkeiten für das Arbeiten mit Modellen im Ecore und XMI-Format:

- Anlegen und Speichern neuer Modelle
- Löschen von Modellen
- Versionierung von Modellen
- Mergen von Modellen
- Modellsuche
- Arbeiten im Offline-Modus möglich

Aktuelle Nachteile und fehlende Features:

- keine Architektur- und Schnittstellendokumentation
- derzeit nur Revision 7 auf dem SVN verfügbar
- keine Versionierung von Modellelementen

Der Hauptgrund der gegen das Eclipse Model Repository spricht ist die Tatsache, dass die Entwicklung bereits in einer frühen Phase eingestellt wurden schien, da seit Anfang April kein Update der Projektseite oder des SVN stattgefunden hat und lediglich eine frühe Version verfügbar ist.

Auch ist nicht ersichtlich, wann das Projekt fortgeführt werden soll und welche zusätzlichen Anforderungen zukünftig umgesetzt werden sollen.

3.1.3. EMFStore

Das EMFStore Repository [EMFSTORE] erfüllt in seiner aktuellen Version bereits einen Großteil der gestellten Anforderungen und kann vor allem in den Bereichen Dokumentation und Weiterentwicklung überzeugen. Das einzige wünschenswerte Feature welches momentan noch nicht integriert ist, ist die Suche nach Modellen bzw. einzelnen Modellelementen. Die restlichen Anforderungen werden jedoch bereits wie folgt abgedeckt:

- Anlegen und Speichern neuer Modelle
- Löschen von Modellen
- Versionierung von Modellen
- Mergen von Modellen
- Arbeiten im Offline-Modus ist möglich

Darüber hinaus bietet EMFStore eine Reihe weiterer Vorteile, die es zu einem geeigneten Kandidaten für die Wahl des zu verwendenden Repository machen:

- umfangreiche Dokumentation
- Video Tutorials zu den Themen Installation, Einrichtung, Modellintegration und Entwicklung eigener Clients
- EMFStore wird seit mehreren Jahren von verschiedenen Firmen und Institution genutzt und stetig weiterentwickelt
- gute Kontaktmöglichkeiten mit den Entwicklern des Repository

Dazu bietet EMFStore von Haus aus die Möglichkeit, auf eigene Bedürfnisse abgestimmte Clients zu entwickeln und zusammen mit EMFStore zu nutzen. Durch die Schaffung von Komponenten für die automatisierte Abhängigkeitsanalyse und Modellvergleiche bietet dies somit die notwenige Grundlage, um weitere Funktionalität in das Repository zu integrieren.
Ein weiterer Aspekt der für die Nutzung von EMFStore spricht, ist die Art und Weise wie neue Modelltypen in das Repository integriert werden können. Da neue Modelle lediglich durch neue Plug-ins in das Repository eingebunden werden müssen, können jederzeit neue Elemente aufgenommen werden, ohne das bestehende Funktionalität verändert oder ergänzt werden muss. Dabei müssen neue Metamodelle lediglich das von EMFStore verwendete Unicase-Metamodell erweitern, womit

EMFStore auch der Anforderung nach einem möglichst flexiblen und leicht anpassbaren Repository gerecht werden kann.

Unter Nutzung von EMFStore wird von dessen Entwicklern an einem weiteren Projekt, dem UNICASE Client, gearbeitet. Nach einer genaueren Analyse dieses Projektes stand jedoch fest, dass UNICASE nicht für das Erreichen der Ziele dieser Diplomarbeit geeignet ist. Der Hauptgrund hierfür ist der Tatsache geschuldet, dass mit UNICASE die Schaffung eines CASE-Werkzeuges verfolgt wird, welches Auszüge verschiedener Modellierungssprachen in sich vereinigt, ohne den Anspruch zu erheben die Sprachspezifikationen vollständig umzusetzen. Zudem würde die Nutzung von UNICASE im Gegensatz zu der Anforderung stehen, dass Modelle verschiedenster CASE-Werkzeuge im Repository aufgenommen werden können und jederzeit wieder im ursprünglichen Werkzeug bearbeitbar bleiben sollen.

3.2. Traceability-Metamodelle

Aufgrund der Anforderungen des Repository muss nach einem allgemeinen Konzept für ein Traceability-Metamodell gesucht werden, mit dem sich Links zwischen den verschiedensten Modellelemente und Modelltypen realisieren lassen.
In [MPR07] und [LG05] wurde analysiert, aus welchen Komponenten ein Traceability-Link bestehen sollte um diesen Anforderungen gerecht zu werden und in [DKPF09] wurde ein Metamodell vorgeschlagen, mit dem Traceability-Beziehungen dargestellt werden können.

Daraus resultieren folgende Eigenschaften, über die ein Traceability-Link verfügen sollte:

- Verweis auf den Quellknoten
- Verweis auf den Zielknoten
- Beschreibung des Linktyps
- Informelle Beschreibung des Links
- Autor des Links
- Relevanz des Links

Für ein Traceability-Metamodell ergibt sich somit die in Abbildung 2 gegebene, vereinfachte Darstellung.

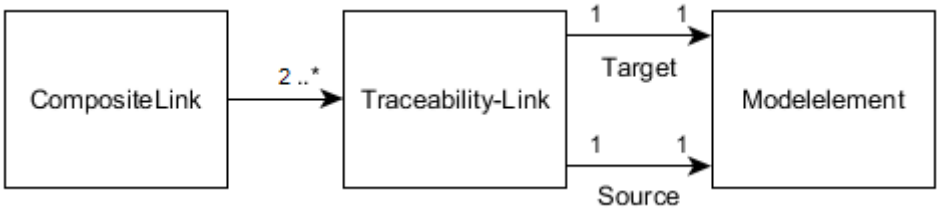


Abbildung 2: Vereinfachtes Traceability-Metamodell

Einzelne Traceability-Links können dabei zu transitiven bzw. zusammengesetzten Traceability-Links zusammengefasst werden, sofern die einzelnen Links über gemeinsame Start- oder Endpunkte verfügen. Durch diese transitiven Verknüpfungen kann man somit Abhängigkeiten verdeutlichen, die man mit einzelnen Traceability-Links nicht ausdrücken kann.

Folgendes Beispiel soll diesen Sachverhalt verdeutlichen:

- Seien A, B und C Modelle bzw. Modellelemente
- A und B sind durch einen Traceability-Link miteinander verbunden: A → B
- B und C sind durch einen Traceability-Link miteinander verbunden: B → C

Daraus folgt, dass auch Modell A und Modell C durch eine transitive Relation verbunden werden, da auch zwischen diesen Modellen eine Abhängigkeitsbeziehung besteht, d.h.:

A → C

Mit diesem Mittel kann man beispielsweise zusammengehörige Anforderungen, Entwurfsmodelle und Testfälle geeignet verbinden und jederzeit nachvollziehen, durch welchen Anwendungsfall eine Anforderung beschrieben wird und welche Klasse bzw. Komponente den Anwendungsfall realisiert.

3.3. Automatisierte Methoden zur Traceability-Generierung

Für die maschinelle Suche und Erzeugung von Traceability-Links gibt es eine Reihe von Konzepten und Verfahren die in [RKA07] und [SZ05] analysiert und bewertet wurden. Im folgenden Abschnitt sollen die 4 wichtigsten Konzepte kurz analysiert werden, um ein geeignetes Verfahren bzw. eine Kombination der geeignetsten Methoden für Traceability-Analysen im Repository zu finden.

3.3.1. Erzeugung von Traceability-Links durch Methoden des Information Retrieval

Im Allgemeinen werden bei Retrieval-Verfahren Ähnlichkeits- und Distanzmaße genutzt, um Übereinstimmungen zwischen den Attributen einzelner Modelle oder Modellelemente zu finden. Bei diesem Vorgehen besteht der 1. Schritt in der Verarbeitungskette meist aus einer Analyse der Modelle und der anschließenden Extraktion aller relevanten Begriffe, die dabei häufig in Zwischenstrukturen wie Wortmatrizen oder Vektorräume eingeordnet werden.
Sobald sämtliche vorkommenden Wörter entsprechend einsortiert wurden, können anschließend mittels verschiedener Vergleichsalgorithmen zusammenhängende Wortpaare gefunden werden und die entsprechenden Modelle können dann durch Traceability-Links miteinander verbunden werden.

Mögliche Schritte der Modellanalyse umfassen u.a.:

- Entfernung von Stoppwörtern (z.B. Artikel)
- Expandieren von Abkürzungen (z.B. „desc." → „description")
- Anreicherung mit Synonymen (z.B. „note" → „hint", „annotation", „memo")
- Wortstammbildung
- Indexierung von Wörtern

Mögliche Vergleichs- bzw. Suchverfahren sind dabei:

- direkter Wortvergleich auf vollkommene Übereinstimmung
- Suche nach übereinstimmenden Wortteilen
- Suche nach prozentualer Übereinstimmung (z.B. N-Gram-Algorithmus)
- vektorbasierte Distanzvergleiche

Der Hauptvorteil dieser Methoden ist der hohe Recall-Faktor, da alle Modelle und Modellelemente unabhängig von ihrer semantischen Bedeutung gleichermaßen in die verschiedenen Analyseschritte einbezogen werden. Allerdings werden dadurch im Vergleich zu anderen Verfahren öfters vermeintliche Zusammenhänge erkannt, da es entsprechende Übereinstimmungen in Bezeichnern gegeben hat. Der Hauptnachteil dieses Verfahrens liegt jedoch in der geringen Aussagekraft der generierten Traceability-Links, da es mit Information Retrieval Methoden schwer fällt, erzeugte Links in Kategorien wie „Defines", „Consists-Of" oder „Uses" einzuordnen und dieser Schritt meist vom Anwender erledigt werden muss. Die Beschreibung eines solchen Verfahrens findet sich beispielsweise in [IK06].

3.3.2. Regelbasierte Erzeugung von Traceability-Links

Präzisere Ergebnisse erhält man dagegen durch die Verwendung fest definierter Regeln, mit deren Hilfe Traceability-Links erzeugt werden können. Jede Regel besteht dabei aus 3 Teilen, die folgende Aufgaben erfüllen:

1. Kopfteil: Definition der betreffenden Elemente und informale Beschreibung der Regel
2. Anfrageteil: Formulierung der Anfrage durch Suchausdrücke
3. Aktionsteil: Formulierung der Ergebnisverarbeitung

Im Anfrageteil kommen dabei Konstrukte zum Einsatz, die den Anfragesprachen für Datenbanken, wie etwa SQL, sehr ähnlich sind. Mit Hilfe der eingebetteten Suchausdrücke wird dabei navigierend auf die Entwurfsmodelle und deren Elemente und Attribute zugegriffen.
Für die Verwendung der Regeln muss jedoch ein zusätzlicher Interpreter geschaffen werden, der die vorzugsweise in einem Regelkatalog definierten Regeln einlesen und verarbeiten kann.
Durch die exakte Spezifikation von Quell- und Zielelement im Kopfteil der Regel ist es zudem möglich, den genauen Typ des erzeugten Traceability-Links anzugeben. Der Benutzer hat somit nicht nur die Information das zwei Modelle oder Modellelemente miteinander in Beziehung stehen, sondern auch die wesentlich interessanteren Angaben „warum“ und „wie“. Somit ist besonders der Precision-Faktor im Vergleich zum Information Retrieval-Ansatz als besser zu bewerten [SZ05]. Ein derartiger Ansatz wird beispielsweise in [JZ09], [SGZ03] und [SZMK04] verfolgt.

3.3.3. Event-basierte Wartung von Traceability-Links

Zur Aufrechterhaltung der Konsistenz und Gültigkeit von Traceability-Links während der Entwicklung eines Softwaresystems werden bestimmte Ereignisse definiert, die eine Überprüfung und eine eventuelle Anpassung von Traceability-Links zur Folge haben [MGP08]. Solche Ereignisse sind beispielsweise die Veränderung, das Löschen, oder das Hinzufügen eines neuen Modells in das Repository. Jedes Modell muss sich dabei bei einer zentralen Komponente registrieren und diese Komponente im Falle einer Änderung benachrichtigen. Diese zentrale Komponente sorgt anschließend dafür, dass alle von diesem Modell abhängigen Modelle und Traceability-Links benachrichtigt werden und entsprechende Reaktionen, meist die Abarbeitung von Regeln, ausgelöst werden. Mit diesem Ansatz ist es somit besonders einfach, Änderungen an Traceability-Links weiterzugeben und auf die Evolution der Software zu reagieren.

3.3.4. Hypertext-basierte Erzeugung von Traceability-Links

Das Hauptmerkmal dieses Ansatzes ist die umfassende Verwendung von XML als Basistechnologie. Entwurfsmodelle, Regeln und Traceability-Links werden als XML-Dokumente aufgefasst, die zudem in einem Repository versioniert werden können. Desweiteren beschränken sich die Vertreter dieses Ansatzes nicht auf einige ausgewählte Modelltypen, wie etwa die Verbindung von informalen Anforderungsbeschreibungen und UML-Modellen. Für das eigentliche Auffinden von Traceability-Beziehungen kommen dabei sowohl Methoden des Information Retrieval, als auch fest definierte Regeln zum Einsatz.
Dieser Ansatz ist somit vielmehr als eine Verbindung von Information Retrieval und regelbasierter Suche zu verstehen, mit dem Hauptziel das Modelle und Links aus der gleichen XML-Technologie aufgebaut werden können.

3.3.5. Schlussfolgerungen für das Konzept dieser Arbeit

Für das in dieser Arbeit in Kapitel 4 zu entwickelnde Konzept ist eine Kombination der hier vorgestellten Methoden einer der vielversprechendsten Ansätze.
Da sich das Modell der regelbasierten Traceability-Erzeugung aufgrund der genauen Spezifikation der dabei erzeugten Links als besonders geeignet erwiesen hat, wird dieser Ansatz als einer der Hauptbestandteile in das hier zu entwickelnde Konzept aufgenommen. Da viele Regeln auf dem Vergleich von Modellnamen bzw. Modelleigenschaften aufbauen, bietet es sich an für diese Vergleiche Methoden aus dem Information Retrieval zu nutzen, um möglichst viele der bestehenden Abhängigkeiten zu erkennen. Aus dem Bereich der Hypertext-basierten Verfahren lässt sich die Grundidee entnehmen, dass alle Modelle, alle Links und das Repository selber nur aus XML-basierten Daten bestehen. Durch die Nutzung von EMFStore als Basis für das Repository ist diese Idee bereits umgesetzt, da EMFStore sämtliche Modelle, Änderungsoperation und Projektdaten als XML-Dateien abspeichert. Einzig für die Repräsentation der Traceability-Links und Regeln muss noch ein geeignetes XML-Format gefunden werden. Abschließend können von den Event-basierten Ansätzen noch Ideen und Methoden zur Behandlung von Traceability-Links im Falle von Modelländerungen und Ergänzungen verwendet werden.

3.4. Sprachen zur Definition von Traceability-Regeln

In den folgenden drei Abschnitten sollen drei Anfragesprachen bzw. Konzepte für Anfragesprachen vorgestellt und bewertet werden, die für die automatisierte Generierung von Traceability-Links genutzt werden können.

3.4.1. XML Query Language (XQuery)

Die XML Query Language, kurz XQuery [XQUERY] bezeichnet eigentlich eine Abfragesprache, die für XML-Datenbanken entworfen wurde und Anfragen über die logische Baumstruktur eines XML-Dokuments auflöst.In [JZ09] wird die XQuery zur Identifikation möglicher Kandidaten für Traceability-Links genutzt. Jede Regel besteht dabei aus einer XQuery-Anfrage, die in ein eigens definiertes Regelformat eingebettet ist, welches neben dem Typ der Traceability-Beziehung und einer kurzer Beschreibung der Regel auch die eigentlichen Aktionen beschreibt, die im Falle einer erfolgreichen Anfrage ausgelöst werden sollen. Die in Abbildung 3 dargestellte Anfrage kommt beispielsweise zur Suche nach Containment-Traceability-Beziehungen zwischen Prozessmodellen und UML-Klassendiagrammen zum Einsatz:

```
<Query>
    declare namespace UML="org.omg.xmi.namespace.UML";
    declare namespace s="java:synonym.s";
    for $item1 in doc(£2£)//UML:Classifier.feature/UML:Operation/@name
    for $item2 in doc(£1£)//Process/Description
    let $t1 := $item1/../../../@name
    where s:containsSInDistance($item2,$item1, $t1)
</Query>
```

Abbildung 3: XQuery Anfrage

XQuery bietet dabei den Vorteil, dass es bereits 3 verschiedene Prozessoren für die Eclipse Entwicklungsumgebung gibt. Allerdings ist die Lesbarkeit von komplexeren XQuery-Anfragen eher als Schlecht zu bewerten, dazu kommt der Nachteil, dass Modelle immer erst materialisiert werden müssen, bevor sie mit XQuery abgefragt werden können. D.h. ein direktes Arbeiten mit XQuery im Repository ist somit nicht möglich, da die gespeicherten Modelle vorher in einem temporären Arbeitsverzeichnis materialisiert werden müssten, oder ein zusätzlicher Adapter für die Auflösung der Anfrage geschaffen werden müsste. Um mit XQuery Traceability-Links zu generieren, muss man wie in [JZ09] aufgezeigt, trotzdem eine zusätzliche Regelsprache entwerfen, die die XQuery-Anfragen

kapselt und mit weiteren Informationen versieht. Zusätzlich muss für dieses Regelwerk ein eigener Interpreter geschrieben und zudem die Möglichkeit der Syntaxprüfung für Regeln gegeben werden. Aufgrund dieser eher als negativ zu bewertenden Eigenschaften habe ich mich gegen den Einsatz von XQuery als Anfragesprache für das Repository entschieden.

3.4.2. Graph Repository Query Language (GReQL)

Speziell für Entwurfsmodelle die als TGraphen [TGRAPH] vorliegen, wurde eine eigene Anfragesprache, die Graph Repository Query Language [GREQL] entwickelt und in [SEW09] beispielhaft verwendet. Mit dieser Sprache können relevante Modelle und Modellelemente aus der Baumdarstellung extrahiert und entsprechend verarbeitet werden. Durch die in Abbildung 4 dargestellte GReQL-Anfrage werden beispielsweise alle Anwendungsfälle mit Interfaces verknüpft, die Schnittstellen für deren Implementierung bereitstellen.

```
from u:V{UseCase}, i:V{Interface}
with u.name = ''Send Bill''
     and u <--{Satisfies} i
report i.name
end
```

Abbildung 4: GReQL Anfrage

Die Tatsache dass mit GReQL nur auf TGraphen gearbeitet werden kann, schließt eine Verwendung in EMFStore aus, da dort Modelle im Format des Unicase-Metamodells gespeichert werden, welches grundsätzlich dem Format gängiger CASE-Werkzeuge entspricht. Eine Konvertierung der Ausgabeformate der CASE-Werkzeuge in TGraphen wäre grundsätzlich möglich, jedoch wesentlich komplexer und zeitaufwendiger, als beispielsweise die Konvertierung in das von EMFStore genutzte Format. Desweiteren sind mit Attributen versehene Beziehungs-Kanten eines der Hauptmerkmale von TGraphen, welches sich nicht in MOF bzw. EMOF-konformen Modellen ausdrücken lässt, da dieses Konzept dort nicht unterstützt wird [SEW09]. Als Lösung dafür bliebe nur die Schaffung eines neuen Modellelementes, mit dem man diese speziellen Kanten nachbilden kann. Allerdings müsste man somit jede Kante im TGraph durch ein neues Modellelement repräsentieren, was wiederum zu erhöhtem Speicherverbrauch und verlängerten Einlese- und Importdauern der Modelle führen würde. Bedingt durch diesen erhöhten Aufwand bei der Modellierung, dem Import und Export, sowie der Konvertierung von Modellen habe ich mich gegen die Einbindung von GReQL und TGraphen in EMFStore entschieden. Jedoch bleibt die Option bestehen, dass man das Repository zukünftig auch um TGraphen und GReQL erweitern kann, womit die Forderung nach der Erweiterbarkeit des Repository nicht verletzt wird.

3.4.3. XML-basierte Regelsprachen

Neben den bereits vorgestellten Anfragesprachen XQuery und GReQL gibt es auch noch einige andere Anfragesprachen, die direkt in XML-Dokumenten definiert werden und auch nur auf Entwurfsmodellen im XML-Format arbeiten.

Besonders Georg Spanoudakis und Andrea Zisman haben in [SGZ03], [FZS03], [SZMK04] und [ZEF2000] eine syntaktisch an SQL angelehnte Anfragesprache entworfen (siehe Abbildung 5), die neben dem Erzeugen von Traceability-Links [SGZ03, FZS03, SZMK04] auch zur Konsistenzprüfung von Modellen genutzt werden kann [ZEF2000]. Diese Arbeiten haben jedoch auch wieder den Nachteil, dass die darin entwickelten Sprachen nur auf materialisierten XML-Dateien arbeiten können, was wiederrum dem Konzept eines Repository für Entwurfsmodelle wiederspricht. Zudem sind die Prototypen bzw. Implementierung nicht frei verfügbar und lassen sich daher nicht auf ihre Eignung für EMFStore untersuchen. Allerdings sind diese 4 Arbeiten ein guter Ausgangspunkt für die Definition einer eigenen Regelsprache.

```
RTOM_RULE Rule-4
EXISTS
  SEQUENCE( <x1/{NN1, NP1, JJ}>:1, <x2/{NN1, NP1, JJ}>:*, <x3/{NN1}>:1, <x4/{VBZ}>:1) in A1/UC.precondition;
  <x5/CLASS> <x6/OPERATION>     in OM
SUCH THAT
  OPERATION_OF(<x6>,<x5>) and CONTAINS(NAME(<x5>,<x1>) and
  EQUAL_TO(STEREOTYPE(<x6>), "get") and CONTAINS(NAME(<x6>), <x3>)
ACTION
 GENERATE REQUIRES_EXECUTION_OF(A1,<x5>)
RTOM_RULE_END
```

Abbildung 5: XML-basierte Beispielregel

3.4.4. Schlussfolgerungen für das Konzept dieser Arbeit

Wie den in Kapitel 3.3.2. und 3.4. vorgestellten Ansätzen zu entnehmen ist, sollte jede Regel zur Erzeugung von Traceability-Links in 3 Teile gegliedert werden, um eine optimale Trennung von Elementdeklaration, Anfrage und Ergebnisverarbeitung zu erzielen. Die hier untersuchten Lösungen lassen sich jedoch lediglich zur Modellierung des Anfrageteils nutzen und für die Kapselung dieser Anfragen in Regeln müsste trotzdem ein zusätzlicher Interpreter geschaffen werden. Da die Ansätze jedoch nur auf materialisierten XML-Dateien bzw. mit der Schnittstelle einer XML-Datenbank arbeiten können, müsste ein zusätzlicher Adapter für EMFStore geschaffen werden, der die Anfragen interpretieren und auflösen kann, bevor sie an EMFStore weitergereicht werden. D.h. es müssten somit insgesamt zwei Komponenten entworfen werden. Zum einem ein Regelinterpreter und der zusätzliche Adapter für EMFStore. Daher habe ich mich dazu entschlossen, keinen der vorgestellten

Ansätze direkt zu verwenden und stattdessen eine eigene Anfragesprache für das Repository zu entwickeln, die später auch für andere Zwecke genutzt werden kann, so etwa für Konsistenzanalysen. Für den Regelinterpreter ergeben sich damit keine neuen Anforderungen, da sich lediglich die Anfragesprache, nicht aber das Konzept geändert hat und ein Adapter für EMFStore somit nicht mehr notwendig ist. Die dabei zu entwickelnde Anfragesprache soll von Beginn an so entworfen werden, dass sie optimal auf Modellen und deren Attributen in einem Repository und damit unabhängig von Dateiformaten und Datenbankschnittstellen arbeiten kann.

4. Entwurf

In diesem Kapitel soll der Entwurf der EMFTrace-Kernkomponenten schrittweise erläutert werden, sowie die Entwicklung eines Metamodells für Traceability-Links und Traceability-Regeln, als auch deren Einbindung in EMFStore aufgezeigt werden.

4.1. Anforderungen an das Repository

An das Repository bzw. EMFTrace werden eine Reihe von Anforderungen gestellt, die im Folgenden nach funktionalen und nichtfunktionalen Anforderungen gegliedert werden sollen.

Funktionale Anforderungen:

1. Einbindung der UML-, URN- (bestehend aus GRL und UCM) und OWL-Metamodelle um deren Instanzen im Repository verwenden zu können.
2. Import von Instanzmodellen aus CASE-Werkzeugen in das Repository, durch die Schaffung der dafür notwendigen Konvertierungsmöglichkeiten.
3. Export von Instanzmodellen aus dem Repository, um sie mit dem ursprünglichen CASE-Werkzeug erneut bearbeiten zu können. Auch hierfür müssen, falls erforderlich, die entsprechenden Konvertierungsmöglichkeiten geschaffen werden.
4. Speicherung und Pflege von Traceability-Regeln im Repository durch Einbindung eines Metamodells für Traceability-Regeln und Schaffung eines Regelkataloges.
5. Speicherung und Pflege von Traceability-Links im Repository durch Einbindung eines Metamodells für Traceability-Links.
6. Import und Validierung neuer Traceability-Regeln in den Regelkatalog des Repository.
7. Speicherung und Pflege von Traceability-Typen im Repository mit Hilfe eines speziellen Typkataloges.
8. Automatische Suche nach Traceability-Beziehungen durch Abarbeitung der Regeln aus dem Regelkatalog. Dabei sollen gefundene Beziehungen automatisch in entsprechende Links umgesetzt werden.
9. Eine benutzergesteuerte Suche nach Traceability-Beziehung unter Angabe welche Regeln auf welche Modelle angewandt werden sollen, muss möglich sein.
10. Das Repository muss in der Lage sein, transitive Beziehungen automatisch erkennen zu können.
11. Die Kernkomponenten des Repository sollen n-äre Links für zukünftige Erweiterungen unterstützen.

12. Sämtliche Modellinstanzen sollen durch das Repository versioniert werden und jedes Modell soll zu jedem Zeitpunkt in einer beliebigen Version materialisiert werden können.

