Fachhochschule Darmstadt
University of Applied Sciences
Fachbereich Informatik
Studiengang Bachelor-Informatik

Einsatz von Rule-Engines zur flexiblen Wissensverarbeitung im betrieblichen Umfeld

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science in Informatik
(Bachelor of Science in Computer Science)

Hendrik Beck
Matrikel-Nr.: 645274

Referent: Prof. Dr. Klaus Kasper
Korreferent: Prof. Dr. Gerhard Raffius

Darmstadt, 18. Oktober 2005

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form keinem anderen Prüfungsamt vorgelegen.

Hendrik Beck
Darmstadt, 18. Oktober 2005

Sperrvermerk

Diese Arbeit beinhaltet interne, vertrauliche Informationen der Firma camunda GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften, auch in digitaler Form, gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung des Autors oder der Firma:

camunda GmbH
Mühlwehrstrasse 43
97980 Bad Mergentheim

Vielen Dank

Herrn Prof. Dr. Klaus Kasper für die hilfreiche, lehrreiche und sehr angenehme Unterstützung während dieser Arbeit.

Meinem Kommilitonen und Freund Till für das Durchhalten und die gegenseitige Unterstützung.

Meiner guten Freundin Martina für das Korrekturlesen und für das Vertrauen in meine Arbeit.

Meinem Kommilitonen, Mitbewohner und Freund Sascha für die hilfreiche gegenseitige Unterstützung und für die Gesellschaft in unserem Büro.

Meinem Arbeitskollegen und guten Freund Bernd für die ständige Bereitschaft zur Unterstützung und für die große Hilfe.

Meiner Familie. Für alles. Und dafür, dass sie es mir ermöglicht hat, so weit zu kommen, wie ich gekommen bin. Und dafür, dass ich diesen Dank jetzt in meiner Bachelorarbeit ausdrücken kann.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Inhalt und Motivation

Das Thema dieser Arbeit entstand im betrieblichen Umfeld, als das im Unternehmen vorhandene Wissen und Know-How mit dem Ziel einer automatisierten Verarbeitung dokumentiert und erfasst werden sollte. Dabei bestand die Forderung, das Wissen sowohl intern zur Unterstützung verschiedener Abteilungen und Prozesse sowie extern zur Aufwertung der Internetpräsenz einzusetzen. So sollte zum einen der Gesamtnutzen in ein besseres Verhältnis zum eingesetzten Aufwand von Entwicklung und Wissenserfassung gebracht werden und zum anderen die redundante Haltung von Wissen reduziert werden.

Das Wissen, das einem solchen System zu Grunde liegt, ist naturgemäß in den Fachabteilungen und den Köpfen der ihnen angehörigen Mitarbeiter vorhanden und muss möglichst effektiv in ein automatisiert verarbeitbares Format übertragen und zentral gespeichert werden. Neben einer ebenfalls schnellen und kostengünstigen Durchführung der Erfassung besteht hier zusätzlich der Anspruch, die Fakten und Zusammenhänge möglichst intuitiv und verlustfrei erfassen zu können. Dies setzt ein System voraus, das natürliche Denkweisen und -muster unterstützt und wenig Transformation des Wissens auf dem Weg in das System verlangt. Die direkte Erfassung des Wissens durch den jeweiligen Mitarbeiter der Fachabteilung ohne den Weg über einen Mitarbeiter der IT-Abteilung sollte dabei entscheidende Vorteile bringen. Diese Tatsache wird nicht zuletzt durch langjährige Erfahrungen des Anforderungsmanagements der Softwaretechnik gestützt.

Es liegt weiterhin in der Natur von Produkten im IT-Umfeld, dass sich, bedingt durch die kurzen Technologie- und Produktzyklen, die Fakten und Zusammenhänge, die einer unternehmensweiten Wissensbasis zu Grunde liegen, sehr schnell ändern. Deshalb sollte das Wissen mit Hilfe der zu entwickelnden Lösung schnell, flexibel und kostengünstig gepflegt werden können.

Untersuchungen im Bereich des Knowledge-Managements zeigten gerade in den letzten Jahren das Potenzial auf, das durch die Dokumentation von Wissen und dessen sinnvolle und konsequente Verwertung innerhalb eines Unternehmens vorhanden ist.[1] Dieses Potential bleibt aber größtenteils durch nicht vorhandene Systeme und fehlende Richtlinien völlig ungenutzt. Deshalb war eine weitere Anforderung, dieses Wissen nicht isoliert für eine einzelne Anwendung, sondern als allgemein verfügbares und verwendbares Wissen zu erfassen. Dies wiederum erfordert eine allgemein verwendbare Notation zur Wissensrepräsentation, eine Entkopplung des Wissens von den konkreten Anwendungsbereichen sowie Software-Komponenten, die dieses Wissen für verschiedene Zwecke aufbereiten und verarbeiten können.

[1] siehe [Her00].

Der Geschäftsbereich *mad-moxx* des Unternehmens *Computation*, für das diese Arbeit durchgeführt wurde, hat sich spezialisiert auf den Versand von Grafikkarten und Grafikkartenzubehör über das Internet. Gerade bei der Spezialisierung auf eine kleine Nische ist der Transport eines Images hilfreich, das eine hohe Kompetenz im selbst zugesprochenen Fachgebiet zum Ausdruck bringt. Daher sollte eine Möglichkeit evaluiert werden, das erfasste Know-How in einer innovativen Form im Web zur Verfügung stellen zu können. Besonderes Augenmerk lag dabei auf einer individuellen Aufbereitung des Wissens für den Besucher im Internet, das persönliche Informationen in den Verarbeitungsprozess einfließen lässt.

Dies führte zu einem Anforderungsprofil, von dem folgende Punkte in der hier vorliegenden Arbeit untersucht und durchgeführt werden sollten:

- Wahl eines Repräsentationsformalismus', der eine flexible Erfassung und Pflege sowie eine universell verwendbare Verarbeitung des Wissens ermöglicht
- Entwicklung einer Komponente, die eine Verarbeitung dieses Wissens für verschiedene Anwendungen und Anwendungsbereiche ermöglicht
- Entwicklung einer Komponente, die die Einbeziehung von persönlichen Kundeninformationen in den Wissensverarbeitungsprozess ermöglicht
- Identifikation von kritischen Problemen und Absicherung des Konzeptes anhand prototypischer Lösungen

Eine Evaluierung verschiedener Möglichkeiten führte zu regelbasierten Systemen, die ihre Ursprünge schon in den Anfängen der Forschung auf dem Gebiet der künstlichen Intelligenz haben, in letzter Zeit aber als sog. „Rule-Engines“ eine Wiedergeburt im Umfeld geschäftlicher Anwendungen erfuhren. Auf dem Gebiet der wissensbasierten Systeme, dem auch die regelbasierten Systeme angehören, boten sich in ausreichender Form Literatur, Dokumentationen und Referenzen, so dass die Entscheidung für diese Lösung auch theoretisch fundiert begründet werden konnte.

In der Arbeit wird eine Einführung in Theorie und Praxis der Arbeit mit regelbasierten Systemen und Rule-Engines gegeben. Schließlich wird eine mögliche praktische Umsetzung anhand prototypischer Entwicklungen in der Programmiersprache Java und unter Verwendung der Open-Source Rule-Engine Drools vorgestellt. Auf dieser Grundlage wird am Ende die erarbeitete Lösung zusammenfassend bewertet.

1.2 Aufbau der Arbeit

Die Arbeit ist gegliedert in sieben Kapitel. Nach der Einleitung in **Kapitel 1** beschäftigt sich **Kapitel 2** mit den Grundlagen von Regeln und regelbasierten Systemen.

Die **Kapitel drei bis fünf** erläutern verschiedene Aspekte in Bezug auf den Einsatz regelbasierter Systeme. Außer Rule-Engines werden hier noch Geschäftsregeln und Aspekte der Integration einer Rule-Engine innerhalb geschäftlicher Architekturen erläutert.

Kapitel sechs widmet sich dem Praxisbeispiel und der Darstellung des praktischen Einsatzes vieler in den vorangegangenen Kapiteln erläuterter Grundlagen.

Der Schluss in **Kapitel sieben** fasst die wichtigsten Punkte dieser Arbeit bewertend zusammen und versucht, einige Ausblicke auf offene Aufgaben und weitere Möglichkeiten dieser Arbeit zu geben.

Kapitel 2

Grundlagen zu Regeln und regelbasierten Systemen

2.1 Einführung

Regeln bestehen aus zwei Teilen: einer Bedingung und einer Konsequenz. Ihr Grundprinzip ist aus dem Alltag bekannt und lässt sich an einer einfachen Regel leicht erkennen:

„Wenn ich müde bin, dann gehe ich ins Bett!“

Die Bedingung dieser Regel, auch *Prämisse* genannt, wird genau dann wahr, wenn der Sprecher dieser Regel müde wird. Genau in diesem Fall wird die Konsequenz, auch *Konklusion* genannt, ausgeführt. Das bedeutet in diesem Fall, dass der Sprecher ins Bett geht. Diese *„Wenn .. dann ..“* - Struktur ist bei allen Regeln grundsätzlich gleich. Es sind sowohl mehrere Prämissen als auch mehrere Konklusionen innerhalb einer Regel möglich, wobei dann erst bei Eintreffen aller Teilbedingungen auch die gesamte Prämisse wahr wird und alle Konsequenzen ausgeführt werden.[1] Dazu wird die oben genannte Regel noch um eine weitere Regel ergänzt. Diese beiden Beispiele werden im Laufe des Kapitels wieder aufgegriffen.

1. *„Wenn ich müde bin, dann gehe ich ins Bett!“*
2. *„Wenn ich müde bin und es vor 21 Uhr ist, dann lege ich mich auf die Couch!“*

Regeln lassen sich mit den bekannten `if..then..`-Abfragen aus der Programmierung vergleichen. Dies mag an diesem Punkt sehr einleuchtend und logisch erscheinen. Da die Regeln bei der Arbeit mit Rule-Engines aber nicht als `if..then..`-Abfragen fest im Quellcode einer Programmiersprache implementiert werden, werden Regeln für den Moment noch auf einer etwas abstrakteren Ebene betrachtet.

Auch Wissen, das für den Menschen von Natur aus nicht in einer „Wenn .. dann ..“-Struktur ausgedrückt wird, kann in den meisten Fällen durch Umformen durch eine solche Regel formuliert werden. Die beiden Aussagen „Alle Vögel können fliegen“ und „In der Schule darf kein Kaugummi gekaut werden“ kann z.B. durch folgende umgeformte Regeln formuliert werden:

[1] Da Regeln ihren Ursprung in der Prädikatenlogik haben, sind eigentlich auch alle dort verfügbaren Verknüpfungen wie z.B. auch eine ODER-Verknüpfung innerhalb der Prämisse möglich. Im späteren Verlauf der Arbeit wird aber deutlich, warum in der Praxis nur UND-Verknüpfungen verwendet werden. Aus diesem Grund wird schon an diesem Punkt nur auf die Verwendung von UND-Verknüpfungen eingegangen.

- *„Wenn ein Tier ein Vogel ist, dann kann es fliegen!“*
- *„Wenn Du in der Schule bist, darfst Du keinen Kaugummi kauen!“*

An diesen beiden Beispielen lässt sich auch erkennen, dass Regeln teilweise auf mehrere Arten formuliert werden können und jeweils die selbe Aussage besitzen, teilweise aber auch nicht. Werden im ersten Beispiel Prämisse und Konklusion miteinander vertauscht („Wenn ein Tier fliegen kann, ist es ein Vogel“), stimmt die Aussage nicht mehr, denn fliegen können z.B. auch Insekten. Werden sie im zweiten Beispiel vertauscht („Wenn Du Kaugummi kaust, darfst Du nicht in der Schule sein!“), stimmt die Aussage immer noch, auch wenn sie sich vielleicht im Alltag seltsam anhören würde.

Somit können mit Regeln schon sehr viele Aussagen in einer strukturierten Form erfasst werden. Auch bei der Analyse von Software-Systemen lässt sich feststellen, dass sich ein Großteil der dort implementierten Geschäftslogik durch Regeln beschreiben lässt.[2] Im Verlauf der Arbeit werden praktische Anwendungsbereiche für Regeln noch intensiver betrachtet. Um dennoch schon zu Beginn eine Vorstellung vom Einsatz in realen geschäftlichen Anwendungen zu ermöglichen, werden hier einige Beispiele für „richtige“ Regeln aufgezählt:

- *„Wenn der Gesamtpreis der Bestellung größer als 400 Euro ist, dann gewähre 10% Rabatt!“*
- *„Wenn ein Kunde noch nicht registriert ist, dann soll nur die Versandart 'Nachnahme' zur Verfügung stehen!“*
- *„Wenn sich mindestens ein Artikel der Kategorie 'Rotwein' im Warenkorb befindet, dann zeige die Meldung 'Wir empfehlen Ihnen zu Ihrem Rotwein unser exklusives Set an Rotweingläsern!' an!“*
- *„Wenn alle Artikel einer noch nicht ausgelieferten Bestellung auf Lager sind, dann setze den Status der Bestellung auf 'versandfertig' und veranlasse die Auslieferung!“*

Ohne im Moment schon wissen zu müssen, wie genau diese Regeln erfasst, gepflegt und verarbeitet werden, ist schon zu sehen, dass sich viele verschiedene Anforderungen der Praxis mittels Regeln ausdrücken lassen. In der klassischen Programmierung würden solche Regeln im Quellcode programmiert werden, so dass die Anwendung genau die Funktionalitäten erfüllt, die das Unternehmen an sie setzt. Im Gegensatz dazu bieten Regeln Möglichkeiten, die Funktionalitäten von Anwendungen zu beschreiben, ohne sie im Quellcode programmieren zu müssen. Dazu sind Voraussetzungen notwendig, damit die Anwendung diese Regeln lesen und selbstständig verarbeiten kann. Diese Aufgabe wird in der Praxis von speziellen Laufzeitsystemen, sog. „Rule-Engines“, übernommen. Diese Arbeit beschreibt die theoretischen Hintergründe dieser regelbasierten Systeme, und wie sie in der Praxis eingesetzt werden können.

