Technische Universität München
Fakultät für Informatik

Diplomarbeit

Integration einer Regel-Engine in das Editor- und Publikationssystem 4Ever

Moritz Theile

Aufgabensteller:	Prof. Dr. Manfred Broy
Betreuer:	Sebastian Winter
Externer Betreuer:	Siegfried Schäfler
Abgabedatum:	15. November 2004

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

(Moritz Theile)

München, den 15.November 2004

Inhalt

1. Einleitung

1.1. Motivation

Unter einem „Dokument“ verstand man vor der Zeit des Personalcomputers „alle Unterlagen die Informationen beinhalten, also nicht nur publiziertes Wissen, sondern auch Briefe, Akten, Urkunden, Bildsammlungen, Filme u.v.a.“ [BROC1968].

Die Vorteile bei der Dokumentenbearbeitung mit Hilfe des Computers sorgten für eine schnelle Verbreitung von Programmen für diesen Zweck. Ein wesentlicher Vorteil von Textverarbeitungsprogrammen gegenüber einer Schreibmaschine bestand in der Möglichkeit, das Dokument jederzeit beliebig verändern zu können. Man erkannte schnell, dass der Computer bei der Bearbeitung eines Dokumentes noch viel umfangreicher genutzt werden kann. Textverarbeitungsprogramme wurden nach und nach mit immer mehr Hilfsfunktionen erweitert, wie zum Beispiel Formatierung oder Rechtschreibkontrolle. Durch ständige Verbesserung ermöglichen moderne Textverarbeitungsprogramme inzwischen eine höchst komfortable Textbearbeitung. An einem Punkt scheitern sie aber alle: Sie verstehen nicht *was* in einem Dokument steht und können deswegen keine inhaltliche Unterstützung bei der Dokumentenbearbeitung bieten.

Ein erster wichtiger Schritt zur Lösung dieses Problems wurde in den achtziger Jahren vollzogen und bestand darin, zwischen fachlichem Inhalt, logischer Struktur und graphischem Format zu unterscheiden[BRUE]. Die logische Struktur kann in gewisser Weise als Metamodell[1] eines Dokumentes gesehen werden. In Metamodellen können, neben der logischen Struktur, auch alle anderen formalisierbaren Eigenschaften beschrieben werden. Diese Informationen können so umfassend sein, dass in einigen Bereichen sogar eine fachliche Prüfung möglich wird. Bei einfachen Dokumenten wie einem Brief macht das sicherlich keinen Sinn. Betrachtet man als Beispiel aber den Konstruktionsplan eines Passagierflugzeugs, der durch die mannigfaltigen Abhängigkeiten der Komponenten zweifellos ein äußerst komplexes Dokument darstellt, erkennt man die Notwendigkeit einer umfangreichen automatischen Unterstützung. Eine Abhängigkeit zwischen den Komponenten könnte in diesem Beispiel die Beziehung zwischen der Spannweite und dem maximal zulässigen Gesamtgewicht des Flugzeuges sein. Aber was nützt es den Konstrukteuren, wenn ihrem Konstruktionsplan ein Metamodell zugrunde liegt?

Hierzu ein Beispiel: Angenommen, das Metamodell enthält den Zusammenhang zwischen Spannweite und maximalem Gesamtgewicht eines Flugzeuges, und es ist beschrieben, wie sich diese Informationen aus dem Dokument gewinnen lassen, dann könnte ein Programm die Konstrukteure warnen, wenn das Maximalgewicht überschritten wird. Wenn das Metamodell zusätzlich die Position und das Gewicht der Einzelteile kennt, könnte in dem Metamodell noch eine Regel definiert werden, die eine Warnung ausgibt, wenn der Schwerpunkt des Flugzeuges nicht mehr stimmt.

Ist ein ausführliches Metamodell vorhanden, können Programme also auch eine umfangreiche inhaltliche Unterstützung bei der Dokumentenbearbeitung geben.

Man könnte auch so weit gehen und dem Programm erlauben, selbstständig Änderungen an dem Dokument vorzunehmen. Beispielsweise könnte das Metamodell dem Konstrukteur

[1] Ein Metamodell ist das Modell eines Modells. Da das Dokument auch eine Art Modell ist, soll das ihm zugrunde liegende Modell immer als Metamodell bezeichnet werden.

Arbeit abnehmen, indem es Ergänzungen vornimmt die sich aus dem bisherigen Zustand des Dokumentes ergeben. Wurde eine Tragfläche konstruiert, so könnte das Programm automatisch eine gespiegelte Tragfläche erzeugen.

Metamodelle sind also ein wichtiges Hilfsmittel bei der Erstellung von komplexen Dokumenten. Durch sie lassen sich viele Dokumentenbearbeitungsprozesse erheblich verbessern. Diese Prozesse sind in der Geschäftswelt alltäglich. Ob es sich um die Erstellung eines Kataloges, einer Bestellung, einer Visitenkarte, oder im IT-Bereich um Projekthandbücher, Vorgehensmodelle oder Entwurfsdokumente handelt, ein zugrunde liegendes Metamodell ist notwendig, damit ein Computer die Daten differenziert behandeln kann. Je nach Anwendungsgebiet lässt sich dieses „gesteigerte Verständnis“ des Computers vorteilhaft nutzen.

Durch die bereits erwähnte Trennung von fachlichem Inhalt, logischer Struktur und graphischem Format können Dokumente in unterschiedlichste Ausgabeformate gebracht werden, weil dem Computer auf der Metamodellebene beigebracht werden kann, was wie dargestellt werden soll.

Aus einem Flugzeugkonstruktionsplan könnte man zum Beispiel eine einfache Liste der Bauelemente erstellen. Genauso gut könnte man aber auch eine Grafik mit farblich hervorgehobener Massenverteilung erzeugen. In diesem Fall wäre wahrscheinlich nicht nur das Ausgabeformat ein anderes, sondern man würde eine fachlich komplett andere Sicht auf das Projekt erhalten. Ein nicht zu vernachlässigender Punkt ist das Verbesserungspotenzial für die Arbeit im Team durch eine modellgetriebene Dokumentenbearbeitung. Die Strukturinformationen ermöglichen den Einsatz von Konfigurationsmanagement-Werkzeugen. Mit ihrer Hilfe unterliegen die Änderungen einer Versionskontrolle und parallel existierende Versionen verteilter Teams können zusammengeführt werden. Denkbar ist auch der Einsatz von einem Rechteverwaltungssystem, das den Benutzern nur den Zugriff auf bestimmte Bereiche des Dokumentes gestattet. Würde man den Konstruktionsplan im Microsoft-Word-Format zur Bearbeitung herumschicken, könnte ein übereifriger Innenraumdesigner auf die fatale Idee kommen, aus ästhetischen Gründen die Spannweite des Flugzeuges zu ändern.

Die Firma 4Soft hat mit 4Ever ein Programm entwickelt, das genau diese modellgetriebene Dokumentenbearbeitung unterstützt. Bisher sind durch die zugrunde liegenden Metamodelle allerdings nur strukturelle Informationen gegeben. Ziel dieser Diplomarbeit ist es, die Modellierungsmöglichkeiten durch einen Mechanismus zu erweitern, der die Definition von weitergehenden Regeln und deren Anwendung auf das Dokument erlaubt.

Nach Abschluss der Diplomarbeit sollten mit 4Ever beispielsweise fachliche Prüfungen, wie das oben beschriebene Maximalgewichtsproblem, möglich sein.

1.2. Die Firma 4Soft GmbH

Gegründet wurde 4Soft von Dr. Klaus Bergner, Prof. Dr. h.c. Manfred Broy, Prof. Dr. Andreas Rausch und Dr. Marc Sihling aus dem Forschungsverbund Software-Engineering FORSOFT I der Technischen Universität München.

Geschäftsführer sind Dr. Klaus Bergner und Dr. Marc Sihling.

4Soft erbringt seinen Kunden Beratungsleistungen bei der Erarbeitung von IT-Strategien, bei modellbasiertem Software-Engineering und bei der Projektdurchführung selbst. Neben der Beratung in diesen Bereichen führt 4Soft auch eigenständig Projekte durch.

Ein wichtiges Spezialgebiet der Firma ist die Unterstützung von komplexen Entwicklungsprozessen. Zu einem Entwicklungsprozess gehört die Bearbeitung vieler umfangreicher Dokumente. Um diese konsistent zu den Vorgaben zu halten und die Bearbeitung im Team zu erleichtern, hat 4Soft den 4Ever-Editor entwickelt, der im Rahmen dieser Diplomarbeit mit einer Regel-Engine erweitert werden soll.

1.3. Zielsetzung

4Ever ist ein Editor- und Publikationssystem, das modellgetriebene Dokumentenbearbeitung auf Grundlage eines Metamodells unterstützt. Das Metamodell wird durch ein XML-Schema[2] beschrieben. Einige Aspekte können in einem XML-Schema nicht definiert werden. Ziel der Diplomarbeit ist es, eine Möglichkeit zu schaffen, mit der diese Aspekte definiert und von 4Ever überprüft werden können.

Die Erweiterung des Metamodells soll durch die Definition von Regeln erfolgen. 4Ever soll mit Hilfe einer Regel-Engine die Konsistenz des Dokumentes während der Dokumentenbearbeitung überwachen. Falls es bei der Bearbeitung zu Inkonsistenzen kommt, soll dem Dokumentenbearbeiter eine Hilfestellung zur Wiederherstellung der Konsistenz gegeben werden.

Im Hinblick darauf stellen sich folgende Teilaufgaben:

1) Analyse der Anforderungen und Erarbeitung eines Konzeptes für die Bedienoberfläche.
2) Erarbeitung und Dokumentation einer Software-Architektur für eine möglichst standardkonforme Integration von Regel-Engines in 4Ever sowie von Vorgaben und Richtlinien für die Formulierung entsprechender Regeln.
3) Implementierung eines Java-Prototyps auf Basis von 4Ever und einer frei verfügbaren Regel-Engine.
4) Umsetzung einiger Regeln im Rahmen eines Fallbeispiels.

1.4. Aufbau der Diplomarbeit

Der Schwerpunkt dieser Diplomarbeit ist die Entwicklung einer Softwarekomponente. Es ist also notwendig, das Problem zu analysieren und eine Lösung zu erarbeiten. Die hierfür notwendigen Grundlagen werden im ersten Teil der Diplomarbeit (Kapitel 1 bis 3) vermittelt. Der zweite Teil (Kapitel 4 bis 7) enthält das fachliche Konzept und eine Erläuterung der technischen Umsetzung.

Der Inhalt der einzelnen Kapitel wird im Folgenden beschrieben.

In dem ersten Kapitel wurde bereits ein grober Rahmen für die Diplomarbeit gesetzt und anhand eines verständlichen Beispiels eine Vorstellung der modellgetriebenen Dokumentenbearbeitung und den damit verbundenen Vorteilen vermittelt. Ein ziemlich genaues Bild der Aufgabenstellung ergibt sich durch die Zielsetzung der Diplomarbeit.

Das zweite Kapitel ist ein Grundlagenkapitel in dem einige Begriffe eingeführt werden und dem Leser bewusst gemacht wird, dass der modellgetriebenen Dokumentenbearbeitung ein uraltes Prinzip zugrunde liegt. Dieses Prinzip ist durch die Unterstützung von Computern

[2] http://www.w3.org/XML/Schema

lediglich wesentlich flexibler und differenzierter einsetzbar. Anschließend wird im zweiten Kapitel auf 4Ever, ein Programm zur modellgetriebenen Dokumentenbearbeitung eingegangen. Es wird der Mechanismus von 4Ever untersucht, der eine modellgetriebene Dokumentenbearbeitung ermöglicht. Durch diese genaue Betrachtung des Mechanismus werden seine Mängel konkretisiert und führen zu einem technischen Verständnis der Problemstellung. Abschließend wird ein reales Fallbeispiel aus dem Einsatzgebiet von 4Ever vorgestellt.

Das dritte Kapitel ist ebenfalls ein Grundlagenkapitel. In ihm wird zunächst allgemein und dann detailliert auf die Funktionsweise einer Regel-Engine eingegangen. Es deckt theoretische und praktische Aspekte ab. Insbesondere wird der RETE-Algorithmus besprochen, der sich für den Einsatz in einer Regel-Engines hervorragend nutzen läßt. Es ergibt sich, dass eine Regel-Engine als Grundlage für die Überwachung der Konsistenz eines Dokumentes zu seinem Metamodell sehr gut geeignet ist.

In den bisherigen Kapiteln wurden das Problem und seine Lösung sehr abstrakt betrachtet. Das vierte Kapitel stellt die Ergebnisse einer detaillierten Anforderungsanalyse vor. Die Analyse wurde anhand des im zweiten Kapitel eingeführten Fallbeispiels durchgeführt. Die festgestellten konkreten Anforderungen führen zu einem fachlichen Konzept.

Um das fachlich Konzept strukturiert vermitteln zu können, wird im fünften Kapitel zunächst die Problemstellung in mehrere Teilprobleme aufgeteilt. Für jedes dieser Teilprobleme wird ein fachlicher Lösungsansatz präsentiert. Abschließend werden Anwendungsfallszenarien durchgespielt, die die Zweckmäßigkeit der vorgestellten Lösungsansätze zeigen.

Das sechste Kapitel beschreibt, wie das in Kapitel fünf eingeführte fachliche Konzept technisch realisiert wurde. Zunächst wird auf die komponentenbasierte Architektur von 4Ever eingegangen, da dies die Grundvoraussetzung für eine Erweiterung von 4Ever ist. Anschließend wird die entwickelte Komponente detailliert erläutert. Für ihren Einsatz ist eine umfangreiche Konfiguration notwendig. Diese Konfiguration wird in einer Regelbasis definiert. Anhand des Fallbeispiels wird gezeigt, wie eine Erstellung dieser Regelbasis, unter Zuhilfenahme eines Programms mit graphischer Benutzeroberfläche, funktionieren könnte. Des Weiteren werden die einzelnen Elemente der Regelbasis detailliert besprochen.

Im siebten Kapitel wird das Endergebnis an den gestellten Anforderungen gemessen und ein Ausblick auf Erweiterungsmöglichkeiten gegeben.

2. Die modellgetriebene Dokumentenbearbeitung

2.1. Einführung

Modelle spielen in der Informatik eine zentrale Rolle. Ein Modell ist eine Abstraktion eines komplexen Systems. Durch diese Abstraktion, also dem Weglassen unwichtiger Aspekte, wird das beschriebene System vereinfacht. Es wird somit handlicher. Welche Aspekte bei der Modellierung berücksichtigt werden, hängt davon ab, welchem Zweck das Modell später dienen soll.

Modelle sind aber nicht erst seit der Informatik bekannt. Sie sind eine Entwicklung der Evolution. Jeder Mensch arbeitet in seinem Geist mit einem mentalen Modell der Umwelt. Durch dieses Modell ist der Mensch in der Lage die Welt zu verstehen.

Eine besondere Fähigkeit des Menschen ist es, anderen Menschen eine Idee zu übermitteln. Dieser Prozess wird Kommunikation genannt. Eine **Idee** soll dabei ein Teil des mentalen Modells eines Menschen sein.

Grundvoraussetzung für Kommunikation ist eine Darstellungsform der zu übermittelnden Idee. Für eine solche Darstellungsform können Sprache, Schrift, Bild, Gestik, Mimik usw. genutzt werden. Die natürlichen Darstellungsformen, wie die Umgangssprache, haben sich im Laufe der Zeit entwickelt und sind universell einsetzbar. Insbesondere sind sie hervorragend für die Kommunikation im alltäglichen Leben geeignet. Um jedoch komplexere Ideen darzustellen ist es notwendig, neue Darstellungsformen zu finden. Dies sind zum Beispiel Fachsprachen, Diagrammarten oder mathematische Formeln.

Eine immer wichtiger werdende Darstellungsform sind Metamodelle der modellgetriebenen Dokumentenbearbeitung. Sie definieren eine Klasse von Dokumenten, die für einen bestimmten Zweck geeignet sind. Metamodelle machen die logischen Strukturen und teilweise auch den fachlichen Inhalt schriftlicher Dokumente für ein Programm „verständlich“.

Das Metamodell eines Vorgehensmodells (siehe Fallbeispiel V200X-Vorgehensmodell S.26) legt zum Beispiel die einzelnen Bausteine des Vorgehensmodells und ihre Beziehungen untereinander fest. Bei der modellgetriebenen Dokumentenbearbeitung sorgt ein Programm für die Einhaltung der im Metamodell definierten Bedingungen. Diese Bedingungen können sich auf die logische Struktur oder andere Eigenschaften des Dokumentes beziehen.

Das Prinzip der Nutzung eines Metamodells zur Darstellung einer Idee ist nicht neu. Wahrscheinlich haben es die alten Ägypter schon bei der Planung ihrer Pyramiden benutzt. Vielleicht sind sie durch einen Modellierungsvorgang, der insbesondere die menschliche Fähigkeit der Abstraktion in Anspruch nimmt, zu dem Schluss gekommen, dass sich als Metamodell der „Idee Pyramide“ ein Baukastensystem aus vielen kleinen Würfeln eignet. Durch das Baukastenmetamodell ist man in der Lage, viele Bereiche der Problemdomäne Pyramidenbau zu klären. Man könnte herausfinden, wie viele Würfel man benötigt, wenn die Pyramide 50 Schichten hoch werden soll. Für diesen Zweck hätte man auch die Mathematik als Metamodell verwenden können. Dies wäre ein wesentlich abstrakterer Ansatz, wodurch viele Aspekte der Problemdomäne Pyramidenbau verloren gehen würden. Wollte man zum Beispiel wissen, wie der Schatten einer Pyramide aussieht, wäre das mathematische Modell nicht ausreichend.

Die mathematische Darstellung, mit der man die Menge der enthaltenen Würfel errechnen kann, enthält eine Teilmenge der Aspekte, die in der Würfeldarstellung enthalten sind. Man könnte also zu dem Schluss kommen, dass das mathematische Modell überflüssig ist. Neben der Frage der dargestellten Aspekte, stellt sich bei der Wahl eines Metamodells aber auch noch die Frage nach dem praktischen Nutzen. Würde der Einsatz von abstrahierenden Modellen keine Vorteile bringen, könnte man ganz auf sie verzichten. Gäbe es kein geeignetes Metamodell für Pyramiden und man will wissen aus wie vielen Würfeln sie bestehen, müsste man sie bauen. Man sieht aber leicht ein, dass dies an der praktischen Handhabung scheitert.

Durch den Bau einer Pyramide haben die Ägypter ein möglichst genaues Abbild von ihrer „Idee Pyramide“ geschaffen. Ist dies aber nicht auch ein Modell, sozusagen ein Modell ihrer Idee? Dieses Thema war schon bei den Philosophen der Antike von Interesse[3]. Während Platon behauptete, die Wirklichkeit wäre ein Abbild der Idee, behaupteten nachfolgende Philosophen, die Idee würde sich aus der Wirklichkeit ergeben [GAR]. Wir bleiben bei einer oberflächlichen Sicht der Dinge und legen fest, dass die Realität das Original ist und unser Geist ein mentales Modell davon enthält.

Die Computer haben uns die Möglichkeit gegeben, wesentlich freier in der Modellierung sein zu können. Es existieren nicht mehr nur die in der Natur vorhandenen Mittel zur Darstellung von Ideen. Man ist jetzt in der Lage, sich eigene Welten zu schaffen, welche die notwendigen Hilfsmittel zur Modellierung bereitstellen. Eine dieser Welten ist die Welt der Objektorientierung.

Um den Begriff der modellgetriebenen Dokumentenbearbeitung besser einordnen zu können soll dieser noch kurz diskutiert werden. Nach bisheriger Verankerung der Modell- und Metamodellebene müsste man korrekterweise „metamodellgetriebene Dokumentbearbeitung“ sagen, weil das Dokument auf der Modellebene angeordnet ist. Der Begriff der modellgetriebenen Dokumentenbearbeitung hat sich im allgemeinen Sprachgebrauch schon etabliert und soll daher auch hier weiterhin benutzt werden.

In dem einleitenden Kapitel wurde schon eine Definition für den Dokumentenbegriff aus den siebziger Jahren zitiert. In dieser Arbeit soll ein **Dokument** ganz allgemein die Darstellung einer Idee sein, die von einem Menschen interpretiert werden kann. Dokumente sind nicht flüchtig. Durch Bearbeitung lässt sich ein Dokument an eine Idee anpassen.

Bei der modellgetriebenen Dokumentenbearbeitung steht insbesondere der Gesichtspunkt des Getriebenen im Vordergrund. Die modellgetriebene Dokumentenbearbeitung basiert auf Metamodellen, in denen implizit schon sehr viel Wissen aus der Problemdomäne enthalten ist. Das Baukastenmetamodell enthält zum Beispiel das Pyramidenfachwissen, dass Pyramiden durch Steinblöcke errichtet werden. Dieses im Metamodell enthaltene Wissen hilft nun bei der Bearbeitung des Metamodells, indem es sie in die richtige Richtung „treibt“. Unmögliche Konstruktionen, wie weite Überhänge oder frei schwebende Bausteine, sind von vornherein ausgeschlossen. Sie lassen sich durch das Metamodell nicht ausdrücken. Versucht der Bearbeiter eine solche, laut Metamodell unmögliche Idee zu verwirklichen, wird er automatisch korrigiert, indem das Bausteinmodell einstürzt. In Abbildung 1 ist eine solche Situation dargestellt.

Je mehr Wissen ein Metamodell enthält, desto eingeschränkter, aber dafür auch brauchbarer, sind seine Ausprägungen. Bei der modellgetriebenen Dokumentenbearbeitung versucht man, möglichst viel Wissen in das Metamodell einzubringen, um möglichst zweckmäßige Ausprägungen zu erhalten.

[3] Universalienstreit: http://de.wikipedia.org/wiki/Universalienstreit

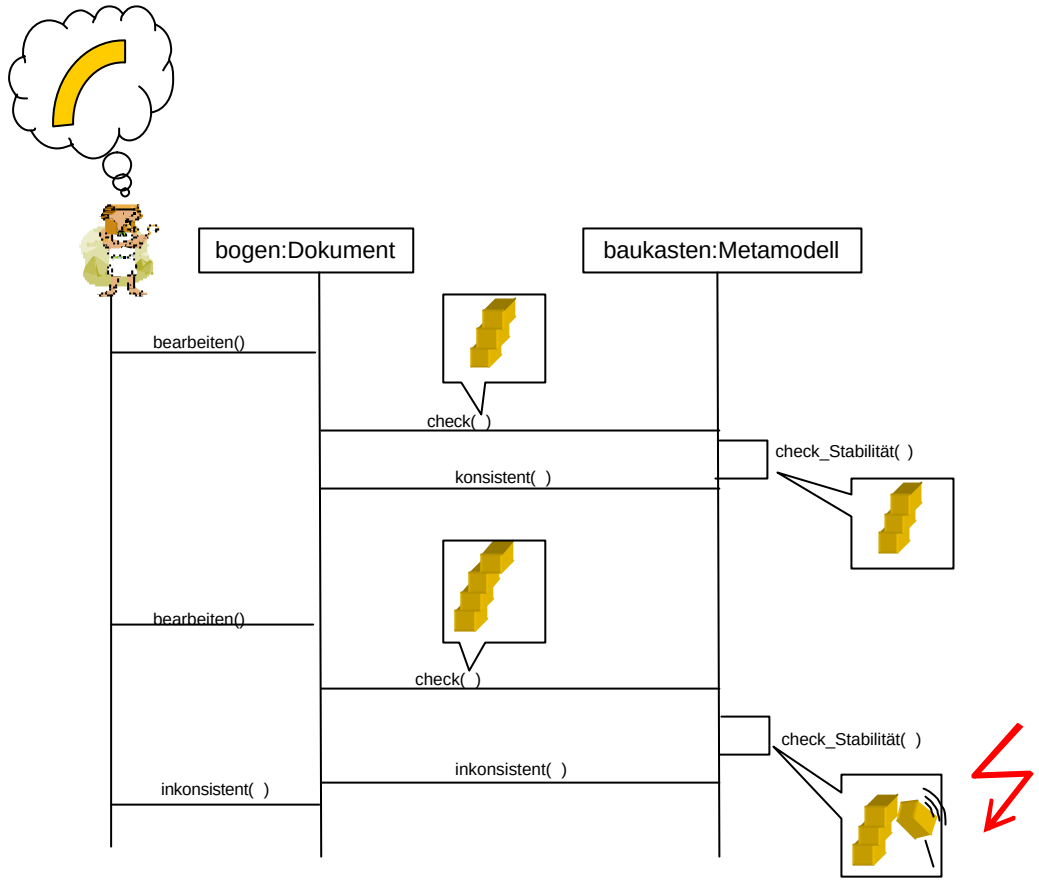

Abbildung 1: Ein Ägypter versucht sein mentales Modell einer Pyramide (eine Bogenform) mit Hilfe eines Baukastens darzustellen.

Ein weiteres, etwas näher an 4Ever liegendes Beispiel für modellgetriebene Dokumentenbearbeitung ist ein Bestellformular. Eine Bestellung enthält auf alle Fälle eine Empfängeradresse, eine Lieferadresse, eine Abrechnungsart mit entsprechenden Angaben, Artikel mit Anzahl und Preis, eine Gesamtsumme und eine Unterschrift. Durch die entsprechenden Felder ist dem Bearbeiter des Metamodells, also des Formulars, nicht mehr so viel Freiraum gelassen. Er wird beim Schreiben einer Bestellung durch das zugrunde liegende Metamodell, dem Formular, unterstützt bzw. „getrieben“. Man sagt, die **Bearbeitungssicherheit** steigt.

Die Menschen sind in der Lage mit Hilfe der Sprache zu kommunizieren. Die Sprache ist optimal, um einfache Sachverhalte schnell zu übermitteln. Die Sprache ist flüchtig, das heißt, dass das Modell einer Idee in Gedanken aufgebaut wird. Das ist auch kein Problem, solange sich das Modell mit unserer Sprache leicht beschreiben lässt und nicht zu komplex ist, um es sich vorzustellen. Problematisch wird es, wenn über einen komplizierten Sachverhalt über einen längeren Zeitraum mit möglicherweise vielen Menschen kommuniziert wird. Das Ergebnis werden unterschiedliche mentale Modelle des selben Sachverhalts sein.

In einem solchen Fall empfiehlt sich der Einsatz einer nicht flüchtigen Darstellungsweise, die von allen Beteiligten auf gleiche Weise interpretiert wird. Dieses Vorgehen wird für die Kommunikation häufig völlig selbstverständlich eingesetzt.

Ein Beispiel hierfür ist das „Mensch Ärger Dich Nicht“-Spiel. Man stelle sich die Schwierigkeiten vor, wenn man ohne Brett spielen würde. Obwohl allen Beteiligten die Idee

hinter diesem Spiel bekannt sein dürfte, würde ein Spiel ohne Brett wahrscheinlich aus folgenden Gründen scheitern:

- Es gibt zu viele Faktoren die man sich merken muss.
- Es gibt keine geeignete Sprache zur Beschreibung von Aktionen.

Ein Spielbrett, also ein Dokument, neutralisiert diese Faktoren. Das Spielbrett und die Figuren stellt die Spielsituation eindeutig dar. Aktionen werden den Mitspielern einfach durch die Bearbeitung des Dokumentes bzw. des Spielbretts mitgeteilt.

Man sieht, dass es für die Kommunikation wichtig ist, eine geeignete Darstellungsform zu haben. Überträgt man das Beispiel auf die modellgetriebene Dokumentenbearbeitung mit 4Ever, ist 4Ever ein Spielbrett, das man durch die Definition eines Metamodells für viele Einsatzzwecke konfigurieren kann. Zum Beispiel wurde 4Ever für den Entwurf eines Vorgehensmodells konfiguriert (siehe Das V200X-Metamodell S.27).

2.2. Modellgetriebene Dokumentbearbeitung mit 4Ever

4Ever ist eine von der Firma 4Soft entwickelte Software, welche modellgetriebene Dokumentenbearbeitung unterstützt. Das heißt, sie stellt eine Umgebung bereit, in der das Dokument nur innerhalb der durch das Metamodell gegebenen Einschränkungen bearbeitet werden kann. Die Bearbeitung des Dokumentes findet über eine graphische Benutzeroberfläche statt. (siehe Abbildung 2)

Auf der linken Seite des Fensters ist der **Strukturbaum** angeordnet, der eine schnelle Navigation zu den Elementen des Dokuments erlaubt. Auf diesem Strukturbaum sind die typischen Operationen (Kopieren, Einfügen, Löschen und Erstellen von Elementen) möglich. Es können nur Operationen angewendet werden, die zu einem weiterhin konsistenten Dokument führen (siehe Konsistenzerhaltung und Konsistenzwiederherstellung S.23). Auf der rechten Seite wird, je nach Typ des momentan im Strukturbaum aktivierten Elements, ein passender Editor geöffnet. Dieser Bereich wird **Editor-Panel** genannt. Er erlaubt die Bearbeitung der Attributwerte.

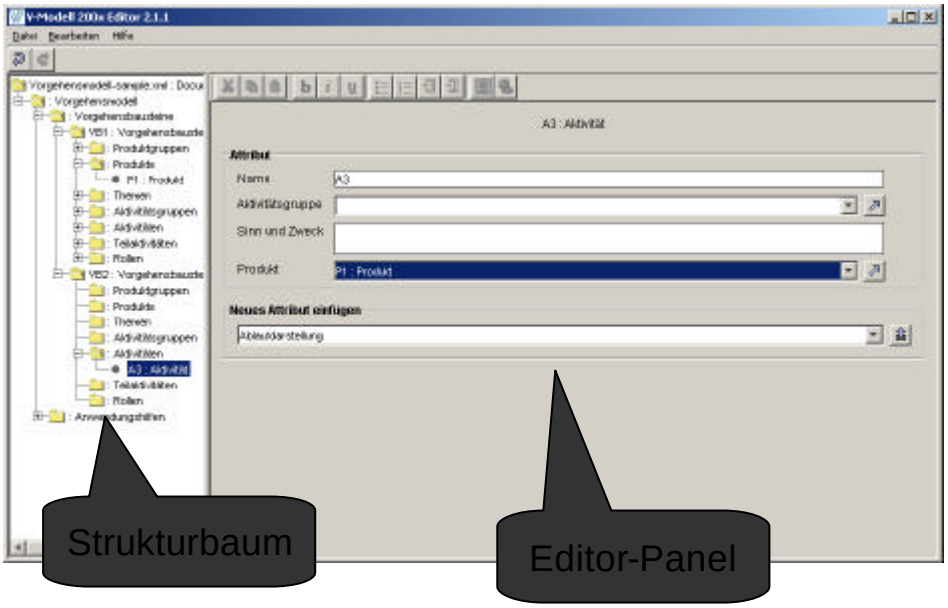

Abbildung 2: Benutzerinterface des Dokumentenbearbeitungssystem 4Ever

Weitere Eigenschaften von 4Ever sind für diese Arbeit eher nebensächlich, aber trotzdem erwähnenswert. Durch das zugrunde liegende Metamodell sind über das Dokument wesentlich mehr Informationen als bei normalen Textdokumenten erhältlich. Diese Informationen können von 4Ever dazu genutzt werden, verschieden Sichten des Dokumentes zu generieren. (z.B. HTML oder PDF) Außerdem werden die Informationen zur Anbindung an ein Versionsverwaltungssystem (z.B. CVS) benötigt. Durch ein Versionsverwaltungssystem können mehrere parallel existierende Versionen zu einer zusammengeführt werden. Das Zusammenführen von verschiedenen Versionen ist notwendig, wenn im Team mehrere Kopien des Dokumentes gleichzeitig bearbeitet wurden.

2.3. Das Objektmodell von 4Ever

Eine Besonderheit an 4Ever ist die generische Metamodellebene. Sie ist nicht im Quellcode festgeschrieben sondern kann flexibel durch Beschreibungssprachen instantiiert werden. Auf diese Weise ist es möglich, das zugrunde liegende Metamodell ohne Änderungen am Programmcode anzupassen oder gänzlich neu zu definieren.

In der aktuellen Version von 4Ever werden fast alle Aspekte des Metamodells durch die Typseite des **4Ever-Objektmodells** realisiert. Das 4Ever-Objektmodell verfügt über eine Typ- und eine Instanzseite. Das Objektmodel ist eine Komponente (siehe Integration von 4EverRulez in 4Ever S.60) von 4Ever.

Ein wesentlicher Ausschnitt des Objektmodells ist in Abbildung 4 als Klassendiagramm dargestellt. Die Trennung zwischen der Typ- und Instanzseite ist durch die gestrichelte Linie hervorgehoben. Beide Seiten benutzen das Composite-Muster [GAMMA] in abgewandelter Form. Die Originalstruktur des Musters ist in Abbildung 3 zu sehen.

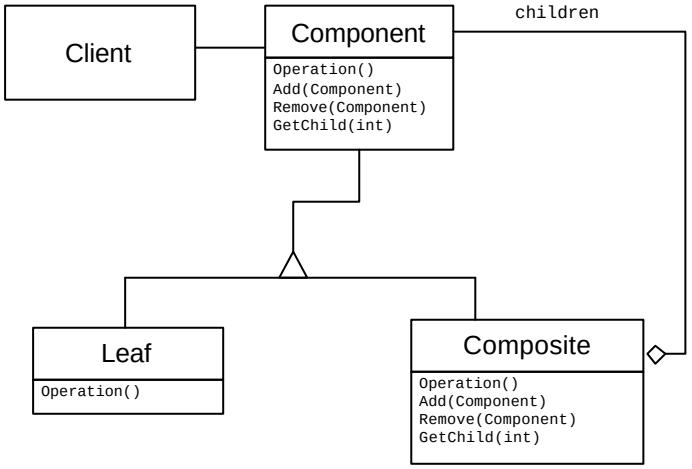

Abbildung 3: Das Klassendiagramm des Composite-Musters.