Nichtfunktionalen Anforderungen:

1. Alle Metamodelle sollen durch austauschbare Eclipse Plug-Ins in das Repository eingebunden werden können.
2. Jede funktionale Komponente soll als austauschbares Eclipse Plug-In realisiert werden.
3. Das gesamte Repository soll möglichst flexibel und erweiterbar gehalten werden.
4. Die Benutzeroberfläche soll dem Eclipse-üblichen Stil entsprechen und deren Handhabung schnell erlernbar sein.

4.2. Das Kernkonzept

Durch die Nutzung von EMFStore als Repository und Grundgerüst für EMFTrace (siehe dazu auch Abschnitt 3.1.3.: „*EMFStore*“) erübrigt sich die Schaffung eines eigenen Repository für das Projekt. EMFTrace kann somit schrittweise auf EMFStore aufgebaut werden bzw. EMFStore durch neue Funktionalität und Modelle ergänzen.
Für die regelbasierte Erzeugung von Traceability-Links in EMFTrace müssen dabei zunächst die folgenden Dinge realisiert werden:

1. Ein explizit modelliertes Traceability-Metamodel
2. Ein Schema für Traceability-Regeln
3. Ein Konzept für einen Regelinterpreter

Das Traceability-Metamodell soll dabei die größtmögliche Menge an Informationen darstellen können, die man mit Traceability-Links ausdrücken kann. Zudem dient die Einbindung des Metamodells in EMFStore zur direkten Speicherung von Links im Repository, womit eine externe Speicherung und all die damit verbundenen Problemstellungen (Verzeichnisstruktur, Dateiaufbau, Synchronisation etc.) von Beginn an vermieden werden.
In Abschnitt 3.4. wurde erläutert, wie man Regeln zur Traceability-Generierung aufbauen kann und warum keiner der dort untersuchten Ansätze für EMFTrace geeignet ist. Die Implementierung einer eigenen Regelsprache erfordert daher zuallererst die Schaffung eines geeigneten Schemas für solche Regeln, mit dem sich diese effizient definieren lassen. Desweiteren soll dieses Schema so erweitert werden, dass sich die Regeln damit in einem speziellen Regelkatalog direkt im Repository abspeichern lassen.

Abschließend muss eine geeignete Architektur für den Regelinterpreter geschaffen werden, die eine größtmögliche Flexibilität und Erweiterbarkeit für neue Regeln gewährleisten kann. Neben dem Regelinterpreter (RuleEngine) müssen dabei noch 2 weitere Komponenten geschaffen werden, um die geforderte Funktionalität möglichst erweiterbar zu halten. Zum einem bedeutet dies die Schaffung einer Zugriffsschicht (AccessLayer), die lesende und schreibende Zugriffe auf das Repository kapselt und zum anderen die Erstellung einer Logikkomponente für das Erstellen, Löschen und Bearbeiten von Traceability-Links (LinkManager).

Wie aus Abbildung 6 ersichtlich ist, verteilt sich die Funktionalität auf die zwei Hauptkomponenten EMFStore und EMFTrace. Während EMFStore für die Speicherung und Versionierung von Modellen, Traceability-Links und Regelkatalogen zuständig ist, stellt EMFTrace die Funktionen zum Auswerten von Regeln, Validieren von Traceability-Links und weitere Analysemethoden bereit. Hierbei wird besonders deutlich, wie der gekapselte Zugriff auf die Modelle im Repository durch die EMFTrace-interne AccessLayer-Komponente realisiert werden soll.

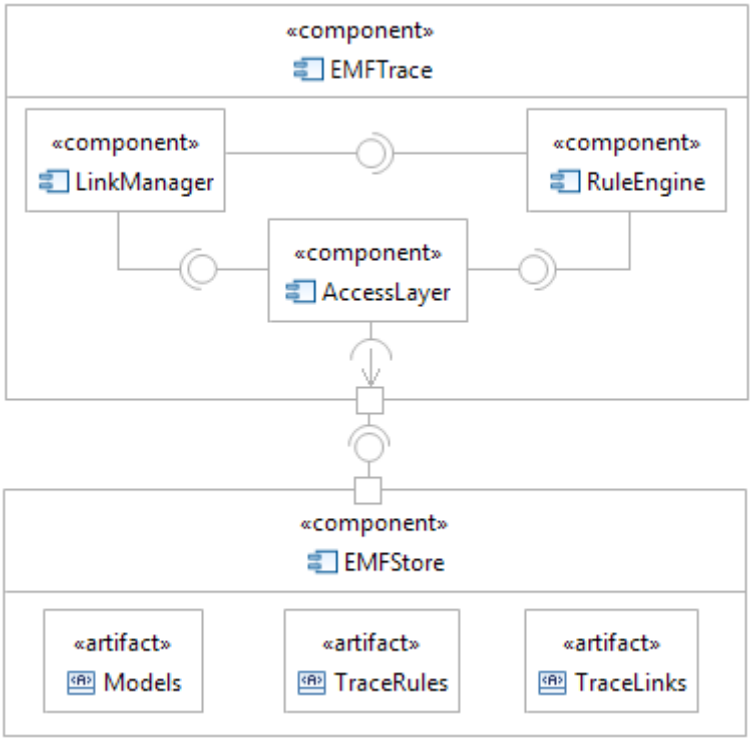


Abbildung 6: Kernkomponenten der Anwendung

Beginnend in Kapitel 4.5. werden die in Abbildung 6 gezeigten Kernkomponenten der Anwendung schrittweise in ihrem Entwurf erläutert und verfeinert. Aus Platzgründen ist das Komponentendiagramm, welches die hier vorgestellten Komponenten, deren Schnittstellen und Beziehungen zusammenfassend darstellt in Anhang K dieser Arbeit untergebracht.

Für zukünftige Erweiterungen bietet dieses Konzept die Möglichkeit, neue Komponenten in EMFTrace und EMFStore zu registrieren. So können Konsistenzanalysen beispielsweise auch über Regeln ausgeführt werden, die vom Interpreter verarbeitet und in einem speziellen Regelkatalog hinterlegt werden. Da die Regeln zudem eine konzeptionelle Überschneidung mit Anfragesprachen wie etwa SQL besitzen, ist eine zukünftige Erweiterung um spezielle Anfrage-Regeln mit denen gezielt nach Modellen im Repository gesucht werden kann durchaus denkbar.

Die möglichen Erweiterungen um Komponenten zur Konsistenzanalyse und Modellanfrage werden dabei in Abbildung 7 beispielhaft skizziert.

Neben den neuen Komponenten, die die zusätzliche Funktionalität in EMFTrace integrieren, sind dabei auch Ergänzungen an EMFStore vorzunehmen. Diese umfassen neben der Einbindung neuer Metamodelle für weitere CASE-Werkzeuge auch die Einbindung neuer Regelkataloge, beispielsweise für Konsistenzanalysen.

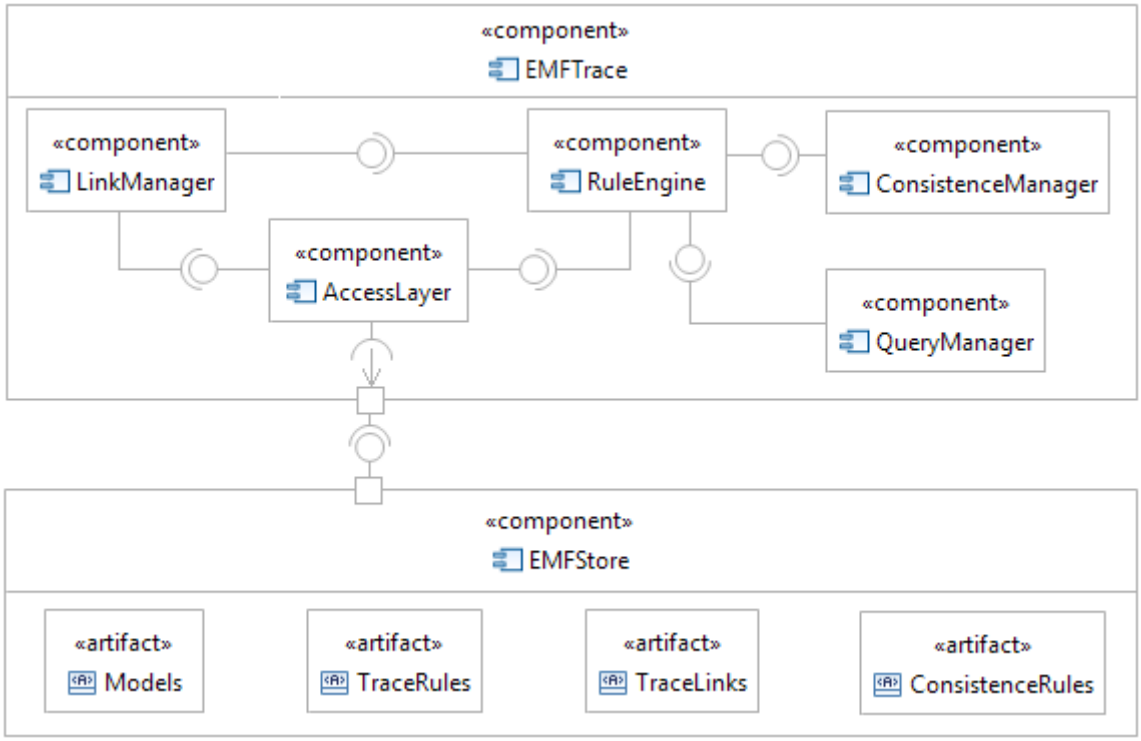


Abbildung 7: Zukünftige Erweiterungen

Sämtliche Kernkomponenten werden dabei als Plug-Ins für Eclipse realisiert, was einerseits die spätere Austauschbarkeit und vor allem das einfache Ergänzen neuer Komponenten wesentlich vereinfacht. Die Metamodelle für Traceability-Links, Traceability-Regeln, UML-Modelle, URN-Modelle und OWL-Modelle werden dabei auch als eigenständige Eclipse Plug-Ins in EMFStore registriert, um Instanzen dieser Metamodelle in EMFStore und damit auch EMFTrace nutzen zu können (siehe dazu auch Abschnitt 5.1.: *„Einbindung der Metamodelle in EMFStore“*).

Insgesamt werden dabei folgende Eclipse Plug-Ins im Rahmen dieser Arbeit entwickelt:

EMFTrace_Core - Kernfunktionalität der Anwendung, siehe Kapitel 4.5.
EMFTrace_GUI - grafische Benutzeroberfläche, siehe Kapitel 4.6.
EMFTrace_TraceLink - Metamodell für Traceability-Links, siehe Kapitel 4.3.
EMFTrace_TraceRules – Metamodell für Traceability-Regeln, siehe Kapitel 4.4.2.
EMFTrace_URN - Metamodell für URN-Modelle, siehe Kapitel 5.1.1.
EMFTrace_OWL - Metamodell für OWL-Modelle, siehe Kapitel 5.1.2.
EMFTrace_UML - Metamodell für UML-Modelle, siehe Kapitel 5.1.3.

4.3. Entwurf des Traceability-Metamodells

In Anlehnung an [DKPF09] muss für die Erzeugung von Traceability-Links ein eigenes Metamodell erstellt werden, um später Traceability-Links als dessen Instanzen erzeugen zu können. In Kapitel 3.2. wurden bereits einige Aspekte eines solchen Metamodells vorgestellt, die hier verfeinert werden sollen. Speziell in EMFTrace sprechen folgende Gründe für die Schaffung eines expliziten Metamodells:

1. Traceability-Links sollen im Repository gespeichert und versioniert werden, dies ist aber nur durch die vorherige Einbindung eines EMF-basierten Metamodells möglich.
2. Ein Metamodell bietet eine genaue Definition wie Traceability-Links aussehen dürfen und aus welchen Daten und Elementen sie bestehen. Somit ist eine feste Referenz vorhanden, an der sich der Nutzer, oder ein Entwickler der Erweiterungen einbindet orientieren kann.
3. Durch das Metamodell ist es prinzipiell möglich Konvertierungsmethoden zu implementieren, um exportierte Traceability-Links in anderen CASE-Werkzeugen verwenden zu können. Auch der umgekehrte Weg, der Import von Traceability-Links in EMFStore, wäre somit prinzipiell denkbar.

Da die Option für eine zukünftige Erweiterung um n-äre Links gleich von Beginn an im Entwurf berücksichtigt werden soll, erfordert es einige Umstrukturierungen des üblichen Konzepts der binären Links (d.h. eine Abbildung einer Quelle auf genau ein Ziel).
Ein n-ärer Traceability-Link besteht nun nicht mehr nur aus einem Ziel- und Quellknoten, sondern unter Umständen aus einer Menge von Ziel- und Quellknoten.

In Abbildung 8 wird das hierfür entwickelte Metamodell mit all seinen Bestandteilen dargestellt, aus Gründen der Übersichtlichkeit wurden jedoch Referenzen auf das zugrunde liegende Unicase-Metamodell ausgespart. Für die Darstellung der Vielzahl an Endpunkten auf die ein n-ärer Link verweisen kann, wird die neue Klasse *LinkEnd* geschaffen. Diese Klasse beinhaltet neben einer Referenz auf das eigentliche Modellelement auch zwei Attribute, die dieses Element entweder als Quell- oder Zielknoten ausweisen.

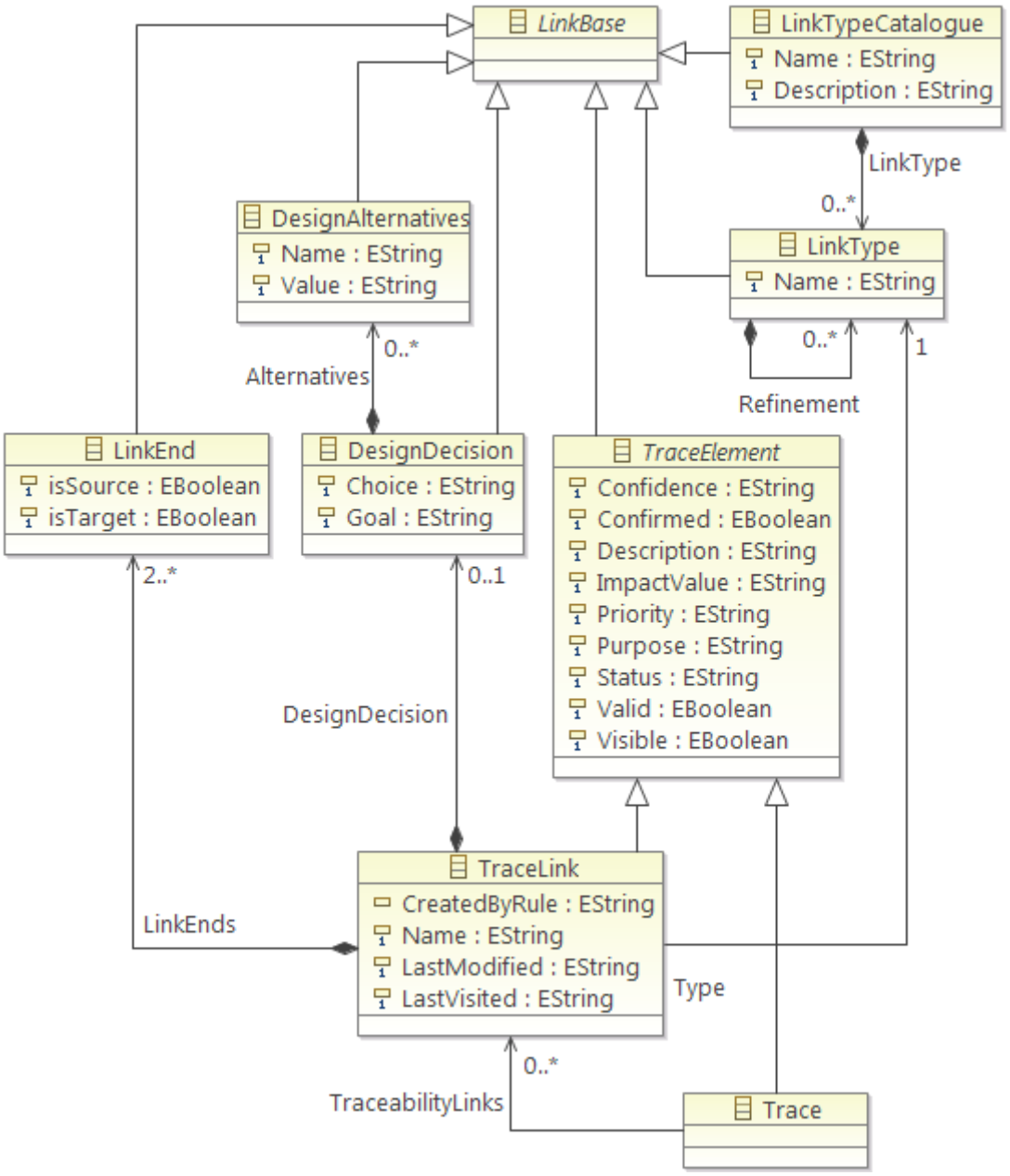


Abbildung 8: Traceability-Metamodell

Die Klasse *TraceLink* verbindet nun mindestens 2 *LinkEnd*-Elemente mittels Traceability-Beziehung und verfügt darüber hinaus über eine Reihe von Attributen, die von der Basisklasse *TraceElement* geerbt werden und den Link näher beschreiben und seinen Zweck erläutern.
Jeder *TraceLink* kann zudem mit einer Instanz der Klasse *DesignDecision* versehen werden, die Auskunft darüber gibt, warum der Link erstellt wurde und zudem mit mehreren *DesignAlternative*-Instanzen angereichert werden kann.

Eines der wichtigsten Attribute eines Traceability-Links ist jedoch der Linktyp, da er aussagt wie Elemente miteinander in Beziehung stehen. Da die Linktypen im Projekteinsatz ständigen Änderungen unterliegen können, bietet sich die Schaffung einer eigenen Klasse für Linktypen an, in der die eigentlichen Typinformationen gekapselt werden. Im gleichen Atemzug stellt sich dabei die Frage nach der Speicherung von Linktypen in EMFStore. Die Instanzen der Klasse *LinkType* werden in der Container-Klasse *LinkTypeCatalogue* zusammengefasst, gespeichert und somit auch durch EMFStore versioniert. Neue Linktypen können dabei auch als Kindelemente bestehender *LinkType*-Instanzen erzeugt werden und stellen somit eine Verfeinerung des Linktyps dar. In Anhang H („Traceability-Klassifizierung“) ist die aktuell verwendete Hierarchie der Linktypen und ihrer Verfeinerungen dargestellt. Durch die zentrale Speicherung und Pflege der Linktypen werden zudem Inkonsistenzen vermieden, die durch eine verstreute bzw. dezentrale Speicherung der Typinformationen entstehen könnten.
Neben *TraceLink*-Instanzen können auch einfache *Trace*-Elemente erzeugt werden, die eine besondere Rolle im Reengineering spielen. So können zu Beginn der Systemanalyse einfache *Trace*-Instanzen erzeugt werden, die mit fortschreitendem Verständnis der Software-Architektur immer weiter verfeinert werden können und schließlich zu einer transitiven Kette von *TraceLink*-Instanzen „heranwachsen“.

Zudem werden alle hier vorgestellten Elemente von der abstrakten Basisklasse *LinkBase* abgeleitet, deren einziger Zweck die Einbindung in EMFStore ist, da sie direkt vom Basiselement des Unicase-Metamodells erbt. Dieses Vorgehen schließt dabei direkt an eine Empfehlung der Unicase-Entwickler an, die besagt das jedes Metamodell über eine eigene Basisklasse mit dem Unicase-Metamodell verbunden werden sollte, um den Aufwand bei späteren Änderungen zu minimieren.

4.4. Traceability-Regeln

Für die Suche nach Traceability-Beziehungen zwischen Elementen aus dem Repository sollen vom Benutzer definierte Regeln zur Anwendung kommen, die aussagen, unter welchen Bedingungen Elemente auf welche Art und Weise verknüpft werden können.
In Anlehnung an die in Kapitel 3.3.2. und 3.4. untersuchten Ansätze und Regelsprachen, sollen die hier entstehenden Regeln dem üblichen Schema des Regelaufbaus [ZEF2000] folgen. Jede Regel besteht dabei aus einem Kopfteil, in dem die betreffenden Elemente deklariert werden. In dem anschließenden Anfrageteil werden Bedingungen definiert, die an die Elemente aus dem Kopfteil gestellt werden. Diese Bedingungen können dabei auch beliebig geschachtelt und durch die logischen Operatoren *Und*, *Oder*, *Nicht* und *Exklusiv-Oder* miteinander verknüpft werden. Den Abschluss der Regeldefinition bildet der Aktionsteil, der festlegt was mit den Elementen passieren soll, die den an sie gestellten Bedingungen genügen. Abbildung 9 liefert dabei ein einfaches Beispiel zur Demonstration des Regelaufbaus.

```
<TraceRule RuleID="TraceRule44" Description="Find equivalent properties in UML and OWL">
    <Elements Type="ObjectProperty" Alias="e1"/>
    <Elements Type="Property" Alias="e2"/>
    <Conditions Type="Or">
        <BaseConditions Type="Contains" Source="e1::IRI" Target="e2::name"/>
        <BaseConditions Type="Contains" Source="e1::AbbreviatedIRI" Target="e2::name"/>
        <BaseConditions Type="Contains" Source="e1::FullIRI" Target="e2::name"/>
    </Conditions>
    <Actions ActionType="CreateLink" LinkType="Equivalent" LinkSource="e1" LinkTarget="e2"/>
</TraceRule>
```

Abbildung 9: Beispielhafte Traceability-Regel

Die Regeln können dabei entweder in einer XML-Datei definiert und via Modellimport in EMFStore geladen werden, oder direkt in EMFStore erstellt werden. Für die Prüfung einer Regel auf syntaktische Korrektheit bietet sich die Definition eines XSD-Schemas zur Validierung der Regeln an, welches direkt beim Import aufgerufen werden kann an und somit den Import von syntaktisch falschen Regeln unterbindet. Für Regeln die direkt im Repository erzeugt oder verändert werden, muss dennoch ein Validierungsalgorithmus entworfen werden, da eine Prüfung mittels XSD-Schema eine Materialisierung der Regel in XML-Form voraussetzen würde.

Auf Modellelemente kann innerhalb der Regel immer über den Alias-Namen zugegriffen werden. Der Zugriff auf Modellattribute erfolgt dabei durch den Scope-Operator („::“). So wird im obigen Beispiel (siehe Abbildung 9) wie folgt auf den Namen einer UML-Property zugegriffen:
e2::name

In den Regeln können dabei folgende Operationen zur Auswertung der Attribute genutzt werden:

- Equals: prüft, ob der Attributwert eines Elements einem geforderten Wert entspricht
- Contains: prüft, ob der Attributwert eines Elements eine Zeichenkette enthält
- IsParent: prüft, ob ein Element in einem anderen Element enthalten ist
- SimilarTo: prüft, ob der Attributwert eines Elements ähnlich zu einem geforderten Wert ist
- LesserThan: prüft, ob der Attributwert eines Elements kleiner als ein geforderter Wert ist
- GreaterThan: prüft, ob der Attributwert eines Elements größer als ein geforderter Wert ist
- NotNull: prüft, ob ein Attribut in einem Modell vorhanden ist

Hierbei sei jedoch noch anzumerken, dass diese Bedingungen in Zukunft noch um weitere Typen ergänzt werden können, etwa dann wenn eine Anfragekomponente in EMFTrace verankert werden soll. Für zukünftige Erweiterungen müssen dafür lediglich 2 Änderungen vorgenommen werden. Zum einen muss das Metamodell um den neuen Typ ergänzt werden und zum anderen muss der Interpreter mit Code zur Abarbeitung des neuen Typs angereichert werden (siehe Kapitel 5.3.2.).

4.4.1. Klassifizierung von Traceability-Regeln

Die in EMFTrace verwendeten Linktypen und die dadurch entstandene, detaillierte Hierarchie von Linktypen kann im Anhang H „Traceability-Klassifizierung“ eingesehen werden. Die hier entstandene Hierarchie ist eine Komposition aus Vorschlägen und Spezifikationen in [JZ09], [FZS03], [LET02], [MPR07], [RBFB2010], [SZ05], [SZMK04], sowie eigenen Ideen und Ergänzungen. Eine Übersicht aller für EMFTrace entwickelten Traceability-Regeln findet sich in Anhang I.

4.4.2. Einbindung der Regeln in EMFStore

Für die Einbindung und Versionierung der Regeln muss wieder ein spezielles Metamodell geschaffen werden, um die Regeln in EMFStore verfügbar zu machen. Das Metamodell ist in Abbildung 10 dargestellt und aus Gründen der Übersichtlichkeit wurden auch hier wieder die Elemente des Unicase-Metamodells ausgeblendet.
In Analogie zum Traceability-Metamodell aus Kapitel 4.3. wurde für die Verknüpfung mit dem Unicase-Metamodell wieder eine zentrale, abstrakte Basisklasse *RuleBase* verwendet.
Regeln zur Prüfung auf Abhängigkeitsbeziehungen werden durch die Klasse *TraceRule* realisiert und lassen sich später auch für andere Prüf- und Suchmethoden, wie etwa Konsistenzprüfungen verwenden. Für die effiziente Verwaltung in EMFStore werden alle Regeln in einem Katalog, dem *TraceRuleCatalogue* zusammengefasst und gespeichert. Für eine spätere Ergänzung von

Konsistenzregeln muss dabei lediglich ein neuer Katalog angelegt werden, der die Konsistenzregeln aufnimmt und es müssen die eigentlichen Konsistenzregeln in EMFStore importiert werden. Wenn EMFTrace zudem um eine Abfragekomponente ergänzt werden soll, würde sich ein weiterer Katalog für Anfrage-Regeln anbieten, der häufig genutzte Anfragen speichern kann um so einen schnellen Zugriff auf diese zu gewährleisten.

Die Bündelung von Regeln in Katalogen verfolgt dabei mehrere Ziele. Zum einen soll durch die Verwendung von Katalogen die Austauschbarkeit der Regeln verbessert werden, da man nun einfach ganze Kataloge und nicht jede Regel einzeln exportieren, oder importieren muss. Zum anderen sorgen in Katalogen gekapselte Regeln für wesentlich mehr Übersicht im Repository-Browser, was wiederum die eigentliche Arbeit mit dem Repository erleichtert.

Die durch die Regeln zu untersuchenden Modellelemente werden durch Instanzen der Klasse *ElementDefinition* beschrieben, die jedem Elementtyp einen Alias-Namen zuordnet, mit dem auf diesen Typ bei der späteren Verarbeitung zugegriffen werden kann.