In den folgenden Abschnitten werden zunächst die wichtigsten Eigenschaften von Regeln erklärt, bevor in den anschließenden Kapiteln Rule-Engines, Regeln und deren praktischer Einsatz erläutert werden.

[2] Werden im Prozess der Anforderungsanalyse von Softwaresystemen nur die fachlichen Anforderungen betrachtet, so wird auch hier versucht, eine strukturierte Form zu finden, in der alle Anforderungen aufgenommen werden. Somit könnte die hier beschriebene Regel-Syntax auch direkt als solche Struktur zur Erfassung aller Anforderungen benutzt werden. Gerade im Zusammenspiel mit der Verwendung einer Rule-Engine zur Implementierung der Anforderungen könnten hier hilfreiche Synergieeffekte entstehen. Als weiterführende Literatur wird z.B. [Rup04a] empfohlen.

2.2 Schließen von neuem Wissen (Inferenz)

Das Schließen von Wissen aus vorhandenem Wissen und gegebenen Regeln wird Inferenz genannt. Im einfachsten Fall kann bei Eintreffen einer Bedingung schon aus einer Regel eine neue Information gewonnen werden. Es ist aber auch möglich, aus der Verkettung mehrerer Regeln Informationen zu gewinnen. Dabei werden zwei Strategien unterschieden, die Vorwärtsverkettung und die Rückwärtsverkettung. Die verschiedenen Möglichkeiten zur Inferenz werden in diesem Abschnitt beschrieben.[3]

2.2.1 Einfache Inferenz

„Wenn ich müde bin, dann gehe ich ins Bett!“

Zu Beginn ist nichts über die momentane Aktivität des Sprechers bekannt. Wenn er nun allerdings müde wird, dann folgt aus dieser Regel, dass er ins Bett geht. Also kann diese Aktivität als neue Information aus der Regel und dem gegebenen Wissen gewonnen werden. Damit erweitert sich auch die aktuelle Wissensbasis um eine neue Information.

„Wenn der Gesamtpreis der Bestellung größer als 400 Euro ist, dann gewähre 10% Rabatt!“

Auch bei dieser Regel, die dem Kunden eines Warenhauses bei einer Bestellung von über 400 Euro 10% Rabatt gewährt, kann bei Eintreffen der Bedingung eine neue Information gewonnen werden: der nun um 10% verminderte neue Gesamtpreis.

Formal lässt sich die einfache Inferenz mit dem Satz des *Modus Ponens* der Aussagenlogik vergleichen. Dieser lautet:

$$(A \Rightarrow B) \wedge A \Rightarrow B$$

Er ist zu lesen als *„Wenn aus A B folgt und A gilt, dann gilt auch B“*. Auf Regeln übertragen ist A die Bedingung, bei deren Eintreffen (*„wenn A gilt“*) die Konsequenz B folgt (*„dann gilt auch B“*).[4]

In der Praxis beeinflussen sich allerdings meist mehrere Regeln gegenseitig. Dieser Einfluss wurde bei der einfachen Inferenz noch völlig außer Acht gelassen und jede Regel isoliert betrachtet. In den folgenden beiden Abschnitt über Verkettungen wird dies erläutert.

2.2.2 Vorwärtsverkettung

Bei der Vorwärtsverkettung wird versucht, die aus Regeln gewonnenen, neuen Informationen wiederum als Eingabewerte für neue Regeln zu verwenden, um so nochmals neue Informationen zu gewinnen.

1. *„Wenn ich müde bin, dann gehe ich ins Bett!“*
2. *„Wenn ich ins Bett gehe, ziehe ich mir einen Schlafanzug an!“*

Bisher wurde aus der Feststellung, dass der Sprecher müde wird, nur gefolgert, dass er ins Bett geht. Nun existiert eine zweite Regel, deren Bedingung wahr wird, sobald der Sprecher ins Bett geht. Sobald die Information bekannt wird, dass er ins Bett geht, könnte sie direkt auf die zweite Regel angewendet werden und damit nun auch geschlussfolgert werden, dass er sich einen Schlafanzug anzieht. Dies würde zu folgender Verkettung beider Regeln führen:

[3] siehe [Bar82], [Bib93], [AR01] und [GG03].
[4] Siehe Literatur zu Grundlagen der Mathematik bzw. der Logik. Hier entnommen aus [Soe74].

1. *Da er müde wird, geht er ins Bett! (Regel 1)*
2. *Da er ins Bett geht (aus Regel 1!), zieht er sich einen Schlafanzug an! (Regel 2)*

Es werden aus einer Information zwei neue Informationen aus zwei verschiedenen Regeln gewonnen! Aus den Konsequenzen von Regeln können also Informationen gewonnen werden, die wiederum zur Prüfung der Bedingungen anderer Regeln verwendet werden können.

Es kann aber noch eine weitere Eigenschaft der Vorwärtsverkettung beobachtet werden: Die Konsequenz der Regel wird genau in dem Moment ausgeführt, in dem die Bedingung erfüllt ist. Im selben Moment steht nun auch eine neue Information zur Verfügung, weshalb auch die Bedingung der zweiten Regel geprüft werden kann. Das bedeutet, dass Bedingungen immer dann neu geprüft werden müssen, wenn Informationen hinzugefügt, geändert oder entfernt werden. Aufgrund dieses zeitlichen Zusammenhangs zwischen dem Moment des Eintreffens neuer Informationen und des erneuten Prüfens aller Bedingungen wird die Vorwärtsverkettung auch als *Ereignis-basierte Inferenz* [5] bezeichnet. Das Eintreffen neuer Informationen wird dabei als das auslösende Ereignis betrachtet.

Auch die Vorwärtsverkettung lässt sich formal mit einem Satz der Aussagenlogik, mit dem Satz des „Modus Barbara“, vergleichen. Dieser lautet:

$$(A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$$

Er ist zu lesen als *„Wenn aus A B folgt und aus B C folgt, dann folgt auch aus A C“*. Wenn B also die Konsequenz der Bedingung A ist, und B selbst wieder eine Bedingung darstellt, aus der dann C folgt, dann folgt implizit aus A auch C. Bei Regeln wird aber entgegen der Mathematik der Zwischenschritt immer ausgeführt.[6]

Da Systeme, die die Vorwärtsverkettung unterstützen, immer neues Wissen, neue Fakten oder allgemein neue Informationen „produzieren“, werden sie auch oft als *Produktionssysteme*[7] bezeichnet.

2.2.3 Rückwärtsverkettung

Da die Vorwärtsverkettung versucht, aus vorhandenem Wissen jede mögliche neue These zu erschließen, ist sie schlecht geeignet bei Fragestellungen zum Wahrheitsgehalt bestimmer Thesen. Sobald im Beispiel bekannt wurde, dass der Sprecher müde ist, wurde auch bekannt, dass er ins Bett geht, und dass er einen Schlafanzug anzieht. Auf die Frage, unter welchen Bedingungen denn (laut den Regeln zumindest) der Sprecher einen Schlafanzug anzieht, müssten alle Regeln vorwärts verkettet werden, um genau die Verkettungen zu erhalten, die mit der gewünschten Information enden. Geschickter und schneller ist aber die Vorgehensweise, von der zu prüfenden Information auszugehen und rückwärts alle Bedingungen zu prüfen, die dafür notwendig sind. Im Beispiel würden folgende Schritte rückwärts durchgeführt werden:

- „Der Sprecher zieht genau dann einen Schlafanzug an, wenn er ins Bett geht.“
- „Und er geht genau dann ins Bett, wenn er müde wird!“

Also hängt der Wahrheitsgehalt der These, dass der Sprecher einen Schlafanzug anzieht, von zwei Bedingungen ab. Dass diese beiden Bedingungen darüber hinaus direkt voneinander abhängen, ist hier nur Zufall. Es könnten auch beliebig viele voneinander unabhängige Bedingungen sein.

[5] engl. *event based inference* oder *triggering inference*.
[6] Entnommen aus [Soe74].
[7] engl. *production systems*.

Da die Rückwärtsverkettung vom gewünschten Ziel (also einer zu überprüfenden Aussage) ausgeht, wird sie auch als Ziel-getriebene Inferenz[8] bezeichnet.

2.2.4 Anwendungsbeispiel in der Praxis

Populäre und in der Praxis gut funktionierende Beispiele von Inferenzmaschinen sind Expertensysteme zur Diagnostik. Dabei besteht z.B. im medizinischen Bereich bei der Diagnose von Krankheiten zum einen die Möglichkeit, aus einer Menge von Symptomen auf alle möglichen Krankheitsbilder zu schließen (Vorwärtsverkettung). Zum anderen besteht aber auch die Möglichkeit, ein bestimmtes Krankheitsbild aufgrund aller dazu hinreichenden Symptome zu überprüfen (Rückwärtsverkettung). Diagnose-Systeme werden außer in der Medizin in etlichen Bereichen eingesetzt, z.B. auch zur Fehlersuche an technischen Geräten.

Produktberatungen im Bereich des Handels können auf ähnliche Weise gelöst werden. Hierbei könnten dann z.B. Voraussetzungen beim Kunden oder spezielle Wünsche des Kunden als Bedingungen formuliert werden (vergleichbar mit Krankheitssymptomen) und empfohlene Produkte oder Dienstleistungen als Schlussfolgerungen der Regeln (vergleichbar mit den Krankheitsbildern). Mit Hilfe der beiden Inferenzstrategien kann dann auf die beiden in der Praxis vorkommenden Fragestellungen Antwort gegeben werden:

- „Dies sind meine Wünsche und die bei mir vorliegenden Voraussetzungen! Welche Produkte sind möglich / werden mir empfohlen?“
- „Ich bin interessiert an diesem Produkt! Was muss ich beachten / welche Voraussetzungen müssen vorliegen?“

2.2.5 Zusammenfassung

Der Begriff der *Inferenz* bezeichnet also allgemein die Verwendung von gegebenem Wissen und den zugehörigen Regeln, um neue Informationen zu gewinnen. Die einfache Inferenz bietet dabei die Grundlage für die beiden Verkettungsstrategien Vorwärtsverkettung und Rückwärtsverkettung. Die Wahl der Strategie hängt dabei von der Aufgabenstellung der Inferenz ab:

- Sollen zum Zeitpunkt des Eintreffens neuer Informationen alle daraus zu schließenden Informationen abgefragt werden, eignet sich die Vorwärtsverkettung.
- Soll dagegen eine bestimmte These auf der Grundlage der bisherigen Informationen und der zugehörigen Regeln geprüft werden, eignet sich die Rückwärtsverkettung.

Für die Praxis kann allgemein gesagt werden, dass die Vorwärtsverkettung flexibler und mächtiger ist als die Rückwärtsverkettung, da sie sich auf vielfältigere Art und Weise in geschäftliche Anwendungen integrieren lässt. In der Praxis gibt es Implementierungen von Inferenzmaschinen, die nur eine der beiden Strategien beherrschen und solche, die beide Möglichkeiten anbieten. Darauf wird bei der Marktübersicht in Abschnitt 3.4 näher eingegangen.

Eine Aufgabenstellung wurde bei der Beschreibung der Inferenz deutlich: Immer wieder muss aufgrund von bekannten Informationen geprüft werden, welche Bedingungen aller Regeln erfüllt sind, damit evtl. deren Konsequenzen ausgeführt werden können. Bei den beschriebenen Beispielen stellte dies noch kein Problem dar. In der Praxis arbeiten große regelbasierte Systeme allerdings häufig mit mehreren hundert bis tausend Regeln und bis zu mehreren zehntausend Informationen. Dann kann das Prüfen der Bedingungen unter Umständen sehr lange dauern. Dieses Problem und eine mögliche Lösung wird im nächsten Abschnitt beschrieben.

[8] engl. *goal driven inference*.

2.3 Auswertung einer Menge von Regeln

Bei der Vorwärtsverkettung besteht eine der Hauptaufgaben darin, regelmäßig zu prüfen, welche Bedingungen aller Regeln anhand der aktuell verfügbaren Informationen zutreffen und welche Konsequenzen daraufhin ausgeführt werden können. Aufgrund der in der Praxis großen Anzahl an Informationen und Regeln (und damit auch an Bedingungen) wird dies zu einer wichtigen und anspruchsvollen Aufgabe. Dies liegt vor allem in der Komplexität dieser Aufgabe und in der Tatsache, dass die Laufzeit (wie bei allen Anwendungen in der Praxis) eine wichtige Rolle bei der Praxistauglichkeit regelbasierter Systeme spielt. In den folgenden Abschnitten werden der leistungsfähigste verfügbare Algorithmus zur Lösung dieser Aufgabe sowie dessen Komplexität erläutert.

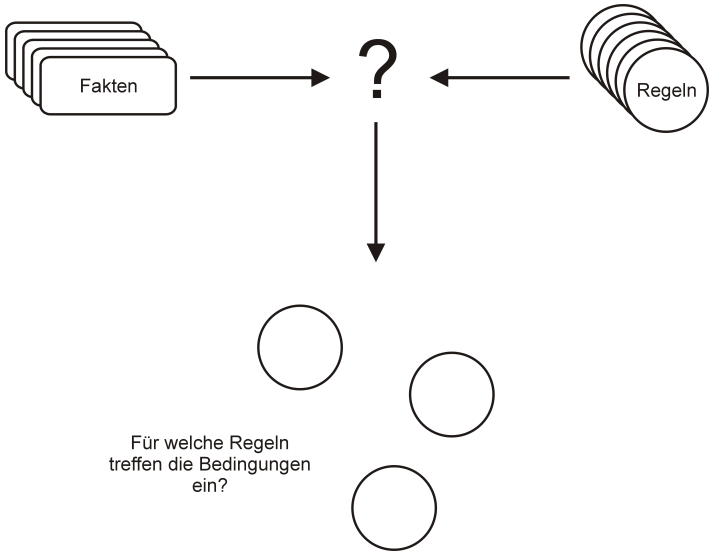


Abbildung 2.1: Auswertung von Regeln gegen Fakten: Pattern-Matching

2.3.1 Das Pattern-Matching-Problem

Das Problem des Prüfens von vielen Bedingungen anhand vieler zu Grunde liegender Informationen wird als Pattern-Matching-Problem[9] bezeichnet. In den folgenden Abschnitten wird erläutert, warum der einfachste Algorithmus zur Lösung dieses Problems in der Praxis nicht anwendbar ist, und durch welche Optimierungen ein Algorithmus entwickelt wurde, der diese Aufgabe sehr viel besser löst.