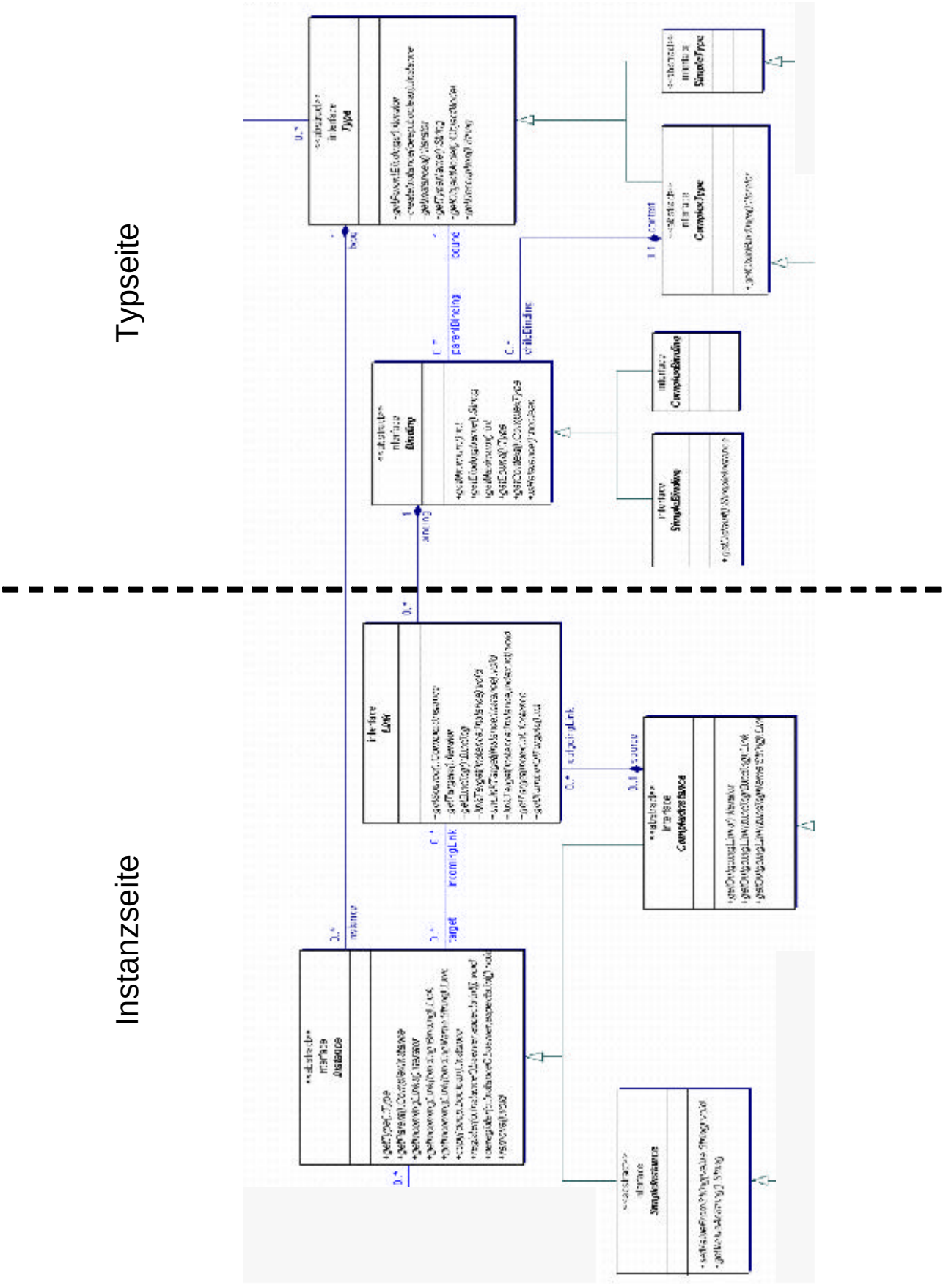

Abbildung 4: Das Objektmodell von 4Ever

Die „children“-Assoziation des Composite-Musters ist sowohl auf der Typ- als auch auf der Instanz-Seite des Objektmodells durch eine musterfremde Klasse realisiert. Auf der Typseite ist dies die `Bound`- und auf der Instanzseite die `Link`-Klasse. Durch sie lassen sich Zusatzinformationen zu den Assoziationen speichern. Diese Zusatzinformationen sind notwendig um eine Beziehung zwischen der Typ- und der Instanzseite herzustellen und um die Struktur des Metamodells besser abbilden zu können.

Bei dem Aufbau der Instanzseite wird jeder Instanz eines `Link` ein `Binding` und jeder `Instance` ein `Type` zugeordnet. Die `Types` kennen alle `Instances` die ihnen zugeordnet wurden. Wie viele `Links` es zwischen zwei `Instances` geben darf und wie viele es geben muss ist durch ein Minimum- und ein Maximumattribut in der Bindingklasse festgelegt.

Ein Dokument wird in dem Objektmodell durch Ausprägungen von `ComplexInstances`, `SimpleInstances` und `Links` dargestellt.

Man kann die `Links` als Kanten und die `Instances` als Knoten eines Graphen betrachten. Um was für eine Art von Graph es sich dann bei einem Dokument handeln kann hängt von der Ausprägung der Typseite ab. Sind nur komponierende Bindings definiert, dann erhält man einen oder mehrere Bäume, also einen Wald. Ein Wald ist ein Graph dessen Komponenten Bäume sind. [STEG, S.34]. Durch die Definition von referenzierenden Bindings erhält man nach obiger Betrachtungsweise einen beliebigen Graphen, weil keine einschränkenden Bedingungen an sie gestellt sind (siehe Abbildung 5).

Bevor die Instanzseite des Objektmodells genutzt werden kann, ist es unbedingt erforderlich, dass der Aufbau der Typseite schon vollständig erfolgt ist. Das ist notwendig, weil das Objektmodell keine evolutionäre Entwicklung der Typseite unterstützt. Mit einer **evolutionären Entwicklung** ist die Änderung der Typseite nach Aufbau der Instanzseite gemeint. Versucht man die Typseite nachträglich zu verändern, so erhält man eine Fehlermeldung (`InstanceEvolutionNotSupportedException`).

Für ein besseres Verständnis sollte hier auf die Parallelen zu dem Pyramidenbeispiel von weiter oben eingegangen werden. Dort besteht das Metamodell aus einer Art Baukasten. Das Metamodell in dem Objektmodell ist auch nichts anderes als ein Baukasten, der vorgibt, aus welchen Bausteinen (`Types`) das Dokument bestehen soll. Außerdem gibt er vor, welche und wie viele Bausteine auf was für eine Weise (komponierend oder referenzierend) miteinander kombinierbar sind (`Bindings`). Die nicht vorhandene evolutionäre Typentwicklung auf das Beispiel übertragen bedeutet, dass nach dem Zusammensetzen der ersten Bausteine eine Änderung des Baukasteninhalts nicht mehr möglich ist.

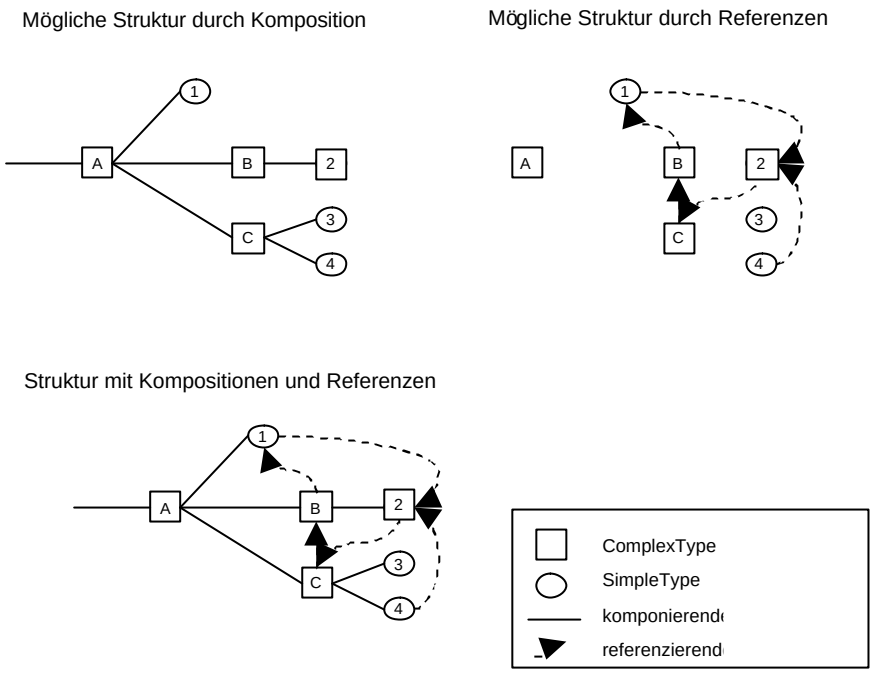

Abbildung 5: Mögliche Strukturen mit komponierenden und referenzierenden Links.

Durch die bisher vorgestellten Methoden ist das Objektmodell in der Lage, einen Teil des Metamodells von 4Ever zu beschreiben. Dieser Teil, der durch `Bindings`, `ComplexTypes` und `SimpleTypes` beschrieben werden kann, soll der **Strukturteil** des Metamodells sein. Ein weiterer Teil des Metamodells betrifft die möglichen Typen der `SimpleInstances`. Jeder Ableitung von der abstrakten `SimpleInstance`-Klasse ist eine Typklasse zugeordnet, die eine Ableitung der abstrakten `SimpleType`-Klasse ist. Auf diese Weise ist für eine Typisierung der `SimpleInstances` gesorgt. Das heißt, 4Ever weiß beispielsweise, ob es sich bei einem gespeicherten String um einen normalen Text oder einen HTML-String handelt. Durch dieses Wissen kann 4Ever den jeweiligen Typ im Editor-Panel (siehe Abbildung 2) in einer geeigneten Weise darstellen. In diesem Fall würde sich für den Text ein Textfeld und für den HTML-String ein HTML-Editor eignen.

Eine aktuelle Version des Objektmodells mit den SimpleInstances und ihren Typen ist in Abbildung 6 zu sehen.

Es ist mit dem Objektmodell möglich, eine Standardinstanz für einen `SimpleType` zu definieren. Diese Standardinstanz wird erzeugt, falls eine Instanz des Typen benötigt wird, diese aber nicht explizit definiert wurde. Außerdem können durch `StringEnumerationTypes` `SimpleTypes` definiert werden, die nur bestimmte Strings als Wertebereich haben.

Das Objektmodell gewährleistet eine ständige Konsistenz des Dokumentes zu allen Aspekten, die durch die Typseite des Objektmodells definiert sind. Das heißt, das Dokument kann nur Strukturen annehmen, die durch die Typseite vorgesehen sind und `SimpleInstances` können nur Werte annehmen, die von ihren Typen erlaubt sind. Eine `IntegerInstance`-Instanz kann beispielsweise nur einen Integerwert tragen. Das Objektmodell folgt also dem Prinzip der Konsistenzerhaltung (siehe Konsistenzerhaltung und Konsistenzwiederherstellung S.23).

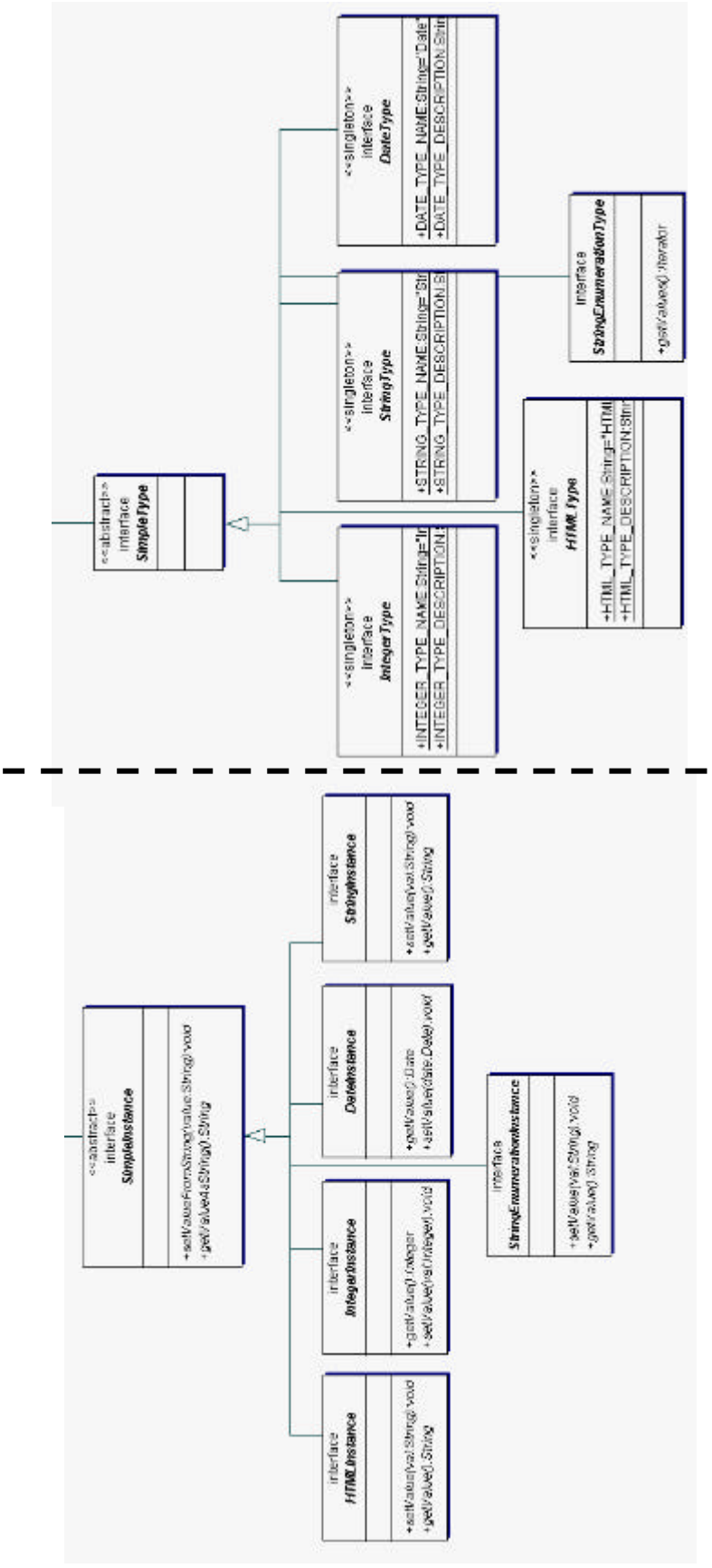

Abbildung 6: Die `SimpleInstances` des Objektmodells mit ihren Typen.

2.4. Persistenz in 4Ever durch XML und XML-Schema

Verantwortlich für die Lade- und Speichervorgänge ist die IO-Komponente[4] von 4Ever. Wie in Abbildung 7 dargestellt, wird die Typseite durch das Laden einer XML-Schema-Datei und die Instanzseite durch das Laden einer XML-Datei instantiiert.
Eine Speichermöglichkeit für das XML-Schema ist nicht notwendig, da die Typseite des 4Ever-Objektmodells zur Laufzeit von 4Ever nicht verändert werden kann. Das Dokument muss nach erfolgten Änderungen als XML-Datei abgespeichert werden können. Das Dokument entspricht dabei einer Ausprägung der Instanzseite des 4Ever-Objektmodells.
Durch die Wahl von XML und XML-Schema als Format für die persistente Speicherung ergeben sich einige Vorteile:

- Es sind weit verbreitete Standards mit einem hohen Bekanntheitsgrad und guter Dokumentation.
- Die Dateiformate sind direkt von Menschen lesbar.
- Es existiert eine Vielzahl an Bibliotheken für ihre Bearbeitung.
- Eine Validierung des Dokumentes zu seinem Schema kann auch ohne 4Ever mit vorhandenen Werkzeugen (z.B. XML-Spy[5]) erfolgen.

Die Nutzung eines Standards kann aber auch als Nachteil gesehen werden, da er nicht einfach an die eigenen Bedürfnisse angepasst werden kann. Wie wir sehen werden wird genau dieser Punkt tatsächlich zu einem Problem werden.

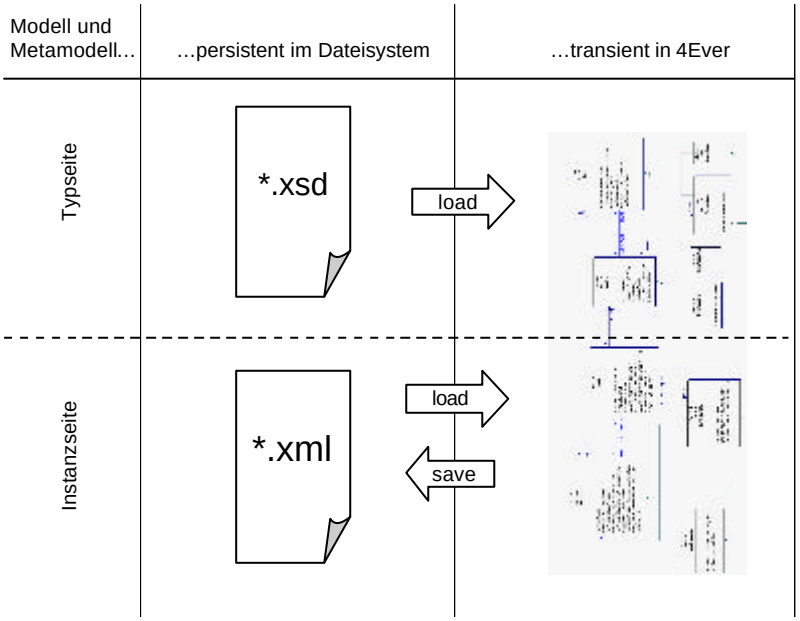

Abbildung 7: Das 4Ever-Objektmodell kann einen XML-String mit zugehörigem Schema abbilden.

[4] In Folge eines Redesigns von 4Ever war während der Diplomarbeit keine IO-Komponente für das überarbeitete Objektmodell verfügbar. Aus diesem Grund wurde von mir ein Loader geschrieben, der die Durchführung der notwendigen Tests erlaubte.
[5] Zu finden unter: www.altova.com

2.5. Erweiterung des Metametamodells von 4Ever

4Ever dient der modellgetriebenen Dokumentenbearbeitung. Für die modellgetriebenen Dokumentenbearbeitung ist eine möglichst genaue Beschreibung des Metamodells unbedingt notwendig. Um das Metamodell beschreiben zu können, ist wiederum ein Metametamodell erforderlich.

In 4Ever ist die Basis des Metametamodells die Typseite des 4Ever-Objektmodells. Durch sie wird bestimmt, welche Metamodelle beschrieben werden können. (siehe

Das Objektmodell von 4Ever S.14) Leider reichen die vom Objektmodell gebotenen Möglichkeiten zur Modellierung bestimmter Sachverhalte nicht aus.

Für ein einheitliches Bild soll an dieser Stelle der Gesamtvorgang der modellgetriebenen Dokumentbearbeitung anhand von 4Ever betrachtet werden. Man kann zwischen zwei 4Ever-Benutzergruppen unterscheiden. Es gibt die **Experten**, die das Metamodell entwerfen und es gibt die **Dokumentenbearbeiter**, die das Dokument auf der Grundlage des Metamodells bearbeiten. In Abbildung 8 sind ihre Rollen dargestellt.

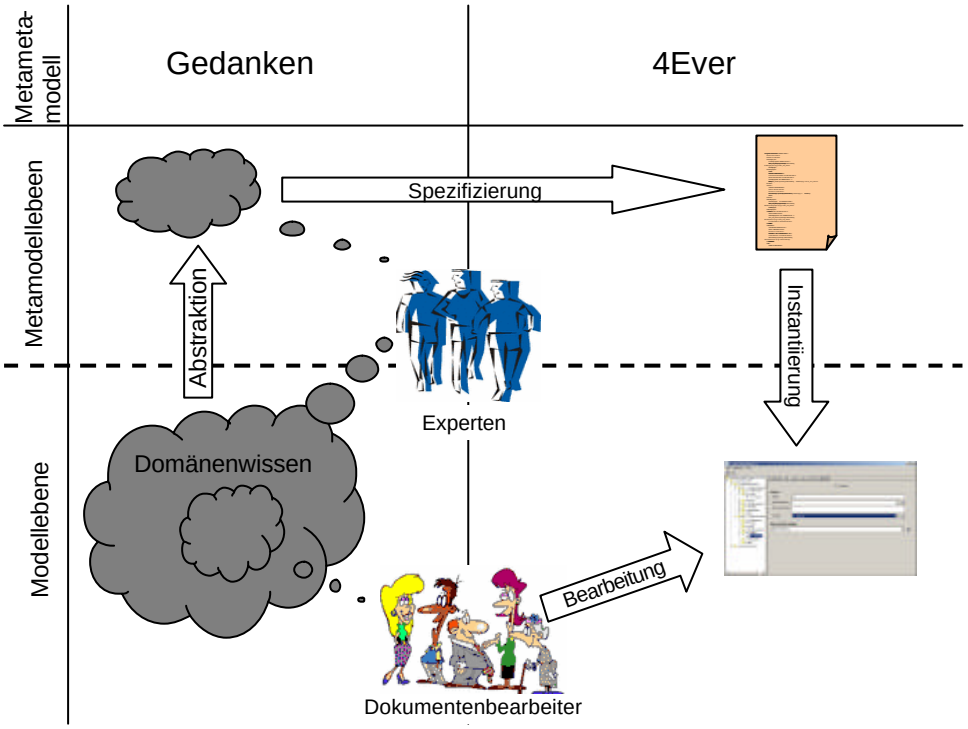

Abbildung 8: Experten können aufgrund ihres umfangreichen Domänenwissens ein Metamodell modellieren.

Die Experten verfügen über einen wesentlich besseren Überblick in einer bestimmten Wissensdomäne als die Dokumentenbearbeiter. Durch diesen Überblick sind sie in der Lage die Problemdomäne zu abstrahieren. In ihren Gedanken entsteht ein Metamodell. Dieses gedankliche Metamodell soll nun als Ausprägung des 4Ever-Metametamodells möglichst vollständig spezifiziert werden. Die Experten spezifizieren es als XML-Schema, welches die IO-Komponente von 4Ever als Anleitung für den Aufbau der Typseite verwendet (siehe Persistenz in 4Ever durch XML und XML-Schema S.19).

In Abbildung 9 ist dargestellt, dass nur ein Teil des Metamodells durch ein XML-Schema definiert werden kann. Ziel der Erweiterung des Metametamodells ist es, eine Möglichkeit für die Definition von zusätzlichen Metabedingungen zu schaffen um einen größeren Teil des mentalen Metamodells beschreiben zu können. Optimal wäre, das gesamte gedankliche Metamodell durch die Metabedingungen ausdrücken zu können.

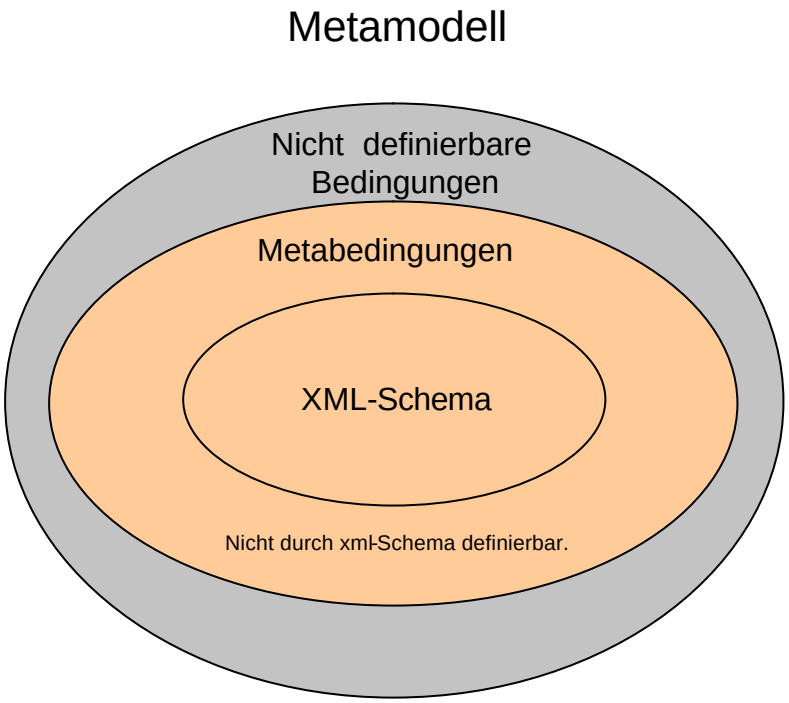

Abbildung 9: Der Bereich der nicht definierbaren Bedingungen soll möglichst klein sein.

Abbildung 10 zeigt einen anderen wichtigen Aspekt der berücksichtigt werden muss. Eine Definitionsmöglichkeit der Metabedingungen alleine reicht nicht aus. Zusätzlich muss es eine Möglichkeit geben, überprüfen zu können, ob ein vorliegendes Dokument den Metabedingungen entspricht.

Für die Konsistenz des Dokumentes zu dem XML-Schema sorgt das Objektmodell selbst. Es lässt nur Operationen zu, die in einen weiterhin konsistenten Zustand des Dokumentes führen (siehe Konsistenzerhaltung und Konsistenzwiederherstellung S.23).

Aufgabe dieser Diplomarbeit ist es

- eine Möglichkeit für die Spezifikation von Metabedingungen zu schaffen.
- eine Komponente in 4Ever einzubinden, durch die die Konsistenz des Dokumentes zu diesen Bedingungen sichergestellt werden kann.

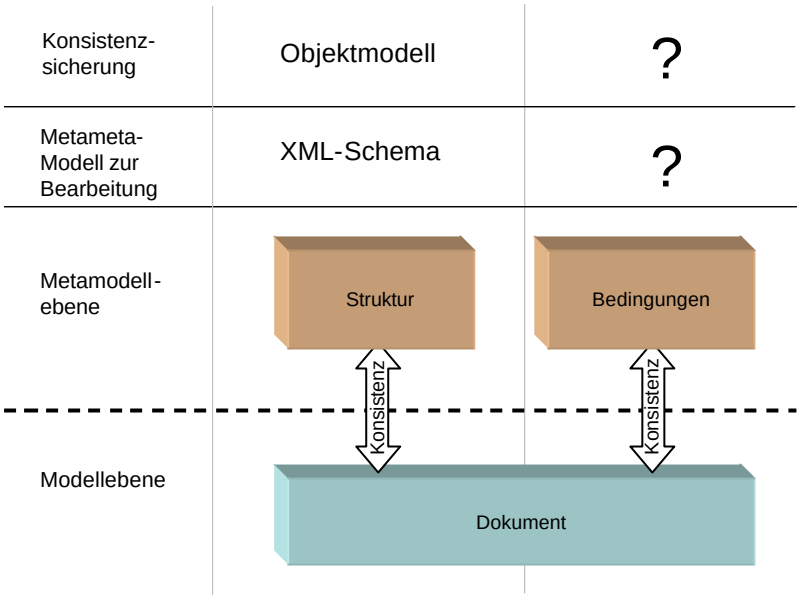

Abbildung 10: Das Metametamodell von 4Ever muss erweitert werden, damit zusätzliche Bedingungen definiert und überprüft werden können.

Die Erweiterung des Metametamodells durch Bedingungen ist in den aktuellen Anwendungsgebieten von 4Ever eine Notwendigkeit. Aus diesem Grund hat man sich zunächst für eine schnell zu realisierende, provisorische Möglichkeit entschieden. Die Metabedingungen werden indirekt, in Form von Algorithmen, im Quellcode spezifiziert. Durch Ausführung dieser Algorithmen kann festgestellt werden, ob das Dokument Metabedingungen verletzt. Übertrüge man diese Lösung in Abbildung 10, müssten die beiden Fragezeichen mit den Worten „Java-Quellcode“ ersetzt werden.

Dieser Lösungsansatz hat aber eine Reihe erheblicher Nachteile:

- Die Metabedingungen sind in Quelltexten verankert und damit nicht generisch.
- Der Experte muss die Programmiersprache, in diesem Fall Java, beherrschen.
- Die Algorithmen können kompliziert sein.
- Es ist detailliertes Wissen über das Objektmodell notwendig, um einen Algorithmus auf ihm schreiben zu können.
- Die Prüfung der Konsistenz ist ineffizient. Es müssen bei jeder Prüfung alle Bedingungen auf dem kompletten Dokument überprüft werden.

Diese Nachteile sind gravierend genug, um die Suche nach einem neuen Ansatz zu rechtfertigen.

2.6. Konsistenzerhaltung und Konsistenzwiederherstellung

Um die Begriffe der Konsistenzerhaltung und der Konsistenzwiederherstellung besser beschreiben zu können, ist es sinnvoll, die Dokumentenbearbeitung zunächst zu formalisieren.

Ein Dokument wird durch seine Bearbeitung in einen gewünschten Zustand gebracht. Die Bearbeitung besteht dabei aus vielen Bearbeitungsschritten. Ausgehend von einem

Ausgangszustand werden also Aktionen auf dem Dokument ausgeführt, die es in andere Zustände und schließlich in den gewünschten Endzustand bringen. Durch einen deterministischen endlichen Automaten lässt sich dieser Vorgang gut beschreiben. Ein deterministischer endlicher Automat *M* wird durch ein 5-Tupel spezifiziert [SCH]:

$$M ? (Z, ? , ?, z_0, E)$$

Z sei die Menge aller möglichen Zustände des Dokumentes. *?* sei die Menge aller möglichen Operationen auf dem Dokument. d ist eine Überführungsfunktion mit $d: Z x ? ? Z$. $z_0 ? Z$ ist der Startzustand, also das unbearbeitete Dokument. *E* ist die Menge der Zustände eines Dokumentes in denen es als fertig betrachtet werden kann. Außerdem gilt $Z n ? = Ø$ und $E ? Z$. Die Menge der Zustände *Z* zerfällt in die zwei Mengen T und F mit $Z = T ? F$ und $T n F = Ø$. Außerdem soll gelten $z_0 ? T$. Dabei ist *T* die Menge der Zustände, in denen das Dokument konsistent ist und *F* ist die Menge der Zustände, in denen es nicht konsistent ist. Man kann also auch die Überführungsfunktion in d^T und d^F aufteilen, wobei $d^T: Z x ? ? T$ und $d^F: Z x ? ? F$ ist.

- **Konsistenzerhaltung** ist ein Verfahren, dessen Ziel es ist, ein Dokument in konsistenten Zuständen zu halten.
- **Konsistenzwiederherstellung** ist ein Verfahren, dessen Ziel es ist, ein Dokument von einem inkonsistenten Zustand in einen konsistenten Zustand zu überführen.

Das Vorgehen der Konsistenzsicherung ist sehr restriktiv. Man verbietet einfach alle Überführungsfunktionen, d^F die in einen inkonsistenten Zustand führen würden. Leider ist dieses Verfahren häufig ungeeignet, da es eventuell dazu führt, dass Endzustände aus *E* nicht erreicht werden können. In Abbildung 11 ist eine solche Situation, in der für deterministische endliche Automaten typischen Darstellungsweise [SCH] gezeigt.

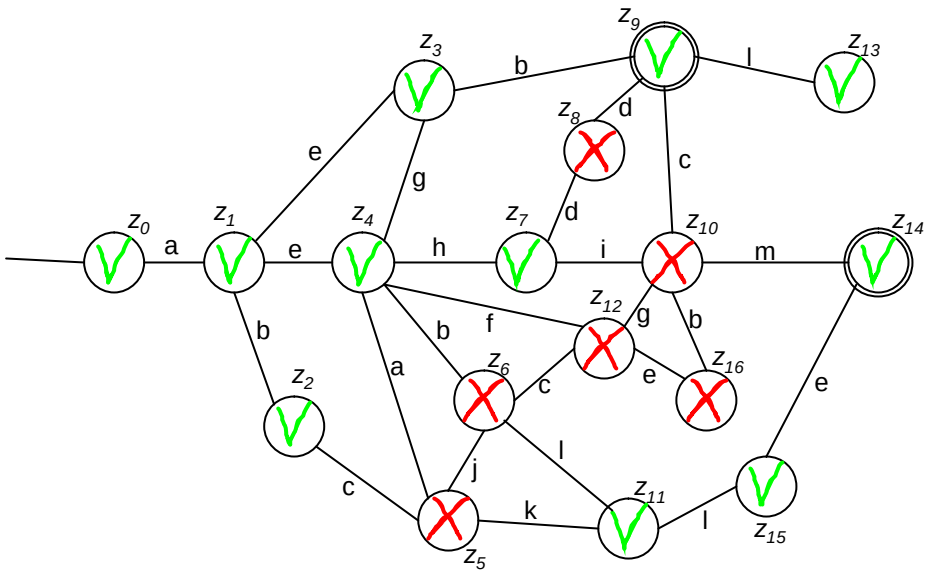

Abbildung 11: Inkonsistente Dokumentenzustände sind mit einem roten Kreuz markiert, konsistente mit einem grünen.

In der Abbildung repräsentiert der Graph einen deterministischen endlichen Automaten $M = (Z, \Sigma, \delta, z_0, E)$ mit $Z = T \cup F$.

$T = \{z_0, z_1, z_2, z_3, z_4, z_7, z_9, z_{11}, z_{13}, z_{14}, z_{15}, z_{16}\}$
$F = \{z_5, z_6, z_8, z_{10}, z_{12}\}$
$\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m\}$
$E = \{z_9, z_{14}\}$
$\delta^T = \{(z_0, a, z_1), (z_1, e, z_4), (z_1, b, z_2), (z_3, e, z_1), (z_3, b, z_9), (z_4, g, z_3), (z_4, h, z_7), (z_5, k, z_{11}), (z_6, i, z_{11}), (z_8, d, z_9), (z_9, i, z_{13}), (z_{10}, m, z_{14}), (z_{11}, l, z_{15}), (z_{15}, e, z_{14})\}$
$\delta^F = \{(z_2, c, z_5), (z_4, f, z_{12}), (z_4, b, z_6), (z_4, a, z_5), (z_5, j, z_6), (z_6, c, z_{12}), (z_7, i, z_{10}), (z_7, d, z_8), (z_9, c, z_{10}), (z_{10}, b, z_{16}), (z_{12}, g, z_{10}), (z_{16}, e, z_{12})\}$

Würde man für die Bearbeitung eines Dokumentes mit einem solchen Zustandsraum ein Konsistenzerhaltungsverfahren einsetzen, könnte der Dokumentenbearbeiter den eventuell gewünschten Endzustand z_{14} nicht erreichen, weil alle Pfade in dem Graphen zu dem Zustand z_{14} über mindestens einen inkonsistenten Zustand führen. Es könnte auch der Fall eintreten, dass man den gewünschten Zustand nur über einen Umweg erreichen kann. Ein Umweg bedeutet aber mehr Arbeitsschritte und entsprechend mehr Zeitaufwand. In diesen Fällen ist ein Verfahren zur Konsistenzwiederherstellung wünschenswert. Für solche Verfahren gibt es unterschiedliche Ansätze. Entweder können sie einen konsistenten Zustand vollautomatisch wieder herstellen, oder sie benötigen die Unterstützung von dem Dokumentenbearbeiter.

2.7. Fallbeispiel V200X-Vorgehensmodell

Allgemein

Momentan wird 4Ever unter anderem als Bearbeitungsprogramm für das V200X Vorgehensmodell eingesetzt. Das V-Modell 200X ist die Weiterentwicklung des V-Modells 97. Das V-Modell 97 geht auf eine Initiative des Bundesamtes für Wehrtechnik und Beschaffung (BWB) zurück. [DRO, S.1] Ziel dieser Initiative war es, den Entwicklungsprozess von Software- und Informationssystemen zu verbessern. Das V-Modell 97 wird seit seiner Fertigstellung von vielen Unternehmen und Behörden als Leitfaden zur Softwareentwicklung eingesetzt. Dadurch konnte eine Verbesserung der Produktqualität als auch eine Verbesserung der Kooperation zwischen den beteiligten Parteien erzielt werden. Seit der Fertigstellung des V-Modells im Jahr 1997 wurde es nicht mehr verändert[6] und ist somit nicht auf dem aktuellen Stand der Informationstechnologie. Mit dem V-Modell 200X soll nun eine zeitgemäße Folgeversion des V-Modells entstehen. Die mit dem V-Modell 97 gesammelten Erfahrungen sollen dabei in die Entwicklung der Folgeversion einfließen. Das V-200X Modell wird im Rahmen des WEIT-Projektes (**W**eiterentwicklung des **E**ntwicklungsstandards für **IT**-Systeme des Bundes auf Basis des V-Modell97) entwickelt. Es wurde von dem BMVg, dem BMI-KBSt und dem BWB beauftragt. Auftragnehmer ist die Technische Universität München mit den Partnern 4Soft, EADS, IABG mbH, Technische Universität Kaiserslautern und der Siemens AG. Die Planung für das WEIT-Projekt sieht eine Entwicklung in drei Phasen vor. In der ersten Phase sollten die inhaltlichen Verbesserungspotentiale des V-Modells analysiert und eine verbesserte Struktur für das V-Modell 200X konzipiert werden. In dieser Phase fand auch der größte Teil der Entwicklung des Metamodells statt. In der zweiten Phase soll das V-Modell mit Unterstützung von 4Ever detailliert ausgearbeitet werden. In der dritten Phase soll es letztendlich durch verstärkte Verbreitung im Internet, Publikationen und Präsentationsveranstaltungen bekannt gemacht werden. Die ursprünglich geplanten Zeiträume für die Phasen sind in der Abbildung 12 zu erkennen.