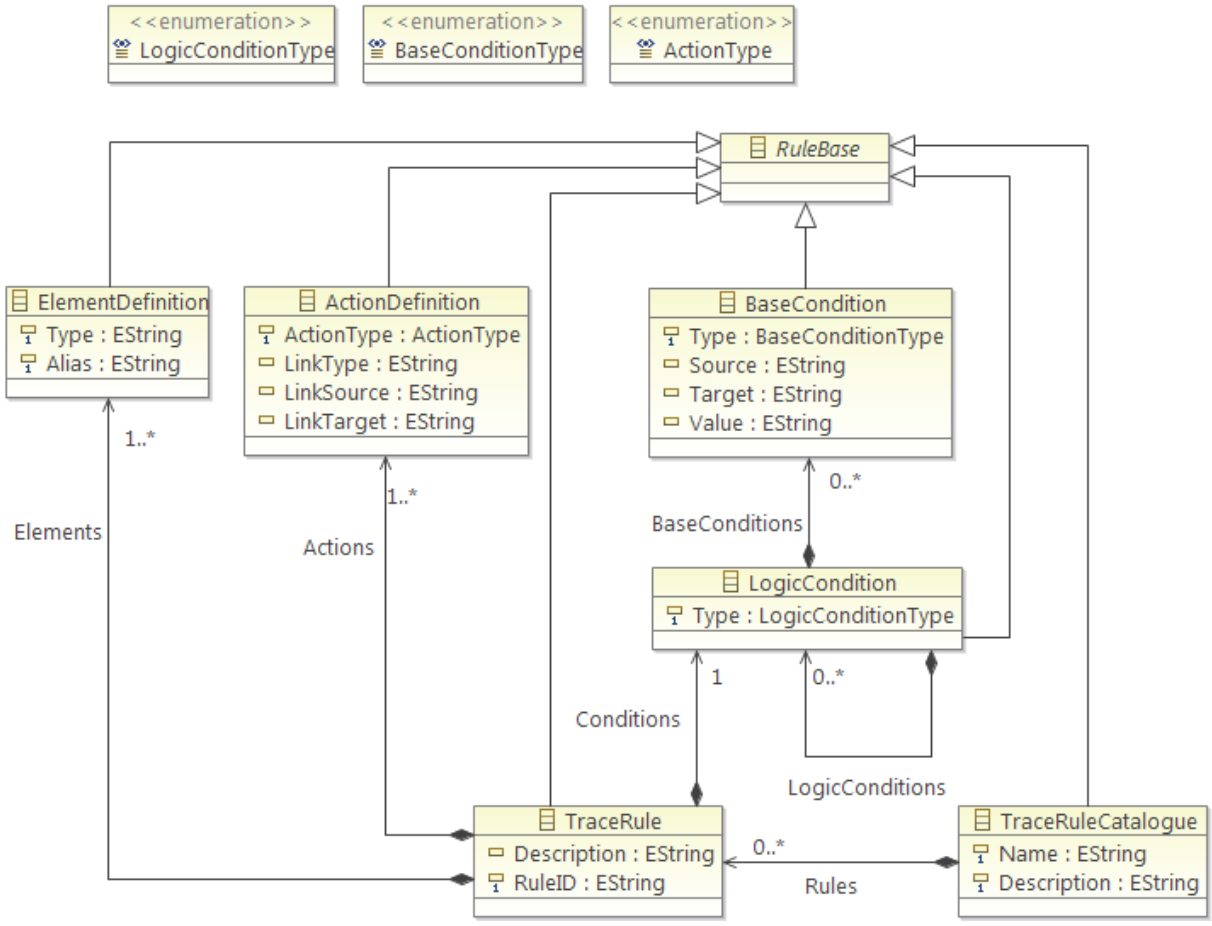


Abbildung 10: Metamodell für Traceability-Regeln

Die logischen Verknüpfungsoperatoren *Und*, *Oder*, *Nicht* und *Exklusiv-Oder* werden durch Instanzen der Klasse *LogicCondition* repräsentiert, die zudem beliebig geschachtelt werden können. Eine Operatorreihenfolge wie *NICHT(x UND y)* wird dabei in der Art in Regeln umgesetzt, wie sie in Abbildung 11 beispielhaft skizziert ist.

```
<Conditions>
    <Not>
        <And>
            x
            y
        </And>
    </Not>
</Conditions>
```

Abbildung 11: Logische Operatoren in Regeln

Die eigentlichen Prüfregeln werden durch die Klasse *BaseCondition* implementiert, die auf die Attributwerte von Modellelementen zugreifen und diese vergleichen kann. Neben verschiedenen Auswertungsvarianten für Attributwerte bietet diese Klasse auch die Möglichkeit, zwei Modellelemente auf eine Eltern-Kind-Beziehung zu überprüfen. Prüfregeln vom Typ *BaseCondition* werden dabei immer sequentiell abgearbeitet, d.h. im obigen Beispiel (siehe Abbildung 11) würde Regel *x* immer vor Regel *y* geprüft werden.
Sobald alle Regeln abgearbeitet wurden, erfolgt die Ergebnisauswertung mittels der in einer Instanz der Klasse *ActionType* vorgegebenen Bedingungen. Momentan ist dabei nur das Anlegen neuer Traceability-Links implementiert, jedoch kann der Aktionsteil noch um weitere Aktionen, wie etwa das Hinweisen des Anwenders auf verletzte Konsistenzregeln erweitert werden.

4.5. Entwurf der EMFTrace-Kernarchitektur

Wie in Kapitel 4.2 und Abbildung 6 dargestellt wurde, besteht die eigentliche Anwendungslogik von EMFTrace aus den drei Komponenten AccessLayer, LinkManager und RuleEngine. Der LinkManager und die RuleEngine greifen dabei stets auf die Funktionalität zurück, die ihnen die Komponente AccessLayer zum Zugriff auf Elemente des EMFStore bietet. Von daher bietet es sich an, eine gemeinsame Basisklasse für beide Komponenten zu entwerfen, die den Zugriff auf den AccessLayer realisiert. Werden später neue Komponenten zu EMFTrace hinzugefügt, kann man diese auch von der Basiskomponentenklasse ableiten um somit den gekapselten Zugriff auf EMFStore zu erhalten. Diese Basisklasse erhält daher den Namen *TraceComponent* und wird durch das Interface *ITraceComponent* definiert (siehe Abbildung 12).

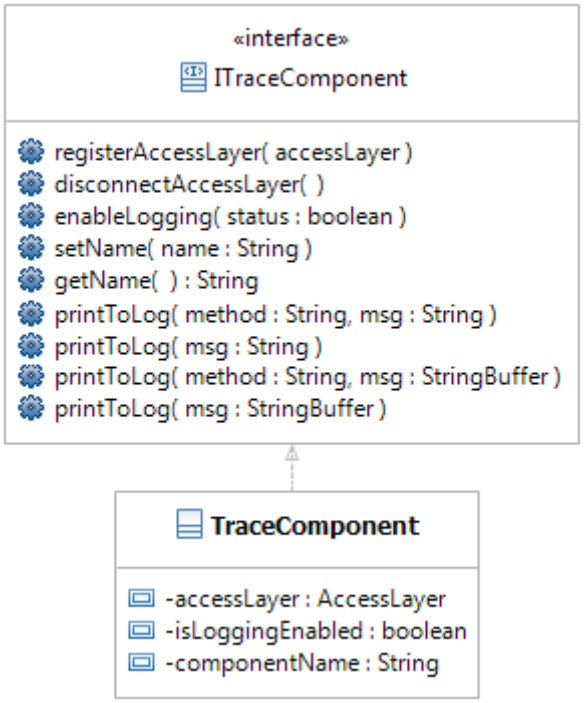


Abbildung 12: Klassendiagramm der TraceComponent

Das Interface der Komponente stellt dabei Funktionen zum registrieren einer AccessLayer-Komponente bereit, durch die anschließend auf die in EMFStore gelagerten Modelle zugegriffen werden kann. Dazu wird in dem Attribut *accessLayer* eine Referenz auf den tatsächlich verwendeten AccessLayer gespeichert.

Desweiteren verfügt die Basiskomponente über Log-Funktionen, die das Verhalten der Komponente auf Wunsch des Anwenders protokollieren können.

Neue Komponenten, wie etwa der LinkManager, müssen diese Basisfunktionen dabei nicht einmal überladen oder neu implementieren. Es muss lediglich vom Interface *ITraceComponent* geerbt und die neue Funktionalität deklariert werden, da das Registrieren des AccessLayers, sowie die Log-Funktionen unverändert genutzt werden können. Jede Komponente sollte dabei auch mit einem einzigartigen Namen versehen werden, um die Log-Einträge später ihrem Verursacher zuordnen zu können. Das Log, welches sich entweder über die Eclipse-Konsole, oder über eine Textdatei realisieren lässt, hat dabei folgenden Aufbau:

Log-Eintrag ::= <Zeitangabe> <Komponenten-Name> [<Methodenname>] <Text>

Die Paketaufteilung der TraceComponent wird dabei wie in Abbildung 13 dargestellt realisiert. Für das Interface, die Implementierung und die Testfälle wird dabei jeweils ein eigenes Eclipse Package verwendet und dieses Schema auch auf alle folgenden Komponenten angewendet.

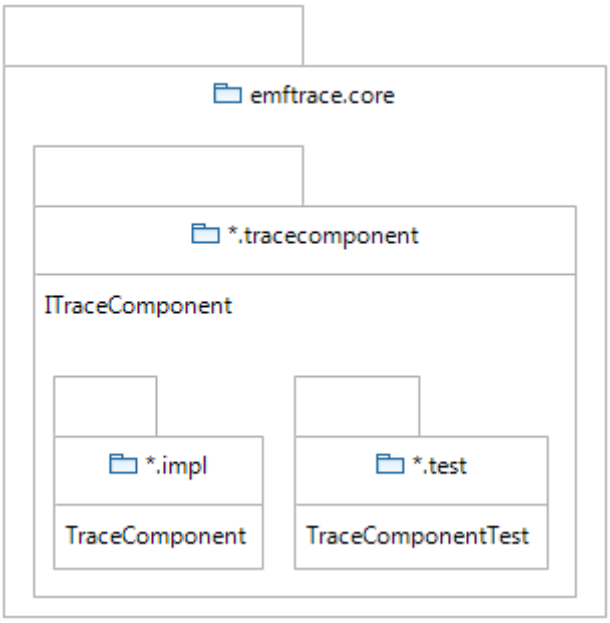


Abbildung 13: Paketaufteilung der TraceComponent

4.5.1. Entwurf der AccessLayer-Komponente

Wie aus Abbildung 14 zu entnehmen ist, wird für die Realisierung der Zugriffsschicht das Interface *IAccessLayer* definiert, welches durch die Klasse *AccessLayer* implementiert wird. Diese Komponente ermöglicht den gekapselten Zugriff auf die im Repository gespeicherten Daten (siehe Kapitel 4.2.). Desweiteren stellt sie eine Reihe von Zugriffsfunktionen bereit, die weit über die von EMFStore bereitgestellten Funktionen hinaus gehen. Dabei werden folgende Basisfunktionen des Repository durch die Zugriffsschicht gekapselt:

addElement – fügt ein neues Modell in ein Projekt ein
removeElement – löscht ein Modell aus einem Projekt
getElement – sucht ein Modell im Projekt anhand seiner ID
getElements – sucht alle Modelle einer bestimmten Klasse im Projekt
commitProject – überträgt lokale Änderungen des Projektes an den Server

Desweiteren bietet die Zugriffsschicht folgende Methoden an, um gezielt Elemente und Daten von Modellen anzusprechen:

getAttribute – sucht im Modell nach einem Attribut mit vorgegebenen Namen
getAttributes – gibt alle Attribute eines Modells zurück

getAttributeValue – gibt den Wert eines Attributes als String zurück

getAllChildren – gibt eine Liste aller Kindelemente (auch indirekte Kinder) des Modells zurück

getAllDirectChildren – gibt eine Liste aller direkten Kindelemente des Modells zurück

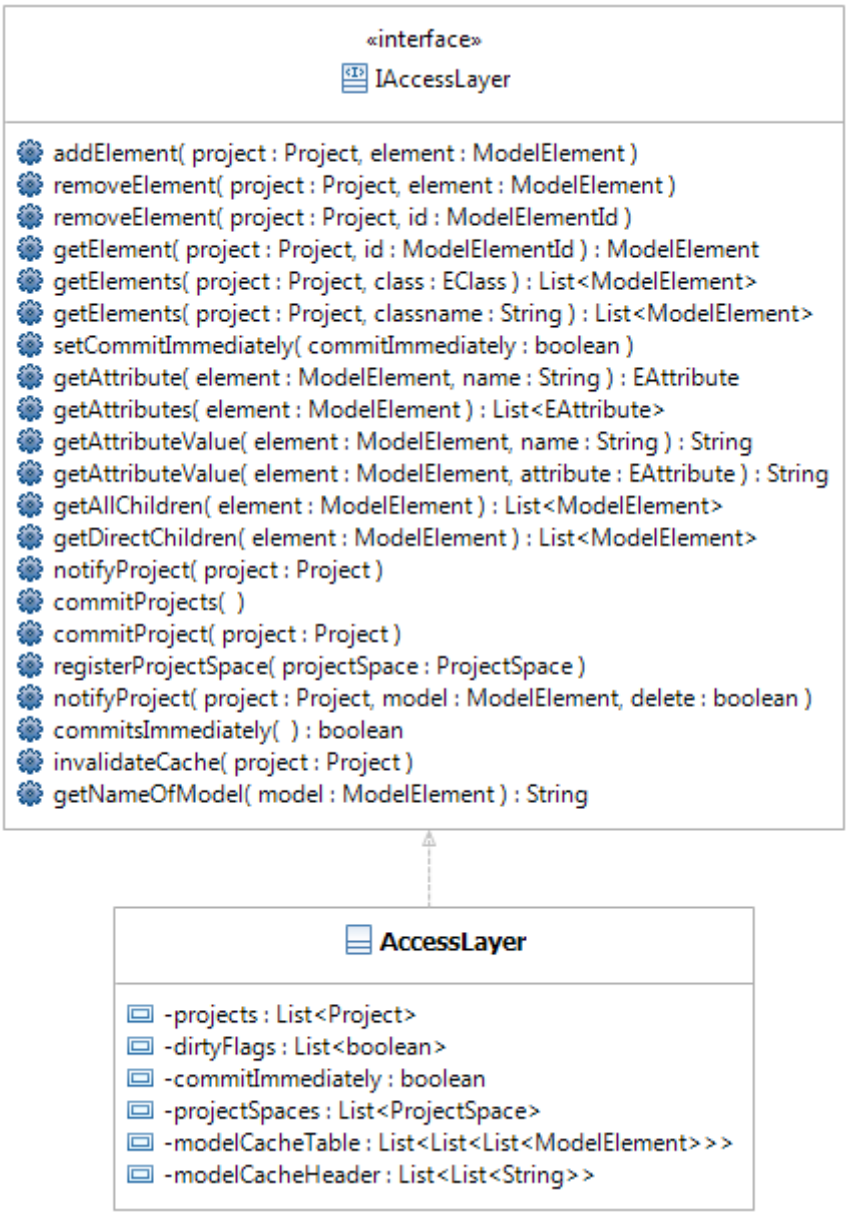


Abbildung 14: Klassendiagramm des AccessLayer

Neben diesen stark modellgebundenen Funktionen kommt der Zugriffsschicht eine weitere, wichtige Aufgabe zu, die Steuerung der Änderungsübertragungen an den Projektserver („commit“).

Über den AccessLayer kann dabei eingestellt werden, wann eine Übertragung erfolgen soll, d.h. ob jede Änderung sofort an den Server übermittelt werden soll, oder ob Änderungen gebündelt und erst zu einem bestimmten Zeitpunkt (z.B. vor Beenden des Programms) übertragen werden sollen.

Für die Aktualisierung und Übertragung von Änderungen an den Projektserver führt der AccessLayer zudem eine Liste aller Projekte, deren Daten manipuliert wurden bzw. von denen Modelle und Daten abgefragt wurden. Für jedes Projekt existiert ein zusätzliches *dirty*-Flag welches auf *true* gesetzt wird, sobald eine schreibende Operation auf dem Projekt ausgeführt wurde. Im Falle einer Übertragung an den Server werden nur die Projekte übertragen, deren *dirty*-Flag gesetzt ist, um leere Übertragungen zu vermeiden.

Im Regelfall ist das gebündelte Übertragen von Änderungen einer sofortigen Ausführung vorzuziehen, da das Aktualisieren eines Projektes auf dem Server eine recht teure Operation ist, deren Ausführung selbst bei kleinen Datenmengen und mit lokal verwendetem Server schon im zweistelligen ms-Bereich liegt. So würde das sofortige Aktualisieren im Falle einer Regelauswertung die Bearbeitungsdauer der Regelverarbeitung enorm verlängern, da nach jedem neu erzeugten Traceability-Link eine Übertragung an den Server erfolgen würde. Ein weiterer unangenehmer Nebeneffekt der sofortigen Übertragung wäre die regelrechte „Explosion“ der Versionsnummern eines Projektes, da diese durch jede Veränderung inkrementiert wird.

Um die spätere Abarbeitung der Regeln optimal durch die Zugriffsschicht zu unterstützen, wird der AccessLayer zudem mit einem Cache für Modelle ausgestattet. Wie in Kapitel 4.4.2. dargestellt wurde, besteht jede Regel aus einem Anfrageteil, in dem Modelle aus dem Repository extrahiert werden, die einer vorgegebenen Modellklasse entsprechen und auf die anschließend die Selektionsbedingungen der Regel angewandt werden.
Um die benötigten Elemente aus dem Repository zu extrahieren, müssen alle Elemente im Repository durchlaufen und auf ihre Klasse untersucht werden. D.h. für jede Regel die nach Elementen vom Typ „UseCase“ sucht, müsste das Repository jedes Mal erneut durchsucht werden.
An dieser Stelle kann ein Cache genutzt werden, um unnötige Berechnungen zu vermeiden. Sobald eine Anfrage ausgeführt wurde, werden die gefundenen Modelle automatisch im Cache an entsprechender Stelle registriert. Bei einer erneuten Anfrage wird somit zuerst der Cache befragt, ob er Modelle des entsprechenden Typs gespeichert hat. Ist dies nicht der Fall, wird normal im Repository gesucht und die Werte anschließend wieder in den Cache übertragen. Der AccessLayer führt deshalb 2 Listen, die den Cache realisieren:

modelCacheHeader – eine Liste aller im Cache gespeicherten Modellklassen
modelCacheTable – eine Liste, die jeweils Listen von Modellen einer bestimmten Klasse speichert

Damit die Konsistenz des Cache jederzeit gewährleistet werden kann, werden durch die Add- und Remove-Operationen auch die gecachten Einträge gelöscht bzw. ergänzt. Eine detaillierte Analyse, sowie eine experimentelle Untersuchung der Laufzeit- und Speicherplatzkomplexität des Cache findet sich dabei in Kapitel 5.3.1.

Die Paketstruktur der AccessLayer-Komponente folgt dabei wieder der der TraceComponent und gliedert sich in die in Abbildung 15 dargestellten Pakete.

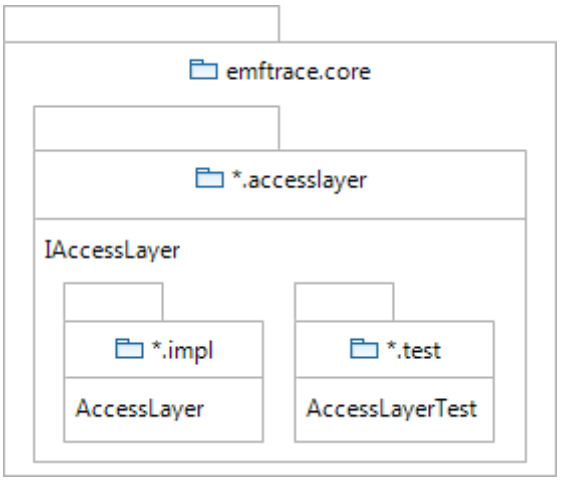


Abbildung 15: Paketaufteilung des AccessLayer

4.5.2. Entwurf der LinkManager-Komponente

Mit dem LinkManager soll nun die nächste, wichtige Komponente von EMFTrace vorgestellt werden. Da der LinkManager mithilfe der AccessLayer-Komponente auf die Modelle des EMFStore zugreifen können muss, erbt das Interface des LinkManagers, *ILinkManager*, direkt vom Interface *ITraceComponent*. Die Klasse *LinkManager* implementiert anschließend das Interface *ILinkManager* und erbt gleichzeitig alle implementierten Methoden der Klasse *TraceComponent*, wie aus Abbildung 16 zu entnehmen ist.

Die Hauptaufgaben der LinkManager-Komponente lassen sich dabei wie folgt charakterisieren:

1. Erzeugen und Löschen von Traceability-Links (binär und n-är)
2. Erzeugen und Löschen von Trace-Instanzen
3. Validierung von Traceability-Links
4. Validierung von Trace-Instanzen
5. Transitive Abhängigkeitsanalysen zwischen Traceability-Links

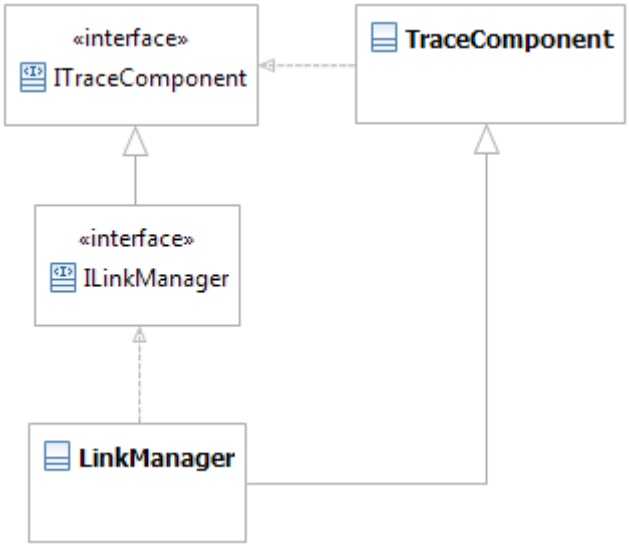


Abbildung 16: vereinfachtes Klassendiagramm des LinkManagers

Im Detail verfügt der LinkManager dabei über die in Abbildung 17 ersichtlichen Methoden.

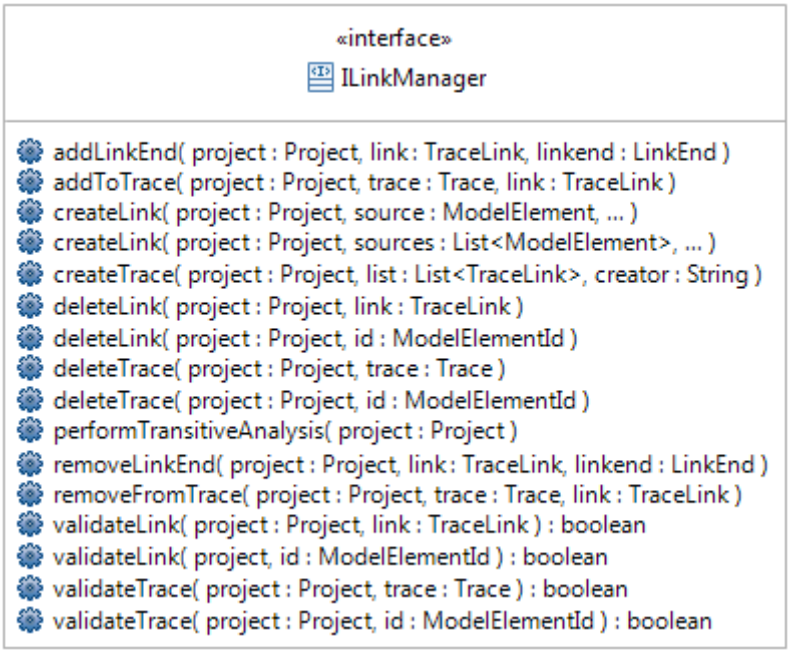


Abbildung 17: Methoden des LinkManagers

Von besonderer Bedeutung sind dabei die Methoden zur Validierung von Traceability-Links (*validateLink*) und Trace-Instanzen (*validateTrace*). Da die Modelle im Repository ständigen Änderungen unterliegen, müssen auch die Traceability-Links, die auf diese Modelle verweisen angepasst werden. Wird zum Beispiel ein Modell gelöscht, auf das vorher ein LinkEnd verwiesen hat, muss der dazugehörige Traceability-Link unter Umständen gelöscht werden, wenn keine Quell- oder Zielknoten mehr vorhanden sind.

Trace-Instanzen müssen zudem auch validiert werden, wenn sie zuvor vom LinkManager bei der transitiven Analyse von Traceability-Links generiert wurden. Sollte ein Traceability-Link bei der Validierung gelöscht werden, kann auch die transitive Relation, die durch einen Trace ausgedrückt wird verloren gehen. Durch das Wegfallen eines Traceability-Links aus der transitiven Ketten von Links in einem Trace kann zudem die Situation entstehen, in der der Trace in 2 oder mehr eigenständige Trace-Elemente aufgespalten werden muss. Jedes Teilelement repräsentiert dabei einen Teil der ehemaligen transitiven Linkkette. Das nachfolgende Beispiel soll eben genannte Situation verdeutlichen.

Sei T ein Trace, bestehend aus den 6 Traceability-Links {L1, L2, L3, L4, L5, L6} und bezeichne „A → B“ die transitive Verbindung zwischen TraceLink A und B und es gilt:
L1 → L2 → L3 → L4 → L5 → L6

Wird nun TraceLink „L3“ gelöscht, ist die transitive Kette in Trace T nicht mehr gegeben, da sie in 2 Teilketten zerfallen ist (L1 → L2 und L4 → L5 → L6). Folglich kann nur eine der Teilketten in Trace T verbleiben und die andere Kette muss in einen separaten Trace T2 ausgelagert werden:
T: L1 → L2
T2: L4 → L5 → L6

Der Paketaufteilung der TraceComponent folgend, stellt sich auch die Aufteilung der Bestandteile des LinkManagers durch das in Abbildung 18 skizzierte Paketdiagramm dar.