2.3.2 Ein einfacher Pattern-Matching-Algorithmus

Bei jedem Durchlauf[10] muss für jede Regel geprüft werden, ob alle Prämissen erfüllt sind, und in diesem Fall müssen die entsprechenden Aktionen ausgeführt werden. In Pseudo-Code könnte ein Algorithmus zur Lösung dieses Problems also folgendermaßen aufgebaut sein:

```
Für jede Regel:
```

[9] Dies könnte frei als „Problem zur Prüfung der Gleichheit von Mustern“ übersetzt werden. Muster sind in diesem Fall die Bedingungen und deren zu Grunde liegende Informationen.

[10] Ein „Durchlauf“ beginnt immer dann, wenn eine neue Information auftaucht. Siehe dazu die Erläuterung der Vorwärtsverkettung in Abschnitt 2.2.2.

```
{
  Für jede Prämisse:
  {
    Ist eine Information vorhanden, damit diese Prämisse wahr wird?
  }

  Wenn alle Prämissen erfüllt sind, dann:
  {
   Führe die entsprechenden Aktionen aus!
  }

}
```

Für jede definierte Regel wird also jede Bedingung darauf hin geprüft, ob eine Information existiert, durch die die entsprechende Bedingung wahr wird. Obwohl dieser Algorithmus sicherlich korrekt ist, ist er aus mehreren Gründen sehr ineffizient:

- Viele Fakten ändern sich von einem zum nächsten Durchlauf nicht. Trotzdem würden alle Prämissen neu berechnet werden, obwohl sich deren Werte seit dem letzten Durchlauf nicht geändert haben.
- Viele Regeln beinhalten in der Praxis gleiche Prämissen. Trotzdem würden innerhalb eines Evaluierungslaufes alle Prämissen einzeln berechnet werden, was zu einer mehrmaligen Prüfung der gleichen Prämisse innerhalb verschiedener Regeln führt.

1. *„Wenn ich müde bin, dann gehe ich ins Bett!“*
2. *„Wenn ich müde bin und es vor 21 Uhr ist, dann lege ich mich auf die Couch!“*
3. ..
4. ..
5. Regel n-1
6. Regel n

Angenommen, es existieren n Regeln und m Informationen. Dann ändern sich aller Voraussicht nach die Informationen, auf denen die Bedingungen der ersten beiden Regeln beruhen, nur sehr selten im Vergleich zu allen anderen Informationen. Trotzdem würde der hier vorgestellte einfache Algorithmus in jedem Durchlauf auch die Bedingungen der ersten beiden Regeln prüfen, obwohl sich die Information vielleicht erst nach 10, 20 oder 1000 Durchläufen ändern.

Die ineffiziente Auswertung der Regeln sowie die Tatsache, dass viele Regeln teilweise identische Prämissen besitzen, sind die beiden Hauptprobleme des einfachen Pattern-Matching-Algorithmus' in der Praxis. Zur Lösung dieser Probleme hat Charles Forgy 1982 den Rete-Algorithmus entwickelt. Er hat sich seitdem als Standard für das Pattern-Matching in vorwärtsverkettenden regelbasierten Systemen entwickelt. Die in Abschnitt 3.4 beschriebenen Rule-Engines *JRules*, *Jess* und Drools nutzen z.B. (leicht abgewandelte) Implementierungen dieses Konzeptes, das im folgenden Abschnitt beschrieben wird.

2.3.3 Der Rete-Algorithmus

Der Rete-Algorithmus, der 1982 von Charles Forgy entwickelt wurde, benutzt zur Lösung des Pattern-Matching-Problems die beiden folgenden Ansätze:[11]

- Da sich bei jedem Durchlauf nur wenige Informationen ändern, werden auch nur diejenigen Bedingungen neu berechnet, die von den neuen bzw. geänderten Informationen betroffen sind.
- Er speichert alle Ergebnisse von bereits ausgewerteten Bedingungen, so dass diese Ergebnisse in jedem Durchlauf nur um die neu berechneten Ergebnisse ergänzt werden müssen.

Der Rete-Algorithmus ist eine effiziente Implementierungsbeschreibung dieses Lösungsansatzes. Dazu baut Rete einen Baum aus Knoten verschiedener Typen auf:[12]

- **Prämissenknoten:** Diese Knoten stellen eine Prämisse dar, wie z.B. *„Ist der Gesamtpreis größer als 400 Euro?“*, d.h. sie stellen einen Test auf *genau einen Fakt* dar. Jede einzelne Prämisse erzeugt solch einen Knoten.
- **Verbindungsknoten:** Enthält eine Regel mehr als eine Prämisse, so werden die Verknüpfungen zwischen jeweils zwei Prämissen durch einen Verbindungsknoten dargestellt. Das Ergebnis eines solchen Knotens ist genau dann wahr, wenn beide Prämissen erfüllt sind.[13] Das Ergebnis eines solchen Knotens kann dann auch wieder als Input für einen weiteren Verbindungsknoten dienen. Bei drei Prämissen werden z.B. die ersten beiden Prämissen durch einen Verbindungsknoten miteinander verknüpft, und das Ergebnis dieses Knotens zusammen mit dem Ergebnis des dritten Prämissenknotens wird durch einen weiteren Verbindungsknoten verknüpft. Es werden also so viele Verbindungsknoten eingesetzt, bis alle Prämissen am Ende in einem einzigen Ergebnis enden.
- **Aktionsknoten** Dieses Ergebnis am Ende dient dann als Eingabewert für den Aktionsknoten, der die Aktion der Regel repräsentiert.

Anmerkung: Die Literatur nennt diese beiden ersten Knotentypen im Allgemeinen *one-input nodes* und *two-input nodes*, da sie entweder einen (Prämissenknoten) oder zwei (Verbindungsknoten) Eingabewerte besitzen, auf deren Grundlage die Evaluierung für diesen Knoten erfolgt. Zum besseren Verständnis werden im Folgenden die hier gewählten Übersetzungen verwendet.

Beim ersten Evaluierungsdurchgang werden alle Prämissenknoten berechnet, und die Ergebnisse werden gespeichert. Ab dem zweiten Durchgang werden dann nur die Prämissen berechnet, deren Fakten sich geändert haben. Ändert sich damit auch das Ergebnis der Prämisse, wird das alte Ergebnis damit überschrieben. Das im Folgenden beschriebene Verhalten gilt für jeden Evaluierungsdurchlauf.

Ist eine Prämisse erfüllt (wenn z.B. analog zum oben genannten Beispiel der aktuelle Gesamtpreis über 400 Euro ist), gibt der Prämissenknoten sein Ergebnis an den folgenden Knoten weiter. Nun gibt es zwei Möglichkeiten:

[11] [For90], [Lab05], [FH03], S.133 ff.

[12] Ein Baum im Allgemeinen Sprachgebrauch der Graphentheorie besitzt eine Wurzel (meist oben dargestellt) und endet in Blättern. In diesem Fall beginnt der Baum mit den Prämissenknoten als Blätter und endet im Aktionsknoten als Wurzel. Diese Verdrehung der eigentlichen Logik kann deshalb akzeptiert werden, da sie nur für das Verständnis des Algorithmus' in der einfachsten Form notwendig ist und in den anschließenden Optimierungsschritten entfällt. Dort wird statt Bäumen ein Netz aus allen Regeln erzeugt.

[13] Wie weiter oben erwähnt, werden in der Praxis nur UND-Verknüpfungen verwendet.

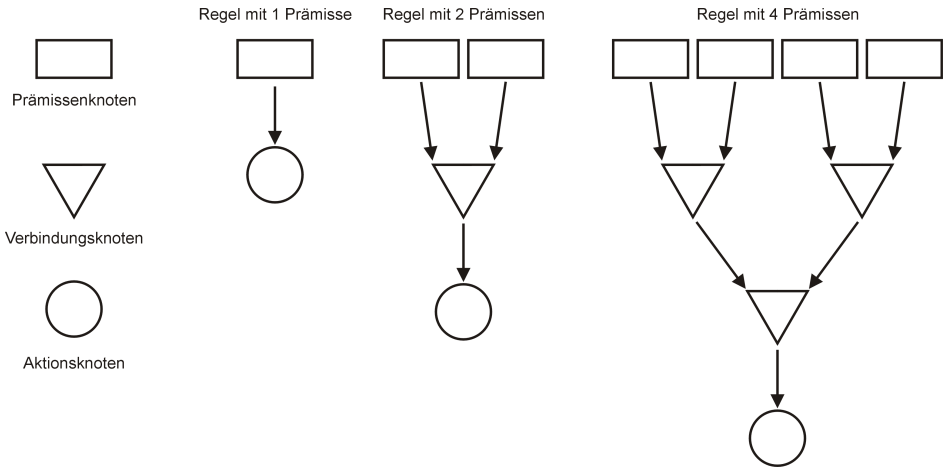


Abbildung 2.2: Knotentypen beim Rete-Algorithmus

- Ist der folgende Knoten ein Aktionsknoten, so besitzt dieser Aktionsknoten nun eine gültige Prämisse als Eingang und somit wird die Aktion ausgeführt
- Ist der folgende Knoten ein Verbindungsknoten (der zwei Eingänge besitzt, siehe oben), so liegt nun auf einem der beiden Eingänge ein gültiges Ergebnis. Wenn auf dem zweiten Eingang nun ebenfalls schon ein gültiges Ergebnis liegt (dies kann auch aus dem vorherigen Evaluierungslauf stammen), sind beide Eingangsprämissen erfüllt, und damit ist auch das Gesamtergebnis dieses Knotens erfüllt. Er kann dieses Ergebnis nun seinerseits an den nachfolgenden Knoten weitergeben. Ab hier gelten wieder die beiden hier beschriebenen Möglichkeiten.

Eine Änderung in den Fakten und somit eine evtl. Änderung in den Prämissenknoten ganz oben pflanzt sich also immer weiter nach unten fort und gelangt genau dann bis ganz nach unten zum Aktionsknoten, wenn *alle* Prämissen erfüllt sind. Denn genau in diesem Fall soll ja die Aktion der Regel ausgeführt werden. Wenn eine Prämisse nicht erfüllt ist, dann wird auch der folgende Aktions- oder Verbindungsknoten nicht wahr und die gesamte Regel ebenfalls nicht.

Dieser Algorithmus berechnet also in jedem Durchgang nur Prämissen, deren Fakten sich geändert haben. Allerdings besitzt auch der Rete-Algorithmus in der hier beschriebenen Grundform noch enormes Optimierungspotenzial, das im nachfolgenden Abschnitt erläutert wird.

2.3.4 Optimierungen des Rete-Algorithmus'

Der Rete-Algorithmus besitzt in seiner Grundform noch weiteres Potenzial zur Optimierung, das anhand des folgenden Beispiel erläutert wird.

Beispiel: Die Aufgabenstellung lautet, ein passendes Abendprogramm zu finden. Dazu sind folgende drei Regeln gegeben:

1. „Wenn es stürmt, dann bleiben wir auf jeden Fall zu Hause!"
2. „Wenn es regnet und ein guter Film im Fernsehen kommt, dann bleiben wir zu Hause!"

3. „Wenn es regnet, aber nicht stürmt und auch kein guter Film im Fernsehen kommt, dann gehen wir aus!“

Regel 1 erzeugt einen recht einfachen Graphen mit einem Prämissenknoten und einem Aktionsknoten. Die Regeln 2 und 3 erzeugen schon kompliziertere Graphen mit einem bzw. zwei Verbindungsknoten.

Wie im vorherigen Abschnitt erläutert, würde der Rete-Algorithmus bei einer Änderung eines Faktums alle zugehörigen Prämissen neu berechnen. Ändert sich nun das Wetter von *„Sonne“* auf *„Regen“*, würden also die beiden Prämissen in Regel 2 und 3 neu berechnet werden, die prüfen, ob es regnet. Da diese beiden Prämissen aber genau gleich sind, muss eigentlich nur einmal geprüft werden, ob es regnet, und im geeigneten Fall können dann beide anschließenden Knoten von dieser Prämissenänderungen benachrichtigt werden. Es werden also alle mehrfach vorhandenen Prämissenknoten aus dem Graphen entfernt und die ausgehenden Kanten des einen verbleibenden Knotens mit den Knoten, mit denen die ursprünglichen Prämissenknoten verbunden waren, verbunden.

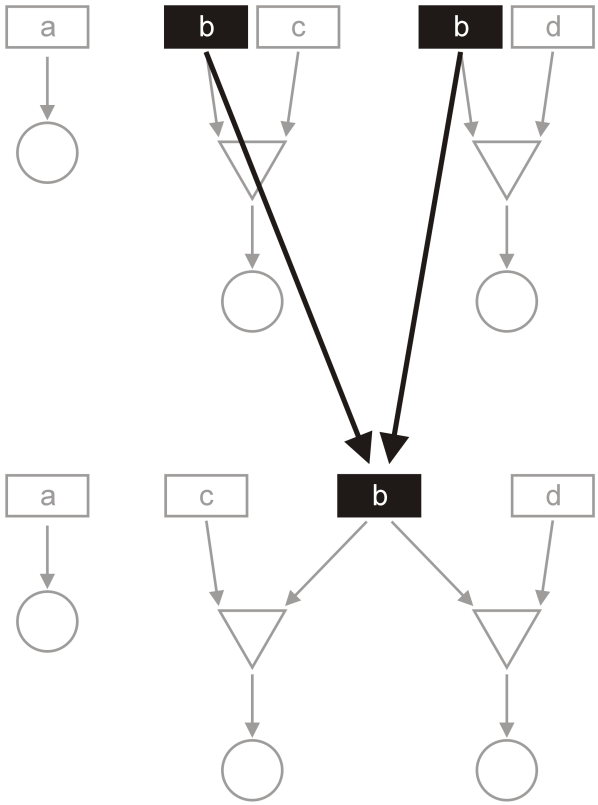


Abbildung 2.3: Reduktion von Prämissenknoten beim Rete-Algorithmus

Somit wird der Aufwand der Berechnungen pro Durchgang noch weiter reduziert.[14]

Wenn sich verschiedene Regeln nun nicht nur eine, sondern mehrere gleiche Prämissen teilen, so kann eine weitere Optimierung dadurch vorgenommen werden, dass nicht nur Prämissen, sondern auch deren Verbindungsknoten geteilt werden. Das Ergebnis eines solchen

[14] Auch diese Tatsache, dass sich viele Regeln gleiche Prämissen teilen, kann an regelbasierten Systemen in der Praxis bestätigt werden.