Das V200X Vorgehensmodell eignet sich als Fallbeispiel für die modellgetriebene Dokumentenbearbeitung mit 4Ever, weil das Metamodell des V200X Modells relativ komplex ist und sich nicht komplett durch das Objektmodell von 4Ever abbilden lässt (siehe Das V200X-Metamodell S.27). Die in 4Ever nicht realisierten Aspekte des Metamodells werden momentan teilweise durch Java-Routinen verwirklicht und auf expliziten Wunsch des Benutzers überprüft. Dies führt jedoch zu Problemen, die in der Anforderungsanalyse (S.48) genauer betrachtet werden.

[6] http://www.v-modell-200x.de/beschreibung.html

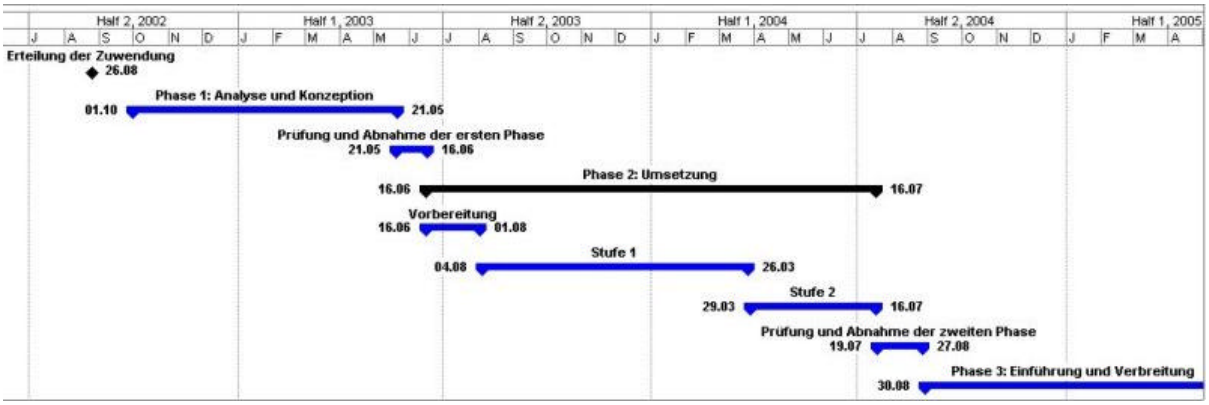

Abbildung 12: Die Entwicklungsphasen des V200X-Vorgehensmodells.

Das V200X-Metamodell

Das Vorgehensmodell setzt sich aus Vorgehensbausteinen zusammen. Vorgehensbausteine sind modular. Das heißt, sie können unabhängig voneinander verwendet und bearbeitet werden. Ein Vorgehensbaustein enthält Aktivitäten, Produkte und Rollen. In Abbildung 13 ist ein solcher Vorgehensbaustein mit seinen Elementen und den Beziehungen zwischen ihnen dargestellt.

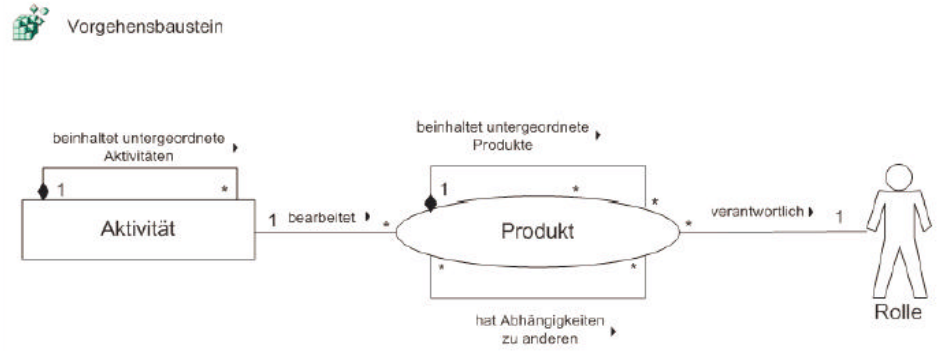

Abbildung 13: Die Struktur der Vorgehensbausteine des V200X-Vorgehensmodells

Das Metamodell eines solchen Vorgehensbausteines könnte mit folgenden Metabedingungen beschrieben werden:

a) Ein Vorgehensbaustein enthält Aktivitäten, Produkte und Rollen
b) Eine Aktivität beinhaltet beliebig viele untergeordnete Aktivitäten.
c) Ein Produkt wird genau von einer Aktivität bearbeitet.
d) Für ein Produkt gibt es genau eine verantwortliche Rolle.
e) Ein Produkt beinhaltet beliebig viele untergeordnete Produkte.
f) Produkte haben beliebige Abhängigkeiten untereinander.

Damit 4Ever als Bearbeitungsprogramm für das V200X-Vorgehensmodell dienen kann müssen diese Bedingungen in eine für 4Ever verständliche Form gebracht werden. Diese verständliche Form ist ein XML-Schema.

Es sollen nun die Bedingungen a) und c) in ein XML-Schema übertragen werden. Auf diese Weise bekommt der Leser eine Vorstellung von dem Aufbau des V200X-Metamodells als XML-Schema. Es wird sich außerdem die Beschränktheit eines XML-Schemas zeigen, da sich die Bedingung c) nicht vollständig in das Schema übertragen lässt.

Durch ein XML-Schema könnte man die Bedingung a) mit dem XML-String aus Listing 1 beschreiben.

Listing 1

```
...
<xs:element name="Vorgehensbaustein" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
    ...
      <xs:element name="Produkt" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Name" type="simpleString"/>
       ...
            </xs:sequence>
       ...
        </xs:complexType>
      </xs:element>
       ...
      <xs:element name="Aktivitaet" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Name" type="simpleString"/>
            ...
          </xs:sequence>
            ...
        </xs:complexType>
      </xs:element>
       ...
      <xs:element name="Rolle" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Name" type="simpleString"/>
            ...
          </xs:sequence>
            ...
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
   ...
```

Dieses Schema sagt aus, dass ein Vorgehensbaustein aus beliebig vielen (`minOccurs="0" maxOccurs="unbounded"`) `Produkten`, `Aktivitäten` und `Rollen` besteht. Obwohl die Reihenfolge der Elemente rein fachlich keine Rolle spielt wird sie hier durch das `<xs:sequence/>`-Element festgelegt.

Das in Abbildung 13 zu sehende Grundgerüst eines Vorgehensbausteins ist durch die Definition der Typen noch nicht vollständig. Zusätzlich sollen die Beziehungen zwischen den Elementen dargestellt werden. Dies wird durch die Definition von Referenzen erreicht. Soll die in Bedingung c) spezifizierte Beziehung zwischen Produkten und Aktivitäten in dem XML-Schema berücksichtigt werden, ließe sich das wie in Listing 2 realisieren.

Listing 2

```
<xs:element name="Vorgehensbaustein" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      ...
      <xs:element name="Produkt" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Name" type="simpleString"/>
            <xs:element name="VerantwortlicherRef" minOccurs="0">
              <xs:complexType>
                <xs:attribute name="link" type="simpleString" use="required"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="id" type="simpleString" use="required"/>
        </xs:complexType>
      </xs:element>
      ...
      <xs:element name="Aktivitaet" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Name" type="simpleString"/>
            <xs:element name="ProduktRef" maxOccurs="unbounded">
              <xs:complexType>
                <xs:attribute name="link" type="simpleString" use="required"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="id" type="simpleString" use="required"/>
        </xs:complexType>
      </xs:element>
     ...
    </xs:sequence>
  </xs:complexType>
</xs:element>
 ...
<xs:unique name="ProduktKey">
  <xs:selector xpath=".//Produkt"/>
    <xs:field xpath="@id"/>
</xs:unique>


<xs:keyref name="ProduktKeyRef" refer="ProduktKey">

  <xs:selector xpath=".//ProduktRef"/>
    <xs:field xpath="@link"/>
  </xs:keyref>
...
```

Hierbei entsprechen die fett gedruckten Elemente den Erweiterungen des Listing 1. Sie sind notwendig, um die Bedingung c) abzubilden. Zunächst wird durch ein `<xs:unique/>`-Element dafür gesorgt, dass das Produkt einen eindeutigen Schlüssel trägt. Auf diesen Schlüssel kann sich nun ein als Zeiger ausgezeichnetes Attribut eines Elementes beziehen. Ein Attribut lässt sich durch ein `<xs:keyref/>`-Element als Zeiger auf ein Attribut auszeichnen.

Durch die fett gedruckten `<xs:unique/>`- und `<xs:keyref/>`-Elemente ist also dafür gesorgt, dass jede Ausprägung dieses XML-Schemas nur `Produkt`-Elemente mit eindeutigem `id`-Attribut enthält, auf die sich die `link`-Attribute der `Aktivitaet`-Elemente beziehen können.

Eine Ausprägung dieses Schemas ist in Listing 3 zu sehen:

Listing 3

```
                …
<Vorgehensbaustein>
                                        …
            <Produkt id="ID_ProdB" >
                  <Name>ProdB</Name>
            </Produkt>
                                        …
            <Aktivitaet id="ID_Akt1">
                  <Name>Akt1</Name>
                  <ProduktRef link="ID_ProdB" />
            </Aktivitaet>
            <Aktivitaet id="ID_Akt2">
                  <Name>Akt2</Name>
                  <ProduktRef link="ID_ProdB" />
            </Aktivitaet>
                                        …
</Vorgehensbaustein>
                  …
```

Der in Listing 3 aufgeführte XML-String enthält zwei `Aktivität-Elemente`, die dasselbe `Produkt` referenzieren. Diese Situation ist gemäß dem obigen XML-Schema valide, obwohl sie von dem ursprünglichen Metamodell aus Abbildung 13 nicht erlaubt ist. Der XML-String besagt, dass ein Produkt von zwei statt nur von einer Aktivität bearbeitet wird.

Die Spezifikation des XML-Schemas [XMLSCH] stellt leider keine Möglichkeit bereit, die Anzahl der eingehenden Referenzen zu beschränken. An dieser Stelle ist also eine Erweiterung bzw. Ergänzung notwendig.

Das V200X-Metamodell enthielt bis zum 13.03.2004 noch circa 20 weitere Regeln, die mit dem XML-Schema nicht beschrieben werden können. Diese Regeln sind in dem Dokument [V200XMeta] aufgeführt.

3. Regel-Engines

3.1. Allgemein

Regel-Engines können feststellen, ob auf einer Datenmenge bestimmte Bedingungen erfüllt sind. Wenn sie erfüllt sind, werden bestimmte Aktionen ausgelöst. Diese Funktionalität wird in vielen Gebieten der Informationstechnik benötigt. Ursprünglich aus dem Bereich der Expertensysteme stammend, erleben Regel-Engines in den letzten Jahren im Bereich des E-Business eine Neubelebung. Dort versucht man, die Anwendungslogik von den Geschäftsabläufen zu trennen, indem die Geschäftsabläufe einfach in einer Menge von Regeln, so genannten Business-Rules[7], festgelegt werden. Diese Regeln können dann nach Belieben hinzugefügt, entfernt oder geändert werden. Wurde für die Regeln auch noch eine einfache Pseudoumgangssprachliche Regelsyntax gewählt, sind nun zum Beispiel auch programmierunkundige Mitarbeiter aus der Marketingabteilung in der Lage, die Geschäftssoftware an neue Anforderungen anzupassen, ohne von einem trägen Entwicklungszyklus in der Softwareabteilung abhängig zu sein.

Die Vereinfachung kommt in erster Linie daher, dass es nicht mehr notwendig ist, den prozeduralen Ablauf eines Moduls zu verstehen, um ihn anpassen zu können. Es reicht, dem System in Form einer Regel zu sagen, wie es sich in Zukunft verhalten soll. Diese Art der Programmierung nennt man **deklarative Programmierung**.

Nimmt man als Beispiel einen Online-Shop für Bastelartikel, den man um einen intelligenten Angebotsservice erweitern will, könnte eine Aufgabenstellung folgendermaßen aussehen:

„Wenn ein Kunde einen Holzbaukasten bestellt, soll ihm ein Angebot für Holzleim in einem Popup-Fenster gemacht werden."

Dieses Verhalten ließe sich in etwa durch den Pseudocode in Listing 4 ausdrücken.

Listing 4

```
IF
      Warenkorb enthält Holzbaukasten
THEN
      showPopup(„Brauchen Sie auch Holzleim?“)
```

Wenn man diese Regel in eine herkömmliche Geschäftssoftware integrieren wollte, müsste man den Code nach einer geeigneten Stelle durchsuchen, an der geprüft werden soll, ob die Bedingung erfüllt ist. Hat man jedoch eine Regel-Engine zur Verfügung, reicht es, diese Regel zur Menge aller Regeln, der **Regelbasis**, hinzuzufügen. Da die Regel-Engine die Fakten quasi ständig beobachtet, würde sie merken sobald ein Warenkorb einen Holzbaukasten enthält und die zugeordnete Aktion auslösen.

[7] **Business-Rule**: „… eine Aussage die bestimmte Aspekte eines Geschäftes definiert oder einschränkt…sie ist atomar, d.h. sie kann nicht in mehrere, feinere Regeln zerlegt werden"(GUIDE Business Rules Project, Nov. ´93 – http://www.businessrulesgroup.org/first_paper/br01c3.htm#s3c

Dieses Verhalten könnte man natürlich auch mit einem einfachen Algorithmus erreichen, der nach jeder Änderung der zugrunde liegenden Fakten überprüft, ob die Bedingungen einer Regel erfüllt sind. Ein solch einfacher Algorithmus würde aber schon bei einer geringen Anzahl von Fakten und Regeln zu inakzeptablen Laufzeiten führen. Regel-Engines hingegen können sehr effizient bestimmen, ob eine Änderung der Fakten Konsequenzen hat. Diese Fähigkeit wird den allermeisten von ihnen durch den weiter unten besprochenen RETE-Algorithmus (siehe Der RETE-Algorithmus S.42) verliehen. Ein weiterer Vorteil des Regel-Engine Ansatzes ist es, dass man die Änderungen zur Laufzeit vornehmen kann, da man den eigentlichen Programmcode nicht neu kompilieren muss [ix0203110, S.110].

Nachdem die Vorteile des Einsatzes einer Regel-Engine in einer Business-Software diskutiert wurden, stellt sich die Frage, inwiefern eine Regel-Engine dafür geeignet ist, eine Konsistenzprüfung auf einem Dokument auszuführen. Die festgestellten Vorteile einer Regel-Engine bei der Anwendung in einer Geschäftssoftware seien hier noch mal aufgelistet:

- zentrale Definition der Geschäftsprozesse in der Regelbasis
- einfache Regelsprache durch deklaratives Vorgehen
- ständige Beobachtung der Fakten mit guter Effizienz
- Austausch der Regeln zur Laufzeit

Wenn man nun das Metamodell eines Dokumentes mit den Regeln und das Dokument mit den Fakten gleichsetzt, lassen sich die ersten drei Vorteile direkt auf den Mechanismus zur Konsistenzprüfung übertragen:

- zentrale Definition des Metamodells in der Regelbasis
- einfache Regelsprache durch deklaratives Vorgehen
- ständige Beobachtung der Fakten und damit ständige Überprüfung der Konsistenz[8] mit guter Effizienz

Man kann also sagen, dass es sich bei der Steuerung von Geschäftsabläufen, ab einem gewissen Abstraktionsniveau, um ein sehr ähnliches Problem wie bei der Konsistenzsicherstellung auf einem Dokument handelt. In beiden Fällen muss nach jeder Änderung einer Objektmenge geprüft werden, ob die Bedingungen einer oder mehrerer Regeln erfüllt sind.

3.2. Fakten und Regeln

Eine Regel-Engine soll erkennen, wenn in einem System bestimmte Situationen eintreten um dann entsprechende Aktionen auszulösen. Um dazu in der Lage zu sein, muss die Regel-Engine zunächst darüber informiert werden, in was für einem Zustand sich das System befindet. Dies geschieht über kleine Wissensfragmente, die in Form von Fakten in der **Faktenbasis** bzw. dem **Arbeitsspeicher** der Regel-Engine abgelegt werden.

Um ein geeignetes Mittel zur Beschreibung von Fakten und Regeln zur Verfügung zu haben, wird im Folgenden die Prolog-ähnliche Jess-Syntax[9] zur Anwendung kommen. Jess[10] ist eine

[8] Hier ist die Konsistenz eines Dokumentes bzgl. seines Modells gemeint.
[9] http://herzberg.ca.sandia.gov/jess/docs/70/language.html

in Java implementierte, sehr gut dokumentierte Regel-Engine. Die Bedeutung der hier verwendeten Sprachkonstrukte sollte jeweils aus dem Kontext hervorgehen. Eine formale Einführung der Sprache ist deswegen nicht notwendig.

Fakten sind Objekte einer Klasse mit beliebig vielen Attribut-Wert Paaren. In Jess können Fakten nach dem Muster in Listing 5 definiert werden:

Listing 5

```
(<Kategorie>  (<Attribute_1>  <Wert_1>)  (<Attribute_2>  <Wert_2>)  …
(<Attribute_n> <Wert_n>))
```

Einige Fakten in dem Bastelshop könnten dementsprechend wie in Listing 6 definiert werden.

Listing 6

```
(Kunde (name Müller) (anrede Herr ))
(Warenkorb (id 0815) (anzahlAngebote 1))
```

Diese Darstellungsform ist sehr vorteilhaft, da man die Fakten in der Faktenbasis in einer datenbankähnlichen Strukturen halten kann, die durch ihre Indizierung ein schnelles Durchsuchen erlauben [FRIED, S.81]. Seit einiger Zeit gibt es auch Ansätze, Objekte einer objektorientierten Sprache der Faktenbasis direkt hinzuzufügen[11]. Die allermeisten Regel-Engines arbeiten aber noch auf Fakten, die wie obige Listen aufgebaut sind. Bei der Verwendung einer solchen faktenbasierten Regel-Engine ist es notwendig, den Zustand eines objektorientierten Systems in Fakten zu übersetzen.

Beispiel: Angenommen das dem Bastelshop zugrunde liegende Objektmodell entspricht dem in Abbildung 14 dargestellten, dann reichen die Fakten aus Listing 6 (`Kunde` und `Warenkorb`) noch nicht aus, um den Zustand des Systems zu beschreiben. Man kann aber bewährte Methoden aus der Datenbankmodellierung verwenden, um zunächst ein relationales Schema zu erhalten, welches sich dann direkt in die notwendigen Fakten übersetzen lässt.

[10] http://herzberg.ca.sandia.gov/jess/
[11] http://www.drools.org/ReteOO

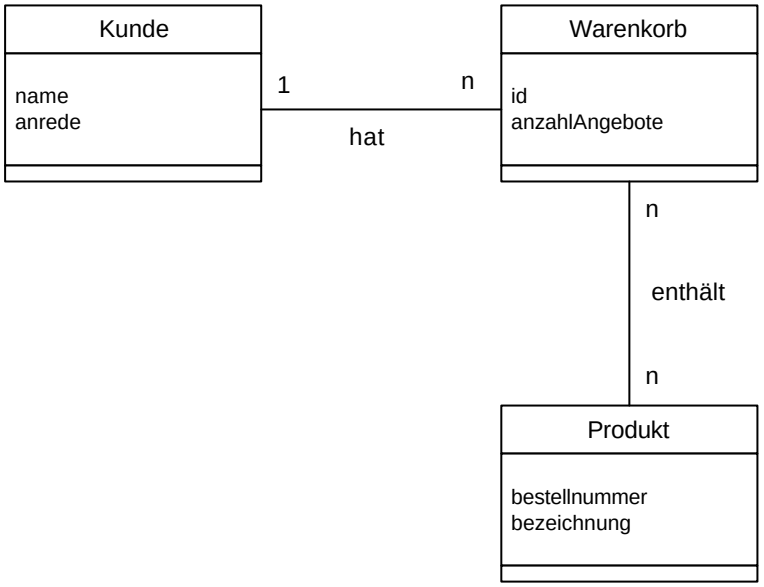

Abbildung 14: Das Modell eines Online-Shops als Klassendiagramm.

In diesem Fall würde sich das in Listing 7 aufgeführte relationale Schema ergeben.

Listing 7

```
Kunde: (name anrede)
Warenkorb: (id anzahlAngebote name)
Produkt: (bestellnummer bezeichnung)
enthält: (id bestellnummer)
```

Hierbei stellen `Kunde`, `Warenkorb`, `Produkt` und `enthält` die Relationen dar. In den Klammern sind ihre Attribute aufgeführt. Die unterstrichenen Attribute dienen als Schlüssel zur Identifikation der Tupel. Die Assoziationen aus dem Klassendiagramm in Abbildung 14 werden gemäß dem typischen Vorgehen[12] in das relationale Schema übernommen. Die 1-zu-n-Beziehung zwischen `Kunde` und `Warenkorb` wird durch die Aufnahme des Fremdschlüssels `name` in die Relation `Warenkorb` abgebildet. Die n-zu-n-Beziehung zwischen `Warenkorb` und `Produkt` bleibt durch eine eigene Relation `enthält` erhalten.

Wenn der Bastelshop zurzeit einen männlichen Kunden, mit einem Holzbaukasten im Warenkorb, hat, dem noch kein Angebot gemacht wurde, dann könnte diese Situation nun durch die Fakten wie in Listing 8 beschrieben werden.

[12] [KEM] S. 76

Listing 8

```
(Kunde (name Müller) (anrede Herr))
(Warenkorb (id 0815) (anzahlAngebote 0) (name Müller))
(Produkt (bestellnummer 111) (bezeichnung Holzbaukasten))
(enthält (id 0815) (bestellnummer 111))
```

Die Fakten alleine reichen der Regel-Engine aber noch nicht aus. Um sie interpretieren zu können, werden ihr die semantischen Zusammenhänge in Form von Regeln übergeben. **Regeln** bestehen aus einem IF- und einem THEN-Teil. Der IF-Teil wird auch als LHS (left hand side) oder **Bedingungsteil** bezeichnet. Der THEN-Teil wird auch als RHS (right hand side) oder **Aktionsteil** bezeichnet. Sind die Bedingungen des Bedingungsteils erfüllt, wird der Aktionsteil ausgeführt.

In dem Online-Bastel-Shop soll die Regel-Engine zum Bespiel erkennen, wenn in einem Warenkorb ein Holzbaukasten ist. Eine entsprechende Regel in Jess-Syntax, bei der der Aktionsteil leer ist, könnte wie in Listing 9 aussehen.

Listing 9

```
(defrule Angebotsregel1
      (Warenkorb (id ?id))
      (Produkt (bestellnummer ?bn) ( bezeichnung Holzbaukasten))
      (enthält (id ?id)(bestellnummer ?bn))
=>
)
```

Die erste Zeile teilt Jess lediglich mit, dass es sich um eine Regeldefinition handelt und dass die Regel den Namen `Angebotsregel1` hat. Die Zeilen zwei, drei und vier sind der Bedingungsteil der Regel. Durch die im Beispiel gegebenen Fakten ist die Bedingung erfüllt, da es eine Kombination von Fakten gibt (in diesem Fall die einzige) die diesem Pattern entspricht. Die Bedingung ist rein deklarativ definiert. Es wird die Bedingung definiert, aber nicht festgelegt, wie die Regel-Engine vorgehen soll, um festzustellen, ob die sie erfüllt ist. Obwohl der Bedingungsteil der `Angebotsregel1` erfüllt ist, würde sie nichts bewirken, da ihr Aktionsteil leer ist. Der Bedingungsteil und der Aktionsteil werden in Jess durch das „daraus folgt“-Zeichen ‚=>‘ getrennt. Angenommen die Regel-Engine sollte den String, „Herr Müller, brauchen Sie noch Holzleim?“ bei erfülltem Bedingungsteil ausgeben, müsste die Regel wie in Listing 10 erweitert werden.

Listing 10

```
(defrule Angebotsregel1
      (Kunde (name ?n) (anrede ?a))
      (Warenkorb (id ?id) (name ?n))
      (Produkt (bestellnummer ?bn) ( bezeichnung Holzbaukasten))
      (enthält (id ?id)(bestellnummer ?bn))
=>
      (printout t ?a ?n „, brauchen Sie noch Holzleim?“)
)
```

3.3. Die Arbeitsweise einer Regel-Engine

Eine klassische Regel-Engine besteht aus 3 Teilen:

1) In der **Faktenbasis** werden die aus den vorhandenen Daten erhobenen Fakten gespeichert. Prinzipiell reicht es, die Fakten in irgendeiner Form in einer Textdatei abzulegen. Um jedoch ein effizientes Arbeiten zu ermöglichen sind komplexere Datenstrukturen notwendig. Zum Beispiel indizierte Tabellen.[FRIED]

2) In der **Regelbasis** werden die Regeln gespeichert. Sie liegt meistens, ebenfalls in indizierter Form, im Arbeitsspeicher um auch hier einen schnellen Zugriff zu haben.

3) Die **Inferenzmaschine** ist der aktive Teil einer Regel-Engine. Sie besteht aus einem **Patternmatcher**, einer **Agenda** und einer **Execution-Engine**.[13]

Das Zusammenspiel dieser Komponenten ist in Abbildung 15 dargestellt. Die gesamte Regel-Engine ist in ein System integriert, welches die Regel-Engine mit einem `execute()`-Aufruf starten kann. Nachdem sie gestartet ist durchläuft sie einen Zyklus, den **Inferenzen-Zyklus** ein- oder mehrmals**.** In diesem überprüft der Patternmatcher zunächst, von welchen Regeln der Bedingungsteil erfüllt ist. Diese Regeln werden dann in einer Konfliktmenge gespeichert. Anschließend findet die Konfliktauflösung statt. Im Rahmen dieses Prozesses wird entschieden, in welcher Reihenfolge die Regeln in der Agenda eingefügt werden. Als Entscheidungsbasis kann hierbei unter anderem eine vom Regelersteller vergebene Priorisierung der Regeln dienen. Nachdem nun alle Regeln in der Agenda eingereiht wurden, führt die Execution-Engine den Aktionsteil der ersten Regel in der Agenda aus. Man spricht hier auch vom „**Feuern**“ einer Regel. Dabei kann beliebiger Code ausgeführt werden, der beispielsweise ein Fenster erscheinen lässt. Würde es bei solchen Nebenwirkungen bleiben, reichte es aus, wenn die Execution-Engine nun die auf der Agenda stehenden Einträge nach und nach abarbeitete. Der von der Execution-Engine ausgeführte Code kann aber auch die Faktenbasis ändern. Das kann bedeuten, dass durch die Änderung nun eine auf der Agenda stehende Regel eigentlich gar nicht mehr gefeuert werden darf, da ihr Bedingungsteil nicht mehr erfüllt ist. Um dies zu überprüfen, ist ein neuer Durchlauf des gesamten Inferenzen-

[13] Die Begriffe Patternmatcher und Execution-Engine wurden aus dem Englischen übernommen, da keine passenden Übersetzungen gefunden wurden.

Zyklus notwendig. Er wird so lange wiederholt, bis die Agenda keine Aktionsteile mehr enthält.

Eine Regel-Engine in dieser Form wäre wegen einer zu hohen Laufzeit in vielen Fällen nicht praktisch einsetzbar, wenn der Patternmatcher jedes Mal aufs Neue alle Bedingungen auf allen Fakten überprüfen würde. Um hier effizient zu arbeiten nutzt er die Ergebnisse der vorigen Durchläufe. (siehe Der RETE-Algorithmus)

Eine präzisere Beschreibung der Vorgänge während eines Inferenzen-Zyklus lässt sich durch die Angabe eines Algorithmus in Pseudocode erreichen (siehe Listing 11).

Listing 11

```
class RuleEngine{
     ...
     public execute(){
           patternmatcher();
           if(agendaContainsRule()){
                 executionengine();
                 execute ();
           }
     }


     // Diese Methode entfernt alle Regeln von der Agenda,
     // deren Bedingungsteil nicht mehr erfüllt ist und baut
     // die Agenda gemäß einer Konfliktlösungsstrategie mit
     // den hinzugekommenen Regeln neu auf.
     private void patternmatcher(){
           …
           conflictResolution();
     }

     // Diese Methode führt den Aktionsteil der auf der
     // Agenda oben stehenden Regel aus. Anschließend
     // entfernt sie die gefeuerte Regel von der Agenda.
     private void executionengine(){…}

     //Diese Methode bestimmt ob die Agenda eine Regel enthält
     private boolean agendaContainsRule(){…}
}
```

Ein Inferenzen-Zyklus umfasst die Zeitspanne von einem `execution()`-Aufruf bis zum nächsten rekursiven `execution()`-Aufruf oder bis zum verlassen der `execution()`-Funktion.

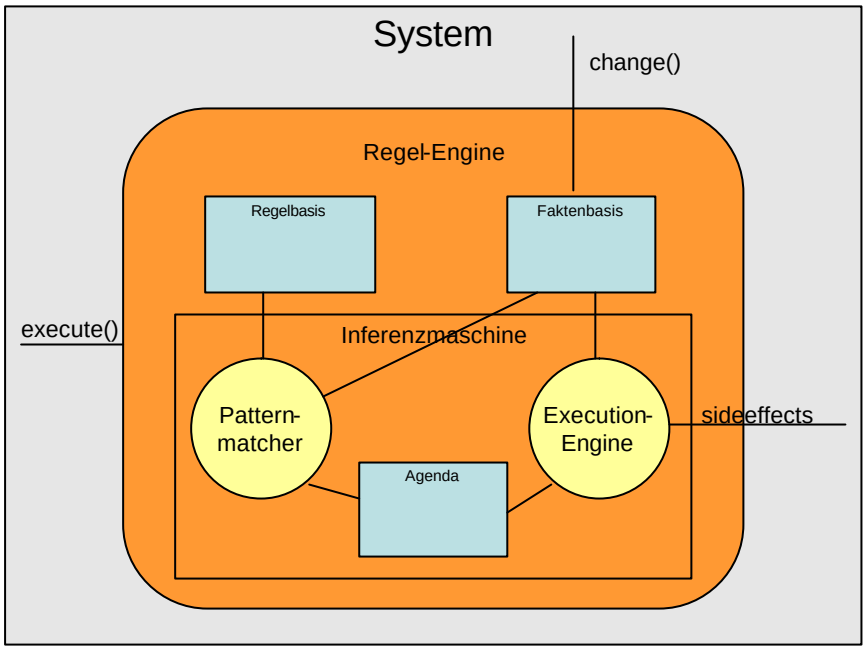

Abbildung 15 : Die Architektur einer klassischen Regel-Engine und das Zusammenspiel ihrer Komponenten.

Inferenzen-Zyklus (Beispiel 1):

Wie würde der Inferenzen-Zyklus nun aussehen, wenn zwei Regeln in einen Konflikt geraten? Nehmen wir als Beispiel zwei Angebotsregeln, die denselben Bedingungsteil haben:

Listing 12

```
Angebotsregel1:
      IF
            Warenkorb enthält Holzbaukasten
      THEN
            zeigeAngebot(„Brauchen Sie auch Holzleim?“)

Angebotsregel2:
      IF
            Warenkorb enthält Holzbaukasten
      THEN
            zeigeAngebot(„Brauchen Sie auch Schmirgelpapier?“)
```

Der Ablauf der Vorgänge nach einem `execute`()-Aufruf durch das System ist in Abbildung 16 dargestellt. Die Ausgangssituation ist durch die in der Abbildung in „snapshot 1“ gezeigte Faktenbasis und eine Regelbasis mit den Regeln `Angebtostregel1` und `Angebotsregel2` aus Listing 12 gegeben.

Wie in „snapshot 2“ zu sehen, stellt der Patternmatcher fest, dass die Bedingungsteile beider Regeln aufgrund der Faktenlage erfüllt sind. Die Bedingung lautet „Warenkorb enthält Holzbaukasten“ und dieser Fakt ist in der Faktenbasis enthalten. Dementsprechend hat er beide auf die Agenda eingetragen. In welcher Reihenfolge sie eingetragen werden, hat er aufgrund seiner Konfliktlösungsstrategie entschieden. In diesem Fall steht die `Angebotsregel1` an erster Stelle auf der Agenda. Die Agenda ist also nicht leer, was bedeutet, dass die Execution-Engine zum Einsatz kommt und die Angebotsregel1 feuert. Das heißt, er führt den im Aktionsteil der Regel enthaltenen Code aus. In diesem Fall enthält der Code keine Anweisungen die Faktenbasis zu ändern. Es handelt sich also nur um Seiteneffekte auf das System. Nachdem der Code abgearbeitet wurde, löscht die Execution-Engine den Aktionsteil der `Angebotsregel1` von der Agenda. Dies ist in „snapshot3“ zu sehen. Da die Execution-Engine nicht weiß, ob die Faktenbasis durch ihre Aktionen verändert wurde, muss sie einen neuen Inferenzen-Zyklus einleiten. Es könnte ja sein, dass der Bedingungsteil der als nächstes auf der Agenda stehenden Regel gar nicht mehr erfüllt ist und deswegen die Regel auch nicht mehr gefeuert werden darf. In dem neuen Zyklus stellt der Patternmatcher keine Veränderungen der Faktenbasis fest. Dementsprechend bleibt es auch bei dem einen Eintrag auf der Agenda. Dies ist in „snapshot 4“ zu sehen. Die Execution-Engine feuert die `Angebotsregel2` nun auch noch und entfernt sie ebenfalls von der Agenda, die nun, wie in „snapshot 5“ zu sehen, leer ist. Da sie wieder nicht weiß, ob sie die Fakten verändert hat, leitet sie erneut einen Zyklus ein. Der Patternmatcher stellt wieder keine Änderungen fest und die Agenda bleibt leer. Da die Execution-Engine nicht aufgerufen wird, wenn die Agenda leer ist und auch kein neuer Zyklus eingeleitet wird, endet der Vorgang hier und der Kontrollfluss geht wieder an das aufrufende System zurück.