Abbildung 18: Paketaufteilung des LinkManagers

4.5.3. Entwurf der RuleEngine-Komponente

Der Interpreter für die Traceability-Regeln (RuleEngine) besteht aus insgesamt 4 einzelnen Komponenten und erbt seine Basisfunktionalität von der Klasse *TraceComponent*, die wieder für den gekapselten Zugriff auf die AccessLayer-Komponente zuständig ist und somit auch den Zugriff auf die in EMFStore gespeicherten Modelle bietet. Jede der 4 Teilkomponenten des Interpreters ist dabei für genau einen Teil der Regelverarbeitung zuständig, angefangen bei der Syntaxprüfung bis hin zur Ergebnisauswertung nach erfolgter Regelabarbeitung in EMFTrace. Durch diese klare Trennung der Komponenten nach ihrer Zuständigkeit, sind ein späterer Austausch oder Ergänzungen wesentlich einfacher zu realisieren, als in einer monolithisch aufgebauten Komponente. Die Funktionalität wird dabei auf die folgenden 4 Teilkomponenten verlagert:

ElementProcessor:
Beschafft die in den Regeln definierten Elemente aus EMFStore und ordnet sie ihrem Alias-Namen zu. Dabei entsteht pro Element genau eine Liste von Modellelementen, die die potenziellen Kandidaten einer bestimmten Klasse enthält.

RuleProcessor:
Arbeitet die in den Regeln definierten Bedingungen ab und entfernt alle Elemente aus den Ergebnislisten, die die Bedingungen verletzen.

ResultProcessor:
Verknüpft die verbliebenen Elemente der Ergebnislisten gemäß den Vorschriften aus den Traceability-Regeln.

RuleValidator:
Prüft Regeln auf syntaktische Korrektheit und gibt Warnungen und Hinweise über die Eclipse Konsole aus.

Die Komponenten ElementProcessor, RuleProcessor und ResultProcessor werden von dem Interface *IProcessingComponent* abgeleitet (siehe Abbildung 19), welches eine abstrakte *run*-Methode definiert, die von den einzelnen Komponenten überladen werden muss. Die RuleValidator-Komponente hingegen wird direkt vom Interface *ITraceComponent* abgeleitet, da sie keine gemeinsamen Schnittstellen mit den anderen 3 Komponenten der RuleEngine besitzt und dieser Zwischenschritt somit nicht notwendig ist und unnötige Abhängigkeiten vermieden werden.

Der eigentliche Regelinterpreter wird durch das Interface *IRuleEngine* definiert und durch die Klasse *RuleEngine* implementiert und beinhaltet jeweils eine Instanz der 4 oben vorgestellten Verarbeitungskomponenten. Die RuleEngine ist für die Erzeugung, Initialisierung und Aktualisierung aller Teilkomponenten verantwortlich, etwa wenn eine Instanz des AccessLayers oder LinkManagers an den Komponenten registriert werden soll. Der Klassenaufbau des Regelinterpreters wird dabei in Abbildung 19 verdeutlicht.

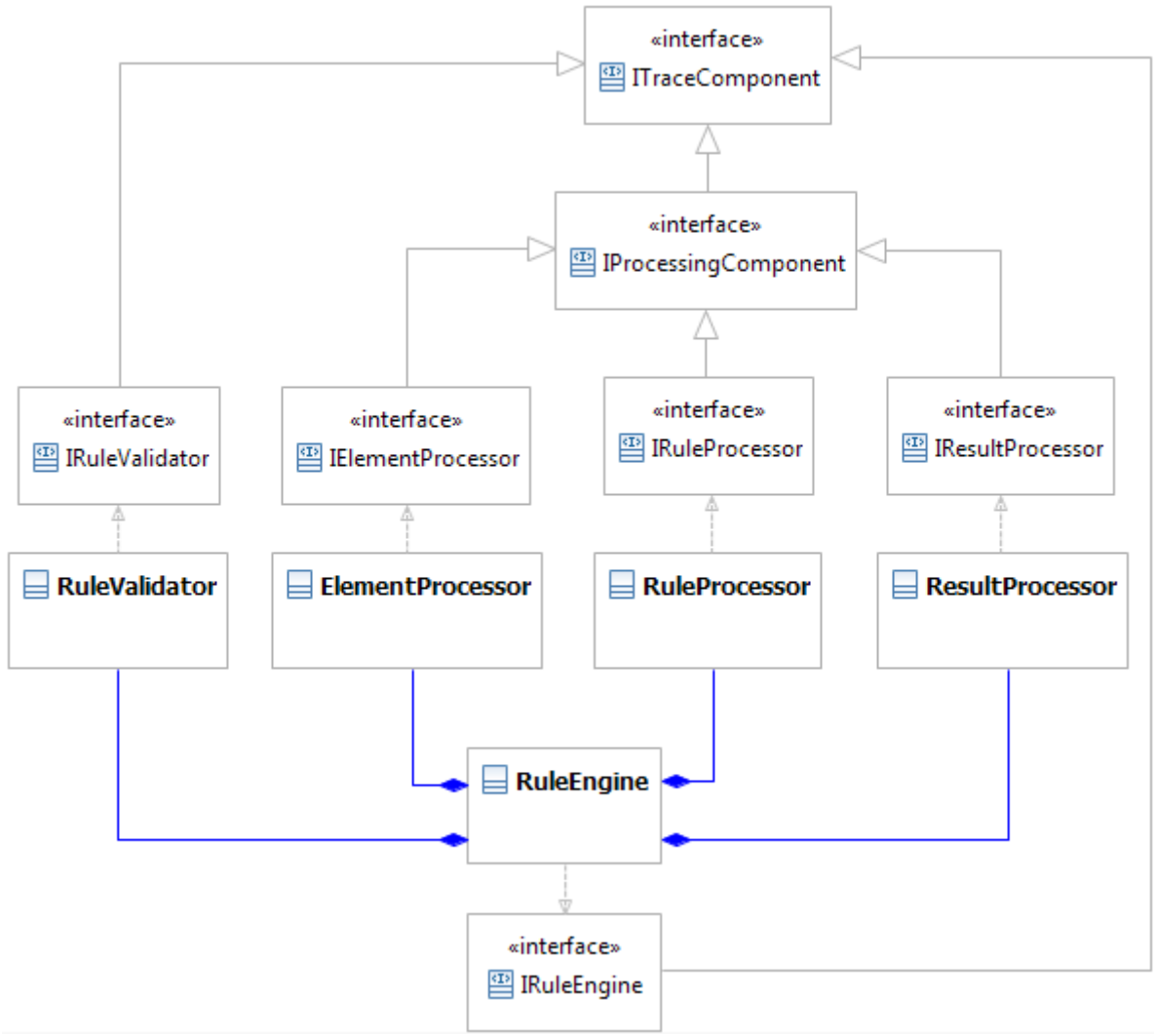


Abbildung 19: interner Klassenaufbau des Regelinterpreters

Für die Abarbeitung von Regeln, egal ob Traceability- oder Konsistenzregeln, stellt die RuleEngine 2 Schnittstellen bereit, über die die Verarbeitung gestartet werden kann. Die beiden Versionen der Methode *applyRules* (siehe Abbildung 20) erlauben dabei die Spezifizierung einer Reihe von Parametern, um eine möglichst flexible Regelverarbeitung zu gewährleisten. Mittels der Liste *elements* kann ausgewählt werden, ob alle Elemente im Projekt für die Regelverarbeitung genutzt

werden sollen, oder nur die in der Liste enthaltenen Modelle. Ähnliches gilt für die Liste *rules,* mit deren Hilfe sich die Regeln angeben lassen, die verarbeitet werden sollen. Alternativ dazu kann auch ein Regelkatalog (*catalogue*) übergeben werden, dessen darin enthaltene Regeln anschließend abgearbeitet werden.

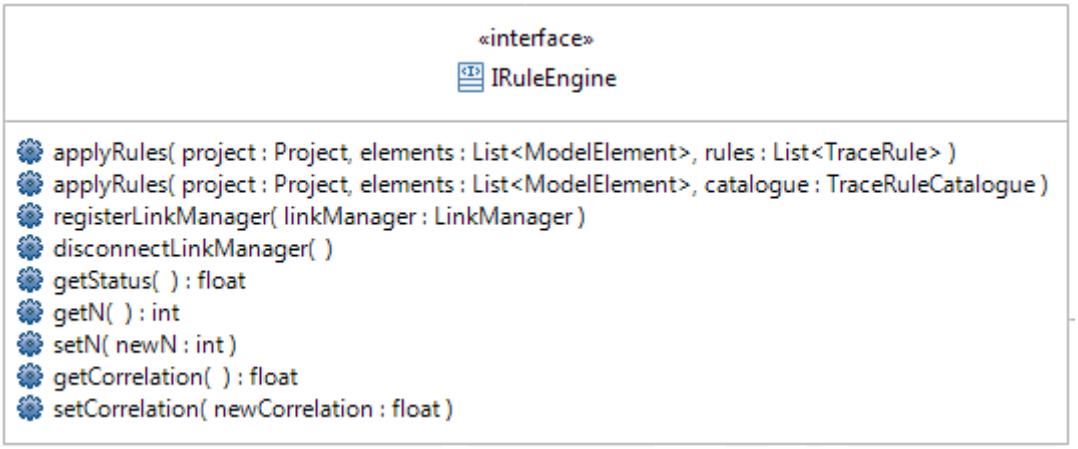


Abbildung 20: Schnittstellen des Regelinterpreters

Somit ist sichergestellt, dass der Benutzer genau festlegen kann, welche Regeln auf welche Modelle angewandt werden sollen. Sofern die RuleEngine aktiv ist (d.h. sie arbeitet Regeln ab) lässt sich mittels der Methode *getStatus* der aktuelle Verarbeitungsfortschritt in Prozent ausgeben.

Da der Regelinterpreter zudem über einen Algorithmus aus dem Bereich Information Retrieval verfügt, bietet er 2 Schnittstellen um die Parameter dieses Algorithmus anzupassen (*setN* und *setCorrelation*). Für mehr Informationen zu diesem Algorithmus und dessen Parametern sei an dieser Stelle jedoch auf Kapitel 5.3.3. („Ausgewählte Aspekte der Implementierung“) verwiesen, in dem dieser Algorithmus explizit erläutert wird.

In Abbildung 21 ist abschließend noch die Paketstruktur des kompletten Regelinterpreters, inklusive aller Teilkomponenten und Testpakete dargestellt, wie sie auch im Eclipse Plug-in EMFTrace_Core durch Java Packages umgesetzt wird.

In den folgenden 4 Teilkapiteln sollen nun die einzelnen Komponenten der RuleEngine näher erläutert, sowie ihre Funktion und Aufbau genauer beschrieben werden.

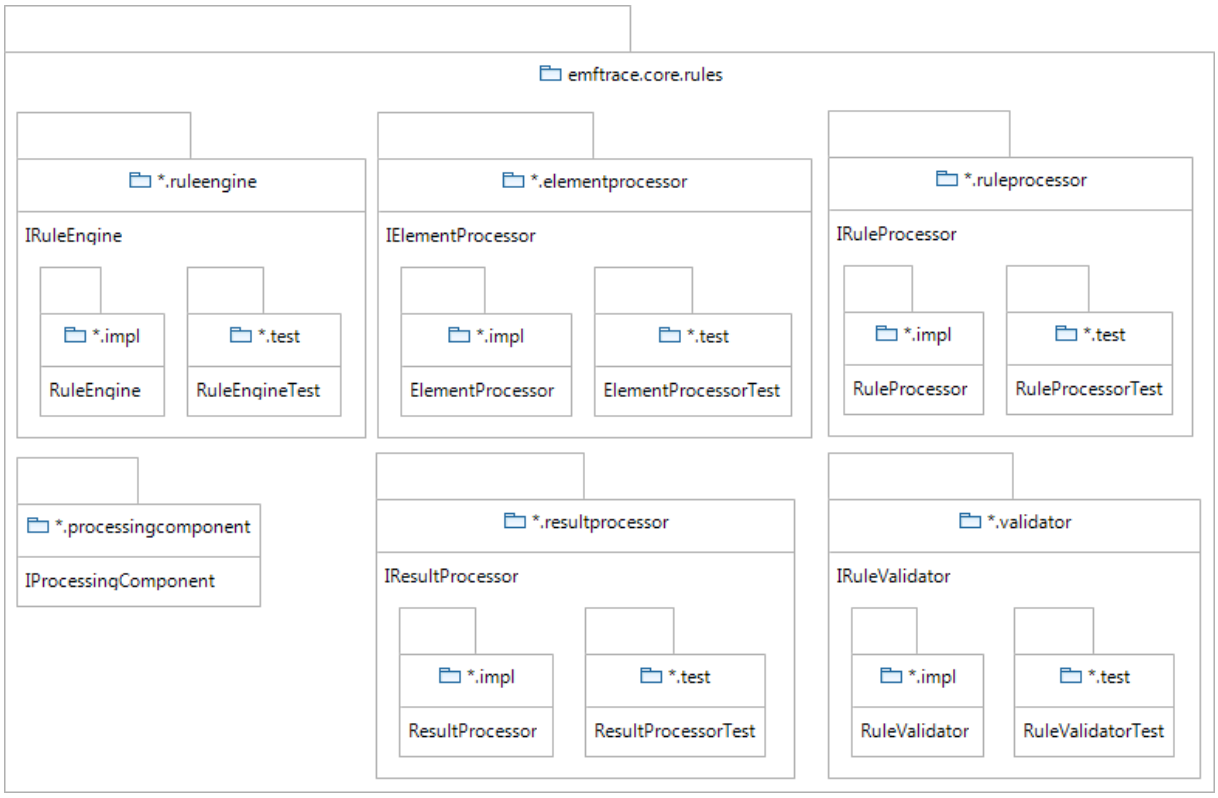


Abbildung 21: Paketaufteilung des Regelinterpreters

4.5.3.1. ElementProcessor-Komponente

Bevor der Entwurf der eigentlichen ElementProcessor-Komponente erfolgen kann, muss zuvor eine Hilfsdatenstruktur für die Anfrageelemente eingeführt werden, die Klasse *QueryElement* (siehe Abbildung 22). Mit der Einführung dieser Klasse werden vor allem zwei Ziele verfolgt:

1. Kapselung des Unicase-ModelElements um spätere Änderungen zu erleichtern.
2. Speicherung von Referenzen auf Modelle, die in Verbindung mit dem eigentlichen ModelElement den Bedingungen der Regeln genügt haben, um die korrekte Verknüpfung der Ergebnisse nach der Regelabarbeitung zu gewährleisten.

Ein kurzes Beispiel soll hierbei das 2. Ziel verdeutlichen:
Gegeben seien 2 Listen von QueryElements, sowie eine Regel, die die Namen der Listenelemente auf Gleichheit überprüfen soll. Sei nun *e1* ein Element der 1. Liste und *e2* ein Element der 2. Liste.
Wenn *e1* und *e2* der Bedingung genügen (d.h. ihre Namen gleich sind), wird *e2* in die *relatedElements*-Liste von *e1* aufgenommen. Wenn nun nach erfolgter Regelabarbeitung die Auswertung der Ergebnisse vorgenommen wird, werden *e1* und *e2* verknüpft, da *e2* in der Liste von

e1 vermerkt wurde. Auf diese Weise wird vor allem verhindert, dass nicht zusammengehörige Elemente verknüpft werden, da man nicht einfach eine Kreuzproduktrelation auf die Ergebnislisten anwenden kann.

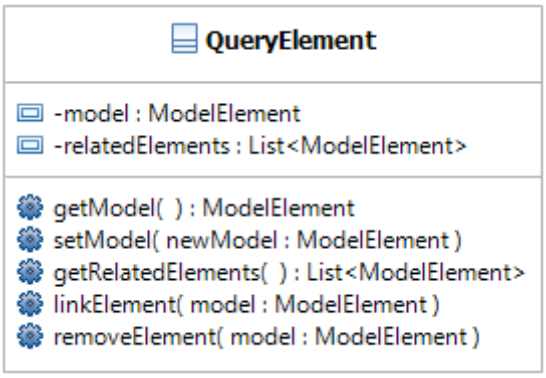


Abbildung 22: Definition der Hilfsstruktur „QueryElement“

Die ElementProcessor-Komponente bietet nun 2 Methoden (*retrieveElements*), um die Listen der möglichen Kandidaten für die Regelabarbeitung zu füllen, wie aus Abbildung 23 ersichtlich ist.
Variante 1 füllt die Liste mit allen Modellen, deren Klassename dem Methodenparameter *name* genügt. Variante 2 wandelt die in der Liste *models* referenzierte Modelle in gültige Instanzen der Klasse *QueryElement* um und fügt diese anschließend in die Ergebnisliste ein.

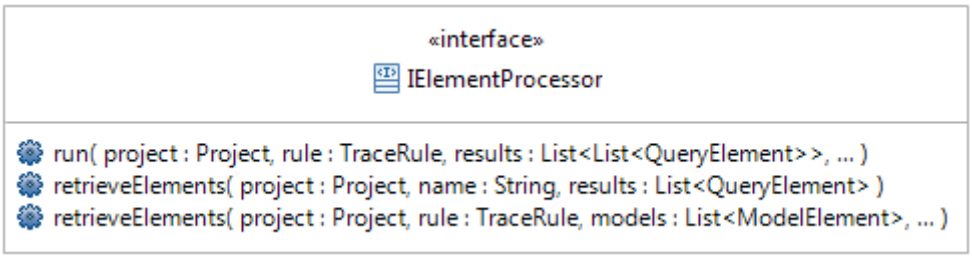


Abbildung 23: Schnittstellen der ElementProcessor-Komponente

Nach Ausführung der *run*-Methode wurden alle für die Regelverarbeitung in Frage kommenden Modelle aus dem Repository extrahiert und die eigentliche Verarbeitung der Regeln kann beginnen. Durch die Kapselung dieses Schrittes in der ElementProcessor-Komponente ist eine spätere Erweiterung um zusätzliche Vorbereitungsschritte vor der eigentlichen Regelabarbeitung ohne Änderungen an anderen Komponenten möglich.

Die Paketaufteilung der ElementProcessor-Komponente kann dem Paketdiagramm der RuleEngine in Abbildung 21 entnommen werden.

4.5.3.2. Die RuleProcessor-Komponente

Die RuleProcessor-Komponente, die für die eigentliche Abarbeitung der Regeln zuständig ist, besitzt für jeden Bedingungstyp, d.h. *BaseCondition* und *LogicCondition* (siehe Kapitel 4.4.2.), eine eigene Verarbeitungsfunktion, deren Abarbeitung jeweils über die von der *IProcessingComponent* geerbten *run*-Methode gesteuert wird (siehe Kapitel 4.5.3.).
In Abbildung 24 ist die RuleProcessor-Komponente mit all ihren Methoden dargestellt. Während die Methode *executeLogicCondition* lediglich die geschachtelten Ergebnisse und Aufrufe von Unteranfragen verwaltet, realisiert die Methode *executeBaseCondition* die tatsächliche Verarbeitung der Regeln. In dieser Methode wird je nach Typ der Regel die Methode *executeCompareCondition* (für EQUALS, SIMILAR_TO, LESSER_THAN, GREATER_THAN sowie NOT_NULL) oder *isParent* (für IS_PARENT) mit den entsprechenden Ergebnislisten und Eingabeparametern aufgerufen.

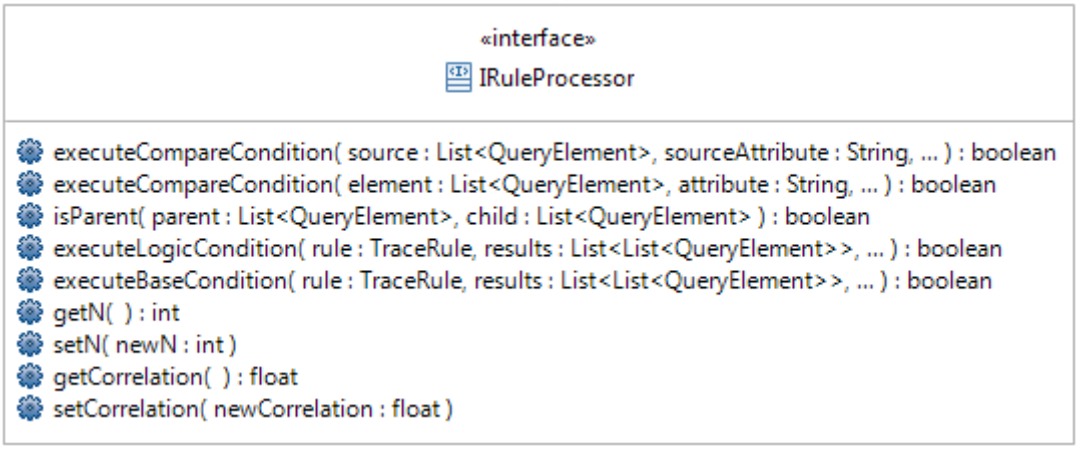


Abbildung 24: Schnittstellen der RuleProcessor-Komponente

Sobald die *run*-Methode die Abarbeitung der letzten Bedingung abgeschlossen hat, befinden sich nur noch die Elemente in den Ergebnislisten, die den Bedingungen tatsächlich genügen. In Analogie zur Anfrageverarbeitung in Datenbanksystemen werden hierbei Elemente schrittweise durch Selektionsbedingungen herausgefiltert, die zuvor in den Regeln definiert wurden.

Für detaillierte Informationen zur Verarbeitung von Regeln durch diese Komponente sei an dieser Stelle jedoch auf das Kapitel 5.3.2. „Regelverarbeitung der RuleEngine“ verwiesen, in dem die wichtigsten Verarbeitungsschritte umfassend erläutert werden.

Die Paketaufteilung der RuleProcessor-Komponente kann dem Paketdiagramm der RuleEngine in Abbildung 21 entnommen werden.

4.5.3.3. Die ResultProcessor-Komponente

Mit der ResultProcessor-Komponente, deren interner Aufbau in Abbildung 25 dargestellt ist, findet die Regelverarbeitung durch die RuleEngine ihren Abschluss. In der von dem Interface *IProcessingComponent* geerbten *run*-Methode werden die Ergebnisse entsprechend der in der Regel definierten Aktion verarbeitet, wobei momentan nur das Erzeugen von Traceability-Links durch die Komponente unterstützt wird und weitere Aktionen, wie etwa die Ausgabe von Konsistenzinformationen noch realisiert werden müssen. Bevor diese Aktionen jedoch ausgeführt werden, wird mittels der Methode *adjustUnrelatedResults* noch überprüft, ob durch die Regeldefinition die Erzeugung eines Kreuzproduktes zwischen allen Ergebnislisten implizit gefordert wird. Dies ist immer genau dann der Fall, wenn alle Elementarbedingungen (*BaseCondition*) jeweils nur Elemente einer einzigen Ergebnisliste umfassen. Hierzu ein einfaches Beispiel, was diesen Fall verdeutlicht:

Gegeben seien eine Regel R mit 2 Bedingungen (B1, B2) und 2 Elementdefinitionen (E1, E2).
Für jede Elementdefinition wird durch die ElementProcessor-Komponente zunächst eine Ergebnisliste erzeugt (L1, L2). Seien die Bedingungen nun vereinfacht wie folgt definiert:
B1 := E1::*someValue* GREATER_THAN *3*
B2 := E2::*anotherValue* EQUALS *5*

Lautet die Aktion nun (vereinfacht dargestellt) “createLink from E1 to E2“, muss das Kreuzprodukt der Listen L1 und L2 bestimmt werden, da durch die Bedingungen B1 und B2 keine Abhängigkeiten zwischen den Ergebnislisten hergestellt werden können und somit die Kreuzproduktrelation implizit gefordert wird.

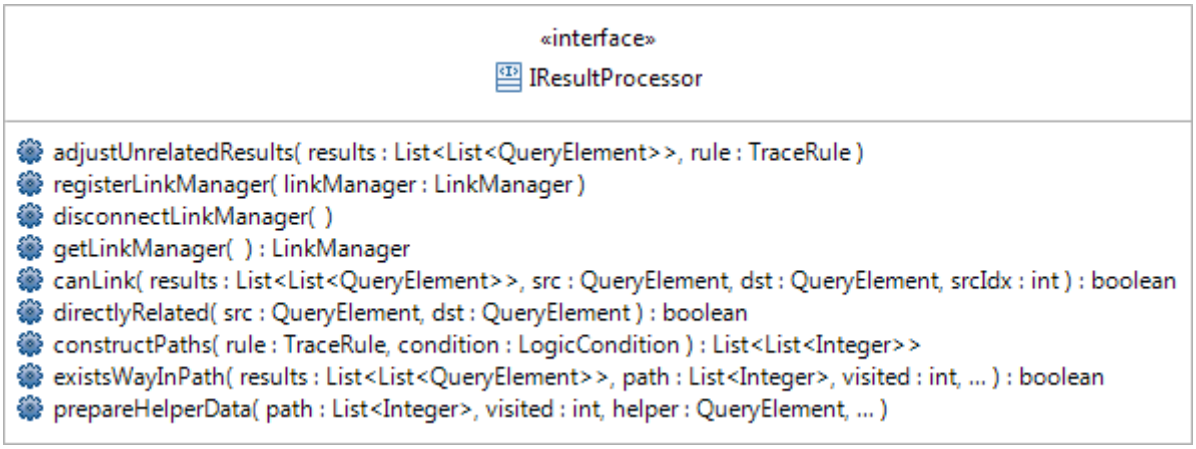


Abbildung 25: Schnittstellen der ResultProcessor-Komponente

Sollen 2 Elemente verlinkt werden, wird zuerst mittels der Methode *canLink* überprüft, ob die Verlinkung erlaubt ist. Dabei wird zunächst nach einer direkten Abhängigkeit zwischen den Elementen gesucht (*directlyRelated*). Falls dabei kein Zusammenhang zwischen Ziel und Quelle erkannt wurde, werden alle durch die Regel bedingten, transitiven Verknüpfungsketten mittels *constructPaths* erstellt und durch *existsWayInPath* überprüft. Sollte auch dabei kein Zusammenhang gefunden werden, können die beiden Elemente nicht miteinander verbunden werden.

Die Paketaufteilung der ResultProcessor-Komponente kann dem Paketdiagramm der RuleEngine in Abbildung 21 entnommen werden.