Verbindungsknoten ist dann verbunden mit den Knoten, mit denen die jeweiligen Verbindungsknoten verbunden waren.

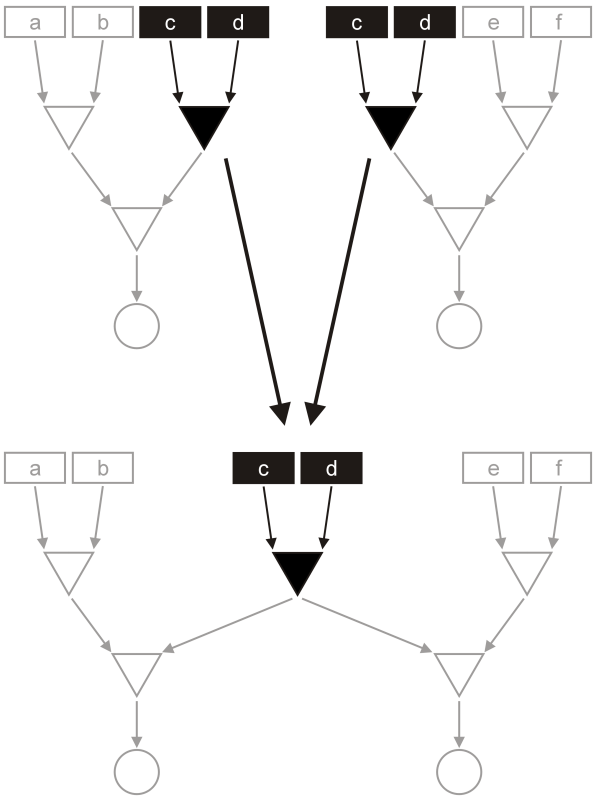


Abbildung 2.4: Reduktion von Verbindungsknoten beim Rete-Algorithmus

Nach diesen Optimierungsschritten entsteht aus anschaulich recht geordneten, einzelnen Graphen für jede Regel ein gesamter, eng vernetzter Graph für das komplette Regelset. Daher auch der Name „Rete“, abgeleitet vom lateininischen *„rete“*, was im Deutschen *Netz* bedeutet.

2.3.5 Komplexitätsbetrachtung

Die Komplexität des zu Beginn betrachteten einfachen Algorithmus' beträgt

$$O((RI)^P)$$

wobei R die Anzahl der Regeln ist, I ist die Anzahl der Informationen und P ist die durchschnittliche Zahl an Prämissen pro Regel. Diese Komplexität lässt sich konstant für jeden Durchlauf berechnen, da der Algorithmus sehr vorhersehbar arbeitet. Schon bei einem recht einfachen Regelsystem mit 20 Regeln, 100 Informationen und 3 Bedingungen pro Regel ergibt sich folgende Anzahl an durchzuführenden Pattern-Matching-Operationen:

$$A = (20 * 100)^3 = 8.000.000.000 \text{ Operationen}$$

Bei einer geschätzten Dauer von $\frac{1}{600.000}$ Sekunde pro Operation dauert die komplette Prüfung aller Regeln 3,7 Stunden.[15]

[15] Die Dauer pro Operation wurde beispielhaft hergeleitet aus der Leistungsfähigkeit der Rule-Engine Jess, die am Ende des Abschnittes dargestellt ist.

Schwerer fällt die Bewertung des Rete-Algorithmus'. Da beim ersten Durchlauf noch keine Ergebnisse bekannt sind, auf die zurückgegriffen werden kann, muss für jede Regel jedes Ergebnis berechnet werden, womit die Laufzeit dem des einfachen Algorithmus' ähnelt. Da gleiche Prämissenknoten und evtl. auch gleiche Verbindungsknoten nur einmal berechnet werden, ist die Laufzeit in der Praxis aber erheblich geringer, kann im schlimmsten Fall aber der oben dargestellten Komplexität entsprechen. Dies kann aber nur bei genauer Kenntnis des verwendeten Regelsets beurteilt werden. Ab dem zweiten Durchlauf muss der Algorithmus nur noch geänderte Prämissen prüfen. Da Evaluierungsdurchläufe im Allgemeinen in sehr kurzen Abständen, oft sogar nach jeder Änderung einer Information, gestartet werden, kann die dafür benötigte Zeit vernachlässigt werden. Wenn der Evaluierungsdurchlauf in größeren Abständen gestartet wird und sich in der Zwischenzeit sehr viele Objekte geändert haben, dauert die Evaluierung natürlich länger.[16] Dennoch liegt die Laufzeit in praktischen allen Fällen deutlich unter der des einfachen Algorithmus. Die Komplexität des Rete-Algorithmus kann allgemein wie folgt angegeben werden:[17]

$$O(RIP)$$

wobei R wieder der Anzahl der Regeln entspricht, I der Anzahl der Informationen und P der durchschnittlichen Anzahl an Prämissen pro Regel.

Wie schon erwähnt kann die oben genannte Rechnung zur Ermittlung einer beispielhaften Laufzeit nicht auf den Rete-Algorithmus übertragen werden. Konkrete Modelle zur Evaluierung von Laufzeit und Speicherbedarf wurden z.B. in [ARW04] und [Hel96] entwickelt und diskutiert, sollen hier aber nicht näher erläutert werden.

2.3.6 Aktueller Stand

Obwohl der verbesserte Rete-Algorithmus vor allem im Vergleich zum einfachsten Ansatz schon eine sehr hohe Effizienz besitzt, gelingen doch immer wieder Verbesserungen bezüglich der Performance. Diese Verbesserungen sind aber zum Teil nicht mehr frei zugänglich[18] oder gehen sehr ins Detail, so dass sie kein Bestandteil dieser Arbeit sein sollen.

Außerdem verwenden die einzelnen Rule-Engine-Produkte meist Varianten des Rete-Algorithmus', die hier auch nicht dargestellt werden sollen. Drools z.B. verwendet einen als *ReteOO*[19] bezeichneten Algorithmus, der ein Pattern-Matching objektorientierter Fakten ermöglicht.

2.4 Konflikte bei der gleichzeitigen Gültigkeit mehrerer Regeln

Das im vorangegangenen Abschnitt beschriebene Pattern-Matching prüfte eine Menge von Regeln gegen eine Anzahl von Fakten und ermittelte daraufhin, welche Regeln erfüllt waren, damit deren Konsequenzen ausgeführt werden können. Das Resultat des Pattern-Matchings ist also eine Menge bzw. eine Liste von Regeln.

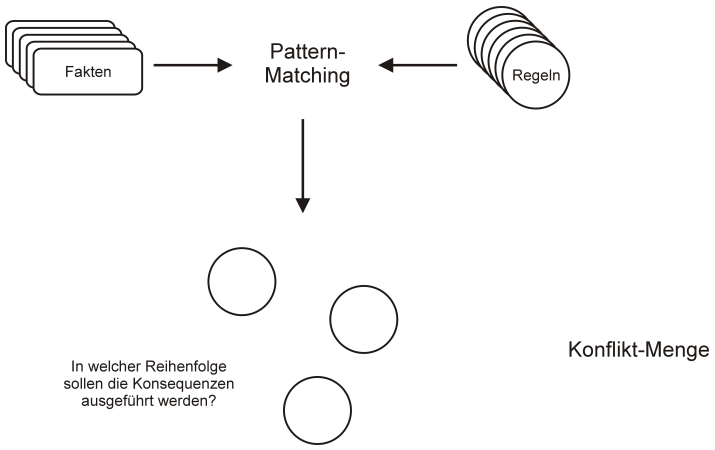


Abbildung 2.5: Die Konflikt-Menge

2.4.1 Die Konflikt-Menge

Diese Liste ist zu diesem Zeitpunkt noch unsortiert.[20] Wenn sie nun mehr als eine Regel enthält, dann muss entschieden werden, in welcher Reihenfolge die Konsequenzen ausgeführt werden. Diese Menge an Regeln, deren Bedingungen gleichzeitig erfüllt sind, wird *Konflikt-Menge* (engl. *conflict set*) genannt.

1. *„Wenn ich müde oder erschöpft bin, dann gehe ich ins Bett!“*
2. *„Wenn ich müde bin und es vor 21 Uhr ist, dann lege ich mich auf die Couch!“*

Angenommen, der Sprecher ist müde und es ist vor 21 Uhr, so sind die Prämissen beider Regeln vollständig erfüllt. Es müssten nun also auch die Konsequenzen beider Regeln ausgeführt werden. Nun kann der Sprecher aber aus ersichtlichen Gründen nicht auf der Couch und im Bett gleichzeitig liegen. Wenn man dieses Problem mit menschlichem Verstand betrachtet, würde man z.B. sagen, dass die zweite Regel eine Ausnahme von der ersten ist und sie deshalb der ersten vorzuziehen ist. Ein Software-System muss diese Entscheidung allerdings eindeutig und möglichst nachvollziehbar treffen können.

2.4.2 Strategien zur Konflikt-Lösung

Daher gibt es in der Praxis Konfliktlösungsstrategien, die versuchen, diese Liste der Regeln möglichst sinnvoll und eindeutig zu sortieren. Die populärsten Strategien zur Konfliktlösung sind:

- **Exakteste Regel:** Die Exaktheit der Regel kann z.B. anhand der Anzahl ihrer Bedingungen bestimmt werden. Dabei wird angenommen, dass die Regel, bei der am meisten Bedingungen übereinstimmen, diejenige ist, die am besten zur momentanen Situation passt.

[16] Bei dem in Abschnitt 5.2.3 beschriebenen Einsatzszenario „Permanenter Working Memory“ wird die Evaluierung asynchron zur Änderung der Faktenbasis gestartet.

[17] Siehe [LA87], [Hel96] und [ARW04].

[18] Charles Forgy, der Entwickler des Rete Algorithmus', hat z.B. kürzlich den Algorithmus *Rete III* fertig gestellt, der einige Verbesserungen vor allem bezüglich der Laufzeit bieten soll. Dieser wird aber in Produkten seiner Firma *RulesPower* vertrieben und nicht veröffentlicht.

[19] *„OO“* steht hier für *object-oriented* bzw. *objektorientiert*.

[20] Daher ist die Bezeichnung *Menge* im Grunde besser geeignet.

- **Aktuellste Regel:** Hierzu kann z.B. das Änderungs- oder Erstellungsdatum der Regeln herangezogen werden. Dieses wird meist als Meta-Information zu jeder Regel erfasst.
- **Reihenfolge in der Regelbasis:** Die Regel, die in der Reihenfolge im Rule-Set weiter vorne steht, wird bei der Konflikt-Lösung bevorzugt. Diese Möglichkeit wird sehr selten verwendet, da die Reihenfolge im Rule-Set gerne im Sinne der Übersichtlichkeit nach anderen Kriterien sortiert wird.
- **Statische Prioritäten:** Die Regel mit der höheren Priorität wird einer anderen Regel bevorzugt. Die Möglichkeit wird oft genutzt, um z.B. einige wichtige Regeln zu markieren. Diese Strategie bietet eine sehr exakte Steuerung der Konflikt-Lösung, erfordert aber auch den größten Aufwand an Pflege der Prioritäten.
- **Fachspezifische Strategien:** In der Literatur existieren viele verschiedene Strategien und Heuristiken, um die jeweils beste Konfliktlösung zu erreichen. Z.B. könnte die Hierarchie des Erstellers der Regel im Unternehmen für die Konfliktlösung herangezogen werden, so dass die Regel des Abteilungsleiters immer der des zugehörigen Mitarbeiters vorgezogen wird. Solche Strategien bieten eine sehr übersichtliche und einfache Möglichkeit der Konflikt-Lösung, da meist keine zusätzlichen Daten erfasst werden müssen.

In der Praxis werden häufig verschiedene Strategien kombiniert. Dies ist unter anderem auch aus dem Grund sinnvoll, da manche Strategien keine eindeutige Konflikt-Lösung ermöglichen. Wird z.B. nur die Strategie der exaktesten Regel herangezogen, kann der Fall eintreten, dass sich mehrere Regeln im Konflikt-Set befinden, die als „gleich exakt“ angenommen werden und deshalb keine eindeutige Aussage getroffen werden kann. Dann kann innerhalb dieser Regeln eine andere Strategie zur Konflikt-Lösung eingesetzt werden.[21] Als letzte zu verwendende Strategie wird meist eine eingesetzt, die eine sicherere Entscheidung bringt, z.B. die der Reihenfolge in der Rule-Base.

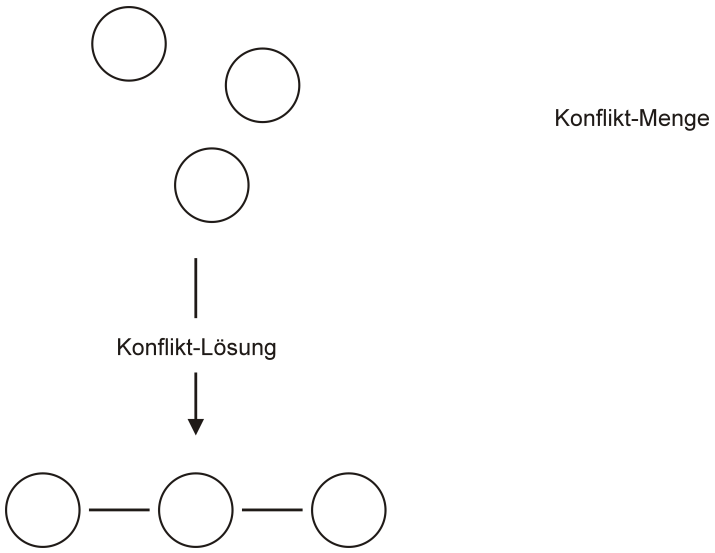


Abbildung 2.6: Durchgeführte Konfliktlösung

[21] Dieses Vorgehen ist vergleichbar mit dem Sortieren einer Tabelle nach mehreren Kriterien. Ein Telefonbuch wird z.B. zuerst nach Nachnahmen und dann innerhalb gleicher Nachnamen nach Vornamen sortiert.