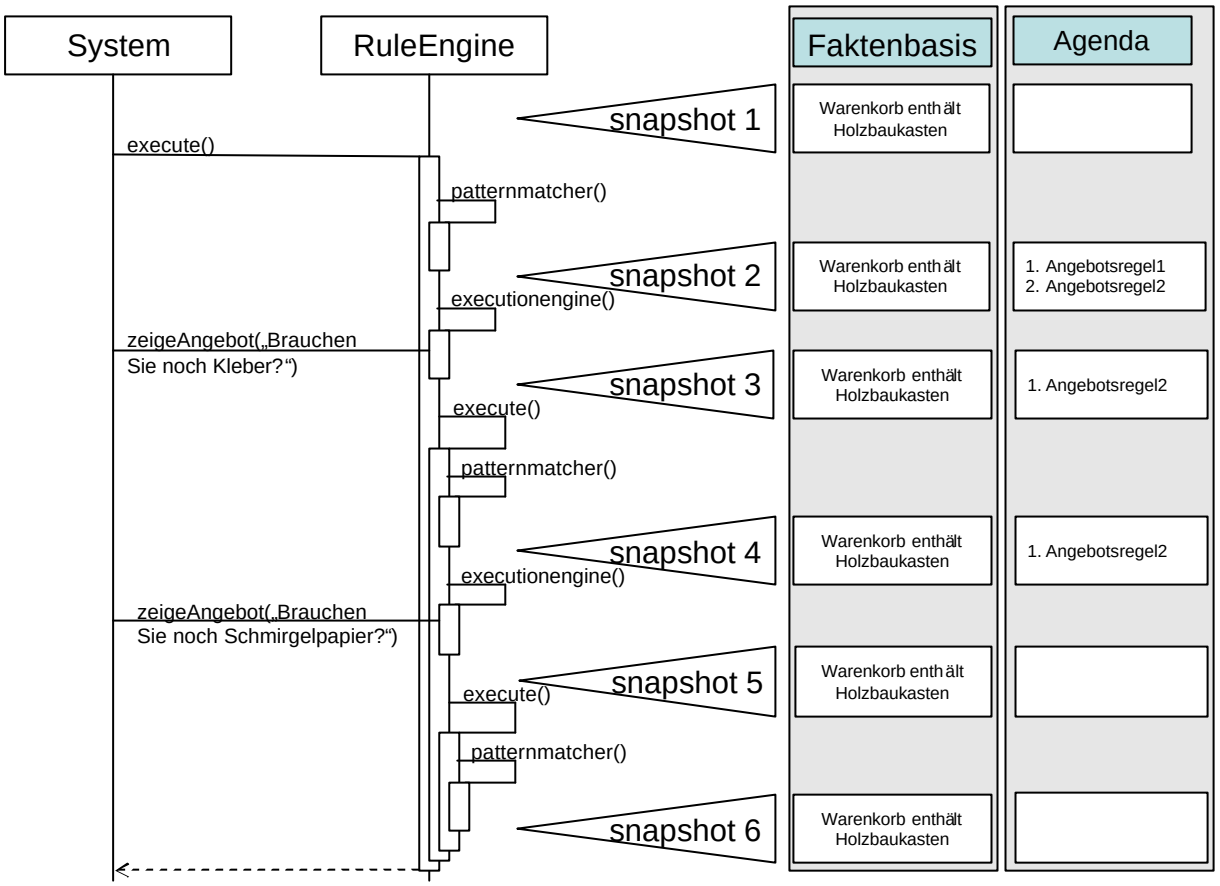

Abbildung 16: In diesem Beispiel wird der Inferenzen-Zyklus dreimal durchlaufen. Die Faktenbasis wird in diesem Beispiel von den Aktionen der Execution-Engine nicht verändert.

Inferenzen-Zyklus (Beispiel 2):

Angenommen dem Kunden soll pro Einkauf nur ein Angebot gemacht werden. Dann müsste man die Regeln wie in Listing 13 erweitern:

Listing 13

```
Angebotsregel1ext:
      IF
            Warenkorb enthält Holzbaukasten AND
            Kunde hat noch kein Angebot erhalten
      THEN
            zeigeAngebot(„Brauchen Sie auch Holzleim?“)
            schreibe in die Faktenbasis: Kunde hat ein Angebot erhalten


Angebotsregel2ext:
      IF
            Warenkorb enthält Holzbaukasten AND
            Kunde hat noch kein Angebot erhalten
      THEN
```

```
zeigeAngebot(„Brauchen Sie noch Schmirgelpapier?“)
schreibe in die Faktenbasis: Kunde hat Angebot erhalten
```

Die Vorgänge nach einem `execute()`-Aufruf in der Regel-Engine mit diesen Regeln als Regelbasis sind in Abbildung 17 dargestellt. Der wesentliche Unterschied zu dem ersten Beispiel spielt sich zwischen dem „snapshot 3“ und „snapshot 4“ ab. Dieses Mal enthält der Aktionsteil der gefeuerten Regel Anweisungen zur Manipulation der Faktenbasis. In ihr wird ein Fakt gespeichert, der aussagt, dass der Kunde bereits ein Angebot bekommen hat. In dem folgenden Inferenzen-Zyklus stellt der Pattermatcher fest, dass der Bedingungsteil der `Angebotsregel2ext` nicht mehr erfüllt ist und löscht sie von der Agenda die nun leer ist. Demzufolge wird der IF-Teil der `execute()`-Funktion nicht mehr ausgeführt und die Rekursion stoppt in diesem Beispiel bereits nach zwei Zyklen. Durch die Modifikation der Regeln ist also der gewünschte Effekt eingetreten: Der Kunde hat bloß ein Angebot erhalten.

Sollte es wichtig sein, welches Angebot dem Kunden offeriert wird, muss die Regel-Engine eine Möglichkeit bereitstellen, den Regeln Prioritäten zuzuordnen. Wenn man in diesem Beispiel nun der Regel `Angebotsregel2ext` eine höhere Priorität als der `Angebotsregel1ext` gegeben hätte, wäre sie auf der Agenda an erster Stelle gelandet und somit auch ausgeführt worden. Der Kunde hätte das Schmirgelpapier statt dem Kleber angeboten bekommen.

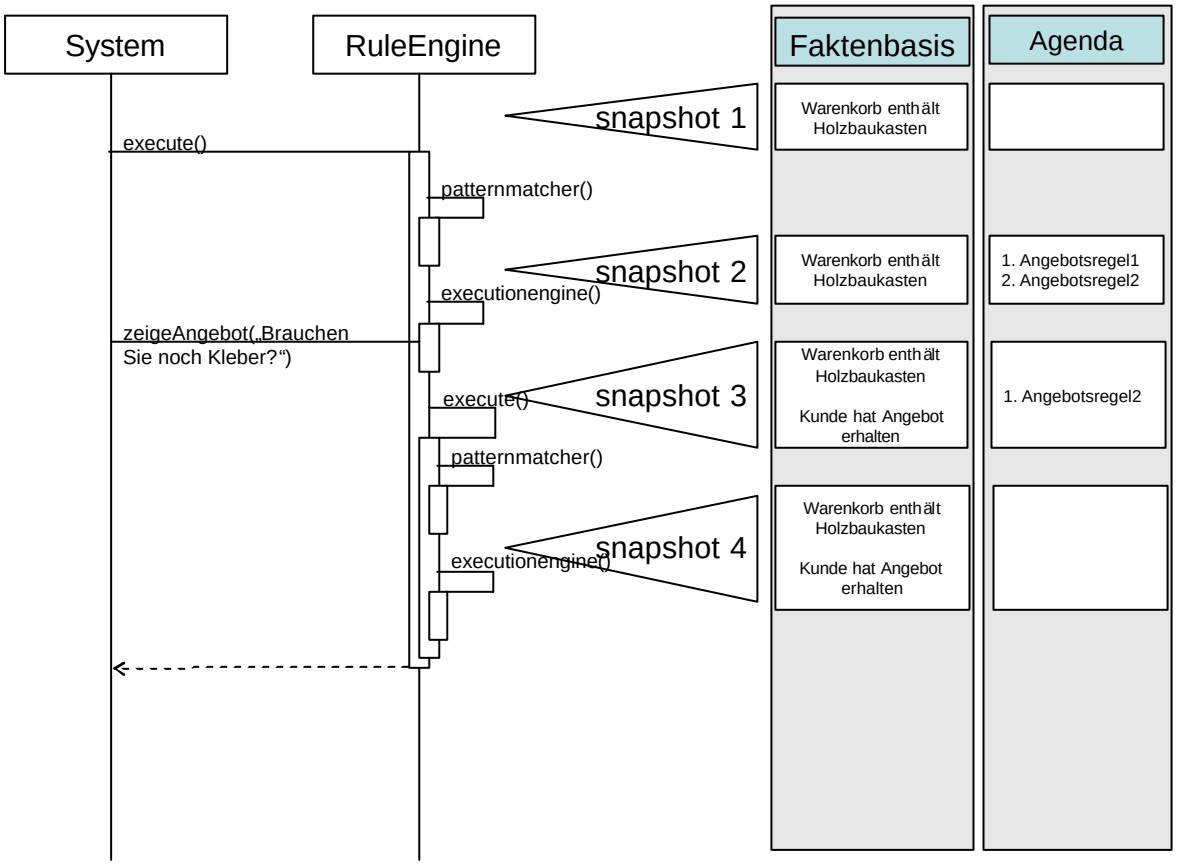

Abbildung 17: In diesem Beispiel wird der Inferenzen-Zyklus nur zweimal durchlaufen, weil die erste Angebotsregel die Faktenbasis so verändert, dass der Bedingungsteil der zweiten Regel auf der Agenda nicht mehr erfüllt ist.

3.4. Der RETE-Algorithmus

Eine Regel-Engine, wie sie bis hierher vorgestellt wurde, scheint eine recht einfache Angelegenheit zu sein. Wenn Zeit keine Rolle spielen würde wäre sie das auch. In jedem Zyklus würde der Patternmatcher mit einem einfachen Algorithmus alle Regel gegen alle Fakten prüfen und auf diese Weise die Regeln mit erfülltem Bedingungsteil finden.

In dem Kapitel

Die Arbeitsweise einer Regel-Engine sollte jedoch deutlich geworden sein, dass eine Regel-Engine bzw. ihr Patternmatcher während ihrer Lebenszeit sehr häufig überprüfen muss, ob Bedingungsteile von Regeln durch die gegebenen Fakten erfüllt sind. Diese Häufigkeit ergibt sich zum einen daraus, dass es rekursive `execute()`-Aufrufe gibt. Insbesondere dann, wenn sich aus der Art der Regeln ein Rule-Chaining[14] ergibt. Zum anderen ist es in vielen Anwendungsfällen notwendig, immer wieder zu überprüfen, ob inzwischen die Bedingungsteile anderer Regeln erfüllt sind.

Sollte ein Kunde des Online-Bastel-Geschäfts sofort informiert werden, wenn ein Angebot für ihn in Frage kommt, dann müsste der Patternmatcher nach jeder Änderung des Warenkorbs bzw. der Faktenbasis die Regeln überprüfen.

Eine wiederholte Überprüfung der Regeln auf einer kaum veränderten Faktenbasis ist also in vielen Fällen sehr typisch. Unter anderem macht sich der RETE-Algorithmus genau diese Tatsache zu nutze. Er merkt sich die Ergebnisse aus vorigen Durchläufen und überprüft nur die Regeln, die auch tatsächlich von der Änderung an den Fakten betroffen sein können.

Aus diesem Grund ist der RETE-Algorithmus auch nicht immer vorteilhaft gegenüber einem einfachen Algorithmus. Wenn einem Kunden des Online-Bastel-Geschäftes die Angebote erst am Ende des Bestellvorganges gemacht werden sollen, sagen wir nach dem Klicken des „Bestellung abschicken“-Buttons, dann wäre das ein einmaliger Vorgang und der RETE-Algorithmus könnte seinen Trumpf, sein Gedächtnis, nicht ausspielen. Es sei denn, die Regeln führen zu Chaining-Effekten, deretwegen es einen tiefen rekursiven Aufruf der `execute()`-Methode gibt.

Das Netzwerk

Rete ist lateinisch und bedeutet Netz. Das von dem RETE-Algorithmus verwendete Netz ist ein azyklischer, gerichteter Graph mit einem Ausgangsknoten von dem aus alle anderen Knoten erreicht werden können. Der Graph besteht neben der Wurzel aus zwei weiteren Typen von Knoten, den alpha- und den beta-Knoten. Die alpha-Knoten haben eine, und die beta-Knoten zwei eingehende Kanten. Beide können keine oder mehrere ausgehende Kanten haben.

Das Netzwerk wird aus den Bedingungsteilen der Regeln erzeugt. Bei den Bedingungen kann man grundsätzlich zwischen zwei Kategorien unterscheiden. Die eine betrifft nur einen isolierten Fakt, die andere betrifft zwei Fakten, die über eine Variablenbindung miteinander verknüpft sind.

Für jede Bedingung der ersten Kategorie entsteht ein alpha-Knoten und für jede Bedingung der zweiten ein beta-Knoten.

[14] Rule-Chaining: Das Ergebnis der Auswertung einer Regel bedingt die Auswertung weiterer Regeln. (http://www.mathema.de/event/publikation/campus_2003_11/RuleEngines.pdf)

Listing 14

```
(defrule KörbeVonMännern
      (Kunde (name ?n) (anrede Herr))
      (Warenkorb (name ?n))
=>
)
```

Aus dieser Regel würde ein Netzwerk wie das in Abbildung 18 erzeugt werden.

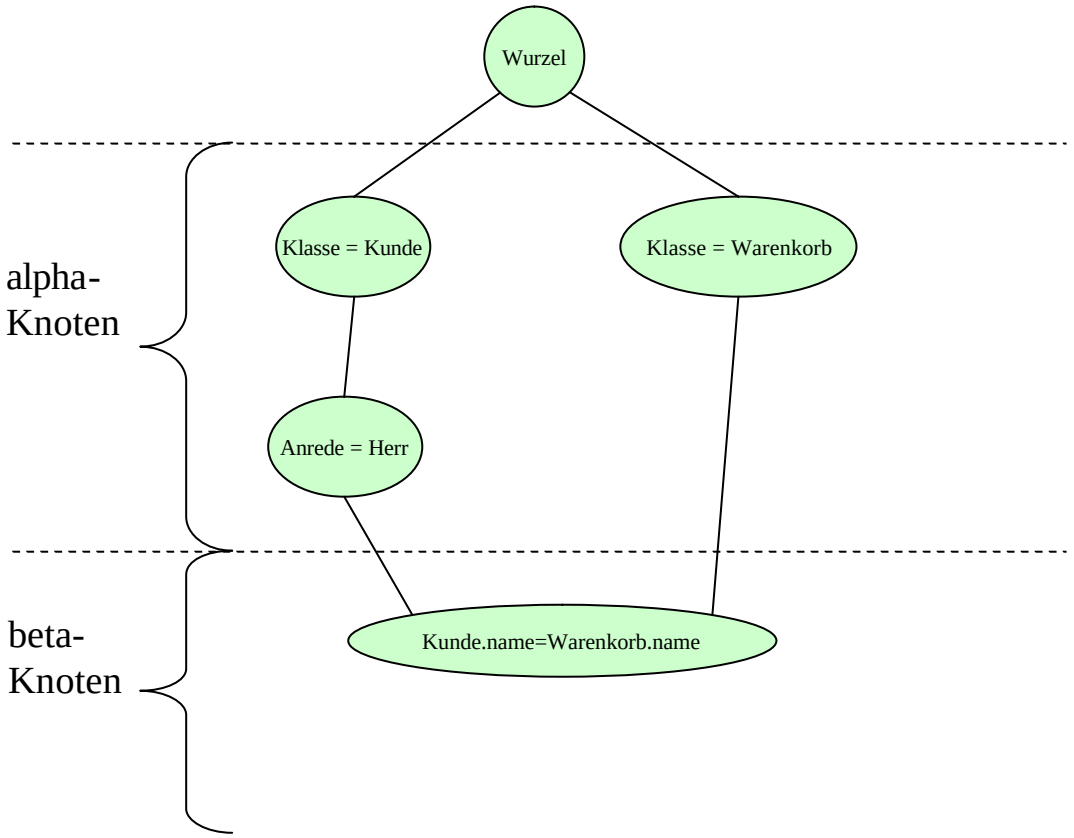

Abbildung 18: Ein RETE-Netzwerk. Alpha-Knoten haben eine eingehende Kanten, beta-Knoten zwei.

In ein solches Netzwerk können nun Fakten eingefügt werden. Ausgehend von der Wurzel werden die Fakten zunächst in den alpha-Knoten auf Bedingungen geprüft. Erfüllen sie die Bedingung, die der Knoten enthält, werden sie in dem Knoten den sie „bestanden“ haben gespeichert und an die folgenden Knoten weiter gegeben. In den beta-Knoten werden die eingehenden Fakten paarweise überprüft, ob sie die Verknüpfungsbedingung erfüllen. Wenn dies der Fall ist, werden sie als Tupel in dem Knoten gespeichert.

Dieser Prozess lässt sich wie in Abbildung 19 anhand eines Beispiels visualisieren. Zu Beginn sind die mit den Knoten assoziierten Speicher leer. Jeder Fakt, der eingefügt wird, durchläuft das Netzwerk ausgehend von der Wurzel auf allen möglichen Pfaden so weit, bis

er an einem Knoten scheitert. Dabei merkt sich jeder Knoten alle Fakten, bzw. die Kombination der Fakten, die ihn passiert haben. Pro Regel gibt es ein Blatt, also einen Knoten ohne ausgehende Kante.

Würde man zum Beispiel die folgenden Fakten in das Netzwerk aus Abbildung 18 einfügen, würden an den Knoten die in Abbildung 19 dargestellte Situation entstehen.

Listing 15

```
(Kunde (name Huber) (anrede Frau))
(Kunde (name Mayer) (anrede Herr))
(Kunde (name Müller) (anrede Herr))
(Warenkorb (id 7070) (anzahlAngebote 0) (name Huber))
(Warenkorb (id 0815) (anzahlAngebote 0) (name Müller))
```

Der einzige beta-Knoten, der gleichzeitig ein Blatt ist und somit einer Regel entsprechen muss, in diesem Fall der `KörbeVonMännern`-Regel, hat ein zweier-Tupel von Fakten gespeichert. Es gibt unter den Fakten also genau eine Kombination von ihnen, die den Bedingungsteil der Regel erfüllt. Es gibt zurzeit also nur einen Warenkorb, der zu einem Mann gehört.

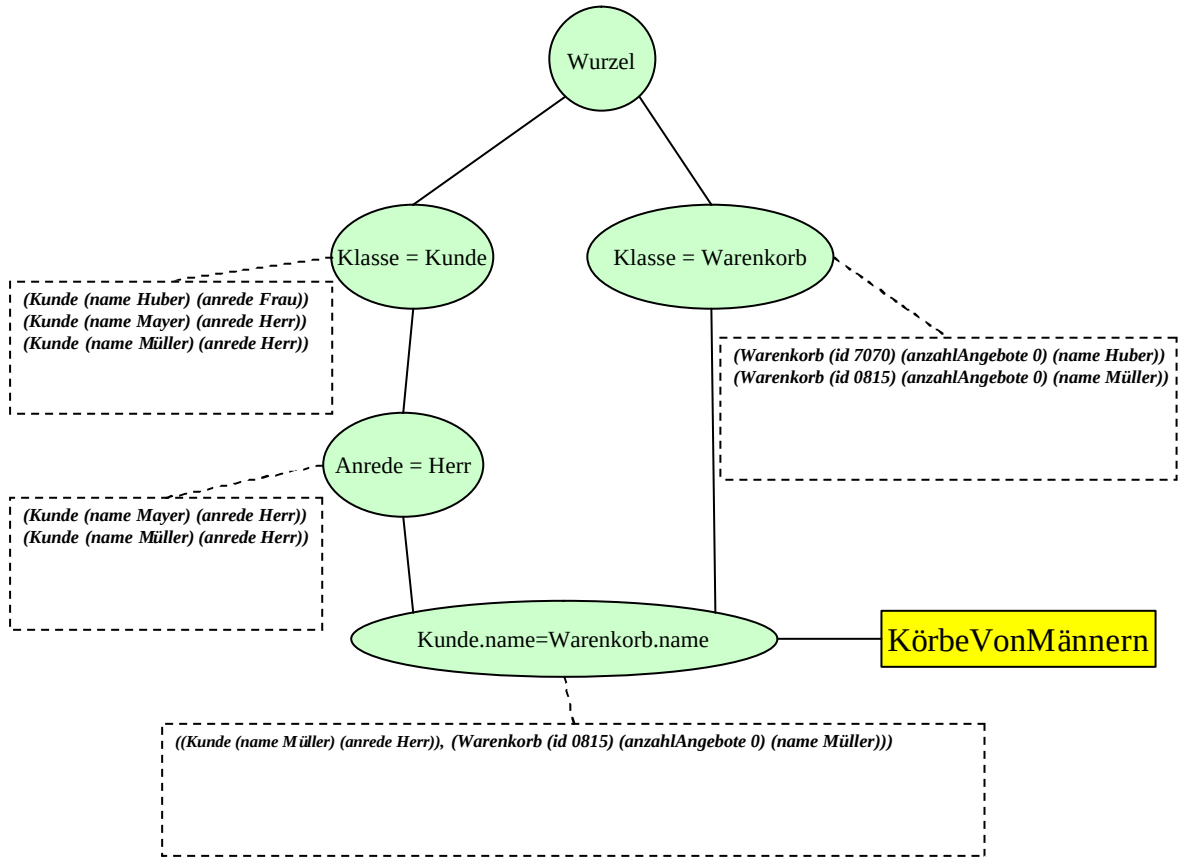

Abbildung 19: Die Fakttupel, die die Bedingungen der vorangegangenen Knoten erfüllt haben, werden gespeichert.

In diesem Fall wurde also eine Regel in ein RETE-Netzwerk übersetzt. Wenn es sich um mehrere Regeln handelt, kann man die sich aus ihnen ergebenen Netzwerke verschmelzen, indem man die Wurzeln zu einer neuen Wurzel zusammenfasst und alle Knoten, die von ihr auf gleichen Pfaden erreichbar sind ebenfalls verschmilzt. Zwei Pfade sind gleich, wenn die Knoten auf ihnen die gleichen Bedingungen in der gleichen Reihenfolge prüfen. Durch dieses Vorgehen wird die Effizienz des RETE-Netzwerks noch gesteigert. Wie viel Verbesserungspotential in diesem Vorgehen steckt ist von den Regeln abhängig.

Angenommen man wollte neben der `KörbeVonMännern`-Regel noch eine `KörbeVonMännernOhneAngebot`-Regel haben, dann könnte man die aus diesen beiden Regeln entstehenden RETE-Netzwerke wie in Abbildung 20 dargestellt zusammenführen.

Das Netzwerk mit den grünen Knoten ist aus der `KörbeVonMännern`-Regel entstanden und das blaue aus der `KörbeVonMännernOhneAngebot`-Regel. Die sich in der Zeichnung überschneidenden Knoten können zusammengefasst werden, da sie einen gleichen Pfad zur Wurzel haben.

Listing 16

```
(defrule KörbeVonMännernOhneAngebot
      (Kunde (name ?n) (anrede Herr ))
      (Warenkorb (name ?n) (anzahlAngebote 0))
=>
)
```

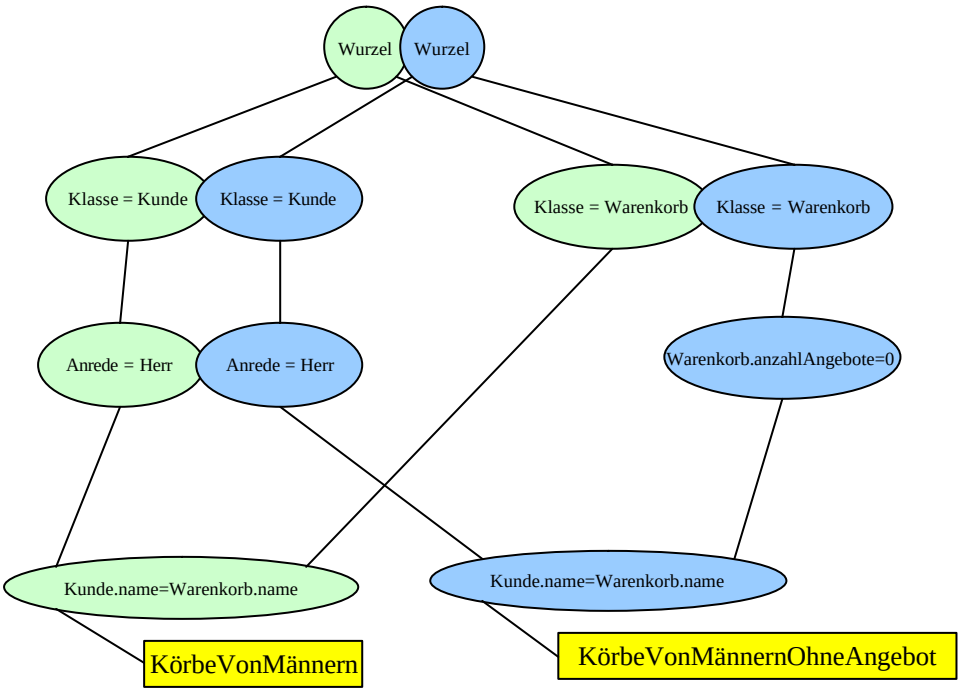

Abbildung 20 : Zur Optimierung können die sich aus den Regeln ergebenen Netzwerke verschmolzen werden.

Effizienz

Der RETE-Algorithmus ist auf viele Inferenzen-Zyklen auf einer sich kaum ändernden Faktenbasis spezialisiert. (siehe

Die Arbeitsweise einer Regel-Engine S.36) Unter solchen Umständen ist er einfachen Ansätzen um Längen voraus.

In [FRIED] S. 141 wird die durchschnittliche Laufzeit pro Zyklus wie folgt abgeschätzt:

$$R'F'^{P'}$$

R' ist kleiner als die Anzahl der Regeln.
F' ist die Anzahl der sich durchschnittlich pro Zyklus ändernden Fakten.
P' ist kleiner als die durchschnittliche Anzahl der Bedingungen pro Regel.

Ein einfacher Algorithmus würde im Wesentlichen die Laufzeit

$$RF^P$$

haben.

R ist die Anzahl der Regeln.
F ist die Anzahl der Fakten.
P ist die durchschnittliche Anzahl der Bedingungen pro Regeln.

D.h. für jede Regel würde er F^P Schritte benötigen, da er für jede Bedingung jeden mit jeden Fakt „matchen" muss.

In dem ersten Zyklus, in dem der RETE-Algorithmus sein Netzwerk mit den Fakten erst einmal füllen muss ähnelt die Laufzeit dem des einfachen Algorithmus. Insbesondere, wenn die Regeln nicht viele Gemeinsamkeiten haben und deswegen nur wenige Knoten der Netzwerke miteinander vereinigt werden können (siehe Das Netzwerk). [FRIED, S. 141]

In den weiteren Zyklen hat der RETE-Algorithmus in vielen Fällen aber große Vorteile.

Ändern sich zum Beispiel von einem Zyklus zum anderen nur $F'=3$ von $F=100$ Fakten und diese 3 Fakten betreffen nur $R'=2$ der $R=10$ Regeln und diese 2 Regeln haben durchschnittlich nur $P'=2$ statt $P=3$ Bedingungen, dann ergibt sich eine Laufzeitverbesserung von

$$RF^P = 10*100^3 = 10\,000\,000$$

zu

$$R'F'^{P'} = 2*3^2 = 18.$$

Zugegebenerweise ist der hier verwendete einfache Algorithmus sehr primitiv. Durch indizierte Listen und andere Vorkehrungen könnte seine Laufzeit erheblich verbessert

werden. Nichts desto trotz zeigt obige Rechnung, dass es sich lohnt, die Ergebnisse aus früheren Matchingvorgängen abzuspeichern.

Es sollte aber nicht vergessen werden, dass man Laufzeitverbesserung durch einen erheblich erhöhten Bedarf an Speicherplatz erkauft hat.

3.5. Verkettung von Regeln

Man spricht von Rule-Chaining, wenn die Auswertung einer Regel die Auswertung weiterer Regeln bedingt[15]. Hierbei kann man zwischen zwei Vorgehensweisen unterscheiden:

- **Forward-Chaining**: Vorgabe einer Faktenbasis auf deren Grundlage nach den möglichen Schlussfolgerungen gefragt wird.
- **Backward-Chaining**: Vorgabe einer Schlussfolgerung zu der die Fakten gesucht werden, die diese Schlussfolgerung erlauben.

Um dies ein bisschen anschaulicher zu machen, betrachtet man am besten ein kleines Beispiel:

Angenommen ein Barkeeper plant den Abend in seiner Bar und überlegt welche Drinks er seinen Gästen mit den vorhandenen Zutaten anbieten kann, dann arbeitet sein Gehirn nach dem Forward-Chaining-Prinzip. Wenn er aber nach einem Drink gefragt wird und überlegt, was er dafür bräuchte, dann arbeitet sein Gehirn nach dem Backward-Chaining-Prinzip.

Der RETE-Algorithmus ist für das Forward-Chaining ausgelegt. Man kann mit ihm aber auch einigermaßen effektiv Backward-Chaining simulieren [FRIED, S.116.]

Die Aufgabe in dieser Diplomarbeit ist es, die Konsistenz eines Dokumentes zu seinem Modell zu überprüfen. Das heißt, es muss überprüft werden, ob eine, das Dokument beschreibende Faktenmenge, bestimmte Bedingungen erfüllt. Für diese Aufgabe benötigt man ein Forward-Chaining Regelsystem. Falls das Dokument nicht mehr konsistent zu seinem Metamodell ist, sollen Regeln gefeuert werden, die dazu führen, dass die Konsistenz wieder hergestellt wird. Dies soll entweder automatisch oder interaktiv über den Dokumentenbearbeiter geschehen.

[15] http://www.mathema.de/event/publikation/campus_2003_11/RuleEngines.pdf S.10

4. Anforderungsanalyse

Durch das zweite Kapitel sollte zunächst ein Überblick über die Aufgabenstellung der Diplomarbeit entstanden sein. Im dritten Kapitel wurde die Funktionsweise von Regel-Engines besprochen. Es hat sich gezeigt, dass sie sich hervorragend für eine ständige Überwachung der Konsistenz eines Dokumentes eignen.

In diesem Kapitel sollen die Anforderungen erläutert werden, die sich während der Analysephase vor allem durch Gespräche mit Klaus Bergner[16] ergaben. Durch weitere Gespräche mit Michael Gnatz[17] als Dokumentenbearbeiter und Regelersteller ergaben sich spezielle Anforderungen, die bei einem Einsatz von 4Ever als 200X-Vorgehensmodell-Editor gestellt werden.[18]

In dem Abschnitt Begriffsdefinitionen werden einige Begriffe eingeführt, die in dem restlichen Teil der Diplomarbeit benötigt werden. Der Abschnitt Anwendungsfälle erklärt die identifizierten Anwendungsfälle eines Dokumentenbearbeiters im Umgang mit 4Ever. Aus diesen Anwendungsfällen ergeben sich die Anforderungen an die Erweiterung von 4Ever. Sie sind im letzten Abschnitt dieses Kapitels, nach funktionalen und technischen Anforderungen geordnet, aufgeführt.

4.1. Begriffsdefinitionen

Eine **Metabedingung** ist eine Aussage über die Elemente eines Dokumentes, die entweder wahr oder falsch sein kann. Ein Metamodell besteht aus einer Menge solcher Bedingungen.

Wenn alle Metabedingungen wahr sind, ist das Dokument **konsistent**. Ansonsten ist es **inkonsistent**.

Eine Metabedingung ist eine Aussage über eine Teilmenge der Elemente. Verletzt ein Dokument eine Bedingung führt dies zu einem **Konflikt**.

Eine **Konfliktstelle** ist ein Element des Dokumentes, durch dessen Änderung der Konflikt aufgehoben werden kann. Es kann mehrere Konfliktstellen pro Konflikt geben. Diese Zusammenhänge sind in Abbildung 21 visualisiert.

Durch eine **Prüfung** wird festgestellt, ob eine Metabedingung des Metamodells durch ein Dokument verletzt wird.

4EverRulez ist die zu entwickelnde 4Ever-Komponente, mit deren Hilfe dem Dokumentenbearbeiter eine Konsistenzsicherung bzw. eine Konsistenzwiederherstellung ermöglicht werden soll.

[16] 4Soft Geschäftsführung
[17] sowohl Experte als auch Dokumentenbearbeiter
[18] z.B. Speichern von Dokumenten muss immer möglich sein

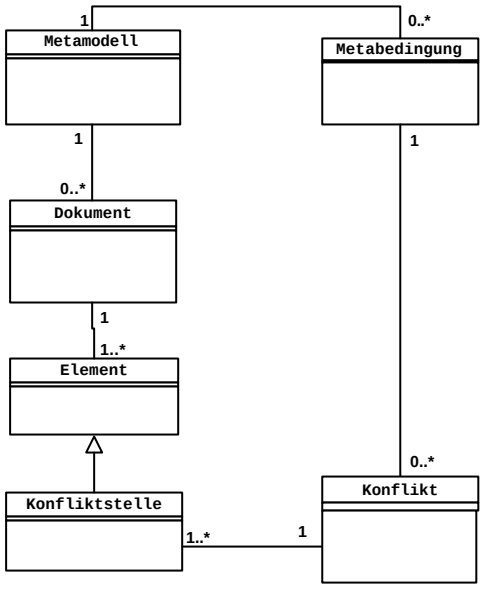

Abbildung 21: Die Begriffe und ihre Zusammenhänge als Klassendiagramm.

4.2. Anwendungsfälle

Die identifizierten Anwendungsfälle des Dokumentenbearbeiters sind in dem Anwendungsfall-Diagramm in Abbildung 22 dargestellt. Im Folgenden werden sie einzeln erläutert.