4.5.3.4. RuleValidator-Komponente

Für die Validierung bzw. syntaktische Überprüfung von Regeln bietet das Interface der RuleValidator-Komponente, dessen Aufbau in Abbildung 26 dargestellt ist, die Methode *validateTraceRule* an. Intern werden dabei 4 Validierungsfunktionen ausgeführt, um alle Bestandteile einer Regel (siehe Kapitel 4.4.) zu prüfen. Im ersten Schritt werden alle in der Regel definierten Elemente durch die Methode *checkElementDefinition* analysiert. Ist dieser Schritt abgeschlossen, werden alle Bedingungen überprüft, je nach Bedingungstyp entweder durch die Methode *checkBaseCondition* (EQUALS, CONTAINS, SIMILAR_TO, IS_PARENT, LESSER_THAN, GREATER_THAN, NOT_NULL) oder durch die Methode *checkLogicCondition* (AND, OR, NOT, XOR). Abschließend werden die einzelnen Aktionen im Schlussteil der Regel mit Hilfe der Methode *checkActionDefinition* validiert.

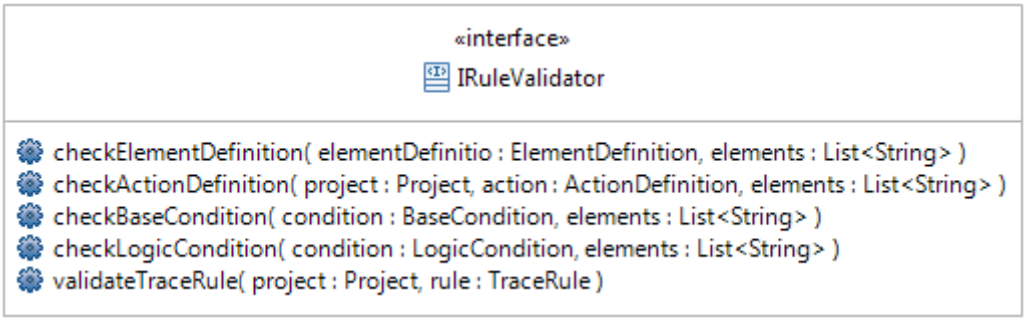


Abbildung 26: Schnittstellen der RuleValidator-Komponente

Sofern die Validierung einer Regel erfolgreich abgeschlossen wurde, kann die eigentliche Ausführung der Regel durch die RuleEngine erfolgen. Sollten bei der Validierung einer Regel Probleme erkannt werden, so wird diese Regel von der weiteren Abarbeitung ausgeschlossen. Zusätzlich werden die erkannten Probleme über das Log an den Anwender weitergeleitet, da das Interface *IRuleValidator*

vom Interface *ITraceComponent* erbt und somit auch Zugriff auf die Logging-Funktionen der TraceComponent hat (siehe Kapitel 4.5. und Abbildung 19).

Die Paketaufteilung der *RuleValidator*-Komponente kann dem Paketdiagramm der *RuleEngine* in Abbildung 21 entnommen werden.

4.6. Entwurf der EMFTrace-Benutzerschnittstelle

Da sich die EMFTrace-Benutzerschnittstelle möglichst in die Benutzeroberfläche des EMFStore integrieren lassen sollte, muss zunächst ein geeigneter Weg für die Realisierung der Benutzerschnittstelle gefunden werden. Dabei ist zu beachten, dass die neue Funktionalität möglichst konform zu der vorhandenen Funktionalität des EMFStore eingebettet wird, um dem Benutzer eine möglichst einheitliche Bedienung des Gesamtsystems zu ermöglichen. Daneben steht auch die Frage nach dem Grad der Abhängigkeit zu Code aus dem EMFStore im Raum. Da sich sowohl EMFStore, als auch EMFTrace noch in der Entwicklung befinden, sind häufige Änderungen nicht auszuschließend und die Benutzeroberfläche sollte möglichst unabhängig von EMFStore gestaltet werden. Für die Realisierung der Benutzeroberfläche stehen dabei die folgenden Optionen zur Auswahl:

1. Schaffung eines neuen Client, der die EMFTrace-Benutzerschnittstelle enthält
2. Einbindung neuer Perspektiven und Views in den vorhandenen EMFStore Client
3. Einbindung neuer Menüfelder, Dialoge und Assistenten in den vorhandenen EMFStore Client

Variante 1 erfüllt die Anforderung nach einer Unabhängigkeit von EMFStore vollständig, bringt jedoch mehrere, entscheidende Nachteile mit sich. Auf den ersten Blick eher unscheinbar, bedeutet ein zusätzlicher Client, dass der Benutzer allein für die Arbeit mit EMFTrace drei Eclipse Instanzen bzw. Eclipse Produkte ausführen muss. Dies bedeutet vor allem eine recht hohe Auslastung des Benutzersystems und einen erhöhten Aufwand für den Benutzer selber, da er nun drei Anwendungen verwalten bzw. bedienen muss.

Variante 2 bietet den Vorteil, dass der vorhandene EMFStore Client für EMFTrace genutzt werden kann und eine zusätzliche Anwendung somit vermieden wird. Die Einbindung der Views und Perspektiven kann dabei über die standardgemäßen Erweiterungspunkte von Eclipse erfolgen, da auch der EMFStore Client über Erweiterungspunkte der Eclipse RCP definiert ist. Nachteilig wirkt sich jedoch aus, dass der Benutzer dadurch mit zu vielen Views und Perspektiven konfrontiert wird und

dass das häufige Umschalten schnell zu Flüchtigkeitsfehlern führen kann, da neben den Views und Perspektiven des EMFStore und EMFTrace auch Views und Perspektiven der UML2 Tools (siehe Kapitel 2.1.) und des jUCMNav-Features (siehe Kapitel 2.3.) durch den Client genutzt werden können.

Variante 3 bietet mit kontextsensitiven Menüs und Dialogen einen Kompromiss zwischen einfacher Bedienbarkeit, Übersichtlichkeit und Änderungsempfindlichkeit. In Analogie zu Variante 2 werden neue Funktionen der Benutzerschnittstelle über Erweiterungspunkte der Eclipse Plug-in Architektur eingebunden, wobei auch neue Erweiterungspunkte des EMFStore genutzt werden. Es ergibt sich somit zwar eine direkte Abhängigkeit zu den Erweiterungspunkte des EMFStore, allerdings sind Änderungen an Erweiterungspunkten eher die Ausnahme und zudem zum Großteil ohne zusätzlichen Programmieraufwand möglich, da Erweiterungspunkte und deren Nutzung in der Konfigurationsdatei des jeweiligen Eclipse Plug-ins in XML-Form gespeichert werden.

Da ein Großteil der Funktionalität von EMFTrace ohnehin nur kontextspezifisch ausgeführt werden kann (z.B. die Validierung von selektierten Traceability-Links), stellt Variante 3 die beste Alternative für die Umsetzung der Benutzeroberfläche von EMFTrace dar. Für die Einbindung neuer Menüs, Dialoge und Assistenten wird der in Abbildung 27 dargestellte Erweiterungspunkt *org.eclipse.ui.menus* benötigt. In der gleichen Abbildung sind auch alle neuen Menüeinträge ersichtlich, die durch das EMFTrace_GUI Plug-in im EMFStore Client registriert werden.

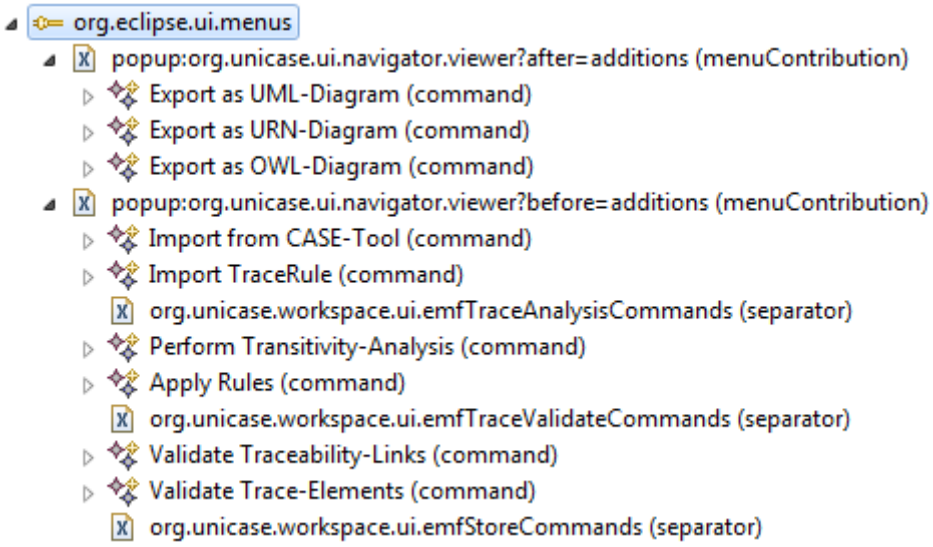


Abbildung 27: Registrierung neuer Menüeinträge

Um auf die neuen Handler-Klassen reagieren zu können, die den Lebenszyklus der Dialoge und Assistenten steuern, werden zudem eine Reihe von Eclipse Commands benötigt. Diese müssen zunächst jedoch, wie in Abbildung 28 dargestellt, an dem Erweiterungspunkt *org.eclipse.ui.commands* registriert werden.

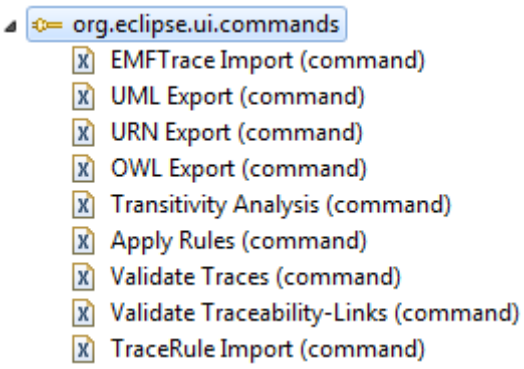


Abbildung 28: Registrierung neuer Commands

4.6.1. Import und Export von Modellen

Da der Export aus EMFTrace kontextspezifisch erfolgen soll, benötigt jedes Metamodell einen separaten Export-Handler und ein dazugehöriges Eclipse Command. Dies bedeutet auf den ersten Blick zwar erhöhten Aufwand, bietet dem Benutzer aber mehr Bedienkomfort und beugt Fehlern vor. Für den Anwender bedeutet das, dass ihm für das aktuell selektierte Modell nur die modellrelevanten Funktionen angezeigt werden, so z.B. die Funktion „Export as UML-Diagram“, wenn ein UML-Klassendiagramm im EMFStore Client ausgewählt wurde. Somit kann der Benutzer nie aus Versehen ein UML-Diagramm als OWL-Ontologie abspeichern (oder auch umgekehrt), da diese Funktion nicht im Kontextmenü verfügbar ist, solange keine Ontologie im Browser ausgewählt wurde.

Für den Import und Export von Modellen gibt es jeweils eine eigene Handler-Basisklasse *EMFTraceImportHandler* bzw. *EMFTraceExportHandler*, welche die grundlegende Funktionalität bereitstellt. Alle spezialisierten Handler-Klassen, wie etwa zum Export von UML-Modellen, können dann von einer der beiden Basisklassen abgeleitet werden. Die Hierarchie der momentan verwendeten Handler-Klassen ist dabei in Abbildung 29 dargestellt, deren oberstes Element die abstrakte Klasse des Eclipse Handlers (*AbstractHandler*) darstellt.

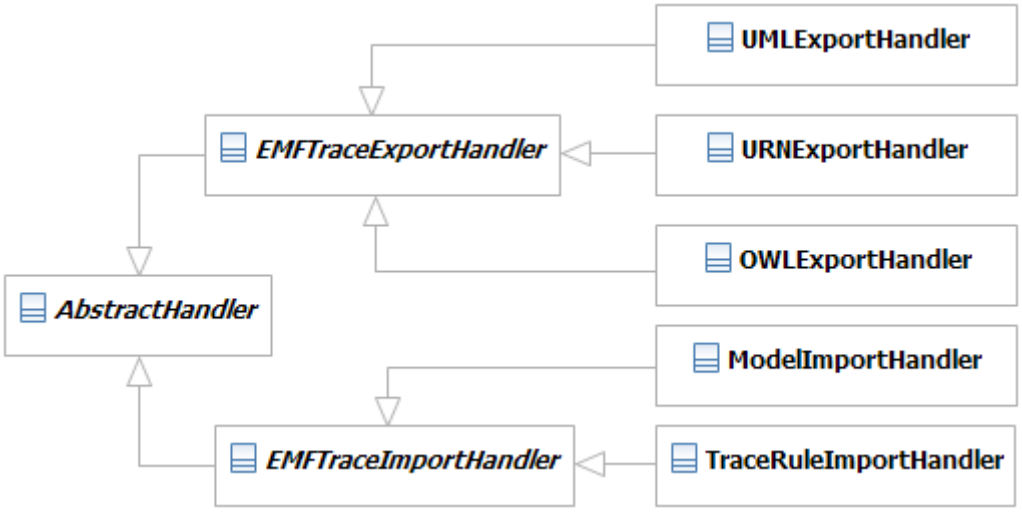


Abbildung 29: EMFTrace Handler-Klassen

Zukünftige Erweiterungen um neue Metamodelle können nun einen eigenen Export-Handler von der Basisklasse *EMFTraceExportHandler* ableiten und müssen dort nur die *execute*-Methode überladen. Um den modellspezifischen Export-Handler nutzen zu können, muss am Erweiterungspunkt *org.eclipse.ui.menus* (siehe Abbildung 27) noch ein entsprechender Eintrag als *menuContribution* angelegt werden, was in Abbildung 30 am Beispiel des UML-Exports demonstriert werden soll.

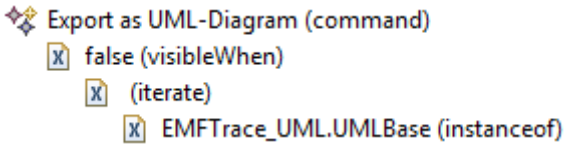


Abbildung 30: menuContribution für den Export von UML-Modellen

Um die eigentliche Funktionalität der Import- und Exportkomponenten von der Eclipse Benutzeroberfläche zu lösen, wurden die eigentlichen Prozeduren in die zwei statischen Klassen *EMFTraceImportHelper* und *EMFTraceExportHelper* ausgelagert, auf die auch unabhängig von der GUI zugegriffen werden kann. Sollen später beispielsweise Importvorgänge mittels Adapter direkt von den CASE-Werkzeugen aus gestartet werden, können dafür die beiden Klassen benutzt werden, ohne in eine Abhängigkeit von den Import- oder Export-Handler-Klassen zu geraten.

4.6.2. Validierung von Traceability-Links und Trace-Elementen

Die Validierung von Traceability-Links und Trace-Elementen wird in EMFTrace global für ein ganzes Projekt ausgeführt. Dabei wird durch die Handler-Instanzen auf die Methoden *validateTrace* und *validateTraceLink* der LinkManager-Komponente zurückgegriffen (siehe Kapitel 4.5.2.).
Die Klassenhierarchie der Validierungs-Handler stellt sich dabei wie in Abbildung 31 skizziert dar.

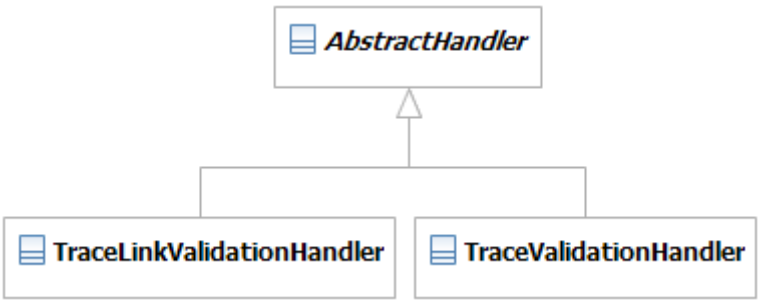


Abbildung 31: EMFTrace Validierungs-Handler

4.6.3. Regelverarbeitung

Für die Abarbeitung von Regeln durch die RuleEngine wird deren Methode *applyRules* genutzt (siehe Abbildung 20), deren Parameter jedoch vor der Ausführung konfiguriert werden müssen. Für den Dialog mit dem Nutzer in dem diese Parameter erhoben werden, kommt ein eigener Assistent (Eclipse Wizard) zum Einsatz, dessen Lebenszyklus von der Handler-Klasse *TraceRuleHandler* verwaltet wird. Wie in Abbildung 32 dargestellt, besteht der RuleApplicationWizard aus insgesamt 4 WizardPages, die den Benutzer schrittweise durch die Konfiguration führen sollen.

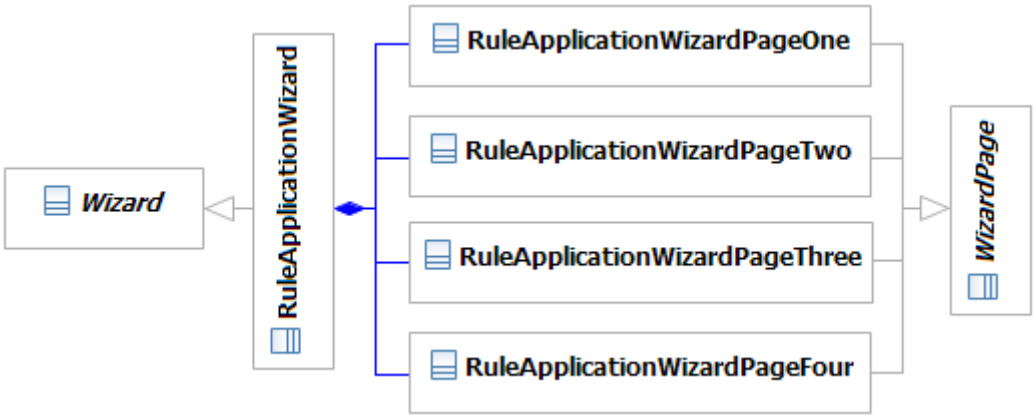


Abbildung 32: RuleApplication Assistent

Der Assistent erhebt die benötigten Daten dabei auf folgende Weise:

Im ersten Schritt (RuleApplicationWizardPageOne) kann ein Regelkatalog ausgewählt werden, dessen Regeln für eine Analyse genutzt werden sollen. Falls im 1. Schritt kein Katalog ausgewählt wurde, muss nun im 2. Schritt (RuleApplicationWizardPageTwo) mindestens eine, im Projekt enthaltene Regel ausgewählt werden. Optional können auch Regeln aus dem zuvor spezifizierten Katalog explizit gewählt werden.
Im dritten Schritt (RuleApplicationWizardPageThree) können zusätzlich die Modelle ausgewählt werden, die für die Regelverarbeitung genutzt werden sollen. Wird dabei kein Modell gewählt, werden alle im Projekt enthaltenen Modelle für die Regelverarbeitung genutzt.
Zum Schluss (RuleApplicationWizardPageFour) können noch die Parameter des bei der Regelauswertung verwendeten N-Gram-Algorithmus eingestellt werden (siehe dazu auch Kapitel 5.3.3.).

Auf der beiliegenden CD befindet sich ein zusätzliches Aktivitätsdiagramm, welches diese 4 Schritte detailliert wiedergibt. Die Datei im Visio-Format (*Regelabarbeitung.vsd*) befindet sich im Unterverzeichnis `\Entwurfsmodelle\GUI\`.

5. Umsetzung

In dem Kapitel „Umsetzung“, welches den inhaltlichen Abschluss der Arbeit darstellt, sollen einige wesentliche Aspekte der Implementierung, Modellkonvertierung und Metamodelleinbindung von EMFTrace näher erläutert und auf die entsprechenden Stellen in der Implementierung verwiesen werden. Im letzten Teil dieses Kapitels wird zudem eine Evaluierung des hierbei entstandenen Regelinterpreters und der Traceability-Regeln durchgeführt.

5.1. Einbindung der Metamodelle in EMFStore

Um neue Modelltypen in EMFStore zu integrieren, muss das von dem Modelltyp verwendete Metamodell zuerst in ein neues EMF Projekt importiert werden und daraus ein Ecore-Modell erstellt werden. In den meisten Fällen liegen dem Metamodell-Standard entsprechende XSD-Schemata [XML2E] für die Validierung von Modellen vor, die direkt für die Generierung des Ecore-Modells genutzt werden können. Sobald das Ecore-Modell generiert wurde, muss es noch um die Elemente des Unicase-Metamodells erweitert werden. Dazu muss zunächst das Unicase-Metamodell in das Ecore-Modell importiert werden und anschließend alle Elemente vom Unicase-Basiselement, dem ModelElement, abgeleitet werden. Das modifizierte Ecore-Modell sollte nun noch validiert werden, um etwaige Fehler frühzeitig zu erkennen.

Nachdem dieser Arbeitsschritt abgeschlossen wurde, muss das Generatormodell erneut geladen werden, um die oben genannten Änderungen zu übernehmen. Nun muss mit Hilfe des Generatormodells lediglich der Model- und Edit-Code generiert werden und das Metamodell kann in EMFStore verwendet werden. Jedoch darf an dieser Stelle nicht vergessen werden, die neu entstandenen Plug-ins in die Startkonfigurationen des EMFStore Servers und des EMFStore Clients zu übernehmen, sofern die Option nur ausgewählte Plug-ins beim Programmstart zu laden aktiviert wurde.

Eine genaue Anleitung, die jeden Schritt beim Einbinden neuer Metamodelle detailliert erklärt, findet sich zudem in Anhang G dieser Arbeit. Zusätzlich können die für EMFStore aufbereiteten Metamodelle der UML, URN und OWL, sowie die Metamodelle für Traceability-Links und Traceability-Regeln als praktische Referenzen für die Einbindung weiterer Metamodelle in EMFStore genutzt werden. Diese befinden sich als fertig konfigurierte Eclipse Plug-ins auf der beiliegenden CD, im Unterverzeichnis `\Quelldateien\` und können in den eigenen Workspace importiert werden.

5.1.1. URN-Metamodell

Bei der Konvertierung des URN-Metamodells (siehe Kapitel 2.3.) traten abgesehen von einer Dopplung der zur Identifizierung der Modellelemente verwendeten Identifier, keine Probleme auf. Da die von jUCMNav exportierten Identifier jedoch der Z.151 Spezifikation wiedersprachen, indem sie als natürliche Zahlen anstelle von NCNames exportiert wurden, bestand nun die einfachste Lösung darin, die eigentlichen URN-Identifier als neues Attribut zu speichern und sie nicht als Unique Identifier zu nutzen. Auf diese Weise wurde das Dopplungsproblem gelöst und sichergestellt, dass keine Informationen bei einem Rückexport aus dem Repository verloren gehen. Zudem wurde die Basisklasse *URNBase* definiert, die für die reibungslose Integration in EMFStore sorgt.

Das URN-Metamodell wird durch folgende Eclipse Plug-ins in EMFTrace eingebunden:
EMFTrace_URN (Modell-Code)
EMFTrace_URN.edit (Edit-Code)

5.1.2. OWL-Metamodell

Das OWL-Metamodell (siehe Kapitel 2.2.) bereitete bei seiner Einbindung in EMFStore einige Schwierigkeiten. Zu allererst ließen sich die XSD-Schemata nicht via EMF in Ecore-Modelle umwandeln, da die Verwendung der neusten Version der Schemadatei in Totalabstürzen von Eclipse nach der Ecore-Umwandlung resultierte. Dieses Problem trat sowohl mit Eclipse Galileo, als auch mit Eclipse Helios unter der testweisen Verwendung verschiedener Ecore Versionen auf. Die Verwendung eines älteren XSD-Schemas von 2002 brachte auch keinen Erfolg, da die Validierung beim Import nach mehr als 50 Fehlermeldungen automatisch abgebrochen wurde.

Nach weiterer Suche bin ich jedoch auf ein, bereits im Ecore-Format vorliegendes OWL-Metamodell [OWLECORE] gestoßen, welches jedoch noch nicht alle Elemente der OWL2 enthielt. Die folgenden Klassen mussten dem Metamodell noch hinzugefügt werden:

- Literal
- AnnotationAssertion
- Prefix
- IRI
- FullIRI
- AbbreviatedIRI
- OWLBase (Basisklasse zur Einbindung in EMFStore)

Das Hauptproblem bei der Einbindung der OWL stellten jedoch wieder die Identifier dar, da die OWL im Gegensatz zum Unicase-Metamodell keine Uniform Resource Identifier (URI), sondern Internationalized Resource Identifier (IRI) verwendet, wie Abbildung 33 zu entnehmen ist.

	Unicase-Metamodell	OWL-Metamodell
Identifier-Typ	URI	IRI
Beispiel	_ok4EJny5Ed-DSYk0NVzYwQ	#Member

Abbildung 33: Vergleich der Identifier

Aufgrund dieser Tatsache ließen sich so keine Referenzen auf andere Modellelemente speichern, da EMFStore die Identifier nicht auswerten kann. Dazu kommt der Umstand, dass EMFStore die Identifier eines Modells sowohl beim Import, als auch beim Export neu generiert und überschreibt. Somit ist diese Information verloren und steht für einen Import in das ursprüngliche CASE-Werkzeug nicht mehr zur Verfügung. Die Lösung bestand nun darin die Referenzen, die zuvor als IRI-Ausdruck gespeichert wurden explizit als zusätzliches String-Attribut zu speichern und somit sämtliche Referenzen durch Strings zu ersetzen. Somit gehen auch diese Referenzen nicht verloren und können wieder aus EMFStore exportiert werden. Lediglich Containment-Referenzen müssen nicht umgewandelt werden, da sie durch die logische Baumstruktur der Modelle erhalten bleiben.