2.5 Deklarative Programmierung

2.5.1 Definition

Die deklarative Programmierung ist ein Programmierparadigma, das häufig als Gegenspieler zur sog. prozeduralen Programmierung positioniert wird. Mit der prozeduralen Programmierung ist der Programmierstil gemeint, der in allen klassischen Programmiersprachen wie Pascal, C, C++ oder Java angewendet wird. Er basiert darauf, Probleme in Sequenzen einzelner Anweisungen zu zerlegen, die am Ende zur Lösung des Problems führen. Diese Befehlssequenzen definieren über Algorithmen, Abfragen und Kontrollstrukturen, wie das Programm zur Lösung des Problems gelangt. Dabei wird die eigentliche Aufgabe des Programms, also *was* zu tun ist, vermischt mit der Art der Implementierung, also *wie* es zu tun ist. Obwohl diese Vorgehensweise bisher zweifellos mit sehr viel Erfolg auch in größeren Projekten durchgeführt wurde, hat sie dennoch einige Nachteile:

Die Logik eines Programms ist nicht durch die Definition der Anforderung, also durch die benötigte Problemlösung, definiert, sondern durch die Art und Weise, wie diese Lösung erreicht wird. Sie ist dadurch nur mit Kenntnissen der verwendeten Programmiersprache lesbar. Zudem können sich durch den Zwischenschritt von der Anforderungsdefinition hin zur Umsetzung Fehler einschleichen. Je stärker ein Problem nun von der Art der Umsetzung, d.h. z.B. von speziellen Konstrukten der Programmiersprache, abhängt, desto schwerer wiegt der oben genannte Punkt und desto schwieriger ist die Lösung des Problems aus dem Quellcode abzulesen. Wird das Problem zu stark von solchen Konstrukten beherrscht, wird es unter Umständen in der Praxis unlösbar.

Die deklarative Programmierung[22] versucht nun, diese beiden Bestandteile voneinander zu trennen. Dafür wird die Struktur des Programms geteilt in zwei Phasen, die Deklaration und die Interpretation. In der Deklaration wird das Ziel bzw. die Aufgabe des Programms in einer für den jeweiligen Zweck definierten Syntax beschrieben. Während der Interpretationsphase wird diese Deklaration gelesen und auf eine spezielle Weise gelöst. So beinhaltet die Deklaration keinerlei Konstrukte der eigentlichen Implementierung, wodurch sie einfacher, kürzer und übersichtlicher sowie auch ohne intensive Kenntnisse der Programmiersprache zu lesen ist.

Ein bekanntes Beispiel dafür sind Datenbankabfragen mittels SQL[23]. Aufgabe von Datenbankabfragen ist z.B. die Abfrage von Datensätzen, die bestimmten Kriterien entsprechen. SQL stellt eine Möglichkeit bereit, in einer definierten Syntax Befehle zu deklarieren, die beschreiben, welche Anforderung die Datenbankanfrage erfüllen muss. Der Datenbank-Treiber schließlich interpretiert diese SQL-Befehle und setzt sie in Anfragen für die konkrete Datenbank um. Eine SQL-Anfrage, wie z.B. `SELECT * FROM Kunden WHERE Name = 'Meier'`, ist daher frei von Konstrukten der Art und Weise der Umsetzung dieser Anfrage. Sie beschreibt lediglich das gewünschte Endergebnis. Damit lassen sich SQL-Anfragen nach einmaliger Deklaration auch von verschiedenen Datenbanktreibern auf verschiedene Datenbanken anwenden. Die Vorgehensweise der Datenbankanfrage mittels SQL kann somit als deklarativer Ansatz bezeichnet werden. Daraus folgen die beiden Hauptbestandteile der deklarativen Programmierung:

- **Deklaration:** In einer bestimmten (aber per se wahlfreien) Syntax wird beschrieben, was als Endergebnis benötigt wird.

[22] siehe [Rup04b], [Ros03] und [FH03].

[23] *„Structured Query Language“*, die verbreitetste standardisierte Sprache für Datenbankzugriffe, siehe [Axm03].

- **Interpretation:** Ein Interpreter bzw. eine Laufzeitumgebung liest die Deklaration und verarbeitet sie entsprechend seiner speziellen Implementierung.

Im Beispiel von SQL bietet die deklarative Vorgehensweise damit die Möglichkeit, auf kurze und prägnante Weise die Anforderung zu beschreiben, ohne auf die Art der Implementierung eingehen zu müssen. Es wird also nur deklariert, *was* benötigt wird, aber nicht, *wie* es interpretiert werden soll.

2.5.2 Regeln als Mittel zur deklarativen Programmierung

Es wurde erwähnt, dass die Vorteile der deklarativen Programmierung umso sichtbarer werden können, je deutlicher ein Problem von der Art der Implementierung abhängig ist. Es wurde ebenfalls erwähnt, dass Regeln eine abstraktere Beschreibung dessen sind, was in der klassischen Programmierung durch `if..then..`-Statements gelöst wird, also die Beschreibung von Abfragen auf Werte des aktuellen Kontextes. Wenn eine Anwendung nun in starkem Maße von solchen Entscheidungen abhängt, kann dies zu Problemen führen. Der zu programmierende Quellcode wird dann dominiert von `if..then..`-Statements und kann bei tiefen und komplexen Verschachtelungen dieser Abfragen evtl. nur noch sehr schwer lesbar sein. Dennoch ist diese komplexe Programmierung von Abfragen eine Umsetzung von Anforderungen, die in natürlicher Sprache meist einfach und unabhängig voneinander beschrieben werden können. In diesem Fall bietet die deklarative Programmierung mit regelbasierten Systemen eine Möglichkeit, die Probleme der Umsetzung von der Beschreibung der Anforderungen zu trennen und darüber hinaus die gesamte Lösung erheblich zu vereinfachen.

Die in Form von Regeln beschriebenen Anforderungen werden dabei durch einen getrennten Interpreter, die „Rule-Engine“, verarbeitet. Dieser Interpreter löst dann die Probleme, Abfragen auf korrekte Weise zu verschachteln und zu jedem Zeitpunkt die jeweils benötigten Aktionen auszuführen. Dabei darf natürlich nicht vernachlässigt werden, dass die Entwicklung des Interpreters eine in höchstem Maße anspruchsvolle und wahrscheinlich auch prozedural ausgeführte Aufgabe sein kann. Daher muss auch für jeden Anwendungsbereich durch eine Kosten-Nutzen-Rechnung ermittelt werden, ob der eingesparte Aufwand bei der Implementierung der Geschäftslogik den Aufwand übersteigt, der zur Entwicklung des Interpreters notwendig ist. Dabei ist es evtl. aber auch möglich, Probleme sehr viel effizienter zu lösen als es mittels prozeduraler Programmierung möglich wäre. Durch die Spezialisierung auf die Verarbeitung von Regeln behandeln Rule-Engines z.B. komplexe Entscheidungsmengen sehr viel effizienter als `if..then..`-Statements im Quellcode. Zudem kann die Implementierung der Problemlösung zu jedem Zeitpunkt ausgetauscht, geändert oder verbessert werden. Regelbasierte Systeme bieten darüber hinaus noch einen weiteren erheblichen Vorteil: Da sie einen weit verbreiteten Anwendungsbereich darstellen, sind mittlerweile viele leistungsfähige Interpreter am Markt erhältlich. Damit entfällt der Aufwand für die Entwicklung fast vollständig, von evtl. entstehenden Kosten für den Kauf und die Einarbeitung abgesehen. In diesem Fall kann von Anfang an von den Vorteilen der deklarativen Vorgehensweise profitiert werden. Der weiten Verbreitung von regelbasierten Systemen ist es z.B. auch zu verdanken, dass sehr effektive Algorithmen zur Lösung dieses Problems entwickelt wurden.

2.5.3 Abgrenzung der deklarativen Programmierung

Betrachtet man die Trennung zwischen Deklaration und Interpretation bzw. zwischen *was* und *wie* als einziges Kriterium zur Definition der deklarativen Programmierung, finden sich leicht Vorgehensweisen zur Erreichung ähnlicher Ziele. Darum soll hier noch eine kurze Abgrenzung des Begriffes der deklarativen Programmierung vorgenommen werden.

Ein ähnlicher Weg wird bei der Generierung von Quellcode im Zusammenhang mit dem Ansatz der Modell-getriebenen Architektur (MDA)[24] beschritten. Hierbei werden die Anforderungen an ein System in einem Modell definiert, das unabhängig von der verwendeten Plattform, Programmiersprache und der genauen Art der Implementierung sein soll. Allerdings wird die in diesem Fall in einer der UML ähnlichen Syntax formulierte Deklaration nicht zur Laufzeit durch einen Interpreter verarbeitet, sondern es wird während der Entwicklung, d.h. spätestens zur Compile-Zeit, daraus Quellcode für die Zielplattform erzeugt. Diese Vorgehensweise wird per Definition nicht als deklarativ, sondern als generativ bezeichnet. Außerdem wird damit nicht die Unabhängigkeit von einer speziellen Implementierung sowie die Abstraktion von einer bestimmten Problemstellung adressiert, sondern auf einem höheren Level die Unabhängigkeit von Plattform und Programmiersprache erreicht.

Ein oft in den Vordergrund gestellter Vorteil der deklarativen Programmierung ist die Auslagerung bestimmter Daten aus dem Quellcode, z.B. in das Dateisystem oder in Datenbanken mit dem Ziel, Daten oder Parameter einer Anwendung ändern zu können, ohne die Anwendung neu kompilieren zu müssen.[25] Obwohl diese Eigenschaft auch den meisten deklarativen Systemen zugesprochen werden kann, muss dabei jedoch differenziert werden, was genau ausgelagert wird. Deklarative Systeme sorgen dafür, dass die Deklaration in eine meist aufwändigere Programmlogik umgesetzt wird. Am Beispiel von regelbasierten Systemen und SQL-Abfragen ist dies leicht zu erkennen. Im Gegensatz dazu können z.B. einfache Konfigurationsdateien noch nicht als deklarativ bezeichnet werden. Denn hier werden lediglich Platzhalter im Quellcode durch Parameterwerte ersetzt. Im Grenzbereich liegen komplexere Formen von Konfigurationsdateien wie z.B. die sog. Deployment-Deskriptoren für Application Server, bei denen das Verhalten von Containern innerhalb dieser Server entscheidend über die Deskriptoren verändert und angepasst wird.

2.5.4 Zusammenfassung

Die deklarative Programmierung trennt im Gegensatz zur prozeduralen Programmierung das *was* vom *wie*, also die gewünschte Lösung von der Art der Implementierung. Dazu wird im ersten Schritt die gewünschte Lösung in einer speziellen Syntax definiert, die ein Interpreter im zweiten Schritt liest und verarbeitet. Dadurch liegen die Anforderungen in einer Form vor, die frei von Implementierungsbefehlen zur Lösungen derjenigen Probleme ist, die der Interpreter löst. Somit sind die Deklarationen einfacher lesbar und ermöglichen eine gute Abstraktion von einer bestimmten Problemstellung.

Regelbasierte Systeme stellen eine spezielle Umsetzung des Prinzips der deklarativen Programmierung dar und abstrahieren dabei insbesondere von der Implementierung einer Logik, die in sehr starkem Maße von Entscheidungen dominiert wird.

2.6 Grundlagen von Expertensystemen

Expertensysteme stellen wie regelbasierte Systeme ebenfalls ein Teilgebiet der wissensbasierten Systeme und damit der künstlichen Intelligenz dar. Kenntnisse darüber sind im alltäglichen Einsatz von Rule-Engines in geschäftlichen Anwendungen nicht relevant. Zum Verständnis des in dieser Arbeit diskutierten Problems und zur fundierten Bewertung des gewählten Ansatzes sind einige Grundlagen jedoch hilfreich. Daher werden die wichtigsten Aspekte hier kurz erläutert.

[24] engl. *„model driven architecture“*, vgl. [Fra03].

[25] In der Praxis ist dieser Prozess meist aufwändiger als ein einfaches Neukompilieren. Meist sind auch anschließende Testläufe, Deployments auf die Zielplattform oder eine erneute Auslieferung an Kunden notwendig.

2.6.1 Definition

Der Begriff *Expertensystem* bezeichnet im Allgemeinen ein System, das es auf der Grundlage einer angelegten Wissensbasis ermöglicht, Probleme eines eng abgeschlossenen Fachgebietes zu lösen. Das Wissen wird in deklarativer Form hinterlegt, und es kommen Inferenzmechanismen zum Einsatz. Die bekanntesten Beispiele für Expertensysteme sind Diagnosesysteme, wie sie schon bei der Rückwärtsverkettung (2.2.3) angesprochen wurden.

Die Differenzierung von Expertensystemen und regelbasierten Systemen, insbesondere beim Einsatz in geschäftlichen Anwendungen, wird im Allgemeinen folgendermaßen vorgenommen:[26]

- Expertensysteme bilden eine sehr viel komplexere Regelbasis ab, die in sehr starkem Maße durch Rückwärtsverkettung ausgewertet wird. In geschäftlichen Anwendungen sind diese Regeln meist einfacher strukturiert, wie später noch zu sehen ist.
- Expertensysteme besitzen meist Mechanismen, um unsicheres Wissen verarbeiten zu können. Es gibt verschiedene Arten von unsicherem Wissen, auf das hier aber nicht näher eingegangen wird. Die Aussagen „Das Auto war ungefähr 50 km/h schnell“ und „Das Auto war zwischen 40 und 50 km/h schnell“ sind Beispiele für unsicheres Wissen. Geschäftliche Anwendungen benötigen diese Eigenschaft nicht, da hier meist nur klar definierte Werte und Zustände existieren bzw. sogar erwünscht sind. In Expertensystemen dagegen liefern oft Menschen Eingabewerte, die nicht exakt sind. Außerdem können manche Aussagen, wie z.B. die Diagnose von Krankheiten, nicht mit Sicherheit getroffen werden. Daher existieren in Expertensystemen meist Mechanismen, um diese Unsicherheit sinnvoll zu verarbeiten.