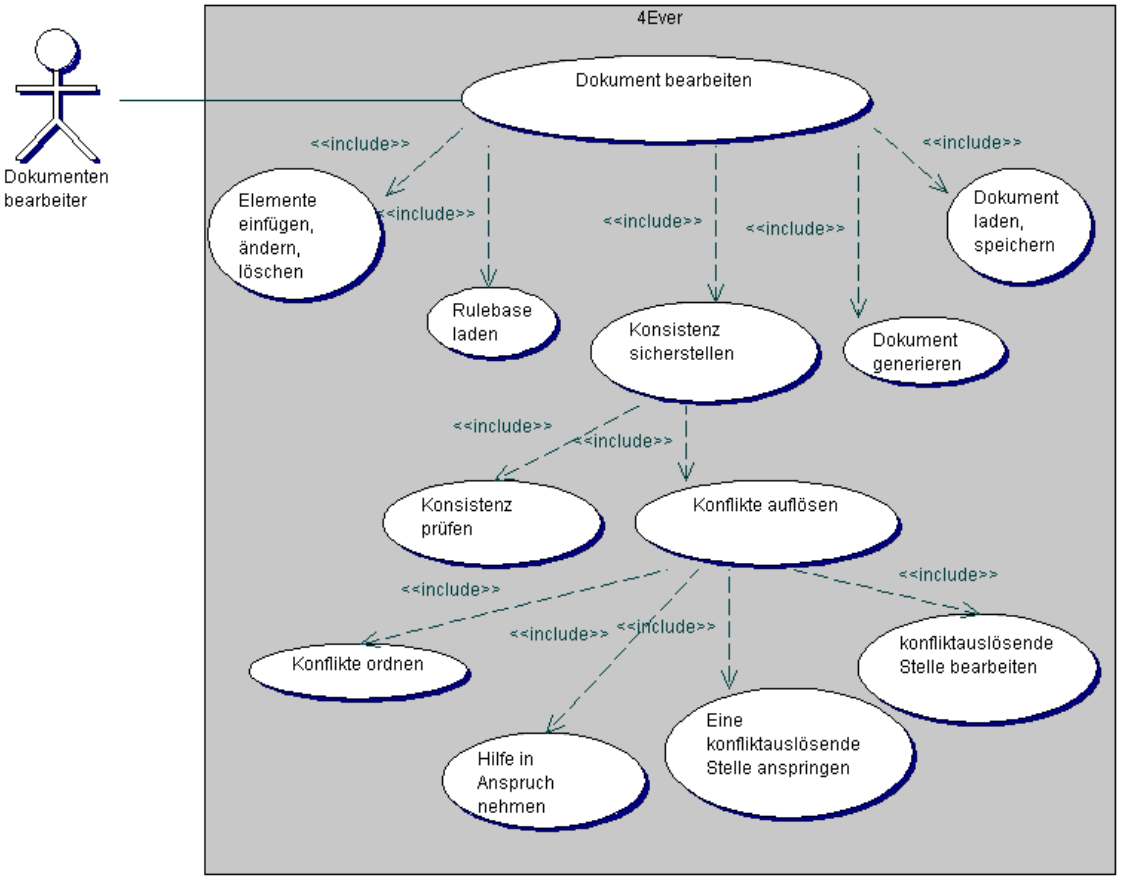

Abbildung 22: Ein Anwendungsfall-Diagramm mit den wichtigsten Anwendungsfällen eines Dokumentenbearbeiters.

Elemente einfügen, ändern, löschen: Dieser Anwendungsfall betrifft das eigentliche Bearbeiten eines Dokumentes. Dazu gehört z.B. das Schreiben eines HTML-Textes in einer 4Ever-EditorPane. Dieser Anwendungsfall ist in 4Ever natürlich schon möglich und bleibt für das 4EverRulez-Projekt auch unverändert.

Rulebase laden: Durch diesen Anwendungsfall werden die vom Experten definierten Bedingungen der 4EverRulez-Komponente zur Verfügung gestellt. (siehe Persistenz in 4Ever durch XML und XML-Schema S.19) Dieser Anwendungsfall sollte auf alle Fälle bei der Planung einbezogen werden. Eine Umsetzung ist vorläufig aber noch nicht notwendig. Insbesondere bei der Bearbeitung von V200X-Dokumenten soll es für den Dokumentenbearbeiter keine Möglichkeit geben, das Metamodell zu ändern.

Konsistenz sicherstellen: Der Anwendungsfall der Konsistenzsicherstellung ist der zentrale Aspekt des 4EverRulez-Projektes. Er umfasst alle Aktivitäten, die seitens des Dokumentenbearbeiters notwendig sind, um die Konsistenz eines Dokumentes zu seinem Metamodell zu gewährleisten.

Konsistenz prüfen: Es soll möglich sein, alle vorhandenen Bedingungen zu einem, vom Dokumentenbearbeiter bestimmten Zeitpunkt zu prüfen.

Konflikte auflösen: Ein sehr wichtiger Anwendungsfall ist das Auflösen von Konflikten. Wenn ein Dokumentenbearbeiter ein Dokument so abgeändert hat, dass es nicht mehr konsistent zu seinem Metamodell ist, muss ihm das System eine Möglichkeit zur Auflösung der Konflikte bieten. Ohne Hinweise wäre für den Anwender wahrscheinlich unmöglich die

Konfliktstellen zu finden. Deswegen könnte er das Dokument nicht wieder in einen konsistenten Zustand bringen. (siehe Konsistenzerhaltung und Konsistenzwiederherstellung S.23)

Konflikte ordnen: Um bei der Auflösung von Konflikten systematisch vorgehen zu können, ist es sinnvoll, dem Dokumentenbearbeiter eine Möglichkeit zu bieten, die Konflikte nach bestimmten Kriterien ordnen zu können.

Hilfe in Anspruch nehmen: Wenn das Dokument inkonsistent ist und der Dokumentenbearbeiter nicht weiß woran es liegt, soll er Hilfe für die Konsistenzwiederherstellung in Anspruch nehmen können.

Eine Konfliktstelle anspringen: Konflikte beruhen auf Stellen im Dokument, die einer Bedingung widersprechen. Um den Konflikt zu beheben ist es notwendig, eine oder mehrerer dieser Konfliktstellen zu bearbeiten.

Durch das Anspringen der Konfliktstellen wird das entsprechende Element im Editor-Panel (siehe Abbildung 2 S.13) geöffnet. Anschließend kann eine Bearbeitung der Konfliktstelle erfolgen.

Dokument generieren: Dieser Anwendungsfall ist in 4Ever bereits realisiert und wird nicht verändert. Allerdings wird das Generieren eventuell bei Inkonsistenz blockiert. Dadurch können keine Sichten, also z.B. PDF-Dateien, eines inkonsistenten Dokumentes erzeugt bzw. generiert werden.

Dateien speichern, laden: Dieser Anwendungsfall ist in 4Ever natürlich auch schon realisiert. Dokumente können als XML-Datei im Dateisystem abgespeichert und wieder geladen werden. (siehe Persistenz in 4Ever durch XML und XML-Schema S.19) Wird vom Dokumentenbearbeiter ein inkonsistentes Dokument gespeichert, sollte er eine Warnung erhalten.

4.3. Anforderungen

In diesem Abschnitt sollen die Anforderungen an 4Ever mit der 4EverRulez-Komponente aufgelistet werden. Sie werden dabei in funktionale und technische Anforderungen aufgeteilt. Zudem erhalten die Anforderungen durch die Schlüsselworte „muss“, „sollte“ und „kann“ eine Priorität. Ihre Bedeutung ist folgendermaßen:

Muss: Ohne Erfüllung dieser Anforderung macht das System wenig oder keinen Sinn.

Sollte: Die Erfüllung dieser Anforderung würde große Vorteile bringen.

Kann: Ohne Erfüllung dieser Anforderungen kann das System noch sinnvoll eingesetzt werden.

Funktionale Anforderungen

- **FA1**: 4EverRulez muss fähig sein, den Dokumentenbearbeiter über die Anzahl der aktuellen Konflikte zu informieren.

? **FA2**: 4EverRulez muss fähig sein, den Dokumentenbearbeiter über die, den Konflikten zugrunde liegenden, Metabedingungen zu informieren.

? **FA2**: 4EverRulez kann dem Dokumentenbearbeiter die Möglichkeit bieten, die aktuellen Konflikte nach verschiedenen Eigenschaften zu ordnen.

? **FA3**: 4EverRulez sollte dem Dokumentenbearbeiter die Möglichkeit bieten, Konfliktstellen schnell und einfach zu erreichen. (Nach maximal zwei Aktionen (z.B. Mausklicks) sollte das zu bearbeitende Element geöffnet sein.)

? **FA4**: 4EverRulez muss fähig sein, den Dokumentenbearbeiter über einen Konflikt so ausführlich zu informieren, dass dieser in der Lage ist, den Konflikt zu beheben.

? **FA5**: 4EverRulez kann dem Dokumentenbearbeiter die Möglichkeit bieten, eine Sprache für die Konflikthilfe auszuwählen.

? **FA6**: 4EverRulez muss dem Dokumentenbearbeiter die Möglichkeit erhalten, alle konsistenten Zustände eines Dokumentes zu erreichen.

? **FA7**: 4EverRulez muss dem Dokumentenbearbeiter die Möglichkeit erhalten, ein Dokument jederzeit abspeichern zu können.

? **FA8**: 4EverRulez sollte dem Dokumentenbearbeiter die Möglichkeit bieten, alle Prüfungen temporär zu deaktivieren.

? **FA9**: 4EverRulez muss dem Dokumentenbearbeiter die Möglichkeit bieten, eine Prüfung aller Bedingungen manuell auszulösen.

? **FA10**: 4EverRulez sollte fähig sein, die Konsistenz im Hintergrund zu überwachen.

? **FA11**: 4EverRulez sollte fähig sein, bestimmte Aktionen, wie das Generieren von Sichten, bei bestimmten Konflikten zu blockieren.

Technische Anforderungen

? **TA1**: 4EverRulez muss eine optionale Komponente von 4Ever sein.

? **TA2**: 4EverRulez sollte durch einen Adapter an unterschiedliche Objektmodelle angepasst werden können.

? **TA3**: 4EverRulez sollte die Möglichkeit bieten, für verschiedene Regel-Engines konfigurierbar zu sein.

? **TA4**: 4EverRulez muss in Java implementiert sein.

? **TA5**: 4EverRulez muss eine Schnittstelle für die Metamodelldefinition bereitstellen.

5. Fachliche Konzeption

Dieses Kapitel diskutiert ein Konzept zur Erfüllung der festgestellten Anforderungen unter Berücksichtigung der gegebenen Randbedingungen. Diese Randbedingungen sind die Nutzung der schon vorhandenen 4Ever-Komponenten und der Einsatz einer Regel-Engine für die Konsistenzprüfung.

Zunächst wird eine Gesamtübersicht über die Problemstellung gegeben, die dann gemäß dem Grundsatz „teile und herrsche“ für eine bessere Handhabbarkeit in mehrere unabhängige Problemfelder aufgeteilt wird. Durch die Analyse dieser Problemfelder ergibt sich ein mögliches Konzept zur Realisierung einer Lösung.

Um dem Leser das Konzept verständlich zu machen, wird eine auf diesem Konzept basierende graphische Benutzeroberfläche vorgestellt. Anhand dieser Benutzeroberfläche werden einige Anwendungsfälle durchgespielt. Die problemlose Durchführung der Anwendungsfälle mit dem Benutzerinterface bestätigt gleichzeitig die Richtigkeit des Konzepts.

5.1. Aufteilung der Problemstellung

Im Folgenden wird die recht komplexe Problemstellung die durch die 4EverRulez-Komponente gelöst werden soll in drei Teilprobleme zerteilt (siehe Abbildung 23). Diese Teilprobleme sind relativ unabhängig voneinander. Aus diesem Grund ist es möglich, den Lösungsansatz für jedes Teilproblem einzeln zu erläutern.

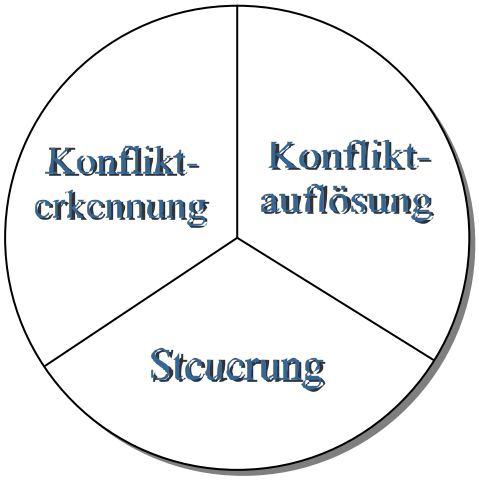

Abbildung 23: Die Aufgabenstellung wird in drei Teilprobleme aufgeteilt.

Konflikterkennung

Für die Erkennung von Konflikten kommt eine Regel-Engine zur Anwendung. Damit die Regel-Engine Konflikte erkennen kann, müssen zwei Vorraussetzungen gegeben sein.

1) Die Regel-Engine muss das Metamodell kennen.
2) Die Regel-Engine muss über den Zustand des Dokumentes informiert werden.

Der erste Punkt wird gelöst, indem jede Metamodellbedingung negiert wird. Anschließend wird für jede Metamodellbedingung eine Regel definiert, deren Bedingungsteil (siehe S. 35) die negierte Metamodellbedingung enthält. Sobald nun eine Inkonsistenz des Dokumentes auftritt, ist die negierte Metamodellbedingung erfüllt und die Regel feuert. Das Feuern der Regel bewirkt die Erzeugung eines Konfliktobjektes. Jedes Konfliktobjekt entspricht also einer Verletzung einer Metamodellbedingung durch das Dokument. Wenn das Dokument dieselbe Metabedingung mehrmals verletz, wird für jede Verletzung ein Konfliktobjekt erzeugt.

Die Lösung des zweiten Punktes basiert auf einem Beobachtungsmechanismus, der jede Änderung im Dokument der Regel-Engine mitteilt. Wegen der Anforderung TA2 (siehe Technische Anforderungen S. 52) wird dieser Beobachtungsmechanismus in einen austauschbaren Adapter gekapselt. (siehe Abbildung 24)

Um eine Beobachtung des Zustandes von dem 4Ever Objektmodell zu realisieren, implementiert der Adapter das `InstanceObserver`-Interface (siehe Abbildung 42) des Objektmodells. Dadurch kann er sich bei dem Objektmodell registrieren und wird gemäß dem Observer-Muster [GAMMA] über alle Änderungen des Objektmodells informiert. Diese Informationen kann der Adapter jetzt in eine für die Regel-Engine passende Form bringen und mitteilen.

Konfliktlösung

4EverRulez stellt ein Konsistenzwiederherstellungsverfahren (siehe Konsistenzerhaltung und Konsistenzwiederherstellung S.23) zur Verfügung. Es wird an dieser Stelle auf jeglichen Automatismus verzichtet. Das heißt, 4EverRulez hat keine Möglichkeit das Dokument zu verändern. Stattdessen wird ein manuelles Verfahren der Konsistenzwiederherstellung angewendet. Es werden also alle, zur Wiederherstellung der Konsistenz notwendigen Änderungen, von dem Dokumentenbearbeiter durchgeführt.

Um diese Änderungen durchführen zu können, muss der Dokumentenbearbeiter zwei Dinge wissen:

1) Welche Stellen in dem Dokument verursachen die Konflikte.
2) Wie müssen diese Stellen bearbeitet werden, damit die Konflikte beseitigt werden.

Genau diese Informationen werden dem Dokumentenbearbeiter von 4EverRulez zur Verfügung gestellt.

Ein Konflikt beruht meistens auf mehreren Konfliktstellen. Wie im vorangegangenen Abschnitt erläutert, enthalten die Bedingungsteile der Regeln negierte Metamodellbedingungen. Sobald eine Menge der Elemente des Dokumentes dieser negierten Metamodellbedingung entspricht, wird ein Konfliktobjekt erzeugt. In diesem Konfliktobjekt werden nun auch noch die für den Konflikt verantwortlichen Elemente bzw. Konfliktstellen gespeichert. Zusätzlich wird in dem Konfliktobjekt vermerkt, von welcher Regel es erzeugt wurde. Durch diese Informationen ist, mit Hintergrundwissen des Dokumentenbearbeiters, eine Behandlung der Konfliktstellen möglich.

Besitzt der Dokumentenbearbeiter das notwendige Hintergrundwissen aber nicht, dann wären Zusatzinformationen sinnvoll, die dem Dokumentenbearbeiter eine Hilfestellung geben. Optimal wäre es, wenn es keine allgemeine, sondern eine konfliktstellenspezifische Hilfestellung geben könnte. Das heißt, für jede an dem Konflikt beteiligte Stelle, kann der Dokumentenbearbeiter eine Hilfe abfragen, die ihm konkrete Anweisung für die Bearbeitung dieser speziellen Stelle im Dokument gibt. Man müsste also für jede Konfliktstelle eine spezielle Hilfe definieren. Dies ist aber nicht möglich, da die Anzahl der Konfliktstellen unter Umständen beliebig groß werden kann. Es ist jedoch so, dass Konfliktstellen in einem Konflikt immer in einem gewissen Kontext erscheinen. Diesen Kontext, in dem eine Konfliktstelle erscheint, wird **Rolle** genannt. Da die Anzahl der Rollen pro Konflikt gleich bleibt, ist es möglich, für jede Rolle eine Hilfestellung zu definieren. Die Hilfestellung, die von 4Ever für eine bestimmte Konfliktstelle gegeben wird, soll abhängig von der Rolle sein, die sie einnimmt. Dies macht auch Sinn, da die Art der notwendigen Behandlung für alle Stellen einer Rolle gleich ist. Kommt es zu einem Konflikt und eine Regel feuert, werden nicht nur die Konfliktstellen bei der Initialisierung des Konfliktobjektes gespeichert, sondern auch eine Rollenzuteilung. Dadurch ist es möglich, anhand des Konfliktobjektes zu erkennen, welche Stellen des Dokumentes welche Rollen in einem Konflikt spielen. Ist diese Information gegeben, lässt sich natürlich auch eine konfliktstellenspezifische Hilfestellung geben.

Steuerung

Das Metamodell kann aus sehr vielen und komplexen Metamodellbedingungen bestehen. Trotz der Nutzung des effizienten RETE-Algorithmus kann eine Konsistenzprüfung sehr viele Ressourcen in Anspruch nehmen, so dass die Dokumentenbearbeitung durch Verzögerungen gestört wird. Für diesen Fall ist es wünschenswert, die Regel-Engine nur zu bestimmten Zeitpunkten zu aktivieren. Noch besser wäre es, wenn man bestimmen könnte, welche Metamodellbedingungen wann geprüft werden sollen. Genau dies ist mit 4EverRulez möglich. Mit 4EverRulez können den Regeln Gruppen zugeordnet werden. Die Gruppen kennen die Zustände „alive“ und „notAlive“. Ist eine Regel mindestens einer Gruppe zugeordnet die im Zustand „alive“ ist, wird sie durch die Regel-Engine überprüft. Der Dokumentenbearbeiter hat über die graphische Benutzeroberfläche von 4EverRulez die Möglichkeit, den Zustand der Gruppen zu ändern und damit die Prüfung von bestimmten Regeln ein- bzw. auszuschalten (siehe Abbildung 24).

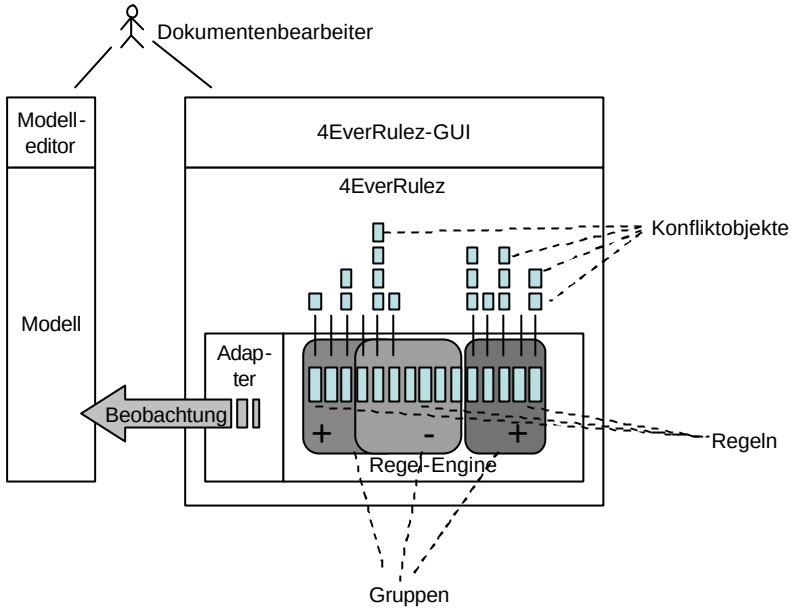

Abbildung 24: Der Modellzustand wird vom Adapter in die Regel-Engine übertragen. Falls der Bedingungsteil einer Regel erfüllt ist und eine ihrer Gruppen den Zustand „alive“ hat, wird ein Konfliktobjekt erzeugt. Aufgrund der im Konfliktobjekt enthaltenen Informationen, ist der Dokumentenbearbeiter in der Lage, den Konflikt im Modell bzw. im Dokument zu beheben.

5.2. Das Konzept im Überblick

Das Konzept basiert auf der Erweiterung des schon im Kapitel Begriffsdefinitionen (S.48) eingeführten Diagramms mit den Assoziationen zwischen den beteiligten Instanzen. Hinzugekommen sind Rollen, Regeln, Gruppen und Eigenschaften.

Mit Rollen sind die Rollen gemeint, die eine Konfliktstelle in einem Konflikt spielen kann. Die Erweiterung des Diagramms mit Rollen ist notwendig, weil man die Rolle einer Konfliktstelle kennen muss, um eine adäquate Hilfestellung zur Konfliktlösung geben zu können.

Durch die Nutzung einer Regel-Engine werden die Metamodellbedingungen negiert in den Bedingungsteilen der Regeln definiert. Dabei ist es durchaus möglich, dass der Bedingungsteil einer Regel mehr als eine Bedingung enthält. Die Regeln werden einer oder mehreren Gruppen zugeteilt. Auf diese Weise ist eine feinkörnigere Steuerung der Konsistenzprüfung möglich.

Für eine bessere Handhabbarkeit der Konflikte werden ihnen Eigenschaften zugeordnet. Dies könnten beispielsweise Prioritäten der Konflikte sein.

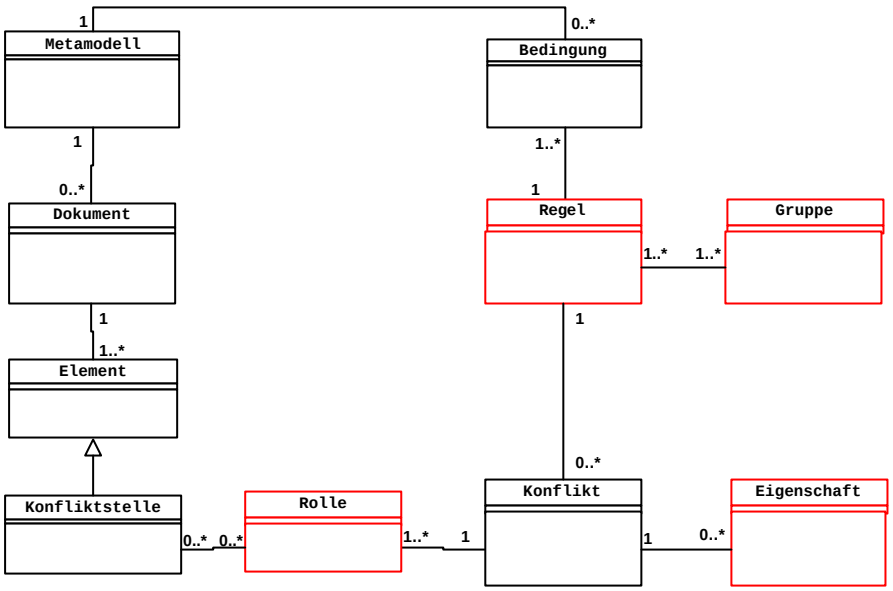

Abbildung 25: Erweitertes Klassendiagramm aus Abbildung 21.

5.3. Anwendungsszenarien

Um der Benutzeroberfläche von 4Ever die für die Konsistenzsicherstellung notwendige Zusatzfunktionalität zu geben ist eine Erweiterung notwendig. Diese Erweiterung besteht aus der 4EverRulez-GUI, die in den 4EverEdit-Frame integriert wurde. In Abbildung 26 ist diese Erweiterung in einem Screenshot hervorgehoben.

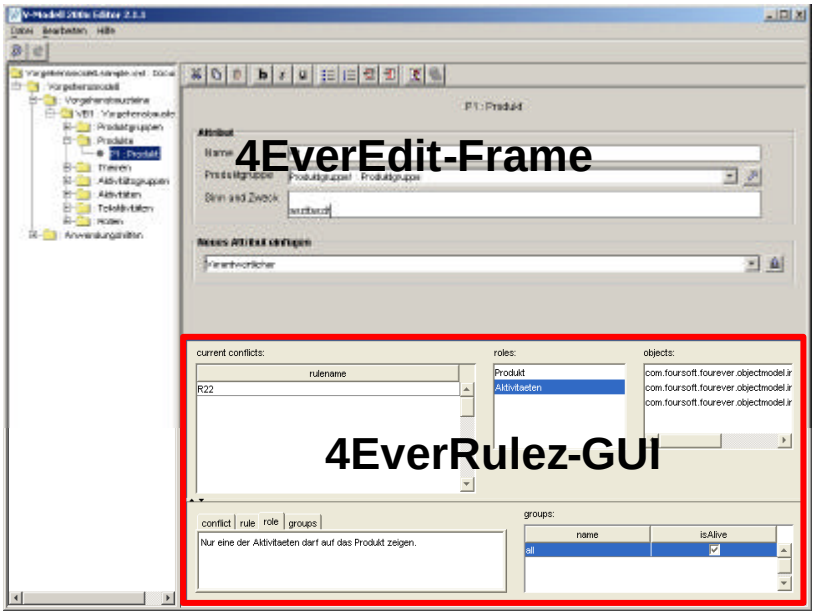

Abbildung 26: Die Benutzeroberfläche von 4Ever mit integrierter 4EverRulez-GUI.

Durch die 4EverRulez-GUI soll dem Dokumentenbearbeiter der Anwendungsfall Konsistenz sicherstellen mit seinen Unterfällen ermöglicht werden (siehe Abbildung 27).

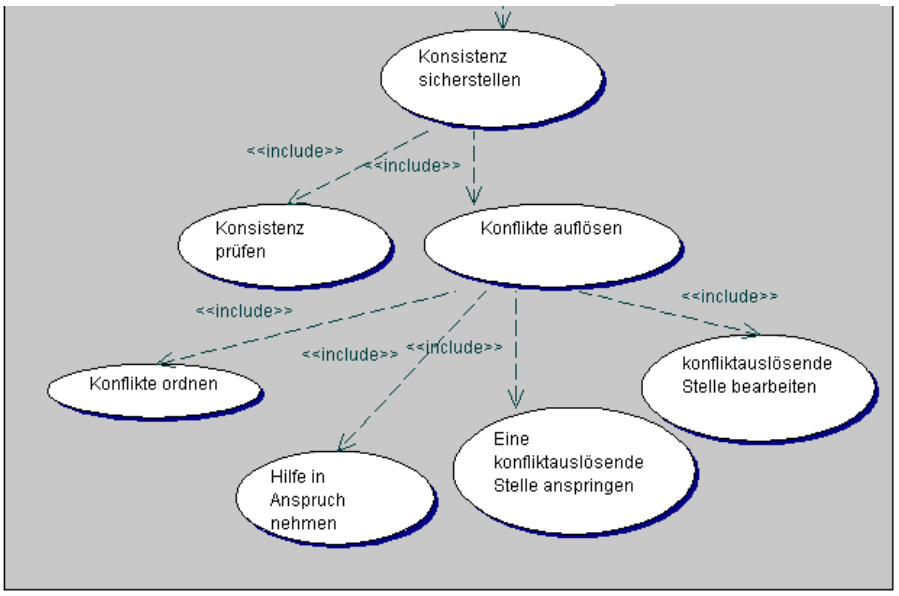

Abbildung 27: Anwendungsfall-Diagramm mit den Anwendungsfällen, die durch die 4EverRulez-Komponente ermöglicht werden sollen.

Die 4EverRulez-GUI ist in eine obere und eine untere Hälfte aufgeteilt. Die obere Hälfte enthält eine Konfliktabelle, eine Rollenliste und eine Objektliste. Die untere Hälfte enthält ein Informationsfenster. In ihm sind ein Konfliktinfo-Tab, ein Regelinfo-Tab, ein Rolleninfo-Tab und ein Gruppeninfo-Tab enthalten. Außerdem enthält die untere Hälfte noch eine Gruppentabelle (siehe Abbildung 28).

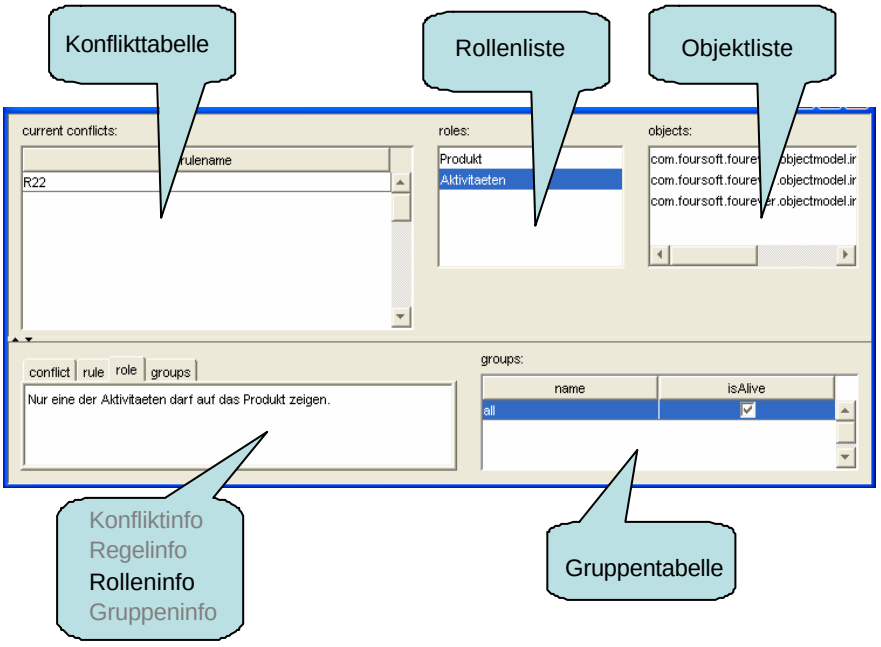

Abbildung 28: Die Benutzeroberfläche von 4EverRulez

Es wird nun gezeigt, wie mit Hilfe dieser Elemente die Anwendungsfälle durchgeführt werden können.

Der Anwendungsfall „**Konsistenz prüfen**“: Die Konsistenz eines Dokumentes wird geprüft, in dem festgestellt wird, ob es Metabedingungen seines Metamodells verletzt. 4EverRulez stellt eine Methode zur Prüfung der Konsistenz zur Verfügung, die es sogar erlaubt, die Konsistenz des Dokumentes auf einer Teilmenge der Bedingungen zu testen. Dies geschieht durch eine Zuteilung der Regeln zu Gruppen. Die Gruppen haben die Zustände „alive“ und „notAlive“. Der Zustand kann in der Gruppentabelle von einem Dokumentenbearbeiter beliebig verändert werden. Ist eine Gruppe in dem Zustand „alive“, wird von 4EverRulez gewährleistet, dass alle in ihr enthaltenen Regeln überprüft werden. Eventuell entstehende Konflikte werden in der Konflikttabelle angezeigt. Diese Gruppierung der Regeln hat den Vorteil, dass besonders Ressourcenintensive Prüfungen von dem Dokumentenbearbeiter manuell ausgeführt werden können.

Der Anwendungsfall „**Konflikte ordnen**“: Vorhandene Konflikte werden in der Konflikttabelle angezeigt. Ein Experte kann bei der Definition der Regelbasis Konflikten beliebige Eigenschaften hinzufügen, nach denen sie sich dann in der Konflikttabelle sortieren lassen. Hat er den Konflikten beispielsweise eine Priorität zugeordnet, lassen sie sich durch einen Mausklick auf den Spaltennamen nach ihnen ordnen. Dasselbe gilt natürlich für alle anderen Eigenschaften, die ihnen zugeordnet wurden, wenn sie in der Tabelle angezeigt werden.

Der Anwendungsfall „**Hilfe in Anspruch nehmen**“: Hilfestellung bekommt ein Dokumentenbearbeiter durch das Informationsfenster. Aktiviert er eine Zeile in der Konflikttabelle, bekommt er zunächst allgemeine Informationen über die Bedingung angezeigt, die zu diesem Konflikt geführt hat. Außerdem werden durch die Aktivierung einer Zeile in der Konflikttabelle, in der Rollenliste alle Rollen aufgeführt, die es für Konfliktstellen dieses Konfliktes gibt. Durch die Aktivierung einer Rolle in der Rollenliste wird in dem Informationsfenster eine rollenspezifische Hilfe gegeben. Sie beschreibt, was zu tun ist, um eine Konfliktstelle mit der entsprechenden Rolle so zu bearbeiten, dass der Konflikt gelöst wird.

Die Anwendungsfälle „**Konfliktstelle anspringen**“ und „**Konfliktstelle bearbeiten**“: Es gibt drei Möglichkeiten zu einer Konfliktstelle zu gelangen. Entweder ein Doppelklick auf eine Zeile in der Konflikttabelle, einen Doppelklick auf einen Eintrag in der Rollenliste oder einen Doppelklick auf einen Eintrag in der Objektliste. Für jeden Konflikt gibt es eine primäre Rolle und für jede Rolle gibt es eine primäre Konfliktstelle. Ein Doppelklick auf eine Zeile in der Konflikttabelle bewirkt, dass die primäre Konfliktstelle der primären Rolle des Konflikts im 4EverEditor geöffnet wird. Bei einem Doppelklick auf einen Eintrag in der Rollenliste wird die primäre Konfliktstelle der Rolle geöffnet. Bei einem Doppelklick auf eine Konfliktstelle in der Objektliste wird genau diese geöffnet. Nachdem die Konfliktstelle in dem 4EverEditor geöffnet wurde, kann sie gemäß den Hilfsanweisungen bearbeitet werden.

6. Technische Umsetzung

6.1. Integration von 4EverRulez in 4Ever

Die Architektur von 4Ever ist komponentenbasiert. Das heißt, 4Ever setzt sich aus mehreren eigenständigen Softwareelementen mit klar definierten Schnittstellen zusammen [HER]. Komponenten lassen sich durch die Nutzung eines Container-Frameworks einfach in ein System integrieren. 4EverRulez nutz das Spring-Framework[19] als Container für seine Komponenten. Die Nutzung eines Komponentenframeworks kann sehr vorteilhaft sein. Einige dieser Vorteile sind:

- isoliertes Entwickeln und Testen einer Komponente möglich
- Wiederverwendbarkeit von Komponenten
- Übersichtlichkeit durch klare Schnittstellen
- einfache Erweiterung eines Systems durch neue Komponenten
- Verlagerung von technischen Problemen (z.B. „Verdrahtung“) in die Containerschicht

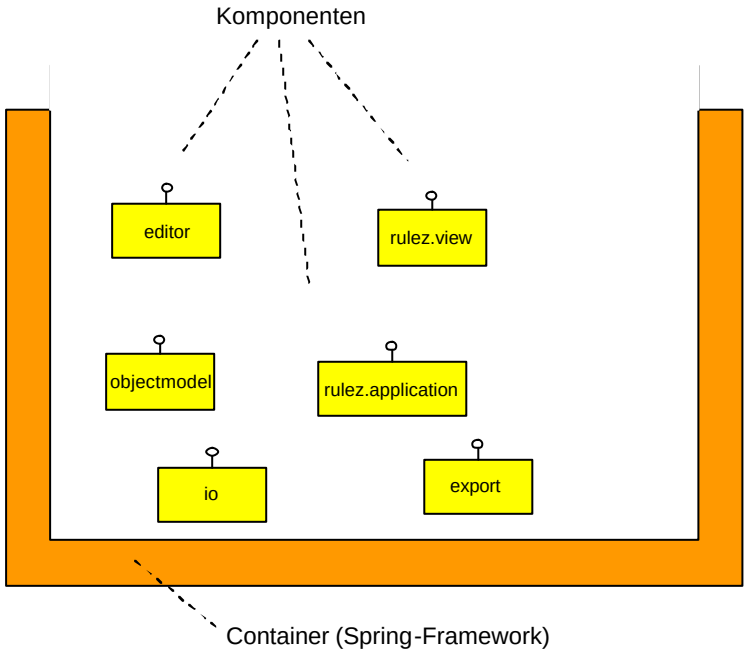

Abbildung 29: 4EverRulez erweitert 4Ever um zwei Komponenten.