Das OWL-Metamodell wird durch folgende Eclipse Plug-ins in EMFTrace eingebunden:
EMFTrace_OWL (Modell-Code)
EMFTrace_OWL.edit (Edit-Code)

5.1.3. UML-Metamodell

Für die Einbindung des UML-Metamodells (siehe Kapitel 2.1.) in EMFStore kann auf das UML-Metamodell der UML2 Tools für Eclipse zurückgegriffen werden, welches bereits als Ecore-Modell vorliegt. Allerdings mussten auch hier Ergänzungen vorgenommen werden:

- die importedElement-Referenz des ElementImport-Elements musste zur Containment-Referenz abgeändert werden
- Hinzufügen der Klasse importedElement mit den Attributen *href* und *type*
- die handler-Referenz des ExecutableNode-Elements musste zur Containment-Referenz abgeändert werden
- Hinzufügen der Basisklasse UMLBase

Aufgrund der Tatsache, dass der EMFStore die Element- und Referenz-Identifier bei jedem Import/Export neu generiert, mussten auch hier die Originalwerte wieder in einem neuen String-Attribut abgespeichert werden. Dafür wurde die UML-Basisklasse (*UMLBase*) um das neue String-Attribut *umlID* erweitert. Darüber hinaus mussten nun noch über 200 Elemente des UML-Metamodells und insgesamt über 700 Referenzen manuell angepasst und zu Strings umgebaut werden. Nach diesem Umbau ließ sich jedoch auch dieses Metamodell in EMFStore verwenden und Instanzmodelle konnten problemlos importiert und exportiert werden, ohne dass dabei Informationen verloren gehen.

Das UML-Metamodell wird durch folgende Eclipse Plug-ins in EMFTrace eingebunden:
EMFTrace_UML (Modell-Code)
EMFTrace_UML.edit (Edit-Code)

5.2. Konvertierung von Instanzmodellen

Sobald die Plug-ins für das neue Metamodell soweit aufbereitet wurden, dass sie in EMFStore verfügbar sind, können auch dessen Instanzmodelle in EMFStore erzeugt, versioniert und exportiert werden. Jedoch ist noch kein direkter Import aus den CASE-Werkzeugen möglich, mit denen die Modelle erzeugt wurden. Durch die Ergänzung des Metamodells um die Elemente aus dem Unicase-Metamodell ist eine Konvertierung der Modelle notwendig, um die fehlenden Informationen hinzuzufügen und andere Umbauten vornehmen zu können.

Ergänzungen die dabei vorgenommen werden müssen umfassen u.a.:

- Hinzufügen eines identifier-Attributes
- Hinzufügen eines timestamp-Attributes
- Hinzufügen eines creator-Attributes

Desweiteren müssen Namensräume unter Umständen angepasst werden, da EMFStore Namenspräfixe zur Identifikation von Modelltypen und Modellelementen verwendet und nicht jedes CASE-Werkzeug Namensräume mit exportiert. Für die eigentliche Konvertierung der in XML-Form abgespeicherten Modelle kommt die in Kapitel 3.5.3 vorgestellte XSTL-Technologie zum Einsatz.

In den folgenden 3 Abschnitten sollen nun kurz die wesentlichen Punkte der einzelnen Konvertierungs-Templates vorgestellt werden.

5.2.1. URN-Modelle

Bei der Konvertierung von URN-Modellen kommt es hauptsächlich auf das Hinzufügen 3 neuer Attribute (*timestamp*, *identifier* und *creator*) an. Da diese jedoch nicht in den Ausgangsmodellen vorhanden sind, werden diese Attribute durch das in Abbildung 34 dargestellte Template mit fest definierten Default-Werten gefüllt.

```
<!-- template for header information -->
<xsl:template name="Header">
    <xsl:attribute name="identifier">
        <xsl:value-of select="$IDENTIFIER"/>
    </xsl:attribute>
    <xsl:attribute name="creator">
        <xsl:value-of select="$CREATOR"/>
    </xsl:attribute>
    <xsl:attribute name="creationDate">
        <xsl:value-of select="$TIMESTAMP"/>
    </xsl:attribute>
</xsl:template>
```

Abbildung 34: *Füllen der Unicase-Attribute mit Default-Werten*

Aufgrund der Tatsache, dass der EMFStore das *identifier*-Attribut der Modelle beim Import überschreibt, ist sichergestellt, dass jedes Modellelement eine einmalige Kennzeichnung erhält. Das *creator*-Attribut wird auch mit einem festen Wert überschrieben, da man nicht auf die Benutzerinformationen des CASE-Werkzeuges zurückgreifen kann, mit der das Modell erstellt wurde. Denn nicht jeder Benutzer der Modelle mit CASE-Werkzeugen erstellt, muss auch über ein gleichnamiges Benutzerkonto im Repository verfügen und umgekehrt.

```
<!-- convert URN spec -->
<xsl:template name="URNspec">
    <xsl:element name="URN:URNspec">
        <xsl:attribute name="xmi:version">
            <xsl:value-of select="$VERSION"/>
        </xsl:attribute>
        <xsl:call-template name="ProcessElem"/>
    </xsl:element>
</xsl:template>
```

Abbildung 35: Konvertierung eines URN-Elementes durch XSLT-Templates

Die zur Konvertierung von URN-Modellen notwendigen XSLT-Templates befinden sich aufgrund ihres Umfangs auf der beiliegenden CD im Verzeichnis `\XSLT-Templates\`, in Abbildung 35 wird jedoch ein Template für die Konvertierung eines URN-Elements beispielhaft skizziert.

5.2.2. OWL-Modelle

Elemente der mit Protégé erstellten Ontologien kann man anhand eines von 3 Attributen exakt identifizieren:

1. IRI
2. abbreviatedIRI
3. fullIRI

Im Grunde handelt es sich dabei immer um den gleichen Attributwert, jedoch in anderer Schreibweise. Da OWL-Modellelemente von anderen OWL-Modellelementen immer durch genau eines dieser Attribute referenziert werden, müssen entsprechende Templates geschaffen werden, die mit jeder der 3 Darstellungsformen exakt arbeiten können. Um die Übersichtlichkeit der XSLT-Transformation zu wahren, wurden zusätzliche Templates für das Auslesen von IRIs entwickelt, die von anderen Templates genutzt werden können. Eines dieser Templates zum Auslesen der IRI ist in Abbildung 36 dargestellt.

```
<!-- convert class IRI -->
<xsl:template name="ClassIRI">
    <xsl:choose>
        <xsl:when test="(Class/@IRI)">
            <xsl:value-of select="(Class/@IRI)"/>
        </xsl:when>
        <xsl:when test="(Class/@abbreviatedIRI)">
            <xsl:value-of select="(Class/@abbreviatedIRI)"/>
        </xsl:when>
        <xsl:when test="(Class/@fullIRI)">
            <xsl:value-of select="(Class/@fullIRI)"/>
        </xsl:when>
    </xsl:choose>
</xsl:template>
```

Abbildung 36: Suche nach dem passenden IRI-Attribut

Bei dem Aufruf eines Templates zur Konvertierung von Elementen muss nun nicht mehr die Art der referenzierten IRI überprüft werden, da ein simpler Template-Aufruf für diesen Fall genügt, wie in Abbildung 37 dargestellt ist.

```
<!-- convert ObjectPropertyDomain -->
<xsl:template match="ObjectPropertyDomain">
    <xsl:element name="OntologyElement">
        <xsl:attribute name="xsi:type">owl:ObjectPropertyDomain</xsl:attribute>
        <xsl:call-template name="Header"/>
        <xsl:attribute name="objectProperty">
            <xsl:call-template name="ObjPropIRI"/>
        </xsl:attribute>
        <xsl:choose>
            <xsl:when test="not(Class)">
                <xsl:apply-templates/>
            </xsl:when>
            <xsl:otherwise>
                <xsl:attribute name="domain">
                    <xsl:call-template name="ClassIRI"/>
                </xsl:attribute>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:element>
</xsl:template>
```

Abbildung 37: Konvertierung eines OWL-Elementes durch ein XSLT-Template

Die zur Konvertierung von OWL-Modellen notwendigen XSLT-Templates befinden sich aufgrund ihres Umfangs auf der beiliegenden CD, im Verzeichnis `\XSLT-Templates\`.

5.2.3. UML-Modelle

Aufgrund der einheitlichen Struktur der von den UML2 Tools für Eclipse exportierten UML-Modelle, musste für die Konvertierung sämtlicher UML-Modellelemente nur das in Abbildung 38 gezeigte Basistemplate *ProcessElem* geschaffen werden.

```
<!--  process basic element information  -->
<xsl:template name="ProcessElem">
    <xsl:for-each select="@xmi:type">
        <xsl:attribute name="xsi:type">
            <xsl:value-of select="."/>
        </xsl:attribute>
    </xsl:for-each>
    <xsl:call-template name="Header"/>
    <xsl:call-template name="ConvertAttr"/>
    <xsl:apply-templates/>
</xsl:template>
```

Abbildung 38: Basistemplate für UML-Elemente

Bis auf 3 Ausnahmen können alle UML-Modellelemente mithilfe dieses Templates in eine EMFStore-kompatible Form umgewandelt werden und es genügt der Aufruf der in Abbildung 39 dargestellten Konvertierungsvorschrift.

```
<!-- convert model elements -->
<xsl:template match="*">
    <xsl:choose>
        <xsl:when test="(local-name() = 'Package')">
            <xsl:call-template name="Package"/>
        </xsl:when>
        <xsl:when test="(local-name() = 'DeploymentSpecification')">
            <xsl:call-template name="DeploymentSpecification"/>
        </xsl:when>
        <xsl:when test="(local-name() = 'Profile')">
            <xsl:call-template name="Profile"/>
        </xsl:when>
        <xsl:otherwise>
            <xsl:element name="{name(.)}">
                <xsl:call-template name="ProcessElem"/>
            </xsl:element>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Abbildung 39: Konvertierung eines UML-Elementes durch ein XSLT-Template

Bei der Verarbeitung der Modellattribute durch das *ConvertAttr*-Template muss zusätzlich die Kardinalität von Attributen angepasst werden, da der Bezeichner für „Unendlich“ in EMF durch eine -1 dargestellt wird, während die UML Tools das übliche *-Format verwenden.

Die zur Konvertierung von UML-Modellen notwendigen XSLT-Templates befinden sich aufgrund ihres Umfangs auf der beiliegenden CD, im Verzeichnis `\XSLT-Templates\`.

5.3. Ausgewählte Aspekte der Implementierung

In diesem Kapitel der Arbeit sollen ausgewählte Aspekte der Implementierung des Regelinterpreters näher betrachtet werden. Dazu zählt ein tieferer Einblick in die Regelverarbeitung, die Umsetzung des N-Gram-Algorithmus, sowie die Transitivitätsanalyse des LinkManagers. Zudem soll kurz auf die Validierung und Dokumentation der Implementierung eingegangen werden.

5.3.1. Transitivitätsanalyse des LinkManagers

Wie bereits in Kapitel 4.5.2. angedeutet, ist die Analyse von Traceability-Links auf darin enthaltene transitive Relationen ein wichtiger Teil der modellübergreifenden Analysefähigkeit des Repository. Zwischen zwei Traceability-Links besteht genau dann eine transitive Relation, wenn sie über mindestens zwei übereinstimmende LinkEnds verfügen, d.h. wenn diese LinkEnds auf dasselbe Modellelement verweisen. Realisiert wird diese Analyse durch folgende Methode des LinkManagers:

```
public void performTransitivityAnalysis(Project project)
```

Als erstes wird in dieser Methode nach allen Instanzen der Klasse *TraceLink* gesucht und diese in die Liste *links* einsortiert, anschließend erfolgt die eigentliche Analyse mithilfe folgenden Methodenaufrufes:

```
// find possible relationships:
List<List<TraceLink>> results = checkForIndirectRelationships(links);
```

Das Resultat dieses Methodenaufrufes ist eine Liste, die transitive TraceLink-Ketten enthält, d.h. jeder Listeneintrag ist eine Liste transitiv zusammenhängender TraceLink-Instanzen.

Das Vorgehen in dieser Methode stellt sich wie folgt dar:

1. iteriere über alle Elemente der Liste *links* und füge das *i*-te Listenelement an Stelle *i* der Ergebnisliste ein
2. überprüfe alle Listenelemente der Ergebnisliste auf direkte Abhängigkeiten mittels der Methode `checkForDirectRelationship(TraceLink link1, TraceLink link2)`
3. iteriere über alle Listeneinträge der Ergebnisliste und verkette die Listen, die mindestens 1 gemeinsames Element enthalten

Im letzten Schritt müssen die gefundenen Abhängigkeiten noch als Trace-Instanzen umgesetzt werden, sofern die Listenelemente noch mehr als einen TraceLink enthalten:

```
// create transitive Traces:
for(int i = 0; i < results.size(); i++)
    if( results.get(i).size() > 1 )
        createTrace(project, results.get(i), this.getName());
```

5.3.2. Regelverarbeitung der RuleEngine

Die für die eigentliche Abarbeitung der Traceability-Regeln zuständige Komponente RuleProcessor wurde bereits in Kapitel 4.5.3.2. vorgestellt und deren Schnittstellen dort erläutert. In diesem Abschnitt soll hingegen die interne Arbeitsweise der Komponente bei der Abarbeitung von Regeln erläutert werden. In der von der RuleProcessor-Komponente implementierten *run*-Methode beginnt die Abarbeitung durch die Methode *executeLogicCondition*, da die Bedingungen in Regeln immer durch eine logische Bedingung vom Typ AND, OR, NOT oder XOR eingebunden werden müssen:

```
public void run(Project project, [...])
{
    if( executeLogicCondition(rule, results, rule.getConditions()) )
    {
        [...]
    }
}
```

In der Methode *executeLogicCondition* werden zunächst die Basisbedingungen vom Typ *Equals*, *SimilarTo* etc. abgearbeitet (sofern vorhanden) und anschließend alle weiteren, geschachtelten logischen Bedingungen (sofern vorhanden):

```
// execute the basic conditions:
for(int i = 0; i < condition.getBaseConditions().size(); i++)
{
    switch(condition.getType())
    {
        case AND :
        {
            if( !executeBaseCondition(rule, [...]) return false;
        }
        [...]
    }
}

// execute the logic conditions:
for(int i = 0; i < condition.getLogicConditions().size(); i++)
{
    switch(condition.getType())
    {
        case AND :
        {
            if( !executeLogicCondition(rule, [...] ) return false;
        }
        [...]
    }
}
```

Bei der Abarbeitung der einzelnen Bedingungen wird dabei immer überprüft, welchem logischen Typ die Elternbedingung entspricht. Ist die Elternbedingung vom Typ *AND*, kann die Abarbeitung genau dann abgebrochen werden, sobald eine Teilbedingung nicht erfüllt wurde. Für die anderen Typen gilt folgendes Vorgehen:

Typ „OR“:
Jede Teilbedingung arbeitet auf einer Kopie der ursprünglichen Eingabeliste. Sollten alle Teilbedingungen fehlschlagen, schlägt die OR-Bedingung fehl. Nach Abarbeitung aller Teilbedingungen werden die einzelnen Ergebnislisten zu einer neuen Rückgabeliste gemischt, sofern die Bedingung nicht fehlgeschlagen ist.

Typ „XOR“:
In Analogie zur OR-Bedingung arbeitet jede Teilbedingung auf einer Kopie der Eingabeliste. Sollten dabei mehr als 1 Teilbedingung erfolgreich abgearbeitet werden, schlägt die XOR-Bedingung fehl, ebenso wenn keine der Teilbedingungen erfolgreich abgearbeitet wurde.

Typ „NOT“:
Vor Abarbeitung einer mit NOT geschachtelten Bedingung wird eine Kopie der Eingabeliste erzeugt, auf der anschließend weitergearbeitet wird. Falls die Abarbeitung erfolgreich verläuft, wird die Schnittmenge zwischen der Eingabeliste und deren Kopie bestimmt und diese Schnittmenge danach aus der ursprünglichen Eingabeliste entfernt. Schlägt die mit NOT geschachtelte Bedingung fehl, kann die ursprüngliche Eingabeliste hingegen unverändert weitergenutzt werden.

In der Methode *executeBaseCondition* werden dann die tatsächlichen Anfragebedingungen wie etwa EQUALS oder CONTAINS abgearbeitet. Das Vorgehen umfasst dabei folgende Schritte:

1. Umwandlung des Element-Alias in einen Listenindex und Zugriff auf diese Liste
2. Iteration über alle Elemente dieser Liste, dabei:
    a. Prüfe ob das spezifizierte Attribut vorhanden ist, wenn ja: weiter, sonst:
       lösche das Element aus der Liste
    b. Vergleiche den Attributwert, schlägt der Vergleich fehl:
       lösche das Element aus der Liste

5.3.3. Implementierung des N-Gram-Algorithmus

Für eine Ähnlichkeitsanalyse von zwei Attributwerten kann der Regelinterpreter auf den N-Gram-Algorithmus [CT94] zurückgreifen, der in dieser Arbeit in zwei Teilmethoden untergliedert wurde:

1. Das Erzeugen der N-Gramme mittels der Methode *createNGram*
2. Der Vergleich der N-Gramme mittels der Methode *compareWords*

Die Aufteilung eines Wortes in seine N-Gramme wird dabei wie folgt durch *createNGram* realisiert:

```
int idx = 0;
for(int i = 0; i < input.length()+1; i++)
{
    if( i == 0 )
    {
        nGrams[i] = "§";
        for(int j = 0; j < n-1; j++) nGrams[i] += input.charAt(j);
    }
    else
    {
        for(int j = 0; j < n; j++)
        {
            if( (idx + j) < input.length() )
                nGrams[i] += input.charAt(idx+j);
            else nGrams[i] += "§";
        }

        if( idx < input.length() ) idx++;
    }
}
```

Die Anwendung dieses Algorithmus würde für das Wort „book“ folgende N-Gramme erzeugen:

2-Gramme: {§b, bo, oo, ok, k§}

3-Gramme: {§bo, boo, ook, ok§, k§§}

Der Vergleich zweier in N-Gramme zerlegte Worte wird anschließend in der Methode *compareWords* durch die Ermittlung des Dice-Koeffizienten ausgeführt:

```
int numberOfMatches = 0;
for(int i = 0; i < nGrams1.length; i++)
    for(int j = 0; j < nGrams2.length; j++)
        if( nGrams1[i].equals(nGrams2[j]) )
            numberOfMatches++;

float correlation = (float)(2*numberOfMatches) /
                    (float)(nGrams1.length + nGrams2.length);

if( correlation >= minCorrelation ) return true;
else return false;
```

5.4. Test und Validierung der Implementierung

Für das Testen der im Rahmen dieser Arbeit entstandenen Funktionalität kamen drei verschiedene Vorgehensweisen zum Einsatz:

1. Praxistest mit aktivem Repository-Server, importierten Modellen, Regeln und Typkatalogen
2. JUnit-Testfälle und Testsuites für jede Teilkomponente des Gesamtsystems
3. Eclipse FindBugs [EFB] zur Quelltextanalyse

In den folgenden 3 Teilkapiteln soll daher das Vorgehen beim Testen näher betrachtet werden.

5.4.1. Praxistest

Der Praxistest von EMFTrace bestand aus insgesamt 4 Teilphasen:

1. Test der Modellkonvertierungen
2. Test des Regelinterpreters
3. Analyse des AccessLayer-Cache
4. Test des fertigen Client

In Phase 1 wurden dabei sämtliche Konvertierungstemplates (siehe Kapitel 5.1.2) bei der Umwandlung von Modellinstanzen geprüft. Jeder Test bestand dabei aus den gleichen 6 Arbeitsschritten:

1. Export des Modells aus dem ursprünglichen CASE-Werkzeug
2. Konvertierung in das Unicase-Format
3. Import in EMFStore
4. Export aus EMFStore
5. Rückkonvertierung
6. Import in das ursprüngliches CASE-Werkzeug

Für den Test des Regelinterpreters in Phase 2 wurden eine Reihe von einfachen Modellen, Regelkatalogen und Linktypen manuell in EMFStore angelegt, mit denen sichergestellt werden konnte, dass jede Kombination von Regelbedingungen mindestens einmal erfasst wurde. Anschließend wurden die vom Regelinterpreter erzeugten Ergebnisse mit den zuvor von Hand erstellten Ergebnissen verglichen. Zusätzlich wurde für jede Traceability-Regel ein eigenes Testprojekt angelegt, mit dem die Regel überprüft werden kann. Diese Testprojekte befinden sich im

Unterverzeichnis `\Validierungsprojekte\` der beiliegenden CD und können in EMFStore importiert werden.

Um das Laufzeitverhalten der AccessLayer-Komponente (siehe Kapitel 4.5.1.) und die Auswirkungen des Cache in Phase 3 zu testen, hat sich die Erstellung eines speziellen JUnit-Tests angeboten, der die Ausführungszeit des Tests stoppt. Innerhalb der Testklasse wurden einfache Anfragen nach Modellelementen einer bestimmten Klasse in einer Schleife durchlaufen, jeweils mit und ohne aktivierten Cache. In der nachfolgenden Tabelle ist ein kleiner Auszug der verwendeten Testparameter und den daraus resultierenden Laufzeiten zu sehen:

Testumgebung	**Laufzeit ohne Cache**	**Laufzeit mit Cache**
7 Modellklassen, 1000 Elemente pro Klasse, 500 Durchläufe	13485 ms	30 ms
7 Modellklassen, 3000 Elemente pro Klasse, 500 Durchläufe	54768 ms	124 ms
20 Modellklassen, 1000 Elemente pro Klasse, 500 Durchläufe	143079 ms	282 ms
20 Modellklassen, 3000 Elemente pro Klasse, 500 Durchläufe	494570 ms	1030 ms

An dieser Stelle soll noch eine Worst-Case Analyse bezüglich des Speicherplatzbedarfs und der Laufzeit des AccessLayers mit aktivem Cache durchgeführt werden. Der Worst-Case bei Verwendung des Cache tritt genau dann ein, wenn pro Modellklasse nur ein Instanzmodell vorhanden ist, und die Anzahl der im Cache gespeicherten Listen genau der Anzahl der Modelle entspricht, d.h.:

Sei *n* die Anzahl der im Repository *R* gespeicherten Modelle und für alle Modelle $m1 \in R, m2 \in R$ gilt: $m1.getClass() \neq m2.getClass()$

Daraus folgt, dass im AccessLayer genau *n* Strings in der Header-Liste des Cache gespeichert werden und der eigentliche Cache genau *n* Referenzen umfasst. Insgesamt ergibt sich somit ein zusätzlicher Speicherverbrauch von genau *2n*, d.h. *O(n)*.

Die Laufzeit mit aktivem Cache entspricht im Worst-Case genau der Laufzeit ohne Cache, denn:

Ohne Cache wird das gesamte Repository durchlaufen und es werden genau *n* Vergleiche ausgeführt, d.h. *O(n)*. Da im Cache nur Listen enthalten sind, die exakt 1 Element enthalten, müssen somit im schlechtesten Fall auch genau *n* Elemente überprüft werden, d.h. auch *O(n)*.

5.4.2. Erstellung von JUnit-Testfällen

Für jede Teilkomponente und für alle verwendeten Hilfsdatenstrukturen (z.B. *QueryElement*) wurde im Rahmen dieser Ausarbeitung ein eigenständiger JUnit-Testfall definiert, der von der JUnit-Basisklasse *TestCase* abgeleitet wurde. Hier sei beispielhaft der Testfall für den AccessLayer genannt:

```
public class AccessLayerTest extends TestCase
```

Eines der größten Probleme bei der Erstellung der Testfälle war die Tatsache, dass viele Funktionen einen laufenden EMFStore Server und damit ein funktionsfähiges Repository benötigen. Da das ständige An- und Abschalten des Servers zu Testzwecken äußerst impraktikabel ist und immer die Gefahr besteht, dass Fehler in EMFStore (da auch noch in einer recht frühen Version vorliegend) mögliche Fehler der Komponenten verschleiern, habe ich mich dazu entschlossen bei sämtlichen Testfällen mit manuell konfigurierten Dummy-Projekten zu arbeiten. Diese Dummy-Projekte werden vor der Ausführung jeder Testklasse durch folgende Methode erzeugt:

```
protected void setUp()
```

Um die Ausführung der Tests schneller und vor allem komfortabler zu gestalten, wurden sämtliche Testfälle abschließend in einer JUnit-Testsuite zusammengefasst:

```
public class AllTests
{
    public static Test suite()
    {
        Class[] testClasses = { AccessLayerTest.class,
                                ListHelperTest.class,
                                NGramCheckTest.class,
                                QueryElementTest.class,
                                LinkManagerTest.class,
                                ElementProcessorTest.class,
                                ResultProcessorTest.class,
                                RuleEngineTest.class,
                                RuleProcessorTest.class,
                                TraceComponentTest.class,
                                RuleValidatorTest.class,

                                [...]
                              };

        TestSuite suite= new TestSuite(testClasses);
        return suite;
    }
}
```

Somit ist jederzeit die Möglichkeit gegeben, den kompletten EMFTrace-Quellcode mit dem Start eines einzigen JUnit-Tests auf mögliche Fehler zu testen.

5.4.3. Quelltextanalyse mit Eclipse FindBugs

Als zusätzliches Werkzeug zur Qualitätssicherung der Implementierung bietet sich die Nutzung von *Eclipse FindBugs* [EFB] an, welches speziell für die Analyse von Java-Quelltext entwickelt wurde.
Durch die einfache Integration als Eclipse Plug-in lässt sich neu erstellter Quelltext schnell auf häufig auftretende Probleme, wie etwa vergessene NULLPointer-Abfragen überprüfen. Zudem bietet FindBugs auch Alternativlösungen für bestimmte Java-Konstrukte an und weist auf mögliche Performanceeinbußen hin, so z.B. entstehend durch die Konkatenation von Strings innerhalb von Schleifenausführungen.
Auch wenn FindBugs das Testen der Komponenten nicht ersetzen konnte, hat es dennoch dazu beigetragen, die Qualität der Implementierung und das Laufzeitverhalten der Komponenten zu verbessern.

5.5. Dokumentation der Implementierung

Für die Dokumentation der Implementierung, sowie die Beschreibung der Funktionalität der Plug-ins und Komponenten dient zuallererst diese Arbeit und deren Anhänge. Darüber hinaus wurden für alle in den Plug-ins definierten Schnittstellen und Klassen eine komplette Dokumentation mit JavaDoc[2] generiert und den Plug-ins beigefügt bzw. kann diese auf Wunsch selber in Eclipse generiert werden.

Um die Erzeugung der JavaDoc zu ermöglichen, mussten sämtliche Interfaces und Methoden mit den üblichen Annotationselementen angereichert werden, die später in die HTML-basierte Dokumentation überführt wurden, was hier beispielhaft an der Methode *createLink* des Interfaces *ILinkManager* demonstriert werden soll:

```
/**
 * Adds a new {@link TraceLink}-object to the {@link Project project}.
 * The link will be of a binary fashion, connecting one source- with one
 * target-{@link LindEnd}.
 *
 * @param project the current project
 * @param source  the LinkEnd-source
 * @param target  the LinkEnd-target
 * @param creator the creator of this object
 * @param type    the type of the new link
 * @param rule    the name of the rule
 * @return        a new instance of TraceLink
 */
public TraceLink createLink(Project project, ModelElement source, [...]);
```

5.6. Evaluierung

Auch wenn der Regelinterpreter und der dazugehörige Regelkatalog im Moment noch von eher prototypischer Natur sind, soll an dieser Stelle eine Evaluierung der damit erzielten Ergebnisse hinsichtlich Precision- und Recall-Faktoren durchgeführt werden, um einen ersten Eindruck von der Qualität dieser Ergebnisse zu erhalten. Für die Ermittlung der Bewertungsfaktoren kommen zwei Praxisprojekte zum Einsatz, die zuerst manuell auf Traceability-Beziehungen untersucht und anschließend maschinell durch EMFTrace analysiert werden. Abschließend erfolgt eine Auswertung, sowie ein Vergleich der Ergebnisse mit denen anderer Arbeiten.

[2] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

5.6.1. RSI-Framework

Das erste Praxisprojekt welches als Untersuchungsgrundlage für die Evaluierung zum Einsatz kam, ist ein Forschungsprojekt des Fachgebietes Neuroinformatik der TU Ilmenau. Aus dem Projekt ließen sich Komponentendiagramme, Sequenzdiagramme, Kompositionsstrukturdiagramme, eine Ontologie und ein GRL-Graph von nichtfunktionalen Anforderungen für die Analyse nutzen. Die maschinelle Analyse durch EMFTrace wurde insgesamt zweimal ausgeführt, wobei einmal ein Ähnlichkeitsfaktor von 75% und einmal ein Ähnlichkeitsfaktor von 90% verwendet wurden. Die Länge der N-Gramme (*n*) wurde dabei konstant bei 3 belassen. Die dabei ermittelten Ergebnisse sind in Abbildung 40 dargestellt. Die detaillierten Analyseergebnisse liegen aufgrund ihres Umfangs auf der beiliegenden CD im Verzeichnis `\Evaluierung\` als PDF-Datei vor

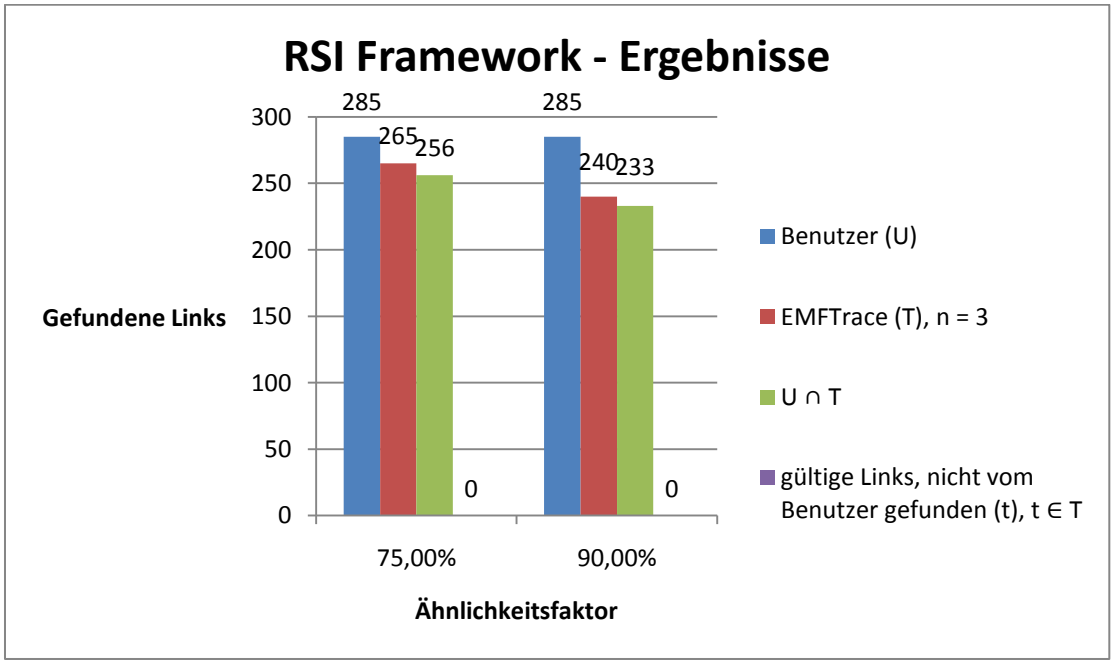


Abbildung 40: Analyseergebnisse des RSI-Projektes

5.6.2. EMFTrace

Als zweites Evaluierungsprojekt kam das EMFTrace-Projekt selber zum Einsatz. Aus dem Projekt konnten Klassendiagramme, Komponentendiagramme und Anwendungsfalldiagramme für eine Analyse genutzt werden. In Abbildung 41 sind die dabei ermittelten Ergebnisse dargestellt. Die maschinelle Untersuchung wurde auch hierbei wieder mit einem Ähnlichkeitsfaktor von 75% und 0%, sowie einem *n* von 3 durchgeführt. Die detaillierten Analyseergebnisse befinden sich dabei wieder in PDF-Form auf der beiliegenden CD im Verzeichnis `\Evaluierung\`.

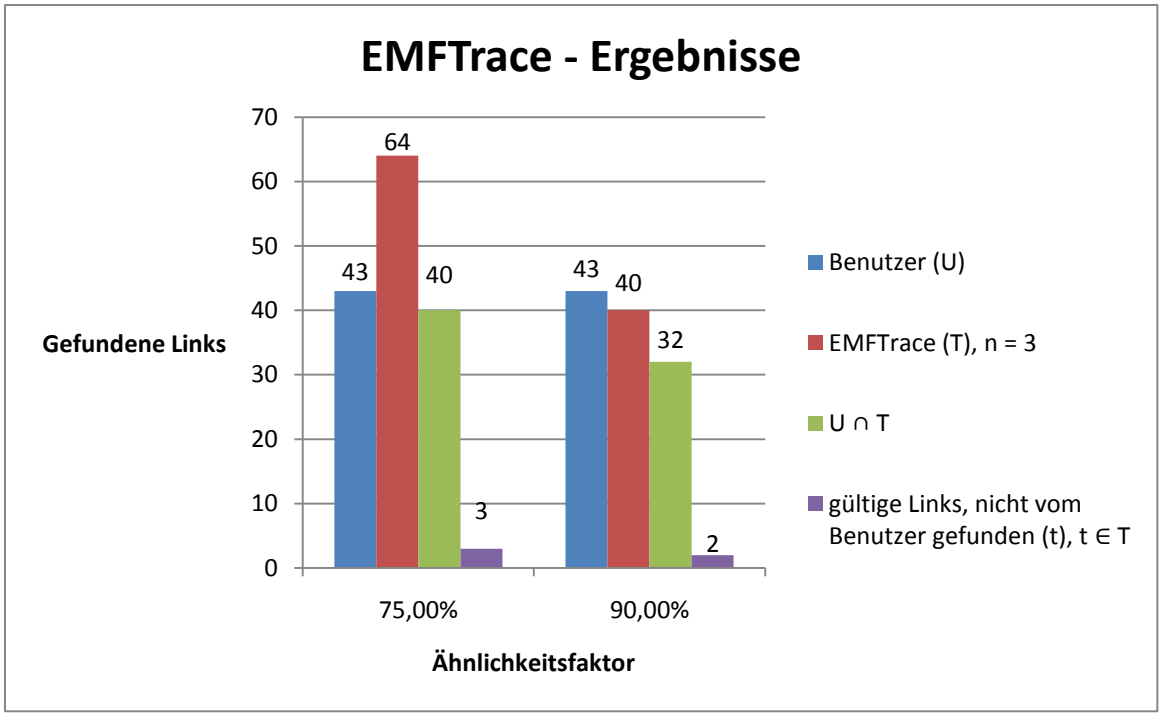


Abbildung 41: Analyseergebnisse des EMFTrace-Projektes

5.6.3. Auswertung

Um eine qualitative Bewertung aus den ermittelten Ergebnissen ableiten zu können, müssen zunächst die Precision- und Recall-Faktoren bestimmt werden. Sei dabei *U* die Anzahl der vom Nutzer gefundenen Beziehungen und *T* die Anzahl der von EMFTrace insgesamt gefundenen Beziehungen. Für die Berechnung dieser Faktoren ergeben sich nun folgende Gleichungen:

$$Precision\ (P) := \frac{U \cap T}{T}$$

$$Recall\ (R) := \frac{U \cap T}{U}$$

Mithilfe dieser Berechnungsvorschriften ergeben sich die in Abbildung 42 dargestellten Werte für Precision und Recall. Aus Abbildung 42 lässt sich zudem auch der durchschnittliche Precision- und Recall-Faktor ableiten, der mit EMFTrace erreicht wurde:

$$P_{avg} = 0{,}840 = 84{,}0\%$$

$$R_{avg} = 0{,}847 = 84{,}7\%$$

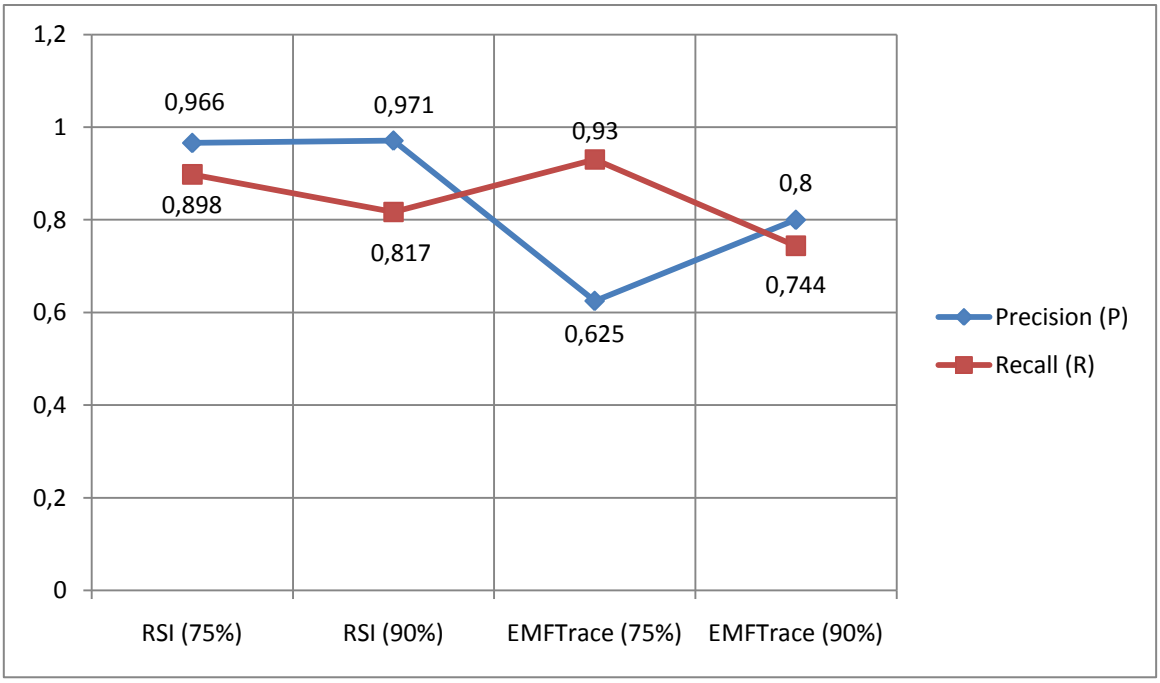


Abbildung 42: Precision und Recall in Abhängigkeit des Ähnlichkeitsfaktors

Der hierbei erzeugte Recall ist Im Vergleich zu anderen Arbeiten [JZ09] als „gut“ zu bewerten. Würde man die von EMFTrace richtig erkannten, aber vom Nutzer nicht gefundenen Beziehungen t ($t \in T \wedge t \cap U = \emptyset$) mit in die Ermittlung des Recall-Faktors einbeziehen, würde sich sogar ein Recall von 85,1% einstellen. Die Formel für R‘ müsste dabei wie folgt angepasst werden:

$$Recall'\ (R') := \frac{(U \cap T) \cup t}{U \cup t}$$

Die Precision ist mit 84% ebenfalls vergleichbar mit anderen Arbeiten und ist unter Berücksichtigung der Tatsache, dass der Regelinterpreter als auch die Regeln momentan noch von eher prototypischer Natur sind als „gut“ zu bewerten. Würde man die vom Tool richtig erkannten, aber vom Nutzer nicht gefundenen Beziehungen t ($t \in T \wedge t \cap U = \emptyset$) mit in die Bewertung einfließen lassen, würde sich eine Precision von 86,5% einstellen. Die Formel für P‘ müsste dabei wie folgt angepasst werden:

$$Precision'\ (P') := \frac{(U \cap T) \cup t}{T}$$

II. Zusammenfassung und Ausblick

In dieser Arbeit wurden eine Reihe von Konzepten entwickelt, die die Einbindung von Modellen in ein Repository und die damit verbundene Fragestellung der Modellkonvertierung lösen. Zugleich wurden Ansätze für Traceability-Beziehungen und Traceability-Regeln entwickelt, die sich problemlos in das Repository integrieren lassen und somit einen einheitlichen Blick auf die darin gespeicherten Modelle wahren. Durch die Bündelung des Konzeptes zur regelbasierten Traceability-Generierung mit Ansätzen und Algorithmen des Information Retrieval wurde gezeigt, dass beide Ansätze sinnvoll miteinander kombinierbar sind und zu einer mächtigeren Regelsprache beitragen können. In der anschließenden Evaluierung der prototypischen Anwendung und der dabei entstandenen Regeln konnte gezeigt werden, dass damit sinnvolle Traceability-Beziehungen erkannt wurden.

In dem einführenden Kapitel dieser Arbeit wurde das zu entwickelnde Repository hinsichtlich seiner Aufgaben und Anforderungen eingeordnet und die Motivation hinter dem Repository erläutert. Zugleich wurde eine genaue Abgrenzung zu vorhergehenden Diplomarbeiten und deren Zielen vorgenommen, sowie eine Skizze des verwendeten Lösungsweges aufgezeigt.

Die für diese Arbeit notwendigen Grundlagen wurden anschließend im nächsten Kapitel kurz umrissen und auf weiterführende Literaturquellen verwiesen. Neben Metamodellen wie das der UML, wurden dabei auch Konzepte zur Modellkonvertierung und das Eclipse Modeling Framework vorgestellt.

Aufbauend auf diesen Grundlagen konnten in Kapitel 3 zahlreiche Untersuchungen bestehender Ansätze und Technologien durchgeführt werden, die alle einen Beitrag zur Lösung der Aufgabenstellung dieser Arbeit geleistet haben. Es wurden dabei bestehende Repository Projekte auf ihre Eignung als Basis für EMFTrace analysiert, sowie Konzepte und Methoden zur Traceability-Findung bewertet.

Im Hauptteil dieser Arbeit wurde das eingangs skizzierte Lösungskonzept umfassend erläutert und die Schritte zur Lösungsfindung detailliert beschrieben. Es wurden EMF-basierte Metamodelle für Traceability-Links und Traceability-Regeln entworfen, sowie deren Einbindung in das Repository erläutert. Anschließend wurde der Entwurf des Regelinterpreters schrittweise verfeinert, angefangen bei der Findung von Teilkomponenten durch die Trennung nach Zuständigkeiten, bis hin zur Entwicklung detaillierter Klassendiagrammen, die die Funktionalität jeder Teilkomponente wiederspiegeln.

Den Abschluss dieser Arbeit bildet ein Einblick in wichtige Teile der Umsetzung vorgestellter Konzepte, d.h. die Konvertierung und Einbindung von Metamodellen, die Implementierung der Regelverarbeitung, sowie den Test und die Evaluierung des dabei entwickelten Regelinterpreters.

Ein Aspekt der im Rahmen dieser Ausarbeitung nicht untersucht werden konnte, ist die automatische Synchronisation des Repository mit den beteiligten CASE-Werkzeugen. Aktuell ist es noch dem Anwender überlassen, Modelle in das Repository zu importieren und die Synchronisation mit dem CASE-Werkzeug über einen erneuten Import bzw. Export zu lösen. Ideal wäre hierbei die Schaffung von Adaptern für die CASE-Werkzeuge, die solche Vorgänge ereignisgesteuert (z.B. bei Modelländerungen) anstoßen können und den Benutzer auf Wunsch über synchronisationsbedingte Änderungen informieren. Für weitere Arbeiten bietet das hier entwickelte Konzept und die dabei entstandene, prototypische Anwendung eine Reihe von interessanten Fragestellungen, die noch zu bearbeiten sind. Zum Einem steht die Frage nach der Integration weiterer Metamodelle, wie etwa BPMN im Raum. Neben der Integration des eigentlichen Metamodells müssen dabei auch wieder die Fragestellungen der Konvertierung und Regelfindung beachtet werden, schließlich verlangen neue Modelltypen auch nach neuen Regeln und Konvertierungsvorschriften. Neben diesen modellbasierten Erweiterungen bietet das Konzept auch noch Anknüpfungspunkte für Konsistenzanalysen, die eine qualitative Analyse von Modellen im Repository erlauben. Für derartige Analysen lassen sich die Konzepte der Traceability-Regeln und Traceability-Regelkataloge wiederverwenden und nach Möglichkeit erweitern. Abschließend bietet auch die Visualisierung von Traceability-Beziehungen im Repository noch Potential für weitere Arbeiten, insbesondere die Komplexitätsbewältigung durch die schiere Menge an möglichen Traceability-Links und die damit verbundene, optimale Aufbereitung für den Anwender.

Zusammenfassend lässt sich jedoch feststellen, dass mit den im Rahmen dieser Arbeit entwickelten Konzepten, dem dabei entstandenen Regelinterpreter und den prototypischen Traceability-Regeln eine solide und praktisch anwendbare Werkzeugergänzung für Software-Architekten und Entwickler entstanden ist. Mit EMFTrace lässt sich die zeitaufwändige Suche nach Traceability-Beziehungen zwischen verschiedenen Modellen wesentlich schneller und vor allem automatisiert bewältigen. Dazu kommen die Validierungs- und Verknüpfungsfeatures die EMFTrace anbietet, die eine weitere Ergänzung des Nutzungsumfangs von EMFTrace darstellen und eine zusätzliche Zeitersparnis für den Anwender bedeuten. Durch die Offenheit für neue Metamodelle, Regeln und Analysemethoden ist EMFTrace zudem eine geeignete Ausgangsbasis für weitere Forschungstätigkeiten und darauf aufbauenden, praktischen Anwendungen für den Einsatz in der Softwareentwicklung.

III. Literaturverzeichnis

[AKR06] **Amyot, Daniel; Kealey , Jason; Roy, Jean-Francois:**
Towards Integrated Tool Support for the User Requirements Notation
In: SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modeling Volume 4320 of Lecture Notes in Computer Science., Springer (2006), S. 198–215

[ATL] **Atlas Transformation Language Project:**
http://www.eclipse.org/atl/
(Stand: 24.06.2010)

[BEEKKT07] **Biermann, Enrico; Ehrig, Karsten; Ermel, Claudia; Köhler, Christian; Kuhns, Günter; Täntzer, Gabi:**
Tiger EMF Model Transformation Framework Manual
TU Berlin, 2007

[CDO] **CDO Model Repository Project:**
http://www.eclipse.org/cdo/
(Stand: 30.01.2009)

[CH03] **Czarnecki, Krzysztof; Helsen, Simon:**
Classification of Model Transformation Approaches
OOPSLA2003 Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA, 2003

[CT94] **Cavnar, William B.; Trenkle, John M.:**
N-Gram-Based Text Categorization
In: Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval (Las Vegas, US, 1994), S. 161–175.

[DKPF09] **Drivalos, Nikolaos; Kolovos, Dimitrios S.; Paige, Richard F.; Fernandes, Kiran J.:**
Engineering a DSL for Software Traceability
In: 1st International Conference on Software Language Engineering (SLE 2008), Revised Selected Papers, S. 151-167, Springer-Verlag, 2009

[EFB] **Eclipse FindBugs:**
http://findbugs.sourceforge.net/
(Stand: 22.08.2009)

[EMF] **Eclipse Modeling Framework:**
http://www.eclipse.org/modeling/emf/
(Stand: 20.06.2010)

[EMR] **Eclipse Model Repository:**
http://modelrepository.sourceforge.net/
(Stand: 04.04.2010)

[EMFSTORE] **EMFStore:**
http://code.google.com/p/unicase/wiki/EMFStoreNavigation
(Stand: 04.10.2010)

[EGM05] **Eberle, Stephan; Gerhardt, Frank; Moroff, Dieter:**
Das Eclipse Modeling Framework
Java User Group Stuttgart: Special Interest Group (SIG) “Eclipse”, Stuttgart, 2005

[FZS03] **Filho, Gilberto A. A. Cysneiros; Zisman, Andrea; Spanoudakis, George:**
A Traceability Approach for i* and UML Models
In: Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS), Lecture Notes in Computer Science, Portland, OR, May 2003. Springer

[GAO10] **Gao, Yan:**
Import/Export of URN Models in Z.151 XSML File Format with jUCMNav
Master Thesis, University of Ottawa, Canada, 2010

[GREQL] **Graph Repository Query Language:**
http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/Graphentechnologie/GReQL
(Stand: 30.03.2009)

[HENSHIN] **Henshin Project:**
http://www.eclipse.org/modeling/emft/henshin/
(Stand: 23.06.2010)

[IK06] **Ivkovic, Igor; Kontogiannis, Kostas:**
Towards Automatic Establishment of Model Dependencies using Formal Concept Analysis
In: International Journal of Software Engineering and Knowledge Engineering (IJSEKE), 2006, 16(4): S. 499-522

[JZ09] **Jirapanthong, Waraporn; Zisman, Andrea:**
XTraQue: traceability for product line systems
In: Software and Systems Modeling 2009, Volume 8, Number 1, S. 117-144, Berlin / Heidelberg, Springer 2009

[JUCMNAV] **jUCMNav Project:**
http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome
(Stand: 02.10.2010)

[LG05] **Limón, Angelina E.; Garbajosa, Juan:**
The Need for a Unifying Traceability Scheme
In: Proceedings of the European Conference in Model Driven Architecture Traceability Workshop (ECMDA-TW), S. 47–55, Nürnberg, 2005

[LET02] **Letelier, Patricio:**
A Framework for Requirements Traceability in UML-based Projects
In: Proceedings of the 1st International Workshop on Traceability for Emerging Forms of Software Engineering (TEFSE’02), Edinburgh, UK, September 2002.

[MIC05] **Michael, Dirk:**
Konzept zur CASE - Werkzeugkopplung zwecks modellübergreifender Konsistenzprüfung
Diplomarbeit, TU Ilmenau, 2005

[MPR07] **Maeder, Patrick; Philippow, Ilka; Riebisch, Matthias:**
Customizing Traceability Links for the Unified Process
Proceedings of the 3rd International Conference on the Quality of Software-Architectures (QOSA2007), Medford MA, USA, 12.-13. Juli, 2007,
Springer: LNCS, 2007

[MGP08] **Maeder, Patrick; Gotel, Orlena; Philippow, Ilka:**
Rule-Based Maintenance of Post-Requirements Traceability Relations
In: Proceedings of the 16th IEEE International Requirements Engineering Conference, S. 23-32 (2008)

[OWL] **Web Ontology Language, W3C Spezifikation:**
http://www.w3.org/TR/owl-features/
(Stand: 12.11.2009)

[OWLECORE] **OWL-Metamodell:**
http://www.w3.org/2007/OWL/wiki/MOF-Based_Metamodel
(Stand: 17.06.2010)

[PROTEGE] **Protégé:**
http://protege.stanford.edu/
(Stand: 30.07.2010)

[RBFB2010] **Riebisch, Matthias; Bode, Stephan; Farooq, Qurat-Ul-Ann; Brcina, Robert:**
Types of Dependency Relationships for Change Impact Analysis and Related Tasks
Technical Report, TU Ilmenau, 2010 (to appear)

[RKA07] **Rochimah, Siti; Kadir, Wan M. N. Wan; Abdullah, Abdul H.:**
An Evaluation of Traceability Approaches to Support Software Evolution
In: 2nd International Conference on Software Engineering Advances,
France: IEEE Computer Society, 2007

[ROH06] **Rohe, Daniel:**
Konsistenzprüfung von Modellen auf Konformität mit Komponentenmodellen
Diplomarbeit, TU Ilmenau, 2006

[RQZ07] **Rupp, Chris; Queins, Stefan; Zengler, Barbara:**
UML 2 Glasklar – Praxiswissen für die UML-Modellierung
3. Auflage, Hanser Verlag, 2007

[SEW09] **Schwarz, Hannes; Ebert, Jürgen; Winter, Andreas:**
Graph-based traceability: a comprehensive approach
In: Software and Systems Modeling Volume9, Number 4, S. 473-492,
Berlin / Heidelberg, Springer, November 2009

[SGZ03] **Spanoudakis, George; Garcez, A. d'Avila; Zisman, Andrea:**
Revising Rules to Capture Requirements Traceability Relations:
A Machine Learning Approach
In: Proceedings of the 15th International Conference in Software Engineering and Knowledge Engineering (SEKE 2003), San Francisco, USA, July 2003

[SZ05] **Spanoudakis, George; Zisman, Andrea:**
Software Traceability: a roadmap
In: Chang, S.K. (ed) Handbook of Software Engineering & Knowledge Engineering: Recent Advances, Vol. 3, S. 395–428. World Scientific Publishing Company, River Edge, NJ (2005)

[SZMK04] **Spanoudakis, George; Zisman, Andrea; Pérez-Miñana, Elena; Krause, Paul:**
Rule-based generation of requirements traceability relations
In: Journal of Systems and Software, 72(2), S. 105-127, 2004

[TIGEREMF] **Tiger EMF Transformation Project:**
http://user.cs.tu-berlin.de/~emftrans/
(Stand: 08.01.2010)

[TGRAPH] **TGraphen:**
http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/Graphentechnologie/TGraphen
(Stand: 30.03.2009)

[UML] **Unified Modeling Language, OMG Spezifikation:**
http://www.uml.org/
(Stand: 06.02.2010)

[UML2TOOLS] **Eclipse Modeling - UML2 Tools:**
http://www.eclipse.org/modeling/mdt/downloads/?project=uml2tools
(Stand: 19.06.2009)

[URN] **User Requirements Notation, ITU Spezifikation:**
http://www.itu.int/itudoc/itu-t/aap/sg17aap/history/z150/z150.html
(Stand: 15.01.2003)

[XML2E] **XML Schema To Ecore Mapping:**
www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf
(Stand: 28.06.2004)

[XQUERY] **XML Query Language, W3C Spezifikation:**
http://www.w3.org/TR/xquery/
(Stand: 23.01.2007)

[XSLT] **Extensible Stylesheet Language Transformation, W3C Spezifikation:**
http://www.w3.org/TR/xslt20/
(Stand: 23.01.2007)

[ZEF2000] **Zisman, Andrea; Emmerich, Wolfgang; Finkelstein, Anthony:**