Aufgrund des sehr begrenzten Fachgebietes schaffen es diese Werkzeuge meist sehr gut, auch neue Sachverhalte aufgrund des bisherigen Wissens zu erschließen und vernünftige Ergebnisse zu liefern. Am besten funktioniert dieses Prinzip bei Problemen, die gut formalisiert werden können und deren notwendiges Wissen nahezu komplett erfasst werden kann. Beispiele dafür sind Spiele wie Schach oder Diagnosesysteme für Ärzte.

Weniger gut eignen sich viele Probleme des Alltags, die häufig ein großes Allgemeinwissen benötigen, das nur schwer komplett erfasst werden kann. Häufig finden sich auch Probleme, die zwar durch Fachwissen recht gut gelöst werden können, aber in vielen Fällen weiter gehendes Allgemeinwissen benötigen, um auch die restlichen Fälle lösen zu können. Dabei scheitern diese Systeme in erster Linie nicht daran, dass das benötigte Allgemeinwissen nicht erfasst und eingebunden werden könnte, sondern dass die Menge an benötigtem Allgemeinwissen schlicht zu groß ist, um sie in der Praxis erfassen zu können. Dieses als *„Kiff and Plateau - Effekt“* bezeichnete Problem des fehlenden Allgemeinwissens von Expertensystemen wird im nächsten Abschnitt erläutert.

Aufgrund des teilweise nicht lösbaren Seiteneffektes dieses Problems werden Expertensysteme häufig so ausgelegt, dass nur Probleme gelöst werden können, deren Problembereich innerhalb der Regeln erfasst ist. Dabei wird oft die Wissensbasis während des Betriebs, z.B. bei Bekanntwerden neuer Informationen, erweitert, womit das Expertensystem insgesamt über die Zeit mächtiger im Bezug auf die Problemlösungsfähigkeiten wird. Die Annahme, auf der diese Vorgehensweise basiert, wird als *„Closed World Assumption“* bezeichnet und im übernächsten Abschnitt erläutert.

Es wird betont, dass dieser Abschnitt kein Grundlagenwissen zu Expertensystemen vermitteln soll, sondern nur zum vollständigen Verständnis der Argumentation am Ende dieser Arbeit benötigt wird. Daher sind die Ausführungen kurz gehalten.

[26] Siehe [vH02].

2.6.2 Kliff and Plateau-Effekt

Dieses Problem des nur schwer erfassbaren Allgemeinwissens wird in der Literatur als *Kliff and Plateau-Effekt*[27] bezeichnet. Bei einem Expertensystem ist es möglich, in einem eng begrenzten Gebiet eine hohe Kompetenz (Plateau) zu erzeugen, die bei Verlassen dieses Gebietes steil abfällt (Kliff). Bei einem menschlichen Experten dagegen ist das Expertenwissen von mehreren Schichten immer allgemeiner werdenden Wissens (das Allgemeinwissen) umgeben, weshalb es beim Verlassen des Gebietes eher flach abfällt. Dadurch ist ein Mensch in der Lage, auch Fragen zu beantworten, die er nicht direkt weiß, indem er intuitiv sein Allgemeinwissen einsetzt und Transferleistungen erbringt. So hat z.B. ein Informatiker ein enormes Wissen in seinem Fachgebiet, ein immer noch fundiertes Wissen in allgemeinen Gebieten der Informatik, ein recht gutes Wissen in allgemeinen Fragen rund um den Computer und im Allgemeinen auch ein gutes modernes Technikverständnis. Diese verschiedenen Schichten immer allgemeiner werdenden Wissens ermöglichen es ihm, Situationen zu meistern, die auch nicht direkt in sein Fachgebiet fallen.

Dieser Effekt beschreibt die Problematik in der künstlichen Intelligenz, Expertensysteme zu entwickeln, die ein breites bzw. mehrere verschiedene Wissensgebiete bedienen können. Der einzige bekannte Ansatz besteht bisher darin, herkömmlichen Expertensystemen eine sehr breite Wissensbasis zur Verfügung zu stellen. Aufgrund des hohen Aufwandes in der Erfassung sowie in der Auswertung der Daten ist dieser Ansatz jedoch nur relativ begrenzt tauglich.

Bei den meisten Expertensystemen, wie auch bei dem hier beschriebenen Beispiel einer virtuellen Verkaufsberatung, ist es jedoch nicht notwendig, dieses Allgemeinwissen zu erfassen. Meist sind nur wenige bzw. einzelne Fakten des Allgemeinwissens notwendig, die dann als Teil der normalen Wissensbasis sozusagen „mit“-erfasst werden.

2.6.3 Vollständigkeit des Wissens - Closed World Assumption

Der aus der Theorie der Wissensrepräsentation stammende Begriff *Closed World Assumption* (übersetzt etwa „Annahme einer abgeschlossenen Welt“) besagt, dass alles falsch ist, was nicht explizit als wahr definiert ist. Das Gegenteil dazu ist die *Open World Assumption*, die nicht definiertes Wissens, wie auch im Alltag üblich, als unbekannt ansieht. Ähnlich dem Kliff and Plateau-Effekt hat auch diese Annahme Konsequenzen auf den Umgang mit nicht bekanntem Wissen.

2.7 Zusammenfassung

Regelbasierte Systeme stellen eine wirkungsvolle Methode dar, um Informationen zu verarbeiten, die in Form von Regeln nach der gegebenen Definition formuliert werden können. Durch die deklarative Vorgehensweise kann die mit Regeln definierte Geschäftslogik sauber von der eigentlichen Implementierung der Ausführung getrennt werden. Anzumerken ist dabei aber, dass natürlich auch weiterhin Code geschrieben werden muss, nämlich der Quellcode, der auch etwas innerhalb der Anwendung tut. Die Rule-Engine als Interpreter des deklarativen Mechanismus' kümmert sich allerdings um viele Probleme, die bei anderen Ansätzen auftauchen würden. So sind die Probleme der Konflikt-Lösung keine Regel-spezifischen Probleme, sondern tauchen allgemein auf, wenn Code programmiert wird, der auf vielen Entscheidungen basiert. In normalem Quellcode programmiert wäre es Aufgabe des Entwicklers, durch geeignete Verschachtelungen die Konflikt-Lösung schon während der Programmierung festzulegen.

[27] Zu übersetzen etwa als „Klippe und Hochebene“.

Die Rule-Engine dagegen handhabt dieses Problem dynamisch, womit es recht einfach und bei geeigneten Voraussetzungen auch noch zur Laufzeit geändert werden kann. Weiterhin ermöglicht der Rete-Algorithmus eine effiziente Verarbeitung von Bedingungen und Aktionen, was bei großen Systemen zu erheblichen Performance-Gewinnen führen kann.

Dadurch, dass Regeln völlig unabhängig voneinander erfasst werden können, wird eine einfache und schnelle Erfassung auch sehr vieler Regeln ermöglicht. Dieser Effekt wird nochmals durch die Tatsache gestärkt, dass auch die meisten in der Praxis vorliegenden Regeln tatsächlich voneinander unabhängig sind. Die Problemstellung der Verknüpfung bzw. Verschachtelung der Regeln wird weg vom Quellcode (und damit weg vom Entwickler) hin zur Logik der Rule-Engine verlagert.

Der deklarative Ansatz, der es ermöglicht, die Regeln separat vom Quellcode zu erfassen, ermöglicht es darüber hinaus, die Regeln zur Laufzeit zu ändern und so schnelle und kostengünstige Änderungen der Geschäftslogik durchzuführen.

Da die Interpretation der Regeln getrennt von deren Deklaration erfolgt, besteht darin eine Möglichkeit, die Regeln für verschiedene Zwecke wieder zu verwenden. Allerdings besteht dabei in der Praxis das Problem, dass die Syntax jedes Interpreters verschieden ist, und dadurch recht früh eine Festlegung auf einen bestimmten Interpreter erfolgen muss.

Außerdem ist zu beachten, dass die Aktionen, die innerhalb der Regeln ausgeführt werden, meist sehr abhängig von der Plattform sind bzw. direkt in Anwendungen eingreifen, weshalb es nicht ohne weiteres möglich ist, Regeln für komplett verschiedene Zwecke zu verwenden.

Kapitel 3

Rule-Engines

Im vorherigen Kapitel wurden die Grundlagen von Regeln und regelbasierten Systemen erläutert. Es wurde ebenfalls erläutert, dass für die Verarbeitung von Regeln ein Interpreter benötigt wird, der die Regeln einlesen und durch die Mechanismen der Inferenz und der Konflikt-Lösung benötigte Informationen ableiten kann. In den Anfängen der regelbasierten Systeme sowie der Expertensysteme waren diese Interpreter Programme, die auf die Lösung eines bestimmten Problems spezialisiert waren. In den letzten Jahren wurden nun Interpreter entwickelt, die universell verwendbar sind, die sog. *„Rule-Engines“*.

In diesem Kapitel wird der Begriff Rule-Engine und dessen Herkunft erläutert und es werden die einzelnen Bestandteile erklärt. Anschließend wird ein kurzer Überblick über die Kategorisierung und über erhältliche Produkte am Markt gegeben. Ziel dieses Kapitels ist es, die Grundlagen für die Auswahl und den Einsatz einer Rule-Engine in der Praxis zu geben. Eine vollständige Übersicht aller hier genannten Produkte und Standards einschließlich der jeweiligen URL befindet sich im Anhang A.

3.1 Historie und Definition

Rule-Engines haben ihren Ursprung in den Anfängen der künstlichen Intelligenz und der wissensbasierten Systeme. Dort wurden die Grundlagen zu regelbasierten Wissensrepräsentationsformalismen, zu Inferenzmaschinen und zur Wissensverarbeitung gelegt.[1] Lange Zeit wurden diese Systeme nur in Forschung und Lehre oder in hoch spezialisierten Expertensystemen eingesetzt. Das Expertensystem *MYCIN*[2] z.B. wurde 1972 von der Stanford University entwickelt und zur Diagnose von Infektionskrankheiten und deren Therapie durch Antibiotika eingesetzt. Obwohl das System sehr hohe Trefferquoten lieferte und noch heute als Meilenstein in der Entwicklung der Expertensysteme gilt, war es aber spezialisiert auf dieses eine Gebiet.

Den nächsten Schritt markierten Werkzeuge, die universell zur Entwicklung von Produktions- und Expertensystemen eingesetzt werden konnten. Diese Systeme boten universelle Regelsprachen, mit denen Expertensysteme für unterschiedlichste Anwendungsbereiche realisiert werden konnten. Der erste bekannte Vertreter dieser Art war CLIPS[3], das 1984 am Johnson Space Center der NASA entwickelt wurde und 1986 öffentlich zugänglich wurde. CLIPS kann als Vorläufer heutiger Rule-Engines angesehen werden, nicht zuletzt da aus diesem System auch Jess (siehe Abschnitt 3.4), eine der heute erfolgreichsten Rule-Engines hervorgegangen ist.

[1] Die Grundlagen zu wissensbasierten Systemen beziehen sich auf [GG03], [Bar82], [AR01].
[2] Siehe [E.H77].
[3] Abk. für *C Language Integrated Production System*.

Eine Rule-Engine kennzeichnet sich also hauptsächlich durch die Eigenschaft aus, freie Definitionen von Regeln und deren Anwendung auf beliebiges Wissen zu ermöglichen. Die Art und Weise, wie eine solche Verarbeitung stattfindet, wie Regeln definiert werden, wie sie auf eine Datenbasis angewendet werden können, und wie ein Austausch der Ergebnisse mit anderen Anwendungen oder Anwendungsteilen erfolgen kann, wird in den folgenden Abschnitten und Kapiteln erläutert.

In den letzten Jahren begannen diese Systeme nun, sich in mittleren und großen Softwaresystemen und industriellen Projekten durchzusetzen. Das hat mehrere Gründe:

- Mit Java und .NET stehen zwei Plattformen zur Verfügung, die sich weltweit als Standard für große und sehr große Softwareprojekte durchgesetzt haben. Ihre objektorientierten und komponentenbasierten Prinzipien erlauben eine komfortable und effiziente Einbindung von regelverarbeitenden Komponenten.

- In den 90er-Jahren standen mit objektorientierten Programmiersprachen, der UML, mit MDA und ähnlichem Entwicklungen im Vordergrund, die technische, organisatorische oder entwicklungsspezifische Probleme lösten. Auf der daraus entstandenen, soliden Grundlage zur effizienten Entwicklung großer Projekte rückten in den letzten Jahren nun fachliche Probleme in den Mittelpunkt, wie z.B. die Definition und Implementierung von Geschäftsprozessen und deren Integration in geschäftliche Anwendungen bzw. die Trennung von Geschäftsprozessen und Quellcode. Dabei bieten Rule-Engines einen sehr guten Ansatz, Logik getrennt vom Code, deklarativ und in verständlicher Form zu definieren und applikationsübergreifend zu verwenden.

- Die Performance heutiger Computersysteme erlaubt es, auch größere Regelsets schnell und effizient zu verarbeiten.

In diesem Trend ist auch der Begriff „Rule-Engine“ entstanden, der zusammen mit dem Begriff „Business Rules“ einen aussagekräftigen und werbewirksamen Begriff prägen sollte, der die gesamte Thematik abdeckt. Darunter sollten regelbasierte Systeme, die deklarative Programmierung und vor allem deren Einsatz in kommerziellen und geschäftlichen Anwendungen fallen. Selbstverständlich lassen sich diese Techniken aber auch in nicht-kommerziellen und nicht-geschäftlichen Anwendungen einsetzen, es ist aber sehr wahrscheinlich, dass diese Begriffsprägung auch einen kleinen Ausschlag gab, dass „Rule-Engines“ zu einem anerkannten Teilgebiet in der heutigen Welt der Softwareentwicklung wurden.

> „The term *business rules* gives the concept a very marketable tag ... *declarative programming* just won't sell!“[4]

Wichtig ist auch, dass eine Rule-Engine in der allgemeinen (und in dieser Arbeit verwendeten) Definition ein Produkt[5] darstellt, das die oben genannten und in den weiteren Kapiteln erläuterten Eigenschaften von Haus aus mitbringt. Dadurch muss sich der Anwender eines solchen Produktes keinerlei Gedanken um die interne Funktionsweise machen und kann sich vollkommen auf Aufgaben wie die Modellierung des Wissens und die Formulierung der Regeln konzentrieren.[6] Auch wenn die Entwicklung einer Rule-Engine durchaus eine in-

[4] Zitat von Paul Irvine, entnommen aus [Dat00], S.19.