In Abbildung 29 ist die Komponentenarchitektur von 4Ever dargestellt. 4EverRulez setzt sich aus der rulez.application- und der rulez.view-Komponente zusammen.

[19] www.springframework.org

Diese beiden Komponenten werden in den folgenden Abschnitten genauer betrachtet.

6.2. Die rulez.application-Komponente

Die rulez.application-Komponente ist der wichtigste Teil von 4EverRulez. Diese Komponente enthält die Regel-Engine und bietet die notwendige Funktionalität für die Konsistenzbeobachtung, die Konsistenzwiederherstellung und die Steuerung (siehe Aufteilung der Problemstellung S.53). Sie besteht im Wesentlichen aus neun Java-Interfaces. Davon sollen die wichtigsten drei beschrieben werden. Eine Dokumentation sämtlicher Methoden findet sich im Anhang A.

Abbildung 30 zeigt ein Klassendiagramm mit der Komponentenschnittstelle der rulez.application-Komponente.

`RuleSession`: Einer `RuleSession` ist ein Objektmodell zugeordnet. Während der Initialisierungsphase wird ihr ein dazu passendes Metamodell in Form einer Regelmenge für die Regel-Engine übergeben. Über das `RuleSession`-Interface erhält man Zugriff auf alle notwendigen Objekte.

`RulezManager`: Das `RulezManager`-Interface dient der Verwaltung. Mit seiner Hilfe ist es möglich `RuleSessions` zu erzeugen und auf sie zuzugreifen. Das erste Argument der `createRuleSession`-Methode ist ein String, der die Position der Datei mit der Regelbasis enthält (siehe Die Regelbasis S.64). Sie enthält die Regeln für die Regel-Engine sowie `ConflictPropertyType`- und `RuleGroup`-Definitionen. Das zweite Argument kann ein beliebiges Objekt sein auf dem die Bedingungen der Regeln überprüft werden sollen. Vorraussetzung dafür ist, dass der `RulezManager` für die Klasse des übergebenen Objektes eine Adapter-Klasse kennt und dass entsprechende Regeln in der Regelbasis definiert wurden. Übergibt man als zweites Argument einen Null-Zeiger, initialisiert der `RulezManager` einen Shortcut-Adapter, über den direkt auf die Faktenbasis der Regel-Engine zugegriffen werden kann. Zugänglich ist er über die `getModelAdapter`-Methode der `RuleSession`, die von `createRuleSession()` zurückgegeben wird.

`ConflictObserver`: Das `ConflictObserver`-Interface soll einen Beobachter immer wissen lassen wenn ein Konflikt entsteht oder wenn einer verschwindet. Aus diesem Grund stellt es eine `update`-Methode mit zwei Argumenten zur Verfügung. Das erste Argument ist ein Konfliktobjekt und das zweite ein Integer-Wert, der angibt, ob das Konfliktobjekt neu hinzugekommen oder verschwunden ist.

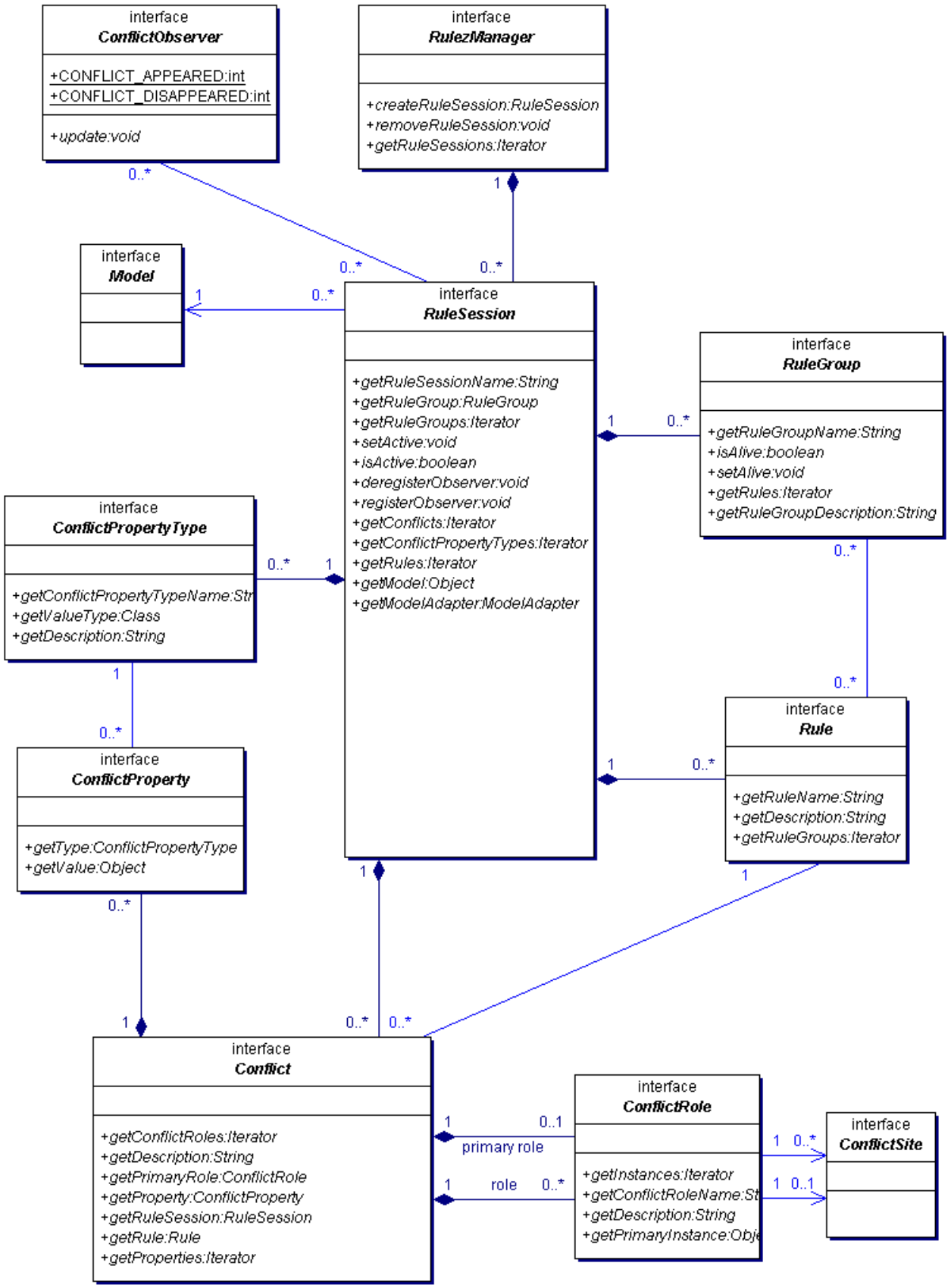

Abbildung 30: Die Schnittstelle der rulez.application-Komponente.

6.3. Die rulez.view-Komponente

Die rulez.view Komponente stellt eine graphische Benutzeroberfläche, die 4EverRulez-GUI zur Verfügung. Ein Klassendiagramm des Benutzerinterfaces ist in Abbildung 31 (S.58) zu sehen. Die rulez.view-Komponentenschnittstelle besteht aus einem Interface und vier abstrakten Klassen deren Funktion im Folgenden erläutert wird.

Mit Hilfe der `ConflictViewerFactory` lässt sich ein `ConflictViewer` erzeugen. Die createViewer-Methode bekommt dabei eine `RuleSession` als Argument, bei der der `ConflictViewer` als `ConflictObserver` registriert wird.

Der `ConflictViewer` ist eine `Jcomponente` und kann somit in eine Java-GUI integriert werden. Bei dem `ConflictViewer` können sich `ConflictSelectionEventListener` registrieren. Dieser `ConflictSelectionnEventListener` wird in der 4Ever-Umgebung wahrscheinlich die editor-Komponente sein.

Ein `OpenConflictObjectEvent` enthält eine Konfliktstelle (siehe Konfliktlösung S.54). Bei diesem Event soll der Editor das entsprechende Dokumentenelement im Editor-Panel (siehe Abbildung 5) öffnen.

Ein `HighlightConflictObjectEvent` enthält mehrere Konfliktstellen. Bei einem solchen Event soll der Editor diese Konfliktstellen im Strukturbaum (siehe Abbildung 2) markieren.

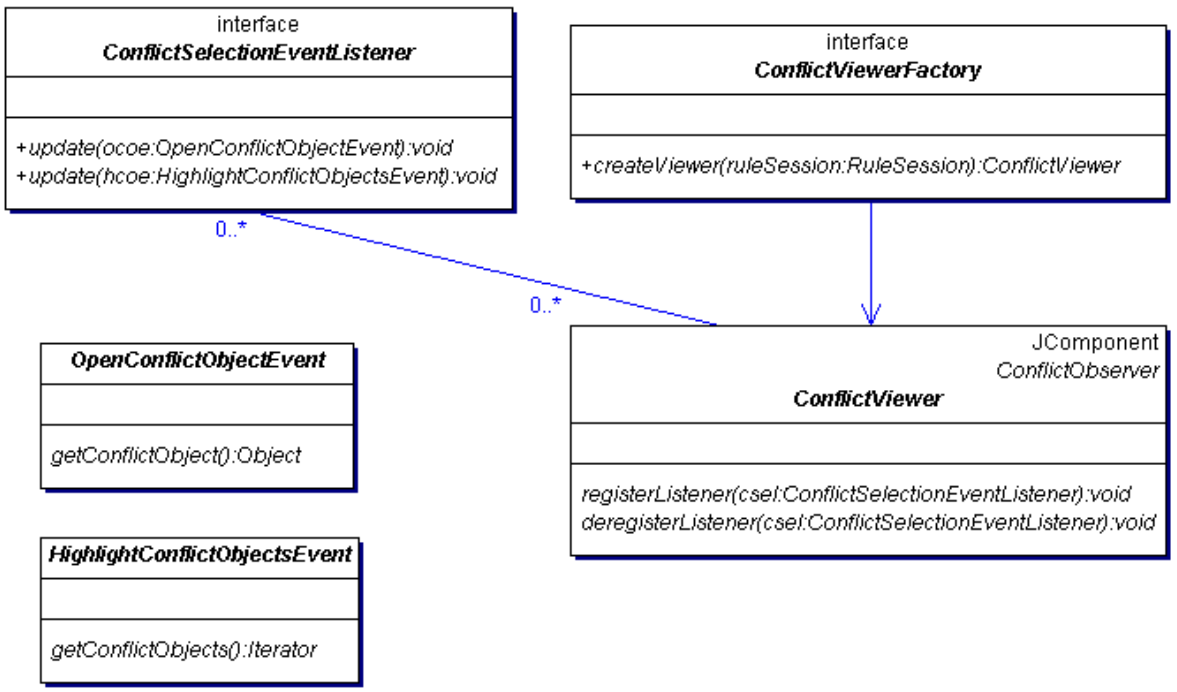

Abbildung 31: Die Schnittstelle der rulez.view-Komponente.

6.4. Die Regelbasis

Die Regelbasis stellt die Konfigurationsschnittstelle von 4Ever für den Experten dar. In ihr werden das Metamodell und die für die Konsistenzsicherung notwendigen Zusatzinformationen gespeichert. Zur Erzeugung einer `RuleSession` (siehe RuleSession S.61) wird dem RulezManager (siehe RulezManager S.61) eine Instanz des Objektmodells und ein Pfad zu einer Regelbasis übergeben. Entsprechend der Regelbasisinformationen werden folgende Aktionen durchgeführt:

- Erzeugung einer `StatefulRuleSession`
- Initialisierung der Regeln
- Initialisierung der Gruppen
- Initialisierung der Eigenschaftstypen

Das Wichtigste in einer Regelbasis sind die Regeln. Um eine Regel für 4EverRulez zu erstellen sind die in Abbildung 32 dargestellten Anwendungsfälle zu durchlaufen.

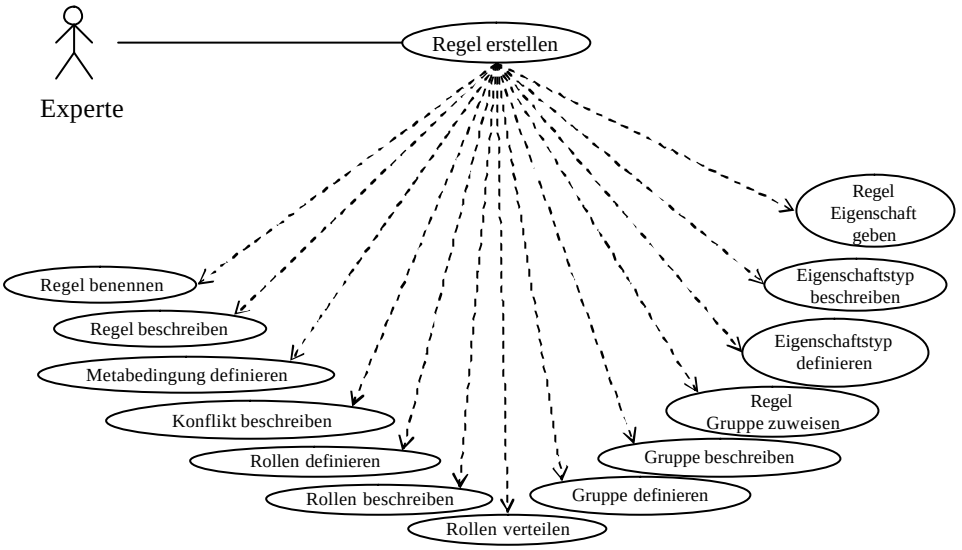

Abbildung 32: Für die Erstellung einer Regel müssen diese Anwendungsfälle durchlaufen werden.

Diese Anwendungsfälle werden im folgenden Kapitel anhand des V200X-Vorgehensmodell-Beispiels erläutert.

Eine Regelbasis ist gemäß dem XML-Schema aus Abbildung 33 aufgebaut. Man erkennt die drei Hauptelemente `RuleSet`, `RuleGroups` und `ConflictPropertyTypes`.

Das `RuleSet` enthält neben dem `jsr94-rule-execution-set` Informationen über die zu verwendende Regel-Engine. Mit Hilfe dieses `jsr94-rule-execution-set` kann eine `StatefulRuleSession` des JSR94-Standards erzeugt werden (siehe Anhang A). Diese `StatefulRuleSession` entspricht der Instanz einer Regel-Engine.

In dem `RuleGroups`-Element sind alle Gruppen (siehe Abschnitt Steuerung S.55) definiert. Die Gruppen enthalten einen Namen, eine Beschreibung, einen Defaultstatus und die Namen aller Regeln die ihnen angehören.

In dem `ConflictPropertyTypes`-Element sind alle Typen der Konflikteigenschaften definiert. Sie enthalten einen Schlüssel, eine Beschreibung und einen Typ. Der Typ ist eine Javaklasse, zum Beispiel String, Integer oder Boolean.

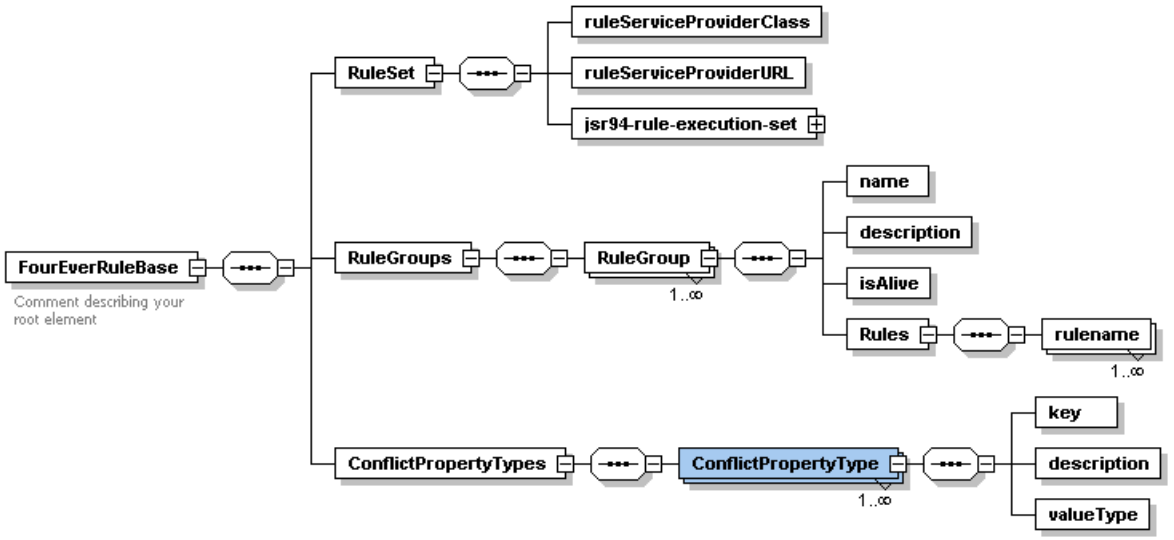

Abbildung 33: 4EverRulez versteht eine nach diesem Schema aufgebaute Regelbasis.

Um eine andere Regel-Engine zu nutzen, ist es lediglich notwendig, das RuleSet in der Regelbasis entsprechend anzupassen. Das heißt, der Inhalt des `ruleServiceProviderClass`- und des `ruleServiceProviderURL`-Elementes müssen, mit den für die neue Regel-Engine gültigen Werte, belegt werden. Außerdem müssen die Bedingungs- als auch Aktionsteile der Regeln in die Syntax der neuen Regel-Engine übersetzt werden.

Eine 4EverRegelbasis lässt sich in zwei Aufgabenbereiche aufteilen. Zum einen ist das der Aufgabenbereich, der die Definition des Metamodells beinhaltet, zum anderen ist es der Bereich, in dem die Hilfsanweisungen für den Dokumentenbearbeiter definiert sind. Die Anpassung dieser Informationen ermöglicht einen Einsatz von 4EverRulez für die Konsistenzsicherung unterschiedlichster Dokumente.

7. Regelerstellung am Beispiel des V200X-Vorgehensmodells

In dem Kapitel Fallbeispiel V200X-Vorgehensmodell wurde das V200X-Vorgehensmodell als Fallbeispiel eingeführt. Es wurde kurz auf die Hintergründe des Vorgehensmodells und anschließend auf sein Metamodell eingegangen. Es hat sich gezeigt, dass durch die Nutzung eines XML-Schemas als Metametamodell nicht alle Aspekte des V200X-Metamodells abgebildet werden können. In diesem Abschnitt soll nun detailliert gezeigt werden, wie man mit 4EverRulez die fehlenden Aspekte des Metamodells durch die Definition einer Regel ergänzen kann.

Grundlage soll das noch mal vereinfachte V200X-Metamodell sein (siehe Abbildung 34).

.

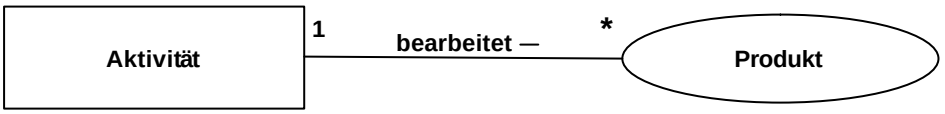

Abbildung 34: Ausschnitt aus dem V200X-Metamodell.

Das XML-Schema aus Listing 2, das einen Teil des V200X-Metamodells enthält, ist in Abbildung 35 wegen einer besseren Übersichtlichkeit in einer Baumansicht dargestellt. Man erkennt, dass ein `Vorgehensmodell` aus einem `Vorgehensbausteine`-Element besteht. Dieses Element kann beliebig viele `Vorgehensbaustein`-Elemente enthalten. Jeder `Vorgehensbaustein` hat ein `Name`-, ein `Produkte`- und ein `Aktivitaeten`-Element. Das `Produkte`- und das `Aktivitaeten`-Element können jeweils beliebig viele `Produkt`- bzw. `Aktivitaet`-Elemente enthalten. Zwischen `Aktivitaet`- und `Produkt`-Elementen soll es die in Abbildung 34 dargestellte Assoziation geben. In dem XML-Schema ist dies durch ein `ProduktRef`-Element realisiert. Mit diesem Referenzelement sind aber auch laut Metamodell verbotene n-zu-m-Beziehungen möglich (n>1) (siehe Das V200X-Metamodell S.27). Das Ziel soll es nun sein, eine Regelbasis zu erstellen, so dass diese verbotenen n-zu-m-Beziehungen entdeckt und beseitigt werden können.

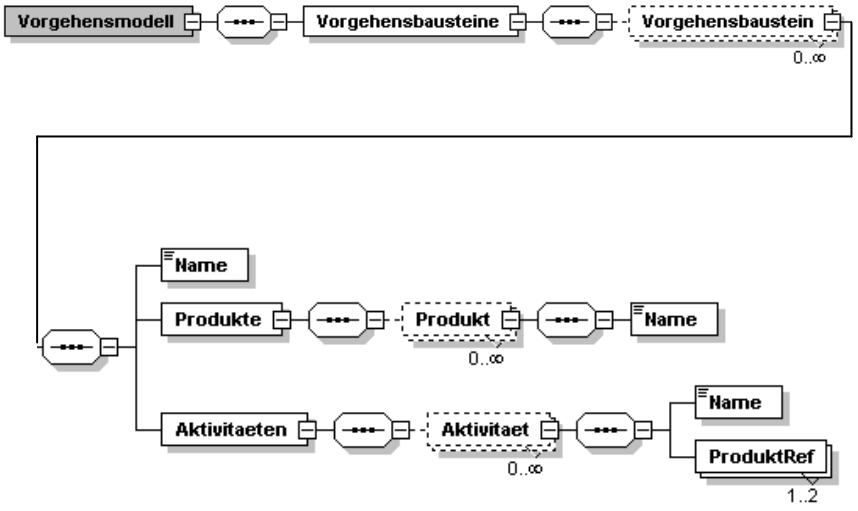

Abbildung 35: Ein Ausschnitt des V200X-Metamodells als XML-Schema.

7.1. Regelerstellung mit RulezEdit

Ein Experte kann die Regelbasis direkt in XML definieren. Dies ist aber sehr umständlich und fehlerträchtig. Besser ist die Nutzung einer grafischen Benutzeroberfläche. Eine solche Benutzeroberfläche soll von RulezEdit bereitgestellt werden. RulezEdit ist ein Werkzeug um eine Regelbasis zu bearbeiten. Man kann mit RulezEdit also eine vorhandene, als XML-Datei abgelegte Regelbasis laden, bearbeiten und wieder als XML-Datei abspeichern. Dieser Sachverhalt ist in Abbildung 36 dargestellt. Natürlich lässt sich mit RulezEdit auch eine Regelbasis erzeugen, ohne sie vorher laden zu müssen.

Das Programm RulezEdit ist keine Komponente von 4EverRulez!

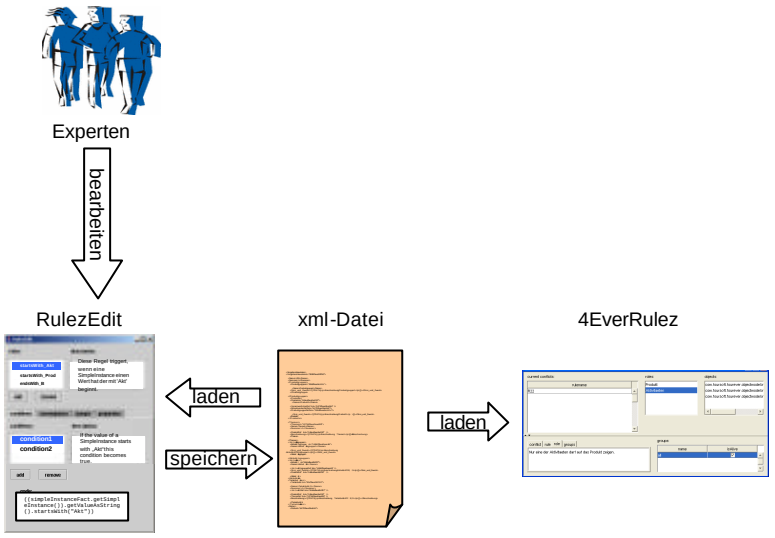

Abbildung 36: Mit dem Programm RulezEdit lässt sich die Regelbasis erstellen und editieren.

Bis zum Fertigstellungstermin der Diplomarbeit existiert noch keine Implementierung von RulezEdit. Alle Abbildungen der Oberfläche sind prototypische Entwürfe. Sie sollen eine Möglichkeit für die Regelerstellung über eine graphische Benutzeroberfläche exemplarisch darstellen.

Um eine Regelbasis zu erstellen sind die in Abbildung 32 dargestellten Anwendungsfälle vom Experten zu durchlaufen. Diese werden anhand des oben eingeführten Beispiels mit der Benutzeroberfläche von RulezEdit durchgespielt.

Anwendungsfall „Regel benennen“: Man betrachte Abbildung 37. Mit dem add-Button lässt sich eine Regel erstellen und benennen. Alle vorhandenen Regeln werden in der Regelliste angezeigt. Die restlichen angezeigten Werte sind abhängig von der in der Regelliste aktivierten Regel. Die aktivierte Regel ist durch einen blauen Hintergrund hervorgehoben.

Die Regel die das Metamodell aus dem V200X-Beispiel (siehe oben) vervollständigen soll, wurde, wie in der Abbildung zu erkennen, der Name `Akt_Prod` gegeben.

Anwendungsfall „Regel beschreiben“: Das Regelbeschreibungsfeld soll eine Beschreibung enthalten, die einem Dokumentenbearbeiter erklärt, welche Metabedingung von der Regel überprüft wird.

In dem V200X-Beispiel lautet die Beschreibung für die `Akt_Prod`-Regel: „Diese Regel feuert, wenn ein Produkt mehr als eine fertig stellende Aktivität hat.“

Anwendungsfall „Metabedingung definieren“: Die Metabedingung wird durch den Metabedingungscode definiert, der in das Feld für den Metabedingungscode geschrieben werden kann. Dieser Code ist abhängig vom verwendeten Adapter und von der verwendeten Regel-Engine.

Sinngemäß lautet die Bedingung in dem V200X-Beispiel: „Anzahl der ProduktRef-Elemente pro Produkt >1“. Wie der Code für diese Bedingung aussehen kann, wird weiter unten in dem Kapitel Definition einer V200X-Metabedingung (S.72) gezeigt.

Anwendungsfall „Konflikt beschreiben“: Durch diesen Vorgang soll der Konflikt eigentlich keine Beschreibung, sondern eine Anweisung zur Lösung des Konflikts für den Dokumentenbearbeiter erhalten.

In dem V200X-Beispiel lautete diese Anweisung: „Dieser Konflikt lässt sich am besten beheben, wenn eine Konfliktstelle mit der Rolle ‚Aktivität’ den Anweisungen entsprechend bearbeitet wird.“

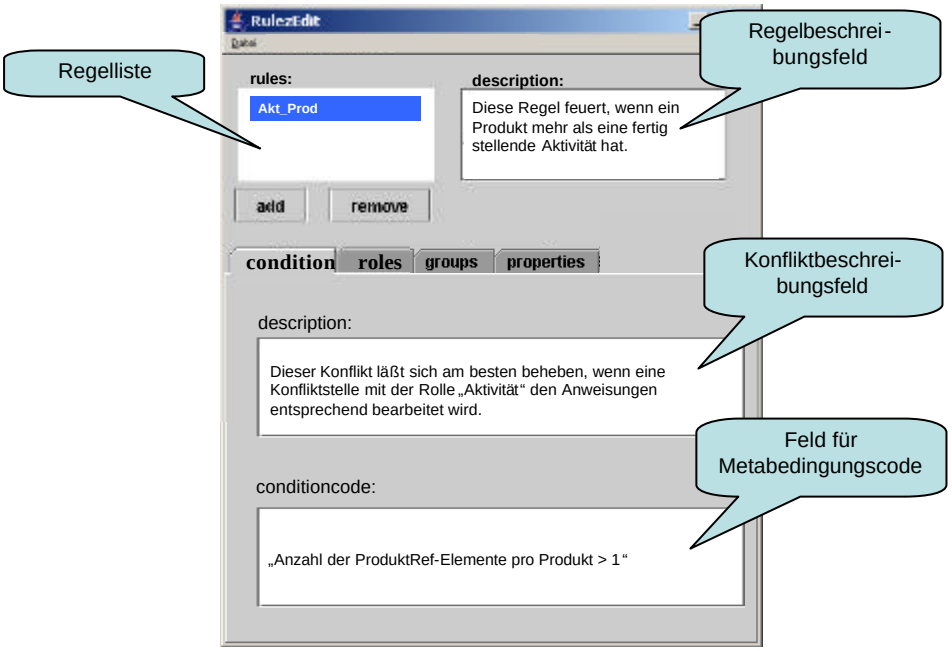

Abbildung 37: Konfliktbeschreibung und Bedingungsdefinition mit RulezEdit.

Anwendungsfall „Rolle definieren“: Dieser Anwendungsfall bedeutet, dass man eine Rolle erzeugt und ihr einen Namen gibt. RulezEdit stellt diese Funktionalität über den add-Button unterhalb der Rollenliste (siehe Abbildung 38) zur Verfügung.

Für die `Akt_Prod`-Regel aus dem V200X-Beispiel wurden eine `Aktivitäten`- und eine `Produkt`-Rolle definiert.

Anwendungsfall „Rolle beschreiben“: Hier ist die Bezeichnung „Rolle beschreiben“ ein wenig irreführend, denn eigentlich soll nicht die Rolle beschrieben werden, sondern es soll eine konfliktstellenspezifische Hilfestellung zur Konfliktlösung gegeben werden.

Für die Konfliktstellen in einer `Aktivitäten`-Rolle wurde in dem V200X-Beispiel folgende Hilfestellung vom Experten gegeben: „Sie können den Konflikt beheben, indem sie die Produktreferenz aus einer Aktivität ändern.“

Anwendungsfall „Rolle besetzen“: Wie die Rolle besetzt wird, muss von dem Experten, wie in dem Anwendungsfall „Metabedingung definieren“, durch Code definiert werden. Der Code ist wiederum abhängig von der verwendeten Regel-Engine und dem verwendeten Adapter. Ein konkretes Codebeispiel wird in dem Abschnitt Konflikterzeugung (S. 77) besprochen.

In dem V200X-Beispiel sollen natürlich alle an dem Konflikt beteiligten `Aktivitaet`-Elemente die Rolle `Aktivitaeten` erhalten. Das `Produkt`-Element das von mehr als einer `Aktivität` referenziert wird, soll die Rolle `Produkt` erhalten.

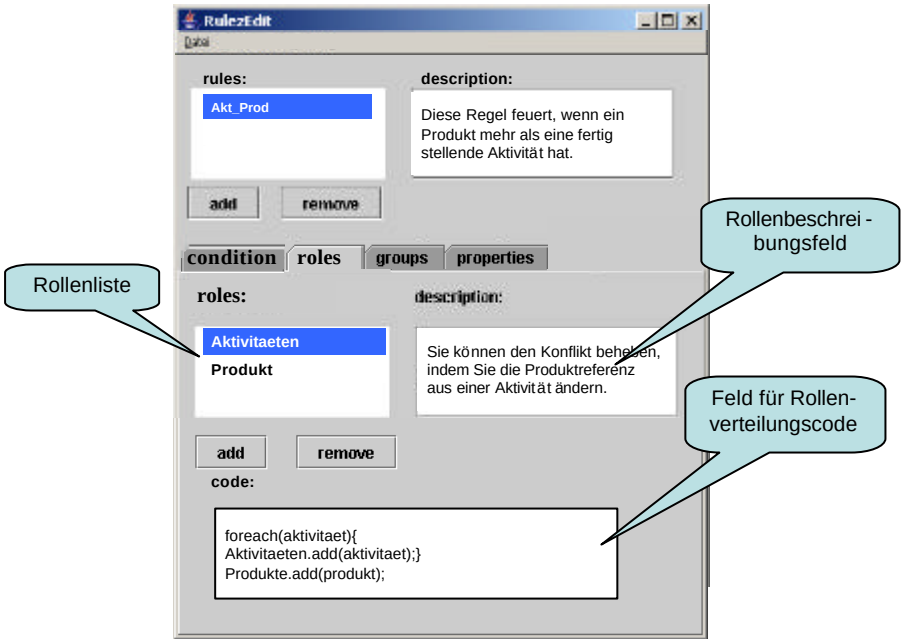

Abbildung 38: Rollendefinition mit RulezEdit.

Anwendungsfall „Gruppe definieren“: Dieser Anwendungsfall umfasst die Erzeugung und die Benennung einer Gruppe. RulezEdit stellt dem Experten diese Funktionalität über den add-Button unterhalb der Gruppentabelle zur Verfügung (siehe Abbildung 39).
Für das V200X-Beispiel wurde nur eine Gruppe mit der Bezeichnung `RefInt` definiert. Sie wird durch die erste Zeile in der Gruppentabelle repräsentiert.

Anwendungsfall „Gruppe beschreiben“: Eine Gruppe erhält eine Beschreibung durch das Ausfüllen des Gruppenbeschreibungsfeldes. Die Beschreibung sollte Informationen darüber enthalten, welche Art von Regeln sie enthält. Es wäre beispielsweise sinnvoll, die Regeln nach ihrem Ressourcenaufwand zu gruppieren.
Die `RefInt`-Gruppe des V200X-Beispiels soll alle Regeln enthalten, die die Integrität der Referenzen überprüfen. Dementsprechend lautet die Beschreibung: „In der Gruppe RefInt sind alle Regeln enthalten, die die Integrität von Referenzen überprüfen. Diese Überprüfung ist nicht sehr teuer und sollte daher immer eingeschaltet sein.“

Anwendungsfall „Regel Gruppe zuweisen“: Durch die zweite Spalte der Gruppentabelle lässt sich eine Regel durch Aktivierung der Checkboxen einer oder mehreren Gruppen zuweisen.
In dem V200X-Beispiel ist die `Akt_Prod`-Regel der RefInt-Gruppe zugeordnet. Auf diese Weise kann der Dokumentenbearbeiter durch Zustandsänderung der RefInt-Gruppe von `alive==true` auf `alive==false` die Überprüfung der `Akt_Prod`-Regel verhindern.