Using XML to Build Consistency Rules for Distributed Specifications
In: Proceedings of 10th International Workshop on Software Specification and Design - IWSSD-10, San Diego, USA, November, 2000

IV. Abbildungsverzeichnis

V. Anhang

Der Anhang dieser Diplomarbeit gliedert sich wie folgt:
Teil A listet die Installationsvoraussetzungen für EMFTrace auf. Anhang B erklärt kurz die wesentlichen Schritte bei der Installation von EMFStore, Teil C gibt die notwendigen Schritte für die Installation von EMFTrace an und Anhang D erläutert das Einrichten und Konfigurieren des EMFStore Servers und des EMFStore Clients. Anhang E gibt einen kurzen Überblick über die wichtigsten Arbeitsschritte mit dem EMFStore Standard-Client. Anhang F ergänzt diese Übersicht um alle Erweiterungen, die durch EMFTrace ergänzt werden. In Anhang G wird das Einbinden eines neuen Metamodells in EMFStore schrittweise erläutert. Es folgen tabellarische Übersichten über Linktypen in Anhang H und Traceability-Linkregeln in Anhang I und eine Übersicht über den Inhalt der Begleit-CD in Anhang J. Abschließend gibt Anhang K einen Überblick über die Kernkomponenten von EMFTrace in Form eines UML-Komponentendiagramms.

A. Installationsvoraussetzungen

Folgende Software muss installiert sein, um EMFTrace nutzen zu können:

1. Eclipse Galileo 3.5.2
2. Ecore Tools SDK 0.9.0
3. Eclipse Modeling Framework SDK 2.5.0
4. Java JDK 1.6.0 und Java JRE6
5. EMFStore 0.6.2 (Installationsanleitung siehe Anhang B)

B. Installation von EMFStore

1. Öffnen Sie den Update-Manager von Eclipse und fügen Sie eine neue Update-Seite ein:
   http://unicase.googlecode.com/svn/updatesite/emfstoreNightly
2. Installieren Sie die folgenden Features:
   - COPE Runtime Feature
   - EMFStore Backchannel
   - EMFStore Client Feature
   - EMFStore Client UI Feature
   - EMFStore Metamodel
   - EMFStore SDK
   - EMFStore Server Feature

C. Installation von EMFTrace

1. Installieren Sie zuerst EMFStore (siehe Anhang B)
2. Öffnen Sie das Verzeichnis `\Installationsdateien\` auf der beiliegenden CD
3. Entpacken Sie den kompletten Inhalt des EMFTrace.zip-Archivs in das Installationsverzeichnis Ihrer Eclipse Entwicklungsumgebung
4. Richten Sie den EMFStore Server und EMFStore Client entsprechend Anhang D ein

D. Einrichten von EMFStore/EMFTrace

1. Legen Sie eine neue Launch/Debug-Konfiguration für den EMFStore Server an:
    1. Launch/Debug -> Launch Configurations/Debug Configurations
    2. Wählen Sie „Eclipse Application“ aus und fügen Sie einen neuen Eintrag hinzu
    3. Nennen Sie die Konfiguration „EMFStore Server“
    4. Wählen Sie den Tab „Main“ aus und wählen Sie bei „Run a product“ „org.unicase.emfstore.server“ aus
    5. Fügen Sie im „Plug-ins“ Tab alle eigenen Metamodell Plug-ins hinzu*
2. Legen Sie eine neue Launch/Debug-Konfiguration für den EMFStore Client an:
    1. Launch/Debug -> Launch Configurations/Debug Configurations
    2. Wählen Sie „Eclipse Application“ aus und fügen Sie einen neuen Eintrag hinzu
    3. Nennen Sie die Konfiguration „EMFStore Client“
    4. Wählen Sie den Tab „Main“ aus und wählen Sie bei „Run a product“ „org.eclipse.platform.ide“ aus
    5. Fügen Sie im „Plug-ins“ Tab alle eigenen Metamodell Plug-ins hinzu*
3. Starten Sie den EMFStore Server
4. Starten Sie den EMFStore Client
5. Aktivieren Sie die beiden Views „Unicase Navigator“ und „EMFStore Browser“
6. Wählen Sie im „EMFStore Browser“ das Repository „unicase.in.tum.de“ aus:
    1. Rechtsklicken Sie auf das Repository, dann auf „New Repository“
    2. Geben Sie als Name und URL „localhost“ ein
    3. Wählen Sie als Port 8080 (Unicase Standard-Port)
    4. Wählen Sie als Zertifikat „unicase.org test test(!!!) certificate“ aus
7. Rechtsklicken Sie auf das neuerstellte Repository:
    1. Klicken Sie auf „Login“
    2. Geben Sie den Benutzernamen „super“ und das Passwort „super“ ein

* aktuell müssen folgende Plug-ins eingebunden werden:
EMFTrace_UML, EMFTrace_UML.edit, EMFTrace_OWL, EMFTrace_OWL.edit, EMFTrace_URN, EMFTrace_URN.edit, EMFTrace_TraceLink, EMFTrace_TraceLink.edit, EMFTrace_TraceRule, EMFTrace_TraceRule.edit, EMFTrace_Core**, EMFTrace_GUI**

** wird nur im EMFStore Client benötigt

E. Arbeiten mit dem EMFStore Client

- Anlegen neuer Projekte:
    1. Rechtsklicken sie auf ihr Repository, wählen Sie „Create new project“ aus.
    2. Geben Sie einen Projektnamen und eine Beschreibung (optional) an.
    3. Rechtsklicken Sie auf das neue Projekt und wählen Sie „Checkout“ aus.

- Import von Unicase-konformen Modellen:
    1. Rechtsklicken Sie auf ihr Projekt, dann „Other -> Import -> Import Modelelement“.
    2. Wählen Sie ihr Modell aus und klicken Sie auf „Ok“.

F. Arbeiten mit EMFTrace

- Import von Modellen aus CASE-Werkzeugen:
    1. Wählen Sie das Projekt aus, in dass das Model importiert werden soll.
    2. Rechtsklicken Sie auf das Projekt und wählen Sie „Import from CASE-Tool“ aus.
    3. Legen Sie den Dateityp fest (*.uml, *.z151, *.owl).
    4. Wählen Sie das Modell aus und schließend Sie den Importvorgang mit „Ok“ ab
        - Erscheint das Modell daraufhin nicht im Projekt, ist ein Fehler beim Import aufgetreten. Die Fehlerursache kann dabei dem Log entnommen werden, welches auf der Eclipse Konsole mitläuft.

- Export von Modellen in das ursprüngliche CASE-Werkzeug:
    1. Wählen Sie im Projekt das Modell aus, das exportiert werden soll.
    2. Rechtsklicken Sie auf das Modell.
    3. Je nach Modelltyp, habe Sie die Möglichkeit das Modell entweder als *.uml, *.z151 oder als *.owl-Datei mittels „Export as […] Diagram“ zu exportieren.

- Import von Regeln und Regelkatalogen:
    1. In Analogie zu „Import von Modellen aus CASE-Werkzeugen“ können auch einzelne Regeln oder Regelkataloge in ein Projekt importiert werden. Wählen Sie dazu jedoch anstelle von „Import from CASE Tools“ die Funktion „Import TraceRule“ aus.

- Ausführen von Transitivitätsanalysen:
    1. Selektieren Sie ein Projekt und öffnen Sie das Kontextmenü über einen Rechtsklick.
    2. Wählen Sie die Funktion „Perform Transitivity-Analysis“ aus.

- Anwenden von Regeln auf Projekte:
    1. Selektieren Sie ein Projekt und öffnen Sie das Kontextmenü über einen Rechtsklick.
    2. Wählen Sie die Funktion „Apply Rules“ aus.
    3. Folgen Sie den Konfigurationsschritten des Assistenten.

- Validierung von Traceability-Links:
    1. Selektieren Sie ein Projekt und öffnen Sie das Kontextmenü über einen Rechtsklick.
    2. Wählen Sie die Funktion „Validate Traceability-Links“ aus.

- Validierung von Trace-Elementen:
    1. Selektieren Sie ein Projekt und öffnen Sie das Kontextmenü über einen Rechtsklick.
    2. Wählen Sie die Funktion „Validate Trace-Elements“ aus.

Für jede der hier beschriebenen Funktionen steht jeweils ein kleines Testprojekt mit vorbereiteten Modellelementen zur Verfügung. Diese Projekte können in den aktuellen Unicase Workspace importiert werden (Other → Import → Import Projectspace) und auf Wunsch in den Projektserver eingecheckt werden. Die Projekte befinden sich auf der beliegenden CD im Unterverzeichnis `\Tutorial-Projekte\`. Desweiteren befindet sich im Verzeichnis `\Tutorials\` eine Sammlung von Anleitungen, die die Erzeugung und den Import von Regeln anhand von Screenshots näher erläutern.

G. Einbindung neuer Metamodelle in EMFStore/EMFTrace

1. Legen Sie ein neues EMF Projekt an:
    1. File -> New -> Project -> Eclipse Modeling Framework -> EMF Project
    2. Geben Sie einen passenden Projektnamen ein, z.B. „UML_2_EMFSTORE“
    3. Wählen Sie nun die Form des Modellimportes aus
    4. Klicken Sie auf „Browse File System“ und wählen Sie ihr Metamodell aus
    5. Beenden Sie die Projekterstellung mit einem Klick auf „Finish“
2. Wählen Sie nun Ihr neues Ecore-Modell aus und klicken Sie rechts in das geöffnete Modell:
    1. Klicken Sie auf „Load Resource“, dann „Browse Registered Packages“
    2. Suche Sie nach „*meta“ und wählen Sie in den Suchergebnissen „http://unicase.org/metamodel“ aus und laden diese Ressource
3. Selektieren Sie nun das Basiselement ihres Metamodells und fügen eine neue Klasse hinzu:
    1. Nennen Sie die Klasse *{NameIhresMetamodells}Base*, z.B.: „*UMLBase*“
    2. Setzen Sie die Property „Abstract“ auf True
    3. Wählen Sie nun die Basisklasse:
        - Klicken Sie auf „ESuper Types“
        - Wählen Sie „ModelElement -> Identifiable Element“ aus
4. Leiten Sie alle Elemente ihres Metamodells ohne Basisklasse von der neu erstellten Klasse ab
5. Rechtsklicken Sie im Projektexplorer auf Ihr Generatormodell und wählen Sie „Reload“
6. Öffnen Sie die Properties des Generatormodells und suchen Sie unter dem Menüpunkt „Model“ nach dem Eintrag „Containment Proxies“ und setzen Sie diese Einstellung auf „True“
7. Wählen Sie Ihr Generatormodell aus und klicken Sie auf „Generate Model Code“
8. Wählen Sie Ihr Generatormodell aus und klicken Sie auf „Generate Edit Code“
9. Compilieren und Erstellen Sie das Model- und das Edit Plug-in
10. Fügen Sie die 2 neuerstellten Plug-ins jeweils in die EMFStore Server und in die EMFStore Client Startkonfiguration hinzu

H. Traceability-Klassifizierung

Traceability-Kategorie	Quelle	Unterkategorien	Quelle
Refinement	[JZ09], [SZ05], [MPR07], [RBFB2010]	Extend Inherit Decomposition Specialization	[RBFB2010] [RBFB2010] [RBFB2010] [RBFB2010]
Generalization	[SZ05], [RBFB2010]	Super Is-A	[RBFB2010] [RBFB2010]
Composition	[RBFB2010]	Contains Part-Of Aggregation	[JZ09] [RBFB2010] [RBFB2010]
Realization	[MPR07]	Implements Instance-Of Type-Of Realizes	[JZ09] [RBFB2010] [RBFB2010]
Defintion	[MPR07], [RBFB2010]	Defines Asserts Contracts	[JZ09] [RBFB2010] [RBFB2010]
Condition	[RBFB2010]	Conforms Compatible Constraints	[RBFB2010] [RBFB2010] [RBFB2010]
Verification	[MPR07], [RBFB2010]	Tests Simulates Interprets	[RBFB2010] [RBFB2010] [RBFB2010]
Use	[RBFB2010]	Imports Uses Calls Activates Triggers Deactivates	[RBFB2010] [RBFB2010] [RBFB2010] [RBFB2010] [RBFB2010]
Dependency	[JZ09], [SZ05]	Requires Provides	[SZMK04] [RBFB2010]
Contribution	[SZ05], [RBFB2010]	Positiv Contribution Negative Contribution Satisfies	[RBFB2010] [RBFB2010] [JZ09], [SZ05]
Overlap	[FZS03], [JZ09], [SZMK04], [SZ05]	Similarities Equivalent	[JZ09]
Conflict	[SZ05]	Resolves Inconsistency	
Evolution	[JZ09], [SZ05]	Replaces Based-On Evolves-To	

I. Übersicht der Traceability-Beziehungen

Quellmodell	Typ	Zielmodell	Quelle	ID
GRL::Actor	Overlap	UML::Use-Case::Actor	[FZS03]	TraceRule01
GRL::Actor	Overlap	UML::Class	[FZS03]	TraceRule02
GRL:: Resource	Overlap	UML::Class	[FZS03]	TraceRule03
GRL::Task	Overlap	UML::Class::Operation	[FZS03]	TraceRule04
GRL::Goal	Overlap	UML::Class::Operation	[FZS03]	TraceRule04
GRL::Softgoal	Overlap	UML::Class::Operation	[FZS03]	TraceRule04
GRL::Goal	Overlap	UML::Use-Case	[FZS03]	TraceRule05
GRL::Softgoal	Overlap	UML::Use-Case	[FZS03]	TraceRule05
GRL::Task	Overlap	UML::Use-Case	[FZS03]	TraceRule05
UML::Use-Case	Overlap	UML::StateMachine	[JZ09]	TraceRule06
UML::StateMachine	Evolution	UML::StateMachine	[JZ09]	TraceRule07
UML::Sequence	Realizes	UML::Use-Case	[JZ09]	TraceRule08
UML::Sequence	Refinement	UML::Use-Case	[JZ09]	TraceRule09
UML::Sequence	Satisfies	UML::Use-Case	[JZ09]	TraceRule10
UML::Class	Implements	UML::Use-Case	[JZ09]	TraceRule11
UML::Sequence	Refinement	UML::Class	[JZ09]	TraceRule12
UML::Sequence	Uses	UML::Class	[JZ09]	TraceRule13
UML::StateMachine	Refinement	UML::Sequence	[JZ09]	TraceRule14
UML::StateMachine	Contains	UML::Class	[JZ09]	TraceRule15
UML::Class	Overlap	UML::StateMachine	[JZ09]	TraceRule16
UML::StateMachine	Satisfies	UML::Use-Case	[JZ09]	TraceRule17
UML::Class	Satisfies	UML::Use-Case	[JZ09]	TraceRule18
UML::Use-Case	Evolution	UML::Use-Case	[JZ09]	TraceRule19
UML::Class	Evolution	UML::Class	[JZ09]	TraceRule20
UML::Class	Dependency	UML::Use-Case	[JZ09]	TraceRule21
UML::StateMachine	Dependency	UML::Use-Case	[JZ09]	TraceRule22
UML::Sequence	Dependency	UML::Use-Case	[JZ09]	TraceRule23
UML::Activity	Realizes	UML::Use-Case	[RQZ07]	TraceRule24
UML::Activity	Realizes	UML::Class::Operation	[RQZ07]	TraceRule25
UML::Class	Refinement	UML::Package		TraceRule26
UML::CompositeStructure	Realizes	UML::Use-Case	[RQZ07]	TraceRule27
OWL::Class	Overlaps	UML::Class		TraceRule28
UML::Activity	Refinement	UML::UseCase		TraceRule29
UML::Class	Part-Of	UML::Collaboration		TraceRule30
UML::Class	Implements	UML::Interface		TraceRule31
UML::Class	Implements	UML::Use-Case::Actor	[RQZ07]	TraceRule32
UML::Sequence	Refinement	UML::Class::Operation	[RQZ07]	TraceRule33
UML::Sequence	Refinement	UML::Class::Attribute	[RQZ07]	TraceRule34
UML::Comment	Overlaps	URN::Coment		TraceRule35
UML::Comment	Overlaps	OWL::Literal		TraceRule36
URN::Comment	Overlaps	OWL::Literal		TraceRule37
UML::Association	Equivalent	URN::ElementLink		TraceRule38
UML::Association	Equivalent	URN::ElementLink		TraceRule39
UML::Association	Equivalent	URN::ElementLink		TraceRule40
OWL::SubClassOf	Equivalent	UML::Generalization		TraceRule41
OWL::SubClassOf	Equivalent	UML::Generalization		TraceRule42

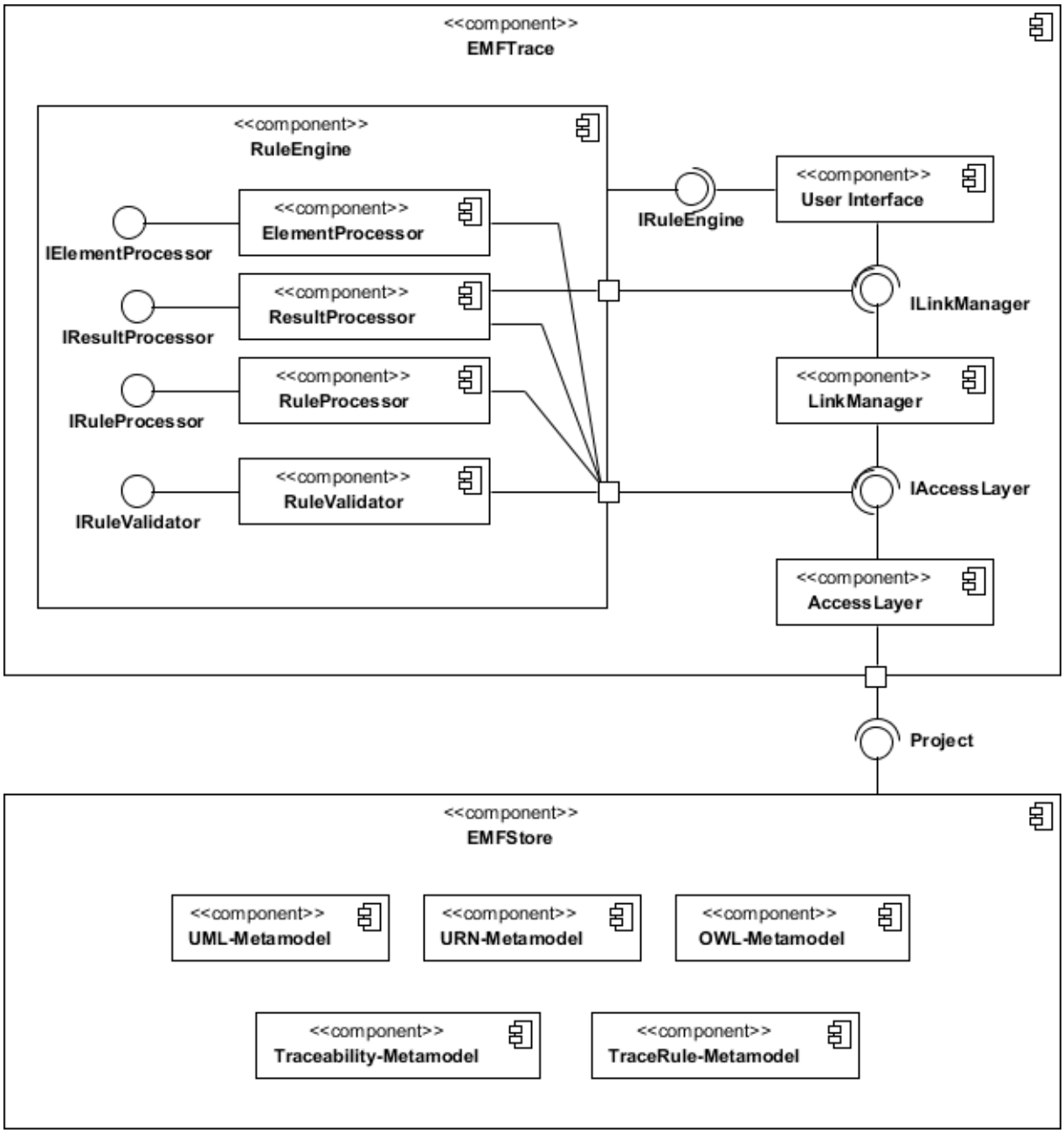
Quellmodell	Typ	Zielmodell	Quelle	ID
OWL::SubClassOf	Equivalent	UML::Generalization		TraceRule43
OWL::ObjectProperty	Equivalent	UML::Property		TraceRule44
OWL::DataProperty	Equivalent	UML::Property		TraceRule45
OWL::Datatype	Equivalent	UML::DataType		TraceRule46
OWL::Datatype	Equivalent	UML::Class		TraceRule47
URN::Goal/Softgoal	Contribution	URN::Goal/Softgoal		TraceRule48
URN::Goal/Softgoal	Decomposition	URN::Goal/Softgoal		TraceRule49
OWL::ObjectIntersectionOf	Overlap	UML::Class		TraceRule50
OWL::ObjectUnionOf	Overlap	UML::Class		TraceRule51
UML::Class	Part-of	UML::Package		TraceRule52
UML::StateMachine	Refinement	UML::UseCase	[JZ09]	TraceRule53
UML:: Sequence	Uses	UML::Class		TraceRule54
UML:: Sequence	Uses	UML::Interface		TraceRule55
UML:: Sequence	Uses	UML::Component		TraceRule56
UML:: Sequence	Uses	UML::Interface		TraceRule57
UML::Component	Part-Of	UML::Component		TraceRule58
UML::Interface	Part-Of	UML::Component		TraceRule59
URN::Goal/Softgoal	Equivalent	OWL::Class		TraceRule60
OWL::Class	Overlap	UML::Component		TraceRule61
UML::Interface	Satisfies	UML::UseCase		TraceRule62
OWL::Class	Is-A	OWL::Class		TraceRule63
UML::Component	Requires	UML::Interface		TraceRule64
UML::Lifeline	Activates	UML::Lifeline		TraceRule65
UML::Lifeline	Deactivates	UML::Lifeline		TraceRule66
UML::Lifeline	Uses	UML::Lifeline		TraceRule67
UML::Interface	Similarities	OWL::Class		TraceRule68

J. Inhalt der Begleit-CD

Auf der Begleit-CD befinden sich folgende Unterverzeichnisse und Inhalte:

\Diplomarbeit\	Enthält diese Ausarbeitung im PDF-Format und als MS Word Datei.
\Entwurfsmodelle\	Enthält Entwurfsdiagramme für die Eclipse UML2 Tools, Visual Paradigm und Microsoft Visio.
\Evaluierung\	Enthält die Evaluierungsprojekte und Evaluierungsergebnisse.
\Grafiken\Entwurfsdiagramme\	Enthält alle in dieser Arbeit verwendeten Entwurfsdiagramme als Bilddateien.
\Grafiken\Screenshots\	Enthält alle in dieser Arbeit verwendeten Screenshots.
\Installationsdateien\	Enthält alle zur Installation der EMFTrace-Anwendung benötigten Dateien.
\Quelldateien\	Enthält den kompletten Quellcode der EMFTrace-Anwendung, sowie dazugehörige Testfälle.
\Tutorials\	Enthält Tutorials zu verschiedenen Themen, u.a. zur EMFTrace-API.
\Tutorial-Projekte\	Enthält eine Sammlung von Beispielprojekten.
\Traceability-Regeln\	Enthält den in dieser Arbeit verwendeten Traceability-Regelkatalog.
\Validierungsprojekte\	Enthält Validierungsprojekte für Traceability-Regeln.
\XSLT-Templates\	Enthält alle Konvertierungstemplates für die Einbindung von Modellen in EMFStore.

K. Kernkomponenten von EMFTrace



VI. Thesen

1. Traceability-Links sind ein wertvolles Mittel, um Abhängigkeiten und Beziehungen zwischen verschiedenen Entwurfsmodellen auszudrücken.

2. Die Suche nach Traceability-Beziehungen in umfangreichen Projekten ist komplex und zeitaufwändig und sollte daher automatisiert erfolgen.

3. Bei der regelbasierten Suche nach Traceability-Beziehungen können die dabei erzeugten Traceability-Links exakt klassifiziert werden.

4. Die Mächtigkeit von Traceability-Regeln lässt sich durch Methoden und Algorithmen des Information Retrieval sinnvoll ergänzen.

5. Traceability-Typen sollten als beliebig spezialisierbare Modellelemente in das Repository eingebunden und versioniert werden.

6. Die Verwendung abstrakter Trace-Elemente ist eine Voraussetzung für die schrittweise Verfeinerung von Architekturwissen beim Reengineering und die Speicherung transitiv verknüpfter Traceability-Links.

7. Traceability-Regeln und deren Interpreter lassen sich auch für die Konsistenzprüfung von Modellen verwenden.

8. Regeln zur Traceability-Erzeugung und Konsistenzprüfung sollten jeweils in einem eigenständigen Katalog zusammengefasst und ständig erweitert und gepflegt werden.

9. Ein Repository mit modellübergreifender Analysefähigkeit ist eine wertvolle Unterstützung, um den Einfluss von Veränderungen an einem Softwareartefakt zu untersuchen.

10. Ein Repository für Entwurfsmodelle mit Import- und Exportfunktionalität kann zu einem zentralen Knoten im Softwareentwicklungsprozess werden.

VII. Erklärung

Ich erkläre, dass ich die hier vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Ich versichere, dass ich das Thema dieser Diplomarbeit bisher weder im In- noch im Ausland in irgendeiner Form als Prüfungsleistung vorgelegt habe.

Ilmenau, den 19. November 2010 Steffen Lehnert