[5] In der Praxis handelt es sich bei diesen Produkten meist um Klassenbibliotheken oder Frameworks. Nur wenige kommerzielle Rule-Engines bieten darüber hinaus Tools etc. an, die es rechtfertigen, als vollständiges „Produkt“ im allgemeinen Sprachgebrauch bezeichnet zu werden. Mehr dazu in der Marktübersicht in Abschnitt 3.4.

[6] Die Kenntnisse um diese Interna sind aber zur effektiven Arbeit mit Rule-Engines dringend empfohlen. Deshalb werden sie in den folgenden Kapiteln erläutert.

teressante Herausforderung darstellen würde, ist sie nicht Teil dieser Arbeit.[7] Sie richtet sich an den Entwickler, der eine Rule-Engine zur Integration in eigene Projekte einsetzen möchte.

3.2 Bestandteile einer Rule-Engine

Mittlerweile existieren viele verschiedene Rule-Engine-Implementierungen, die zum Teil auch unterschiedliche Ansätze verfolgen. Die Unterschiede liegen z.B. in den möglichen Inferenzstrategien (siehe Abschnitt 2.2) oder in den Algorithmen zur Konfliktlösung. Dennoch ähneln sich fast alle Produkte in ihrer prinzipiellen Funktionsweise. Erfreulicherweise nutzen auch die meisten Produkte gleiche oder ähnliche Begriffe zur Bezeichnung der jeweiligen Bestandteile. Dies erleichtert sowohl die Einarbeitung als auch den Umstieg zwischen verschiedenen Produkten. Alle Rule-Engines:

- handeln mit Fakten, auf die Regeln angewendet werden.
- besitzen einen virtuellen Container, in dem sich diese Fakten befinden, den sog. WorkingMemory.
- führen Aktionen aus, wenn die Bedingungen einer Regel eintreffen.
- besitzen einen Regelinterpreter, der die Evaluierung der Regeln sowie die Ausführung der Aktionen vornimmt.

In den folgenden Abschnitten werden diese Bestandteile sowie deren Zusammenspiel erläutert.

3.2.1 Regeln

Wie in Kapitel 2 erläutert, stellen Regeln (engl. *rules*[8]) eine Kombination aus Bedingungen (Prämissen) und Konsequenzen (Konklusionen) dar. Sobald alle Bedingungen der Regeln eintreffen, wird die Konsequenz der Regeln (die Aktion) ausgeführt. Regeln gleichen in ihrer Struktur einer `if..then` - Abfrage aus der Programmierung:

```
WENN
  Bedingung 1 wahr ist
  UND
  Bedingung 2 wahr ist

DANN
  Führe Aktion 1 aus
  Führe Aktion 2 aus
```

Beispiele für Regeln aus der Praxis

- *„Wenn der Gesamtpreis der Bestellung größer als 400 Euro ist, dann gewähre 10% Rabatt!“*

[7] Interessierte seien z.B. an das Open-Source Projekt der Rule-Engine Drools verwiesen: *http://www.drools.org*.

[8] Vor allem in diesem Abschnitt werden wichtige Begriffe zusammen mit ihren englischen Übersetzungen eingeführt, da sie für die Arbeit mit der meist durchweg englischsprachigen Software wichtig sind.

- *„Wenn der Kunde in den letzten zwölf Monaten mehr als 1000 Euro Umsatz gemacht hat und noch kein Premium-Kunde ist, dann nimm ihn in die Kunden-Kategorie 'Premium-Kunde' auf!“*

Regeln werden in einer speziellen Notation erfasst, die die Rule-Engine lesen und verarbeiten kann. Darauf wird in Kapitel 4 näher eingegangen.

Anmerkungen

- Wenn alle Prämissen einer Regel eintreffen, dann wird die Konklusion dieser Regel ausgeführt. Es wird dann gesagt, dass die Regel „feuert“. Da sich dieser Begriff eingebürgert hat, wird er auch hier oft verwendet. Er ist sogar in die Syntax der Rule-Engines eingeflossen, wie später noch zu sehen ist.
- Da in der vereinfachten Schreibweise von Regeln die Prämissen links und die Konklusionen rechts stehen, wird bei beiden auch von der *left-hand-side (LHS)* und der *right-hand-side (RHS)* gesprochen.
- Im Umfeld geschäftlicher Anwendungen werden Regeln auch *Geschäftsregeln* (engl. *business rules*) genannt. In den folgenden Kapiteln wird dieser Begriff auch häufiger verwendet, da er eine gute Differenzierung zur allgemeinen Bedeutung des Wortes „Regel“ ermöglicht.
- Eine Menge von Regeln in einer Anwendung wird *Regelwerk* oder *Regelset* (engl. *rule set*) genannt. In der Praxis existieren häufig mehrere Regelsets, die logisch gegliedert oder gruppiert sind. Dann besteht die Möglichkeit, passend zum jeweiligen Kontext unterschiedliche Regelsets einzubinden. Die Menge der im aktuellen Kontext eingebundenen und damit aktiven Regeln wird *Regelbasis* (engl. *rule base*) genannt.

3.2.2 Fakten

Fakten (engl. *facts*) stellen die Datengrundlage für die Rule-Engine dar. Die Fakten werden bei der Erstellung der Regeln genutzt, um Prämissen zu erstellen. D.h. die einzelnen Prämissen sind jeweils Abfragen auf einen einzelnen Fakt:

„Wenn der Gesamtpreis der Bestellung größer als 400 Euro ist, dann gewähre 10% Rabatt!“

Hier stellt der *Gesamtpreis* den Fakt dar, auf dessen Grundlage die Prämisse *„Ist der Gesamtpreis größer als 400 Euro?“* erstellt wird. Fakten nehmen zur Laufzeit unterschiedliche Werte an, so wie sich für jeden Kunden und für jede neue Bestellung wieder ein anderer Gesamtpreis ergeben wird. Fakten stammen in der Praxis meist aus zwei Quellen:

1. **Daten der Geschäftslogik:** Die Applikation, in die die Rule-Engine eingebettet wird, hält gewisse Daten, die zur Abarbeitung der Geschäftslogik notwendig ist. So hat z.B. jedes Warenhaus einen gewissen Kundenstamm mit den persönlichen Daten der Kunden, einen Artikelstamm mit Daten zu den Produkten wie Bezeichnung und Preis und ein Bestellwesen, in dem Kundenbestellungen erfasst und bearbeitet werden. Das oben genannte Beispiel würde demnach auf die Bestelldaten der Kunden angewendet werden und in diesem speziellen Fall das Attribut *Gesamtpreis* der Bestellung prüfen. In heutigen objekt-orientierten Systemen werden diese Informationen meist in Form von Objekten zur Verfügung gestellt, so dass diese Regel auf alle Objekte des Typs *Bestellung* und deren Attribut *Gesamtpreis* angewendet wird.

2. **Ergebnisse anderer Regeln:** Regeln wiederum können auch verwendet werden, um neue Fakten zu erstellen bzw. existierende Fakten zu manipulieren. Falls im oben genannten Beispiel der Gesamtpreis 400 Euro übersteigt, werden dem Kunden 10% Rabatt gewährt und somit wird natürlich auch der Gesamtpreis neu berechnet. In diesem Fall wird das konkrete Geschäftsobjekt geändert, so dass die Applikation im weiteren Verlauf (z.B. beim Drucken der Rechnung oder beim Verbuchen der Bestellung in der Finanzbuchhaltung) die aktuellen Daten erhält.

Anmerkungen

- Die Menge aller Fakten wird auch als *Faktenbasis* (engl. *fact base*) bezeichnet.
- Fakten und Regeln bilden zusammen die sog. *Wissensbasis*.

3.2.3 Aktionen

„Wenn der Gesamtpreis der Bestellung größer als 400 Euro ist, dann gewähre 10% Rabatt!“

Die Konklusion dieser Regel lautet *„Gewähre dem Kunden 10% Rabatt!“*. Diese Konklusion berechnet im einfachsten Fall nur den Gesamtpreis der Bestellung neu, nämlich auf einen 10% niedrigeren Gesamtpreis als vorher, und erzeugt damit einen neuen Fakt. Nun könnte eine Regel aber auch komplexere Operationen ausführen, wie die nächste Regel zeigen wird:

„Wenn der Kunde in den letzten zwölf Monaten mehr als 1000 Euro Umsatz gemacht hat und noch kein Premium-Kunde ist, dann nimm ihn in die Kunden-Kategorie 'Premium-Kunde' auf!“

Die Aufnahme des Kunden in die Kategorie 'Premium-Kunde' ist nicht durch ein einfaches Ändern eines Wertes getan. In einer objektorientierten Anwendung würde das Geschäftsobjekt „Kunde“ vielleicht eine Methode „aendereKundenKategorie(String nameDerKategorie)“ bereitstellen, die das Ändern der Kundenkategorie durchführt. Da Konklusionen in geschäftlichen Anwendungen sehr häufig Funktionen der Geschäftslogik aufrufen oder komplexe Geschäftsprozesse steuern, wird meist der Begriff *Aktion* bevorzugt.

Zum Starten dieser Aktionen bieten Rule-Engines die Möglichkeit, direkt oder über Schnittstellen Teile der Geschäftslogik anzusprechen. Die in Java implementierte Open-Source Rule-Engine *Drools* (siehe Abschnitt 3.4) bietet z.B. die Möglichkeit, innerhalb der Definition der Regel direkt den auszuführenden Java-Code zu hinterlegen bzw. auch schon in den Prämissen auf Java-Code zuzugreifen. Die praktische Anwendung solcher Mechanismen ist innerhalb der Beispielanwendung ab Kapitel 6 beschrieben. Die oben genannte Regel würde dann im Pseudo-Code folgendermaßen aussehen:

```
WENN
  Kunde.umsatzLetztesJahr() > 1000
  UND
  Kunde.getKundenKategorie() != "PREMIUM"
DANN
  Kunde.aendereKundenKategorie("PREMIUM")
```

3.2.4 Working Memory

Der sog. *Working Memory*[9] ist der Container, in dem sich die Fakten befinden, auf die das Regelset angewendet werden soll. Existierende Fakten wie z.B. die oben angesprochenen Geschäftsobjekte werden dem Working Memory explizit mitgeteilt. Beim nächsten Evaluierungsdurchlauf werden alle Regeln auf alle Fakten und damit auch auf diese Objekte angewandt, die sich im Working Memory befinden.

Beim Einsatz einer Rule-Engine ist der Working Memory ein sehr wichtiger Bestandteil, da auf ihn viele wichtige Operationen angewendet werden:

- Fakten hinzufügen.
- Fakten entfernen.
- Fakten modifizieren.
- Regeln feuern.

Anmerkungen

- Die Menge an Fakten wurde oben definiert als die *Faktenbasis*. Da die Regeln jedoch nur auf diejenigen Fakten angewendet werden, die dem Working Memory explizit mitgeteilt wurden (also evtl. nur auf eine Teilmenge der gesamten Fakten), wird die Menge der Fakten oft auch mit dem Working Memory gleich gesetzt. Dies hat auf der einen Seite durchaus seine Berechtigung, da die im Working Memory enthaltenen Fakten eben die relevanten Fakten sind. Es führt aber im Kontext der Beschreibung der einzelnen Bestandteile leicht zu Missverständnissen. Daher werden die beiden Begriffe in dieser Arbeit konsequent in der bisher eingeführten, getrennten Weise verwendet.

3.2.5 Regelinterpreter (Inferenzmaschine)

Der Regelinterpreter ist der Kern der Rule-Engine. Zur Laufzeit muss ihm die komplette Wissensbasis, d.h. die Fakten in Form des Working Memory sowie die Regeln in Form des Rulesets, zur Verfügung gestellt werden. Daraufhin ist der Regelinterpreter in der Lage, herauszufinden, welche Regeln aufgrund der gegeben Fakten und Bedingungen feuern könnten, und er kann die entsprechenden Aktionen starten.

Die Literatur[10] teilt den hier als „Regelinterpreter“ bezeichneten Bestandteil häufig nochmals in weitere Bestandteile auf. Bei der praktischen Arbeit mit Rule-Engines ist diese Aufteilung bzw. die Kenntnis dieser Teilkomponenten nicht unbedingt notwendig. Deshalb wurden sie in dieser Arbeit auch nicht in die Hauptbestandteile einer Rule-Engine aufgenommen. Da sie aber bei intensiverer Arbeit mit regelbasierten Systemen bzw. bei der Studie theoretischer Grundlagenliteratur dennoch relevant werden können, sollten sie hier zumindest kurz erwähnt sein. Außerdem schließt sich mit dieser Sichtweise ein Kreis mit den Grundlagen zu Pattern-Matching und Konfliktlösung aus dem vorherigen Kapitel.

- **Pattern-Matcher:** Diese Komponente prüft die Bedingungen der Regeln und erstellt eine Liste aller Regeln, die in diesem Durchlauf feuern könnten, die sog. Konflikt-Menge.

[9] Dieser Ausdruck ist auch im Deutschen zu einem stehenden Begriff geworden. Deshalb und in Ermangelung einer vernünftigen Übersetzung wird er in dieser Arbeit so verwendet.

[10] Stellvertretend sei hier z.B. auf [FH03] verwiesen.