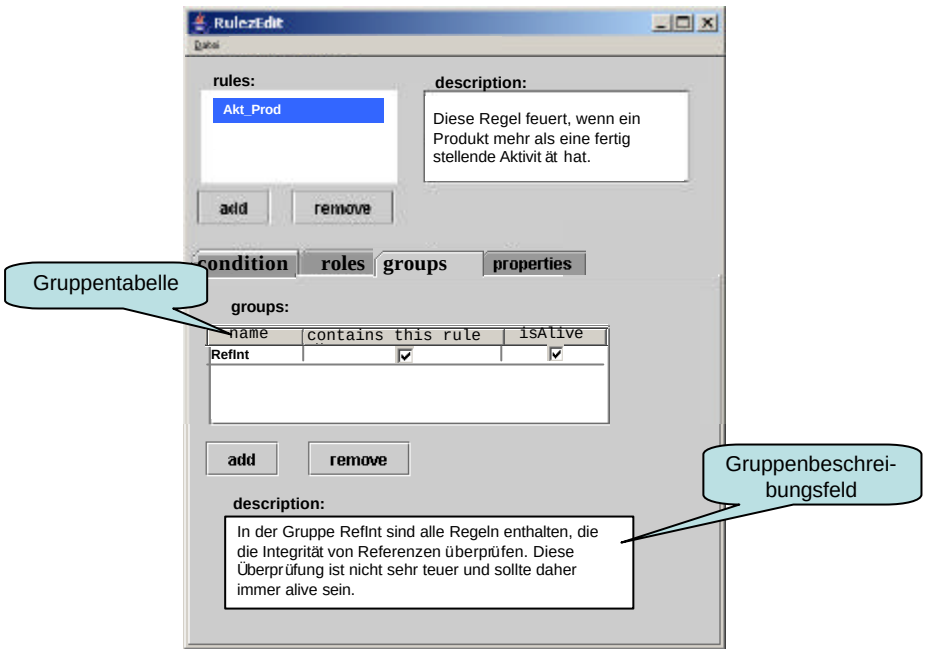

Abbildung 39: Gruppendefinition mit RulezEdit

Anwendungsfall „Eigenschaftstyp definieren“: Durch diesen Anwendungsfall wird ein Eigenschaftstyp erzeugt, benannt und typisiert. Die entsprechende Funktionalität wird von RulezEdit über den add-Button unterhalb der Eigenschaftentabelle bereitgestellt (siehe Abbildung 40).
Die Eigenschaft `rulename` könnte von RulezEdit automatisch erzeugt werden. Sie ist nicht unbedingt notwendig, aber sehr sinnvoll, um zu wissen, von welcher Regel ein Konflikt erzeugt wurde.
Wie in der Abbildung zu erkennen, wurden für das V200X-Beispiel noch drei weitere Eigenschaften definiert.

Anwendungsfall „Eigenschaftstyp beschreiben“: Diese Beschreibung dient dem Experten, den Sinn und die Bedeutung einer Eigenschaft zu dokumentieren.
So ist zum Beispiel für die `priority`-Eigenschaft in der Beschreibung ein mögliches Intervall angegeben. Die Beschreibung für die `noCheckIn`-Eigenschaft könnte so lauten: “Existiert dieser Konflikt in dem Dokument, soll es dem Dokumentenbearbeiter nicht möglich sein, es in das Repository einzuchecken.“

Anwendungsfall „Regel Eigenschaft geben“: Den Regeln bzw. den von ihnen erzeugten Konflikten lassen sich die Eigenschaften direkt durch die Bearbeitung der `value`-Spalte der Eigenschaftentabelle geben.
In dem V200X-Beispiel hat die `Prod_Akt`-Regel die Eigenschaften: `rulename=Prod_Akt, priority=1, author=theile, noCheckIn=true.`

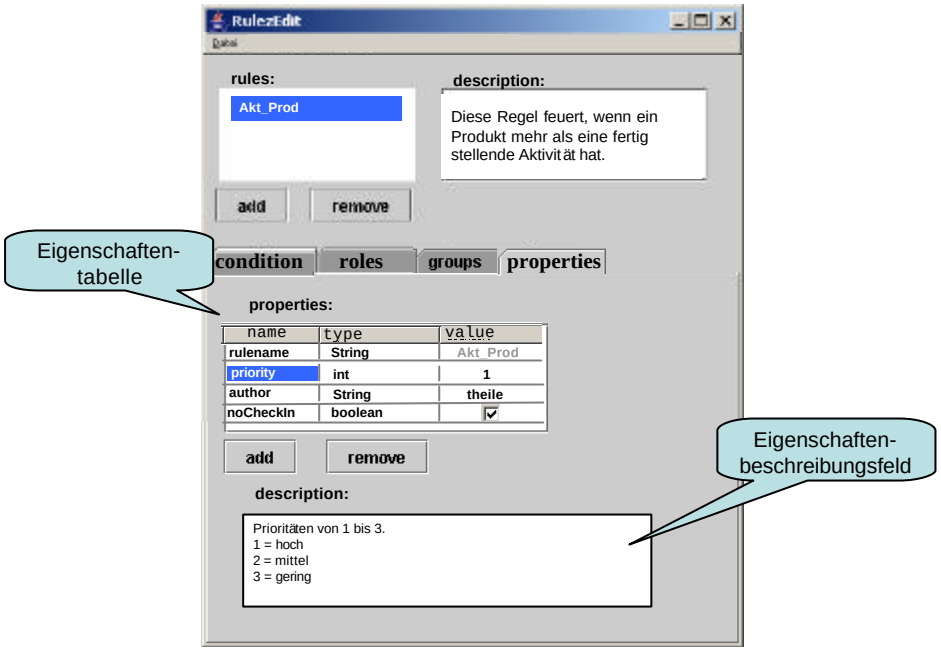

Abbildung 40: Eigenschaftvergabe mit RulezEdit

7.2. Definition einer V200X-Metabedingung

Bis jetzt wurde noch nicht auf darauf eingegangen, in welcher Weise Metabedingungen definiert werden können. In diesem Abschnitt soll nun gezeigt werden, wie sich das in Abschnitt Das V200X-Metamodell (S.27) besprochene V200X-Bedingungsbeispiel definieren lässt. Dabei ist die Syntax für die Bedingungen stark von der verwendeten Regel-Engine und dem verwendeten Adapter abhängig.

Dieses Beispiel geht davon aus, dass eine Drools-Regel-Engine[20] (siehe Anhang A) und der im nächsten Abschnitt erklärte Objektmodelladapter verwendet werden.

Ziel ist es, eine Bedingung zu definieren, die erkennt, ob das V200X-Bedingungsbeispiel durch das Dokument verletzt wird. Dies ist der Fall, wenn zwei Aktivitäten ein Produkt referenzieren. Die Bedingung setzt also drei `ComplexInstances` (siehe

[20] www.drools.org

Das Objektmodell von 4Ever S.14) in Beziehung. Diese müssen zunächst mit folgendem Code als Parameter deklariert werden:

Listing 17

```
<parameter identifier="aktivitaet1">
      <java:class>
            com.foursoft.fourever.objectmodel.ComplexInstance
      </java:class>
</parameter>
<parameter identifier="aktivitaet2">
      <java:class>
            com.foursoft.fourever.objectmodel.ComplexInstance
      </java:class>
</parameter>
<parameter identifier="produkt">
      <java:class>
            com.foursoft.fourever.objectmodel.ComplexInstance
      </java:class>
</parameter>
```

In der Bedingung muss auch angegeben werden, von welchen Typen diese drei in Beziehung stehenden `ComplexInstances` sein müssen. Dies kann durch den Code in Listing 18 erreicht werden.

Listing 18

```
<java:condition>
      (oa.hasType(aktivitaet1," Aktivitaet"))
</java:condition>
<java:condition>
      (oa.hasType(aktivitaet2,"Aktivitaet"))
</java:condition>
<java:condition>
      (oa.hasType(produkt,"Produkt"))
</java:condition>
```

`oa` ist hierbei eine Instanz des Objektmodelladapters, in dem Funktionen definiert werden können. In diesem Fall wurde die `hasType()`-Funktion genutzt.

Als nächstes muss man nun ausdrücken, dass die beiden Aktivitäten dasselbe Produkt referenzieren. Dies geschieht wiederum unter zu Hilfenahme der Objektmodellfunktion `isLinked()`. Der entsprechende Code ist in Listing 19 zu sehen.

Listing 19

```
<java:condition>
      (oa.isLinked(aktivitaet1,"ProduktRef",produkt))
```

```
</java:condition>
<java:condition>
      (oa.isLinked(aktivitaet2,"ProduktRef",produkt))
</java:condition>
```

Die `isLinked()`-Methode bekommt unter anderem den String „ProduktRef“ übergeben. Dies ist der im XML-Schema (siehe Listing 2) definierte Name für Referenzen zwischen Aktivitäten und Produkten.

Nun ist es noch wichtig, der Regel-Engine mitzuteilen, dass die beiden Aktivitäten nicht dieselben sein dürfen, und dass die an einem Konflikt beteiligten Aktivitäten a1 und a2 nur in einer Reihenfolge überprüft werden dürfen. Ansonsten würde die Regel-Engine zwei Konflikte zählen, nämlich den Konflikt (a1, a2) und den Konflikt (a2, a1).

Listing 20

```
<java:condition>
      (aktivitaet1.compareTo(aktivitaet2)<0)
</java:condition>
```

Alle diese Code-Listings zusammen ergäben nun eine Bedingung, die erfüllt ist, sobald die V200X-Beispielbedingung verletzt wird. Diesen Code müsste der Experte bei der Nutzung des RulezEdit-Programms in das in Abbildung 37 (S.69) zu sehende Feld für den Metabedingungscode einfügen.

Aus den Bedingungen der Listings Listing 17, Listing 18, Listing 19 und Listing 20 würde sich das in Abbildung 41 dargestellte RETE-Netzwerk (siehe Das Netzwerk S.42) ergeben.

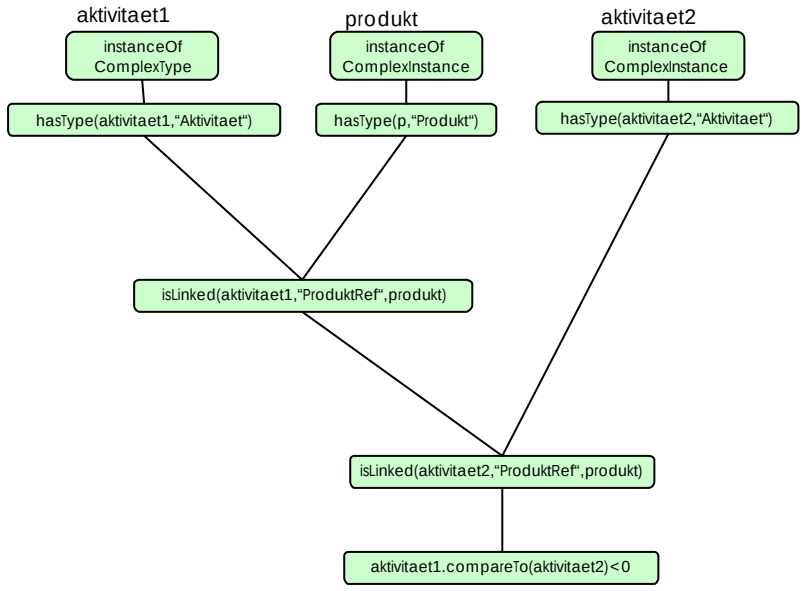

Abbildung 41: Dieses RETE-Netzwerk ergibt sich aus dem V200X-Metabedingungsbeispiel.

7.3. Der Objektmodelladapter

Der Objektmodelladapter ist verantwortlich für die Beobachtung des Objektmodells (siehe Abschnitt

Konflikterkennung S.54). Außerdem stellt er Hilfsmethoden zur Verfügung die in den Bedingungen benutzt werden können.

Der in diesem Beispiel verwendete Objektmodelladapter implementiert das `InstanceObserver`-Interface des 4Ever-Objektmodells (siehe Abbildung 42). Nachdem der Objektmodelladapter bei dem Objektmodell registriert ist, wird er über alle Änderungen in dem Objektmodell, über die Interfacemethode `update`(), benachrichtigt. Diese Änderungen teilt er der Regel-Engine mit. Prinzipiell stehen ihm dafür die Methoden des in dem JSR-94 (siehe Anhang A) definierten `StatefulRuleSession`-Interfaces zur Verfügung. Diese sind im Wesentlichen:

- `addObject`()
- `removeObject`()
- `updateObject`()

Ein Adapter übermittelt den Zustand des zu beobachtenden Dokumentes der Regel-Engine in Form von Fakten. Diese Fakten sind Objekte, die durch die oben aufgelisteten Methoden in der Faktenbasis verwaltet werden (siehe Fakten und Regeln S.32). Von dem Informationsgehalt, der in diesen Objekten steckt, hängt es ab, welche Metabedingungen sich definieren lassen. Für das V200X-Metamodellbeispiel reicht es aus, wenn nur die Instanzen des Objektmodells in der Faktenbasis enthalten sind (siehe Listing 17 bis Listing 20). Um dies zu erreichen, werden je nach Ereignis im Objektmodell unterschiedliche Aktionen ausgeführt. Die Art des Ereignisses im Objektmodell wird der `update`()-Methode, durch einen als Integer codierten Aspekt, mitgegeben. Der Objektmodelladapter behandelt die Ereignisse folgenderweise:

Bei einem `INSTANCE_CREATED`-Aspekt wird die neue Instanz einfach der Faktenbasis hinzugefügt. Dies geschieht durch einen Aufruf der `addObject`()-Methode mit dem Instanzobjekt als Argument. Bei einem `INSTANCE_REMOVED`-Aspekt wird die gelöschte Instanz durch die `removeObject`()-Methode aus der Faktenbasis entfernt. Bei einem `VALUE_CHANGED`-Aspekt wird die Regel-Engine durch ein `updateObject`()-Aufruf auf die geänderte Instanz aufmerksam gemacht. Bei den restlichen Aspekten wird jeweils ein `updateObject`() mit der in der `update`()-Methode mitgegebenen Instanz ausgeführt.

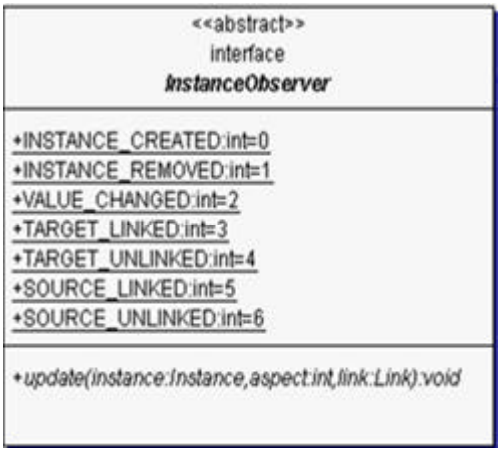

Abbildung 42: Ein `InstanceObserver` wird über Änderungen im 4Ever-Objektmodell benachrichtigt.

Damit die Faktenbasis den Zustand des Dokumentes vollständig wiedergibt, wird nach der Registrierung des Objektmodelladapters bei dem Objektmodell zunächst eine `synchronize()`-Methode des Objektmodelladapters aufgerufen. Diese fügt alle vor der Registrierung erzeugte Instanzen der Faktenbasis hinzu. Dadurch wird die Faktenbasis mit dem Objektmodell sozusagen synchronisiert.

Die bis hierher beschriebenen Mechanismen dienten der Beobachtung des Objektmodells. Die zweite Aufgabe des Adapters ist es, Funktionen bereit zu stellen, die bei der Definition von Bedingungen benutzt werden können. In dem oben ausgeführten V200X-Bedingungsbeispiel waren das die Methoden

- `hasType()` und
- `isLinked()`.

Der Zugriff auf diese Methoden in der Bedingungsdefinition wird dadurch gewährleistet, dass sich der Adapter selbst in der Faktenbasis als Fakt hinzufügt und ein Objekt seiner Klasse als Parameter für eine Bedingung deklariert wird. Dies geschieht durch eine Parameterdeklaration wie in Listing 21.

Listing 21

```
<parameter identifier="oa">
      <java:class>
            ObjectModelAdapter
      </java:class>
</parameter>
```

Durch eine solche Deklaration des Adapters als Parameter ist die Nutzung der Adapterfunktionen, wie zum Beispiel in Listing 19 gezeigt, möglich. Bei der Nutzung eines Programms wie RulezEdit kann diese Deklaration automatisch erfolgen.

7.4. Konflikterzeugung

In dem Abschnitt Definition einer V200X-Metabedingung wurde gezeigt, wie eine Bedingung für eine Regel definiert werden kann. Doch was passiert nun, wenn diese Bedingung erfüllt ist und die Regel feuert? Wenn die Bedingung erfüllt ist, wird der Aktionsteil der Regel von der Execution-Engine (siehe Abschnitt

Die Arbeitsweise einer Regel-Engine S.36) ausgeführt. Prinzipiell können beliebige Aktionen in dem Aktionsteil definiert werden. In dem Aktionsteil einer 4EverRulez-Regel wird, gemäß dem fachlichen Konzept, ein Konfliktobjekt erzeugt (siehe Abschnitt Konfliktlösung S.54). Wenn ein Programm wie RulezEdit für die Regeldefinition verwendet wurde, kann der Code für die Konflikterzeugung vollkommen automatisch aus den vom Experten in RulezEdit eingegebenen Informationen generiert werden. Für ein besseres Verständnis ist es sinnvoll einen Blick auf einen solchen Code für die Konflikterzeugung zu

werfen. Dieser Code ist abhängig von der verwendeten Regel-Engine. In diesem Fall wird davon ausgegangen, dass eine Drools-Regel-Engine (siehe Anhang) eingesetzt wird. Drools nutzt zur Übersetzung des Aktions- und Bedingungsteil Janino[21]. Janino ist ein einfacher Java-Compiler. Dementsprechend wird für die Definition der Regeln Javasyntax verwendet.

Im Folgenden wird der Aktionsteil Stück für Stück ergänzt. Die jeweiligen Ergänzungen sind Fett. Ausgelassener Code wird mit der Zeichenfolge (…) angedeutet. Voll qualifizierte Klassenpfade wurden durch den Klassennamen ersetzt.

Das Ergebnis wird ein Aktionsteil sein, der entstehen würde, wenn man die in dem Abschnitt Regelerstellung mit RulezEdit gemachten Konfigurationen, in einen ausführbaren Aktionsteil übersetzten würde.

Der XML-Tag das den Aktionsteil einer Regel enthält lautet bei Drools `<java:consequence/>`. In ihm wird zunächst ein Konfliktobjekt initialisiert (siehe Listing 22). Dieses Konfliktobjekt wird dann der Faktenbasis hinzugefügt. Auf diese Weise kann er über das `StatefulRuleSession`-Interface des JSR-94 von 4EverRulez ausgelesen und an registrierte `ConflictObserver` (siehe **`ConflictObserver`** S.61) übergeben werden.

Listing 22

```
<java:consequence>
      //---------  Einen Conflict erzeugen
      ConflictImpl conflict = new ConflictImpl();
      conflict.setRulename("Akt_Prod");

      (...)

      drools.assertObject(conflict);
></java:consequence>
```

Anschließend wird eine Rolle erzeugt (siehe Abschnitt Konfliktlösung S.54). In diesem Fall ist es die Rolle „Aktivität“. Ihr wird eine Beschreibung gegeben und Instanzen zugeordnet. Eine der Instanzen wird als primäre Instanz markiert. Zum Schluss wird sie dem Konflikt zugeordnet. (siehe Listing 23)

Listing 23

```
<java:consequence>
      //---------  Einen Conflict erzeugen
      ConflictImpl conflict = new ConflictImpl();
      //---------  Eine ConflictRole erzeugen
      ConflictRoleImpl role1 = new ConflictRoleImpl();

      role1.setRoleName("Aktivität");
      role1.setDescription("Sie können den Konflikt beheben, indem sie die
            Produktreferenz aus einer Aktivität ändern.");
      role1.addInstance(aktivitaet1);
      role1.addInstance(aktivitaet2);
```

[21] www.janino.net

```
      role1.setPrimaryInstance(aktivitaet1);
      conflict.addConflictRole(role1);

      (...)

      drools.assertObject(conflict);
</java:consequence>
```

Die Rolle “Produkt” wird ganz entsprechend erzeugt.

Als nächstes werden die Eigenschaften des Konfliktes erzeugt. Zunächst wird ein Eigenschaftenobjekt initialisiert. Diesem werden ein Typ und ein Wert zugeordnet. Ist dies geschehen wird es dem Konfliktobjekt hinzugefügt. Gibt es mehrere Eigenschaften, wie in dem V200X-Beispiel, müssen diese auf gleiche Weise erzeugt werden (siehe Listing 24).

Listing 24

```
<java:consequence>

      (...)

      conflict.addConflictRole(role1);

      //---------- ConflictProperties erzeugen
      ConflictPropertyImpl prop1 = new ConflictPropertyImpl();
      prop1.setConflictPropertyTypeName("Name");
      prop1.setValue("Akt_Prod-Konflikt");
      conflict.addConflictProperty(prop1);

      (...)

      drools.assertObject(conflict);
</java:consequence>
```

7.5. Gruppenzuteilung

Die Zuteilung von Regeln zu Gruppen dient der Steuerung (siehe Abschnitt Steuerung S.55). Sie wir dadurch realisiert, dass die Gruppenobjekte auch der Faktenbasis hinzugefügt werden und dem Bedingungsteil eine Bedingung hinzugefügt wird, die nur dann wahr ist, wenn eine Gruppe mit einem bestimmten Namen den Status „alive“ hat.

Listing 25

```
      <parameter identifier="group1">
            <java:class> RuleGroupImpl</java:class>
      </parameter>
      <java:condition>
            (group1.getRuleGroupName().equals("RefInt")
            && group1.isAlive())
```

```
      </java:condition>
```

Soll die Regel mehreren Gruppen angehören, muss für jede Gruppe eine entsprechende Bedingung gestellt werden.

7.6. Konfliktentfernung

Durch die weiter oben definierte Bedingung ist es möglich zu erkennen, wenn ein Dokument inkonsistent wird. Bei einem Konsistenzwiederherstellungsverfahren ist es genauso wichtig, zu erkennen, wenn ein Dokument wieder in einen konsistenten Zustand übergeht. (siehe Konsistenzerhaltung und Konsistenzwiederherstellung S.23)

Die bis jetzt besprochenen Maßnahmen dienen der Erkennung und Behebung eines Konflikts. Ist der Konflikt behoben, erwartet der Dokumentenbearbeiter, dass der Konflikt auch nicht mehr angezeigt wird.

Es muss also ein Mechanismus eingeführt werden, der erkennt, wenn die Bedingung auf der ein Konflikt beruht, nicht mehr erfüllt ist. Dieser Mechanismus wird durch eine Konfliktentfernungsregel realisiert.

Ein Konflikt entsteht, weil auf einer Menge von Objekten eine Menge von Bedingungen wahr ist. Eine Konfliktentfernungsregel muss also feststellen, wenn auf dieser Menge von Objekten mindestens eine der Bedingungen nicht mehr erfüllt ist. Ist dies der Fall, muss sie den entsprechenden Konflikt aus der Regelbasis entfernen.

Um eine solche Regel zu schreiben, muss zu jedem Konfliktobjekt gespeichert werden, auf welchen Objekten es beruht. Diese Information wird bei der Konflikterzeugung direkt im Konfliktobjekt gespeichert.

Dieser Code wurde in dem Konflikterzeugungsabschnitt (siehe Abschnitt Konflikterzeugung S.77) unterschlagen, soll aber an dieser Stelle nachgereicht werden:

Listing 26

```
<java:consequence>
      //---------  Einen Conflict erzeugen
      ConflictImpl conflict = new ConflictImpl();
      //---------  Eine ConflictRole erzeugen
      ConflictRoleImpl role1 = new ConflictRoleImpl();

      conflict.addFact(group1);
      conflict.addFact(aktivitaet1);
      conflict.addFact(aktivitaet2);
      conflict.addFact(produkt);

      (...)

      drools.assertObject(conflict);
</java:consequence>
```

Der Konflikt “weiß” also wegen welchen Objekten er existiert. Eine Konfliktentfernungsregel muss nun nichts weiter tun, als jedes Mal, wenn sich eines der an einem Konflikt beteiligten Objekte geändert hat, festzustellen, ob die Bedingungen der erzeugenden Regel noch alle erfüllt sind.

Dies wird mit folgender Regeldefinition erreicht:

Listing 27

```
<rule name="Akt_Prod-RETRACT" >
      <parameter identifier="oa">
            <java:class>MinimalObjectModelAdapter</java:class>
      </parameter>
      <parameter identifier="conflict">
            <java:class>ConflictImpl</java:class>
      </parameter>
      <parameter identifier="group1">
            <java:class>RuleGroupImpl</java:class>
      </parameter>
      <parameter identifier="aktivitaet1">
            <java:class>ComplexInstance</java:class>
      </parameter>
      <parameter identifier="aktivitaet2">
            <java:class>ComplexInstance</java:class>
      </parameter>
      <parameter identifier="produkt">
            <java:class>ComplexInstance</java:class>
      </parameter>


      <java:condition>
            (conflict.getFact(0)==group1)
      </java:condition>
      <java:condition>
            (conflict.getFact(1)==aktivitaet1)
      </java:condition>
      <java:condition>
            (conflict.getFact(2)==aktivitaet2)
      </java:condition>
      <java:condition>
            (conflict.getFact(3)==produkt)
      </java:condition>

      <java:condition>
      (!(
            (group1.getRuleGroupName().equals("RefInt")
      &&
            group1.isAlive())
      &&
            (oa.hasType(aktivitaet1,"Aktivitaet"))
      &&
            (oa.hasType(aktivitaet2,"Aktivitaet"))
      &&
            (oa.hasType(produkt,"Produkt"))
      &&
            (oa.isLinked(aktivitaet1,"ProduktRef",produkt))
      &&
            (oa.isLinked(aktivitaet2,"ProduktRef",produkt))
```

```
      &&
            (aktivitaet1.compareTo(aktivitaet2)<0)
      ))
</java:condition>
<java:consequence>
      drools.retractObject(conflict);
</java:consequence>
</rule>
</rule-set>
```

Diese Regel enthält fünf Bedingungselemente, wobei die ersten vier sicherstellen, dass es sich bei den Konfliktobjekten genau um diejenigen Objekte handelt, auf denen der Konflikt basiert. Das fünfte Bedingungselement prüft, ob alle Bedingungselemente der Konflikterzeugenden Regel erfüllt sind. Ist dies nicht der Fall, ist das Konfliktobjekt nicht mehr gültig. Der Aktionsteil dieser Regel besteht lediglich darin, das nicht mehr gültige Konfliktobjekt aus der Faktenbasis zu entfernen.

Die vollständige Regelbasis für das V200X-Bedingungsbeispiel ist im Anhang zu finden. Diese Regelbasis enthält die Angaben für die Überprüfung einer Metabedingung. Für die Vollständige Konsistenzprüfung des V200X-Vorgehensmodells müssen in etwa 20 weitere Metabedingungen überprüft werden. Für jede dieser Bedingungen muss eine Regel auf die beschriebene Art definiert werden.

8. Zusammenfassung und Ausblick

In unserer modernen Welt sind wir häufig mit Aufgaben konfrontiert, für deren Bewältigung sich spezielle Dokumente (z.B. Vorgehensmodelle) eignen. 4Ever ist ein Programm, mit dem sich solche Dokumente bearbeiten lassen. Bevor dies möglich ist, muss für die zu lösende Aufgabe eine Dokumentenklasse definiert werden. Dies geschieht durch die Spezifikation eines Metamodells als XML-Schema. Häufig ist nur eine unvollständige Spezifikation des mentalen Metamodells möglich, weil sich einige Aspekte durch ein XML-Schema nicht beschreiben lassen.

Die Aufgabe dieser Diplomarbeit war es, die Möglichkeiten für die Spezifikation eines Metamodells zu erweitern und eine Komponente in 4Ever einzubinden, die die Konsistenz des Dokumentes zu dem erweiterten Metamodell überwacht. Falls es zu Inkonsistenzen kommt, soll der Dokumentenbearbeiter so informiert werden, dass er die Konsistenz wieder herstellen kann.

Die für diesen Zweck im Rahmen der Diplomarbeit entwickelte Komponente heißt 4EverRulez. Eine Regel-Engine ermöglicht ihr die ständige Überwachung der Konsistenz. Durch den dabei genutzten RETE-Algorithmus kann dies auf sehr effiziente Weise geschehen. Der Algorithmus macht sich die Tatsache zu nutze, dass sich während der Bearbeitung immer nur wenige Stellen im Dokument ändern. Die Regel-Engine wird nach einem offiziellen Standard[22] eingebunden. Dadurch können verschiedene Regel-Engines, mit geringem Anpassungsaufwand gegeneinander ausgetauscht werden. Ist ein Konflikt von der Regel-Engine erkannt, werden dem Dokumentenbearbeiter detaillierte Informationen über die Gründe des Konflikts und mögliche Maßnahmen für seine Auflösung mitgeteilt. Dies geschieht über eine Erweiterung der graphischen Benutzeroberfläche von 4Ever. Für diese Erweiterung wurde ein Konzept entwickelt, mit dem sich Hilfestellungen zu unterschiedlichsten Konfliktarten geben lassen.

Die bisher durch ein XML-Schema nicht zu definierenden Bedingungen des mentalen Metamodells werden 4EverRulez in Form von Regeln in einer Regelbasis zur Verfügung gestellt. Diese Regelbasis enthält neben dem Metamodell, die für die Konsistenzwiederherstellung notwendigen Hilfsinformationen. Mit RulezEdit wurde eine prototypische Benutzeroberfläche für eine komfortable Definition der Hilfsinformationen vorgestellt. Die Realisierung eines solchen Programms würde die praktische Bedeutsamkeit von 4EverRulez wegen einer starken Vereinfachung der Regelerstellung erheblich steigern.

Es hat sich angeboten 4EverRulez als eine eigenständige Komponente zu konzipieren, obwohl dies in der ursprünglichen Aufgabenstellung nicht gefordert wurde. Die Beobachtung der Dokumente erfolgt über einen leicht austauschbaren Adapter, der die vorliegenden Daten, in eine für Regel-Engines geeignete Form bringen kann. Auf diese Weise lässt sich 4EverRulez problemlos an verschiedene Objektmodelle anpassen.

Die in der Zielsetzung (S.7) gestellten Teilaufgaben wurden mit dieser Diplomarbeit gelöst. Dabei haben sich weitere Aufgaben ergeben. Eine dieser Aufgaben ist es, eine Sprache für die Spezifikation des Metamodells zu standardisieren oder einen vorhandenen Standard für diesen Zweck zu finden. Für diese Sprache müsste ein Compiler entworfen werden, der sie für eine bestimmte Regel-Engine übersetzen kann. Auf diese Weise müssten nicht alle Regelbasen beim Einsatz einer neuen Regel-Engine umgeschrieben werden. Die einmalige Anpassung des Compilers an die neue Regel-Engine würde reichen. Es wäre sehr vorteilhaft,

[22] JSR-94 (siehe Anhang)

eine für den Menschen leicht zu erlernende Sprache zu wählen. Alternativ könnte eine Benutzeroberfläche entworfen werden, die eventuell unter Nutzung des zuvor spezifizierten XML-Schemas eine Spezifikation weiterer Aspekte des Metamodells durch Regeln vereinfacht.

Ein kritischer Punkt, bei dem Einsatz von 4EverRulez in der Praxis, ist die Laufzeit bei der Überprüfung der Konsistenz des Dokumentes zu seinem Metamodell. Der RETE-Algorithmus arbeitet zwar sehr effizient, doch auch er stößt bei der Überprüfung einiger Metamodellaspekte schnell an seine Grenzen. In einem solchen Fall dauern die Prüfungen zu lange, so dass der Dokumentenbearbeiter durch Verzögerungen im Arbeitsfluss gestört wird. Handelt es sich um Zeiträume im Sekundenbereich, könnte man die Regel-Engine in einem eigenen Thread laufen lassen. Dadurch könnte der Dokumentenbearbeiter ungestört arbeiten, während die Regel-Engine die Überprüfungen im Hintergrund durchführt. Wenn die Prüfungen aber länger als einige Sekunden dauern, muss eine Möglichkeit gefunden werden, sie zu beschleunigen. Dies könnte Beispielsweise durch spezielle, im Adapter definierte Algorithmen geschehen.

In dieser Arbeit wurden Regeln behandelt, die Konfliktobjekte erzeugen und damit den Dokumentenbearbeiter über eine Inkonsistenz informieren. Die Architektur von 4EverRulez lässt es aber auch zu, dass andere Aktionen beim Feuern einer Regel ausgeführt werden. Es könnte beispielsweise direkt in die Dokumentenbearbeitung eingegriffen werden. Das heißt, Konflikte oder andere Aufgaben könnten von 4EverRulez automatisch gelöst werden. Die Frage ist, unter welchen Bedingungen solche automatischen Eingriffe sinnvoll sind und wie man dieses Verhalten bei der Metamodellspezifikation beschreiben kann.

Abschließend lässt sich sagen, dass mit 4EverRulez ein robustes Rahmenwerk geschaffen wurde, das viel Potential für Erweiterungen und Fortentwicklungen enthält.

Literatur

[BROC1968] Brockhaus Enzyklopädie. 17. Auflage. 1968.

[BRO] M. Broy. *Informatik eine grundlegende Einführung.* Springer. 1998.

[BRUE] A. Brüggemann-Klein. *Elektronisch Publizieren.* Scriptum . WS00.

[DRO] W. Dröschel , M. Wiemers. *Das V-Modell 97.* Oldenbourg. 2000.

[FRIED] Ernest Friedman-Hill. *Jess in Action.* Manning. 2003.

[GAMMA] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley. 1994.

[GAR] Jostein Gaarder. *Sophies Welt*. Hanser. 1993.

[HER] P. Herzum, O. Sims. *Business Component Factory.* OMG PRESS, 2000.

[ix0203110] Guido Laures. Regelgerecht. *iX 3/2002.*

[KEMP] Kemper, Eickler. *Datenbanksysteme.* Oldenbourg. 2001.

[NORMAN] Donald A. Norman. *The Design of Everyday Things*. MIT Press. 2000.

[RECH] Rechenberg, Pomberger. *Informatik-Handbuch*. Hanser. 1997.

[SCH] Uwe Schöning. *Theoretische Informatik – kurzgefaßt*. Spektrum. 1992.

[STEG] A.Steger, M. Raab. *Diskrete Strukturen I*. Skriptum. WS9899.

[V200XMeta] Friedrich, Gnatz, Meisinger, Rausch, Vogel. *V-Modell-200X Metamodell.* Dokument des WEIT-Projektes. Arbeitspaket: Integration. 2004.

[XMLSCH] http://www.w3.org/TR/xmlschema-0/ .

Anhang A

Marktüberblick der Regel-Engines

Nachdem die prinzipielle Arbeitsweise von Regel-Engines in dem Kapitel Regel-Engines besprochen wurde, ist es nun möglich, sich einen fundierten Überblick über das Angebot der Regel-Engines zu verschaffen.

Auf den ersten Blick ergibt sich eine erschlagende Anzahl von Regel-Engines. Bei genauerem Hinsehen lichtet sich dieser Dschungel aber schnell.

Die unten stehende Liste ist durch Internetrecherche entstanden und erhebt keinerlei Anspruch auf Vollständigkeit. Insbesondere wurde nur CLIPS als exemplarisches Beispiel für eine Regel-Engine aufgeführt, die nicht in Java implementiert ist. Alle anderen nicht-Java Regel-Engines wurden nicht in die Liste aufgenommen.

- CLIPS (http://www.ghg.net/clips/CLIPS.html)
- OFBiz (http://www.ofbiz.org/docs/rules.html)
- Mandarax (http://mandarax.sourceforge.net/)
- Euler (http://www.agfa.com/w3c/euler/)
- JLog (http://jlogic.sourceforge.net/)
- JIProlog (http://www.ugosweb.com/jiprolog/)
- JLisa (http://www.gubaba.com/jlisa/)
- RDFExpert (http://public.research.mimesweeper.com/RDF/RDFExpert/Intro.html)
- JEOPS (http://www.jeops.org/)
- TyRuBa (http://tyruba.sourceforge.net/)
- JRules (http://www.ilog.com/)
- Blaze Advisor (http://www.fairisaac.com/NR/rdonlyres/045E6E22-86E4-4052-83EC-726E48BC428C/0/202638_FIBlazeAdvisorPS.pdf)
- Jess (http://herzberg.ca.sandia.gov/jess/index.shtml)
- Drools (http://drools.org)
- Algernon (http://algernon-j.sourceforge.net/)

Leser, die einen kompletten Produktvergleich der unterschiedlichen Regel-Engines erwarten, muss ich leider enttäuschen. Ziel dieses Marküberblickes ist es, eine geeignete Regel-Engine für die Überwachung der Konsistenz eines Dokumentes zu seinem Metamodell zu finden. Sollte eine Regel-Engine einem KO-Kriterium nicht entsprechen wird diese auch nicht weiter betrachtet. Ein solches KO-Kriterium ist beispielsweise die Implementierungssprache, die für eine nahtlose Integration in 4Ever unbedingt Java sein muss. Aus diesem Grund kommt die weit verbreitete Regel-Engines **CLIPS** nicht in Betracht.

Einige Regel Engines verfolgen viel versprechende Ansätze, kommen aber nicht in Frage, weil sie schon seit einiger Zeit nicht mehr weiter entwickelt werden oder weil keine wirklich aussagekräftigen Referenzen zu finden sind, die belegen würden, dass es sich um eine akzeptable Software handelt.

JEOPS versucht zum Beispiel, die Java Sprache um eine Möglichkeit der deklarativen Programmierung zu erweitern. Die aktuellste Dokumentation die zu finden war, stammt aber

aus dem Jahr 2000 und die Homepage wurde das letzte Mal 2001 aktualisiert. Für **JLisa** und **TyRuBa** sind so gut wie keine Referenzen zu finden.

Bei **RDFExpert** handelt es sich um ein Programm, das für sehr spezielle Anwendungszwecke entwickelt wurde. Es ist der Versuch ein Expertensystem auf der Basis von dem „Ressource Description Framework“ (RDF) aufzubauen. Das würde zu einer höheren Kompatibilität von Wissensbasen und von Expertensystemen mit anderen Prozessen führen.[23] Der Focus liegt also auf Bereichen, die für das 4EverRulez-Projekt nicht von Interesse sind.

Ein weiteres wichtiges Kriterium ist die Unterstützung des RETE Algorithmus oder einer Abwandlung davon. Alle anderen bis jetzt bekannten Algorithmen sind Größenordungen von der Effizienz von RETE entfernt, wenn es darum geht die Faktenbasis gegen die Regelbasis anzuwenden und sich bloß einige kleine Änderungen seit der letzten Anwendung ergeben haben.[24]

Da in den Anforderungen eine Art „Wachhundmodus“ aufgeführt ist, der eine Prüfung der Regeln bei jeder noch so kleinen Änderung der Faktenbasis verlangt, ist die Durchführung des Interferenz-Vorganges auf Basis des RETE-Algorithmus zurzeit unumgänglich. Hiermit fallen **Mandarax**, **Euler, OFBiz** und viele weitere raus.

Des Weiteren sind alle auf Prolog-Ähnlichen Anwendung als Regel-Engine äußerst ungeeignet, da Prolog für ein anderes Anwendungsgebiet entwickelt wurde. Prolog ist darauf ausgelegt, einmalige Anfragen zu beantworten. Wohingegen Regel-Engines auf sich ständig ändernde Fakten reagieren müssen. Außerdem ist das Grundkonzept von Prolog „backward-chaining“. [FRIED] Dieses Vorgehen ist in den Anwendungsfällen von Regel-Engines oftmals ungeeignet und führt zu hohen Laufzeiten. Zu guter Letzt ist Prolog darauf ausgelegt, auf Kosten der Laufzeit, mit wenig Speicher auszukommen. RETE basierte System gehen mit dem Speicherplatz großzügiger um, erreichen dafür aber auch in vielen Fällen eine höhere Performance. Als Prologinterpreter in Java sind zum Beispiel **JLog** und **JIProlog** zu nennen.

Die verbleibenden Regel-Engines zerfallen in zwei Kategorien. Einerseits die sich auf die Kernfunktionalität beschränkenden Engines wie **JESS**, **Algernon** und **Drools** und andererseits kommerzielle Komplettlösungen zur Integration in Unternehmen wie **Blaze Advisor** oder **JRules**. Letztere rechtfertigen ihren Preis durch eine Vielzahl von mitgelieferten Werkzeugen und einen umfassenden Support. Mitgelieferte Werkzeuge sind zum Beispiel komfortable Benutzerinterfaces für die Regelerstellung, integrierte Entwicklungsumgebungen, Regel-Repositories mit Rechteverwaltung, vorgefertigten Deployment-Mechanismen in eine J2EE Umgebungen usw.. Diese Lösungen sind so umfangreich, dass die Anwendung in 4Ever sehr fraglich ist, zumal 4EverRulez kaum etwas mit dem ursprünglichen Einsatzgebiet zu tun hat. Es ist durchaus denkbar, dass man diese Komplettlösungen, mit entsprechendem Aufwand, gut in 4Ever integrieren kann. Aber letztendlich handelt man sich wahrscheinlich nur viel unnötigen und teuer erkauften Ballast ein.

Verbleiben noch **Jess** und **Drools**. Diese beiden Regel-Engines sind aus folgenden Gründen in Betracht zu ziehen: Neben den Grundanforderungen der Implementierungssprache und die Nutzung des RETE-Algorithmus werden sie in vielen ernsthaften Softwareprojekten eingesetzt, sie bringen geeignete Shadowingmechanismen mit und sind hervorragend dokumentiert. Wobei **Jess** in Punkto Dokumentation die Nase weit vorn haben dürfte. Neben den im Internet erhältlichen Informationen ist ein hoch gelobtes Buch, „Jess In Action“ von Friedman-Hill, im Manning-Verlag erschienen. Ich kann mich diesem Lob nur anschließen.

---

[23] http://public.research.mimesweeper.com/RDF/RDFExpert/Documentation/HTML/Overview.html

[24] http://www.theserverside.com/articles/article.tss?l=Drools ()

Durch die grundlegende Einführung und anschließenden Beleuchtung unterschiedlichster, praxisnaher Einsatzszenarien von Jess, gewinnt der Leser schnell einen guten Überblick über die Einsatzgebiete von **Jess.**

Des Weiteren spricht für **Jess** der Status, den es durch die Ernennung zur Referenzimplementierung des JSR94-Standards erlangt hat. Diese Tatsache behebt die letzten Zweifel an der Seriosität des Programms.

Ein Punkt in dem **Jess Drools** noch voraus ist, ist die Unterstützung von Backward-Chaning. Da diese Auflösungsstrategie in 4EverRulez voraussichtlich nie gebraucht werden wird soll dieser vermeintliche Vorteil aber nicht in die Bewertung eingehen. Entsprechendes gilt für die Performance, Jess liegt bei Tests in Sachen Geschwindigkeit ganz vorne.[25] Da dies aber kein besonders Kritischer Punkt bei der Anwendung in 4EverRulez ist, soll er ebenfalls keinen großen Einfluss auf die Entscheidung haben.

Bei all den Vorzügen von **Jess** gibt es zwei große Minuspunkte: Erstens ist es nicht open-source[26], d.h. bei einer Verwendung im kommerziellen Umfeld ist eine nicht unerhebliche Lizenzgebühr fällig und zweitens orientiert sich die Jesssyntax an der etwas kryptischen Prologsytax.

Drools hingegen ist open-source, kostenlos und nutzt eine modifizierte Variante des Rete-Algorithmus, der eine direkte Unterstützung der Objektorientierung möglich macht. Die Regeln werden durch Javacode spezifiziert.

Des Weiteren ist in ernstzunehmenden Artikeln von einer „charming developing community“ die Rede.[27] Man kann also davon ausgehen, dass man bei Problemen nicht auf sich selbst gestellt ist.

Jess ist zwar die Referenzimplementierung des JSR94-Standards, doch auch für **Drools** steht eine komplette JSR94-API bereit.

Die Entscheidung zwischen **Drools** und **Jess** fällt wirklich schwer.

Tabelle 1

	Jess	Drools
Pro	-Dokumentation -Referenzimplementierung	-OO-Unterstützung -opensource
Contra	-Lizenzkosten	-Dokumentation

Glücklicherweise gibt es den JSR94-Standard, der es erlaubt die Regel-Engine mit wenig Aufwand auszutauschen. Allerdings müssen in diesem Fall die Regeln übersetzt werden, da das Format in dem sie definiert werden im JSR94-Standard nicht festgelegt sind.

Jedenfalls kann man durch die Einhaltung von JSR-94 bei der Auswahl der Regel-Engine beruhigt sein, da man bei eventuellen Problemen mit der Regel-Engine jederzeit auf eine andere umsteigen kann.

[25] http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=14&t=000272
[26] http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=14&t=000260
[27] http://www.theserverside.com/articles/article.tss?l=Drools

[Quelle: JLog Whitepaper]
[Quelle: http://www.fairisaac.com/NR/rdonlyres/045E6E22-86E4-4052-83EC-726E48BC428C/0/202638_FIBlazeAdvisorPS.pdf]

Die JSR-94 Spezifikation

Der „Java Specification Request“ JSR-94 ist eine Spezifikation, deren Ziel es ist, ein leichtgewichtiges Programmierinterface zu definieren, durch das eine Regel-Engine konfiguriert und genutzt werden kann. Durch diese Standardisierung ist es möglich, Regel-Engines unterschiedlicher Hersteller mit wenig Aufwand gegeneinander auszutauschen.

Der Standard wurde im Rahmen des „Java Community Process“ (JCP) festgelegt. Dieser Prozess ist ein formales Vorgehen, um in „Internet-Zeit“ brauchbare Spezifikationen hervorzubringen. Neben der Spezifikation wird auch noch eine Referenzimplementierung (RI) und ein „technology compatibility kit“ (TCK) festgelegt. Durch die RI wird unter anderem die Implementierbarkeit der Spezifikation bewiesen. Durch das TCK kann die Kompatibilität anderer Implementierungen zur Spezifikation geprüft werden.

Im JCP entwickelt eine Expertengruppe einen Entwurf der Spezifikation der dann in einem iterativen Prozess verbessert wird. In diesen Verbesserungsprozess fließen Anregungen aus einem breiten Publikum ein. Ist ein Konsens gefunden kann die Spezifikation verabschiedet werden.[28]

Die für den JSR-94 verantwortliche, aus 11 Repräsentanten der Industrie und Forschung bestehende Expertengruppe, formierte sich Ende 2000. Sie sind im Folgenden aufgeführt.

Tabelle 2

ATG Inc.	Allan Scott
BEA Systems Inc.	Alex Toussaint
Fair Isaac Inc.	Johan Majoor
Fujitsu Ltd.	Frank McCabe
IBM Inc.	Rainer Kerth
ILOG SA.	Changhai Ke, Daniel Selman
Oracle Inc.	Mark Hornick
Sandia National Labs	Ernest Friedman-Hill
Silverstream Inc.	Gregg McMullin
Unisys Inc.	Sridhar Iyengar

Ernest Friedmann-Hill von den Sandia National Labs ist der Autor von [FRIED].
Im Dezember 2003 wurde der Entwurf des JSR-94 vom „Executive Committee“ (EC) einstimmig angenommen.[29] Das EC ist ein Konsortium von Firmen, die das Java Projekt stützen (z.B. Sun, IBM, Apache, Borland, SAP). Seine Aufgebe ist es, den Entwicklungsprozess der Javatechnologie zu steuern.[30]
Doch nun zum Inhalt des JSR-94.

Definitionen

[28] http://www.jcp.org/en/procedures/jcp2#1
[29] http://www.jcp.org/en/jsr/results?id=2340
[30] http://www.jcp.org/en/participation/committee

Eine „**Rule Engine**“ ist gemäß dem JSR-94 im Wesentlichen ein ausgeklügelter Interpreter zur Auswertung von IF-THEN-Anweisungen(siehe Allgemein). Als Eingabe bekommt sie ein „**Rule Execution Set**“ (s.u.) und Datenobjekte[31]. Das Output einer Regel-Engine ist abhängig von dem Input. Die Datenobjekte können während einer „**Rule Session**“ (s.u.)manipuliert werden. Es können auch Seiteneffekte die die Fakten nicht verändern ausgelöst werden.

Eine „**Rule**“ ist eine Regel mit einer Bedingung und einer Aktion. Wenn die Bedingung erfüllt ist wird die Aktion ausgeführt. Nach der Definition des JSR-94 gehören zu einer „Rule“ auch noch Metainformationen wie Name und Beschreibung.

Ein „**Rule Execution Set**“ ist eine Menge von Regeln, die wie die Regeln Metainformationen enthalten.

Eine „**Rule Session**“ ist eine Verbindung zwischen Klient und „Rule Engine“ zur Laufzeit. Zu einer „Rule Session“ gehört genau ein „Rule Execution Set“. Eine „Rule Session“ wird beendet, indem der Klient die Ressourcen der „Rule Engine“ freigibt.

Des Weiteren wird zwischen „Stateful Rule Session“ und „Stateless Rule Session“ unterschieden. Der Unterschied besteht darin, dass bei der „Stateful Rule Session“ nach und nach Datenobjekte hinzugefügt, entfernt oder geändert werden können.. Bei der „**Stateless Rule Session**“ hingegen wird ein „Rule Execution Set“ nur einmalig auf einer Menge von Datenobjekten ausgeführt.

Architektur

Die Architektur unterscheidet zwischen einem Laufzeit- und einem Administrations-Paket. Ersteres stellt dem Klienten Methoden zur Verfügung um an eine „Rule Session“ zu gelangen und mit dieser zu interagieren. Das in einer solchen Session genutzte „Rule Execution Set“ wurde vorher mit Hilfe des Administrationspaketes erstellt und registriert. Notwendig ist diese Aufteilung nicht, bringt jedoch den Vorteil, dass eine feinere Unterscheidung unterschiedlicher Benutzergruppen möglich wird.

„Runtime API“

Die „Runtime API“ sorgt dafür, dass man zunächst durch die „RuleServiceProviderManager“ Klasse eine Instanz des „RuleServiceProvider“-Interface des Herstellers erhält. Durch diese kann man sich nun eine Instanz des „RuleRuntime“-Interfaces besorgen, welches wiederum eine Methode enthält, die eine „RuleSession“ zurückgibt. Die Interaktion des Klienten geschieht über genau diese Instanz. Des Weiteren lassen sich durch das „RuleExecutionSetMetadata“ –Interface Metainformationen gewinnen und über das „ObjectFilter“-Interface das Ergebnis einer Ausführung des „Rule Execution Sets“ filtern. Um weiterhin Zugriff auf die zu einer „StatefulRuleSession“ hinzugefügten Objekte zu haben dient das „Handle“-Interface.

Betrachtet man diese Funktionen, fällt auf, dass sie eben genau diejenigen sind, die zwar die Nutzung jedoch nicht die Konfiguration der Regel-Engine gestatten.

Diesem Zweck dient die „Administrator API“.

[31] Die Datenobjekte entsprechen den Fakten.

„Administrator API“

Durch die „RuleServiceProvider“ - Klasse lässt sich zunächst eine Instanz des „RuleAdministrator “- Interfaces organisieren, durch die sich dann ein „Rule Execution Set“ erzeugen lässt. Dieses lässt sich nun durch die „RuleAdministrator“- Instanz registrieren und auch wieder deregistrieren. Man kann sich eine Liste von Regel-Objekten geben lassen und Herstellerspezifische Eigenschaften von Regeln setzen und lesen.

rulez.application Schnittstellenbeschreibung

RuleSession

Einer RuleSession ist ein Objektmodell zugeordnet. Während der Initialisierungsphase wird ihr ein dazu passendes Metamodell in Form einer Regelmenge für die Regel-Engine übergeben. Über das RuleSession-Interface erhält man Zugriff auf alle notwendigen Objekte.

`getRuleSessionName():String:` Ein Name für eine RuleSession ist sinnvoll, weil mehrere RuleSessions parallel existieren können. Durch den Namen lassen sie sich einer Regelbasis zuordnen.

`getRuleGroup(String name):RuleGroup:` Eine RuleSession kann mehrere RuleGroups enthalten. Jede RuleGroup hat einen Namen der in der Regelbasis vergeben wurde. Auf die RuleGroups kann durch den Namen mit dieser Methode zugegriffen werden.

`getRuleGroups():Iterator:` Durch den Aufruf dieser Methode erhält man Zugriff auf alle RuleGroups.

`setActive(boolean active):void:` Mit dieser Methode kann man die Execution-Engine (siehe **Fehler! Verweisquelle konnte nicht gefunden werden.**) der Regel-Engine aktivieren bzw. deaktivieren. Nach der Initialisierung einer RuleSession ist sie zunächst aktiv.

`isActive():boolean:` Mit dieser Methode lässt sich abfragen, ob die Execution-Engine (siehe **Fehler! Verweisquelle konnte nicht gefunden werden.**) der Regel-Engine aktiviert ist.

`registerObserver (ConflictObserver co):void` und **`deregisterObserver (ConflictObserver co):void:`** Mit diesen Methoden lässt sich ein ConflictObserver gemäß des ObserverPatterns [<todo/>] bei einer RuleSession registrieren und deregistrieren.

`getConflicts():Iterator:` Durch diese Methode bekommt man alle aktuell vorliegenden Konflikte. Durch sie lassen sich die eines ConflictObservers bekannten Konflikte mit denen der RuleSession synchronisieren. Das ist von Bedeutung, wenn man einen ConflictObserver „verspätet“, also nach der Entdeckung der ersten Konflikte registriert.

`getConflicPropertyTypes():Iterator:` Durch diese Methode bekommt man alle ConflictPropertyTypes. Sie wird zum Beispiel von der rulez.view-Komponente benutzt um die Konflikttabelle (siehe **Fehler! Verweisquelle konnte nicht gefunden werden.**) aufzubauen.

`getRules():Iterator:` Wie der Name schon sagt…

`getModel():Object:` Diese Methode gibt einem das zu beobachtende Objektmodell also quasi das Dokument zurück. Sie ist bei dem Einsatz in 4Ever nicht unbedingt notwendig. Bei dem Einsatz von 4EverRulez in einer anderen Umgebung kann sie aber durchaus Sinn machen.

`getModelAdapter():ModelAdapter:` Diese Methode liefert den Adapter der für die Aufbereitung des Zustandes eines Objektmodells für die Regel-Engine verantwortlich ist (siehe <todo/>). Sie ist insbesondere zum Testen sinnvoll, da es bei der Erzeugung einer RuleSession durch die Übergabe eines Null-Zeigers anstatt eines Objektmodells möglich ist,

ein Shortcut-Adapter zu bekommen. Mit diesem Shortcut-Adapter hat man direkten Zugriff auf die Faktenbasis der Regel-Engine.

RulezManager

Das RulezManager-Interface dient der Verwaltung. Mit seiner Hilfe ist es möglich RuleSessions zu erzeugen und auf sie zuzugreifen. Die removeRuleSession- und die getRuleSession-Methode sind in ihrer Semantik ziemlich eindeutig und bedürfen keiner weiteren Erläuterung.

Auf die createRuleSession-Methode sollte hingegen genauer eingegangen werden.

`createRuleSession(String file, Object objectModel ):RuleSession:` Das erste Argument ist ein String, der die Position der Datei mit der Regelbasis enthält. Auf die Regelbasis wird noch wesentlich genauer eingegangen. Sie enthält die Regeln für die Regel-Engine sowie ConflictPropertyType- und RuleGroup-Definitionen.

Das zweite Argument kann ein beliebiges Objekt sein auf dem die Bedingungen der Regeln überprüft werden sollen. Vorraussetzung dafür ist, dass der RulezManager für die Klasse des übergebenen Objektes eine Adapter-Klasse kennt und dass entsprechende Regeln in der Regelbasis definiert wurden. Übergibt man als zweites Argument einen Null-Zeiger, initialisiert der RulezManager einen Shortcut-Adapter, über den direkt auf die Faktenbasis der Regel-Engine zugegriffen werden kann. Zugänglich ist er über die getModelAdapter-Methode der RuleSession, die von createRuleSession() zurückgegeben wird.

ConflictObserver

Das ConflictObserver-Interface soll einen Beobachter immer wissen lassen wenn ein Konflikt entsteht oder wenn einer verschwindet. Aus diesem Grund stellt es eine update-Methode mit zwei Argumenten zur Verfügung. Das erste Argument ist ein Konfliktobjekt und das zweite ein Integer-Wert der angibt, ob das Konfliktobjekt neu hinzugekommen oder verschwunden ist.

RuleGroup

Durch das RuleGroup-Interface ist der Zugriff auf Gruppen (siehe 5.1) möglich.

`getRuleGroupName():String:` Die Namen der Gruppen werden in der Regelbasis definiert. Durch diese Methode ist es also möglich, einer RuleGroup eine Gruppendefinition in der Regelbasis zuzuordnen. Beispielsweise wird der Name für die GUI benötigt um einen Namen in der Gruppentabelle anzeigen zu können.

`isAlive():boolean` und

`setAlive(boolean ):void:` Durch diese Methoden kann man den Status der Gruppen setzen und abfragen. Für die Semantik siehe 5.1.

`getRules():Iterator:` Durch diese Methode bekommt man die der Gruppe zugeordneten Regeln. Für die Semantik siehe 5.1.

`getRuleGroupDescription():String:` Für jede Gruppe gibt es in der Regelbasis eine vom Experten erstellte Beschreibung. Diese Beschreibung sollte Informationen zu den Regeln enthalten die der Gruppe zugeordnet sind. Zum Beispiel welchen Aspekt des Dokumentes die Regeln überprüfen oder ob sie besonders Ressourcenintensiv sind.

Rule

Eine Rule entspricht einer Regel (siehe 5.1).

`getRuleName():String:` Durch den von dieser Methode zurück gegebenen Regelnamen lassen sich die Regeln einer Regel in der Regelbasis zuordnen.

`getDescription():String:` Für jede Regel gibt es in der Regelbasis eine vom Experten erstellte Beschreibung. In ihr sollten Informationen enthalten sein, welche Metamodellbedingungen von der Regel geprüft werden.

`getRuleGroups():Iterator:` Mit dieser Methode bekommt man alle Gruppen in denen die Regel enthalten ist. Dies sollte wegen FA8 (siehe 0) mindestens eine sein.

ConflictRole

Eine ConflictRole entspricht einer Rolle (siehe 5.1).

`getInstances():Iterator:` Die mit dieser Methode zurück gegebenen Instances entsprechen den Konfliktstellen (siehe 5.1).

`getConflictRoleName():String:` Mit dem Namen einer Rolle lässt sich ein Bezug zur Regelbasis herstellen und er kann in der GUI genutzt werden.

`getDescription():String:` Für jede Gruppe gibt es in der Regelbasis Beschreibung. Sie sollte konfliktstellenspezifische Angaben für den Dokumentenbearbeiter für die Konfliktlösung enthalten.

`getPrimaryInstance():Object:` Diese Methode ist notwendig, um dem Dokumentenbearbeiter eine Entscheidung für eine Konfliktstelle zu ersparen. Sie gibt eine der mehr oder weniger gleichwertigen Konfliktstellen zurück.

ConflictProperty und ConflictPropertyTypes

Eine ConflictProperty entspricht einer Eigenschaft (siehe 5.1). Diese Eigenschaften haben einen Typ, der ihnen durch einen ConflictPropertyType zugeordnet wird. Die möglichen Typen sowie die Eigenschaften der Konflikte werden in der Regelbasis festgelegt.

Conflict

Ein Conflict entspricht einem Konflikt (siehe 5.1).

`getConflictRoles():Iterator:` Mit dieser Methode bekommt man alle Rollen die in einem Konflikt zu vergeben sind. Welche das sind wird in der Regelbasis festgelegt.

`getDescription():String:` Für jede Gruppe gibt es in der Regelbasis eine Beschreibung. Sie sollte allgemeine Anweisungen enthalten, wie der Konflikt zu beheben ist. Beispielsweise kann sie Empfehlungen enthalten, welche Rollen die Instanzen einnehmen, die bearbeitet werden sollten.

`getPrimaryRole():ConflictRole:` Diese Methode ist notwendig, um dem Dokumentenbearbeiter eine Entscheidung für eine Konfliktstelle zu ersparen. Sie gibt eine Rolle zurück, deren Konfliktstellen sich besonders gut eignen einen Konflikt zu lösen.

`getProperties():Iterator` und

`getProperty(ConflictPropertyType cpt):ConflictProperty:` Für einen Konflikt sind bestimmte Eigenschaften in der Regelbasis definiert. Diese lassen sich mit diesen Methoden auslesen. Diese Eigenschaften können in der GUI oder anders benutzt werden. Zum Beispiel wäre es möglich, bei der Existenz eines Konflikts mit einer bestimmten Eigenschaft eine Aktion zu verbieten. Dies könnte zum Beispiel das Generieren eines PDF-Dokumentes sein.

`getRuleSession():RuleSession:` Gibt die zugeordnete RuleSession zurück.

`getRule():Rule:` Ein Konflikt wird durch den Aktionsteil einer Regel erzeugt. Diese wird von der getRule-Methode zurück gegeben.

Anhang B

Rulebase:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<FourEverRuleBase name="simplestRuleBase">
      <RuleSet>

      <ruleServiceProviderClass>org.drools.jsr94.rules.RuleServiceProviderI
mpl</ruleServiceProviderClass>

      <ruleServiceProviderURL>http://drools.org</ruleServiceProviderURL>
            <jsr94-rule-execution-set>
                  <rule-set name="a RuleSet" description="Eine Regelbasis
für Instanzen des MinimalModel"
                      xmlns="http://drools.org/rules"
                      xmlns:java="http://drools.org/semantics/java">
                      <rule name="Akt_Prod" description="Diese Regel feürt,
wenn ein Produkt mehr als eine fertig stellende Aktivität hat.">
                        <parameter identifier="group1">

<java:class>com.foursoft.fourever.rulez.application.impl.RuleGroupImpl</jav
a:class>
                          </parameter>

                              <parameter identifier="oa">

<java:class>com.foursoft.fourever.rulez.application.impl.adapter.MinimalObj
ectModelAdapter</java:class>
                          </parameter>

                          <parameter identifier="aktivitaet1">

<java:class>com.foursoft.fourever.objectmodel.ComplexInstance</java:class>
                          </parameter>
                          <parameter identifier="aktivitaet2">

<java:class>com.foursoft.fourever.objectmodel.ComplexInstance</java:class>
                          </parameter>
                          <parameter identifier="produkt">

<java:class>com.foursoft.fourever.objectmodel.ComplexInstance</java:class>
                          </parameter>

                          <java:condition>
                              (group1.getRuleGroupName().equals("RefInt")
                              && group1.isAlive())
                        </java:condition>

                        <java:condition>

      (oa.hasType(aktivitaet1,"/Vorgehensmodell/Vorgehensbausteine/Vorgehen
sbaustein/Aktivitaeten/Aktivitaet"))
                        </java:condition>
```

```
                        <java:condition>

      (oa.hasType(aktivitaet2,"/Vorgehensmodell/Vorgehensbausteine/Vorgehen
sbaustein/Aktivitaeten/Aktivitaet"))
                        </java:condition>
                        <java:condition>

      (oa.hasType(produkt,"/Vorgehensmodell/Vorgehensbausteine/Vorgehensbau
stein/Produkte/Produkt"))
                        </java:condition>


                        <java:condition>

      (oa.isLinked(aktivitaet1,"ProduktRef",produkt))
                        </java:condition>

                        <java:condition>

      (oa.isLinked(aktivitaet2,"ProduktRef",produkt))
                        </java:condition>

                        <java:condition>
                              (aktivitaet1.compareTo(aktivitaet2)<0)
                        </java:condition>



<java:consequence>
      System.out.println("Rule Akt_Prod has fired.");

      //---------  Einen Conflict erzeugen
      com.foursoft.fourever.rulez.application.impl.ConflictImpl conflict =
                  new
com.foursoft.fourever.rulez.application.impl.ConflictImpl();
      conflict.setRulename("Akt_Prod");
      conflict.addFact(group1);
      conflict.addFact(oa);
      conflict.addFact(aktivitaet1);
      conflict.addFact(aktivitaet2);
      conflict.addFact(produkt);

      conflict.setDescription("Dieser Konflikt lsst sich am besten beheben,
wenn eine Konfliktstelle mit der Rolle 'Aktivitt' den Anweisungen
entsprechend bearbeitet wird.");

      //---------  Eine ConflictRole erzeugen
      com.foursoft.fourever.rulez.application.impl.ConflictRoleImpl role1 =
                  new
com.foursoft.fourever.rulez.application.impl.ConflictRoleImpl();

      role1.setRoleName("Aktivitäten");
      role1.setDescription("Sie können den Konflikt beheben, indem sie die
Produktreferenz aus einer Aktivität ändern.");
      role1.addInstance(aktivitaet1);
      role1.addInstance(aktivitaet2);
      role1.setPrimaryInstance(aktivitaet1);
      conflict.addConflictRole(role1);

      //---------  Eine ConflictRole erzeugen
      com.foursoft.fourever.rulez.application.impl.ConflictRoleImpl role2 =
      new com.foursoft.fourever.rulez.application.impl.ConflictRoleImpl();
```

```
      role2.setRoleName("Produkt");
      role2.setDescription("Bearbeiten Sie eine Objekt mit der Rolle
'Aktivitäten'");
      role2.addInstance(produkt);
      role2.setPrimaryInstance(produkt);
      conflict.addConflictRole(role2);

      //---------- ConflictProperties erzeugen
      com.foursoft.fourever.rulez.application.impl.ConflictPropertyImpl
prop1 = new
com.foursoft.fourever.rulez.application.impl.ConflictPropertyImpl();
      prop1.setConflictPropertyTypeName("Name");
      prop1.setValue("Akt_Prod-Konflikt");
      conflict.addConflictProperty(prop1);



      com.foursoft.fourever.rulez.application.impl.ConflictPropertyImpl
prop2 = new
com.foursoft.fourever.rulez.application.impl.ConflictPropertyImpl();
      prop2.setConflictPropertyTypeName("Priorität");
      prop2.setValue(new Integer(2));
      conflict.addConflictProperty(prop2);

      com.foursoft.fourever.rulez.application.impl.ConflictPropertyImpl
prop3 = new
com.foursoft.fourever.rulez.application.impl.ConflictPropertyImpl();
      prop3.setConflictPropertyTypeName("Author");
      prop3.setValue("theile");
      conflict.addConflictProperty(prop3);

      com.foursoft.fourever.rulez.application.impl.ConflictPropertyImpl
prop4 = new
com.foursoft.fourever.rulez.application.impl.ConflictPropertyImpl();
      prop4.setConflictPropertyTypeName("noCheckIn");
      prop4.setValue(new Boolean(false));
      conflict.addConflictProperty(prop4);

      //---------- den Konflikt melden
      drools.assertObject(conflict);
</java:consequence>
                     </rule>


                     <rule name="Akt_Prod-RETRACT" >
                           <parameter identifier="conflict">

<java:class>com.foursoft.fourever.rulez.application.impl.ConflictImpl</java
:class>
                           </parameter>
                              <parameter identifier="group1">

<java:class>com.foursoft.fourever.rulez.application.impl.RuleGroupImpl</jav
a:class>
                           </parameter>

                              <parameter identifier="oa">

<java:class>com.foursoft.fourever.rulez.application.impl.adapter.MinimalObj
ectModelAdapter</java:class>
                           </parameter>

                           <parameter identifier="aktivitaet1">
```

```
<java:class>com.foursoft.fourever.objectmodel.ComplexInstance</java:class>
                        </parameter>
                        <parameter identifier="aktivitaet2">

<java:class>com.foursoft.fourever.objectmodel.ComplexInstance</java:class>
                        </parameter>
                        <parameter identifier="produkt">

<java:class>com.foursoft.fourever.objectmodel.ComplexInstance</java:class>
                        </parameter>


                          <java:condition>
                              (conflict.getFact(0)==group1)
                         </java:condition>

                         <java:condition>
                               (conflict.getFact(1)==oa)
                         </java:condition>
                         <java:condition>
                               (conflict.getFact(2)==aktivitaet1)
                         </java:condition>
                         <java:condition>
                               (conflict.getFact(3)==aktivitaet2)
                         </java:condition>
                         <java:condition>
                               (conflict.getFact(4)==produkt)
                         </java:condition>

                           <java:condition>
                          (!(
                               (group1.getRuleGroupName().equals("RefInt")
                               && group1.isAlive())
                               &&

      (oa.hasType(aktivitaet1,"/Vorgehensmodell/Vorgehensbausteine/Vorgehen
sbaustein/Aktivitaeten/Aktivitaet"))
                               &&

      (oa.hasType(aktivitaet2,"/Vorgehensmodell/Vorgehensbausteine/Vorgehen
sbaustein/Aktivitaeten/Aktivitaet"))
                               &&

      (oa.hasType(produkt,"/Vorgehensmodell/Vorgehensbausteine/Vorgehensbau
stein/Produkte/Produkt"))
                               &&

      (oa.isLinked(aktivitaet1,"ProduktRef",produkt))
                               &&

      (oa.isLinked(aktivitaet2,"ProduktRef",produkt))
                               &&
                                     (aktivitaet1.compareTo(aktivitaet2)<0)
                               ))
                         </java:condition>



                           <java:consequence>
                               drools.retractObject(conflict);
                           </java:consequence>
                     </rule>
```

```
                  </rule-set>

            </jsr94-rule-execution-set>
      </RuleSet>

      <RuleGroups>
            <RuleGroup>
                  <name>RefInt</name>
                  <description>In der Gruppe RefInt sind alle Regeln
enthalten, die die Integrität von Referenzen überprüfen. Diese Überprüfung
ist nicht sehr teuer und sollte daher immer eingeschaltet
sein.</description>
                  <isAlive>false</isAlive>
                  <Rules>
                        <rule>Akt_Prod</rule>
                  </Rules>
            </RuleGroup>
      </RuleGroups>


      <ConflictPropertyTypes>

            <ConflictPropertyType>
                  <key>Name</key>
                  <description>Der Name des Konflikts.</description>
                  <valueType>java.lang.String</valueType>
            </ConflictPropertyType>

            <ConflictPropertyType>
                  <key>Priorität</key>
                  <description>1=hoch, 2=mittel, 3=gering</description>
                  <valueType>java.lang.Integer</valueType>
            </ConflictPropertyType>

            <ConflictPropertyType>
                  <key>Author</key>
                  <description></description>
                  <valueType>java.lang.String</valueType>
            </ConflictPropertyType>

            <ConflictPropertyType>
                  <key>noCheckIn</key>
                  <description>Wenn bei einem Konflikt der value==true ist,
wird ein Einchecken verhindert.</description>
                  <valueType>java.lang.Boolean</valueType>
            </ConflictPropertyType>

      </ConflictPropertyTypes>

</FourEverRuleBase>
```