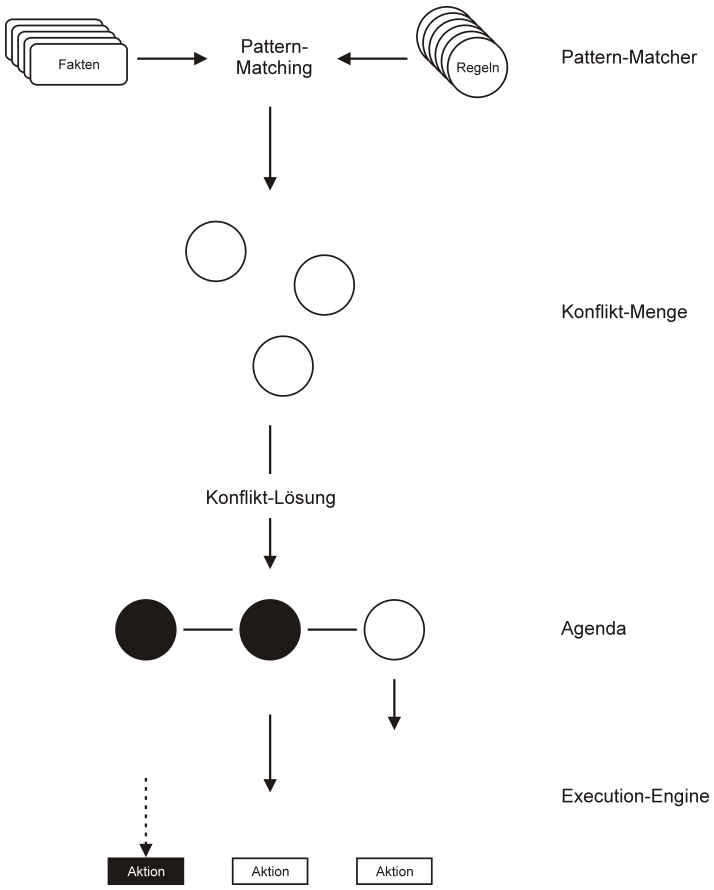


Abbildung 3.1: Die Bestanteile des Regelinterpreters

- **Agenda:** Anhand der verschiedenen Konfliktlösungsstrategien wie Liste der Regeln in der Reihenfolge sortiert, in der die Aktionen ausgeführt werden sollen. Diese sortierte Liste wird als *Agenda* bezeichnet.
- **Execution Engine:** Wenn schließlich die Aktionen der Regeln ausgeführt werden, sorgt die sog. *Execution Engine* dafür, dass z.B. Werte geändert, Methoden aufgerufen oder Objekte erzeugt werden. Sie wird deshalb als eigenständiger Bestandteil betrachtet, da sie Schnittstellen zu der Plattform bereitstellen muss, in der die Rule-Engine eingebettet wird.

3.2.6 Zusammenspiel dieser Komponenten

Das Zusammenspiel dieser Komponenten ist in Abbildung 3.2 dargestellt. Dieses Verhalten ist bei fast allen Rule-Engines ähnlich und wird hier erläutert. Der grundsätzliche Ablauf beim Einsatz einer Rule-Engine ist folgendermaßen:

1. Regeln laden (meist im Dateisystem oder in einer Datenbank abgelegt).
2. Neuen Working Memory erzeugen.
3. Fakten dem Working Memory hinzufügen.
4. Rule-Engine starten.

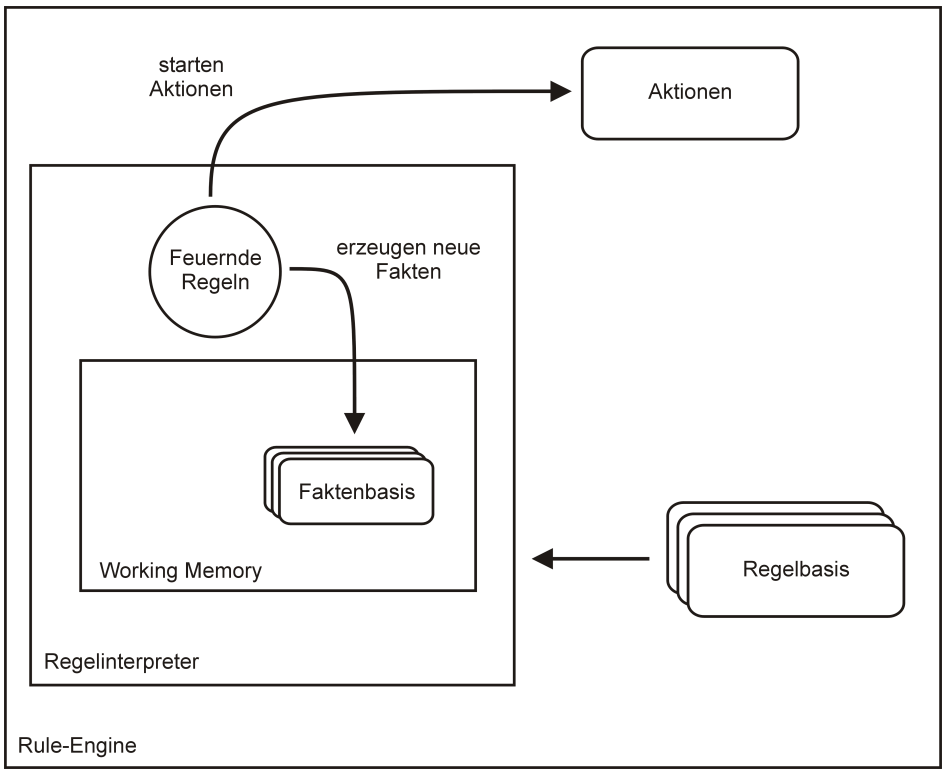


Abbildung 3.2: Bestandteile einer Rule-Engine

Angelehnt an die Anwendungsschnittstelle der Rule-Engine *Drools* ergibt sich damit der folgende Pseudo-Code in Java. Hierbei werden dem Working Memory zuerst beliebige Daten übergeben, die Rule Engine wird dann veranlasst, alle Regeln zu feuern, deren Vorraussetzungen gegeben sind, danach wird ein Fakt verändert, und die Rule Engine wird zu einem erneuten Evaluierungsdurchlauf aufgefordert. Die während des Evaluierungslaufes evtl. veränderten Objekte `object1` und `object2` können danach im Java-Code mit den neuen Daten direkt weiter verwendet werden.

```
RuleEngine ruleEngine = new RuleEngine();
RuleSet ruleSet = RuleEngine.createNewRuleSetFromFile("rules.xml");
WorkingMemory workingMemory = ruleSet.createNewWorkingMemory();


workingMemory.assert( object1 );
workingMemory.assert( object2 );

workingMemory.fireAllRules();


workingMemory.modifyObject( object1 );

workingMemory.fireAllRules();


System.out.println( "Verändertes Objekt: " + object1 );
```

3.3 Anforderungen an eine Rule-Engine in der Praxis

Die Anforderungen an ein bestimmtes Rule-Engine-Produkt sind in der Praxis sehr vielfältig und komplex. Einige dieser Anforderungen gleichen denen bei der Entscheidung über den Einsatz Softwarekomponenten im Allgemeinen, diese sind:

- Die eigene Systemlandschaft und Infrastruktur.
- Know-How im Unternehmen.
- Kosten.
- Leistungsumfang bzw. Erfüllug der eigenen Anforderungen.

Außerdem müssen noch einige Punkte beachtet werden, die Rules-Engines sowie deren Markt im Speziellen charakterisieren:

- Unterstützung der benötigten Inferenzstrategien.
- Umfang des Rule-Engine-Paketes wie z.B. Tools zur Pflege und Überwachung der Regeln bzw. die Verfügbarkeit von Tools für das spezielle Produkt.

Wird die Wahl der Infrastruktur auf den Bereich der beiden populären Plattformen Java und .NET beschränkt, ist die Auswahl an Produkten immer noch komfortabel groß. Eine weitere Eingrenzung lässt sich durch die unterstützten Inferenzstrategien erreichen. Rule-Engines, die nur eine Rückwärtsverkettung unterstützen, eignen sich hauptsächlich für Expertensysteme. Vorwärtsverkettende Systeme sind in der Praxis wesentlich flexibler und erlauben es, das formulierte Regelwissen für viele verschiedene Anwendungsbereiche einzusetzen. Diese Möglichkeiten sollten auf jeden Fall in Betracht gezogen werden. Eine letzte, gröbere Eingrenzung wird dann häufig nach dem Preis erreicht. Hier sind sowohl Open-Source Projekte (wie *Drools*) verfügbar, als auch hochwertige und dementsprechend teure Systeme wie JRules. Diese Systeme bieten aber auch vielfältige Tools für wichtige Aufgaben, wie die Pflege und Überwachung der Regeln. Die wichtigsten Systeme werden im folgenden Abschnitt vorgestellt.

3.4 Marktübersicht

Zu Gunsten der Übersichtlichkeit wurde hier eine Einteilung nach den unterstützten Inferenzstrategien gewählt, da diese Eigenschaft einer Rule-Engine ein eindeutiges K.O.-Kriterium für den jeweiligen Einsatzzweck bedeutet. Der zweite wichtige Faktor des Preises muss dagegen vom jeweiligen Unternehmen bewertet werden und wird hier neutral behandelt.

In Abschnitt 4.2.3 werden außerdem die Regelnotationen der Rule-Engines JRules, Jess und *Drools* näher erläutert.

3.4.1 Vorwärtsverkettende Systeme

Die teuerste und leistungsfähigste Rule-Engine am Markt ist **JRules** der Firma Ilog. Dieses System bietet Schnittstellen für Java und .NET. Im Lieferumfang enthalten sind mächtige Werkzeuge zur Erstellung und Pflege von Regeln, die es einfach und effizient erlauben, Regeln in fast natürlichsprachlichem Englisch zu erfassen. Auch graphische Darstellungen des Regel-Auswertungsablaufes oder die Pflege in alternativen Formen wie Entscheidungstabellen oder

-bäumen wird unterstützt. Gerade diese Tools erlauben eine schnelle und kostengünstige Einführung der Rule-Engine im Unternehmen. Auch die Kosten für eine eigene Anpassung sind aufgrund der vielfältigen Möglichkeiten sehr gering. Der Anschaffungspreis liegt allerdings bei knapp 50000 Euro und steigt nochmals mit der Art des Einsatzes (Betrieb im eigenen Unternehmen oder Integration in verkaufte Produkte) und der Anzahl der Mitarbeiter bzw. der CPUs, auf denen das System installiert ist. Als bekanntestes Unternehmen setzt Ebay diese Rule-Engine seit Jahren erfolgreich ein.

Eine Alternative stellt das Open-Source Projekt **Drools** dar, das mittlerweile in einer stabilen und ausgereiften Version zur Verfügung steht. Dieses System ist eigentlich eine Klassenbibliothek in Java, die sich einfach und problemlos in eigene Java-Anwendungen integrieren lässt. Außer den Kernelementen der Inferenzmaschine sind aber keine weiteren Werkzeuge enthalten. Am Markt sind auch noch keine ausgereiften Werkzeuge erhältlich. Drools bietet die Möglichkeit, Regeln in einem proprietären XML-Format zu erfassen oder mittels XSL-Stylesheet eigene Fachsprachen zu entwickeln. Drools besitzt mittlerweile eine ausreichend große Community, so dass Support durch die Entwickler oder durch andere Anwender sehr schnell verfügbar ist. Das Projekt hat mittlerweile auch eine Größe erreicht, bei der an der Beständigkeit nicht mehr ernsthaft gezweifelt werden muss. So bietet Drools eine der besten Alternativen am Markt, wenn der Preis eine Rolle spielt und ein vorwärtsverkettendes System benötigt wird. Kurz vor Drucklegung dieser Arbeit hat sich das Drools-Projekt dem Unternehmen *„JBoss Inc.“* angeschlossen, unter anderem Entwickler des populären gleichnamigen Application Servers. Gerade bei Open-Source-Projekten fördern solche Schritte Vertrauen und die Bereitschaft zur Nutzung.

3.4.2 Rückwärtsverkettende Systeme

Das Open-Source Projekt **Mandarax** ist einer der bekanntesten Vertreter rein rückwärtsverkettender Systeme. Es ist in Form einer Klassenbibliothek in Java erhältlich. Mandarax unterstützt als erste Rule-Engine am Markt den vielleicht zukünftigen Standard *RuleML* (siehe Abschnitt 3.4.5). Es kann für alle Zwecke kostenfrei verwendet werden.

3.4.3 Hybride Systeme

Jess ist eine der bekanntesten Rule-Engines am Markt. Das Produkt ist für Forschung und Lehre kostenlos und hat für kommerzielle Zwecke einen relativ niedrigen Einstiegspreis von ca. 5000 US-$. Es arbeitet grundsätzlich als vorwärtsverkettendes System, beherrscht aber, wenn benötigt, auch die Rückwärtsverkettung. Jess bietet eine Schnittstelle für Java. Es ist insgesamt sehr leistungsfähig, geht bei der Regelbeschreibung aber eigene Wege: Zur Deklaration der Regeln wird die an die Syntax von Lisp angelehnte Beschreibungssprache *„CLIPS“* verwendet, die im Vergleich zu herkömmlichen Programmiersprachen bzw. Markup-Sprachen wie XML sehr gewöhnungsbedürftig ist, was die Einführung im Unternehmen und die Akteptanz deutlich erschweren könnte. Ansonsten wird Jess aber mit Erfolg in vielen Projekten eingesetzt und genießt in Fachkreisen ein sehr hohes Ansehen.

3.4.4 Tools

Das Werkzeug **Visual Rules** ist ein visuelles Rule-Modellierungstool der deutschen Firma *innovations*. Es ist als Plug-In z.B für die Entwicklungsplattform Eclipse und für die Programmiersprachen Java und Cobol erhältlich. Visual Rules ist keine Rule-Engine im eigentlichen Sinne, sondern ermöglicht es, logische Abläufe und Zusammenhänge grafisch zu modellieren und daraus Quellcode für Java und Cobol zu erzeugen. Somit ist zur Laufzeit keine zusätzliche

Software nötig und der Code kann z.B. direkt in eine Client-Anwendung integriert werden. Alternativ ist auch die Bereitstellung des generierten Codes als Service[11] in einem eigenen mitgelieferten Regel-Server möglich. Damit wird die zentrale Pflege und ein Hot-Deployment der Regeln möglich. Obwohl Visual Rules einen etwas anderen Ansatz als klassische Rule-Engines verfolgt, sind damit je nach Verwendungszweck ähnliche Resultate möglich. Aufgrund der Modellierung mittels Zustandsdiagrammen[12] ist dieses Werkzeug aber eher zur Geschäftsprozessmodellierung geeignet. Für nicht-kommerzielle Zwecke ist das Produkt frei erhältlich. Der Preis für eine kommerzielle Nutzung konnte bis zum Ende dieser Arbeit nicht ermittelt werden.

3.4.5 Standards

Mittlerweile gibt es auch einige Standards für regelbasierte geschäftliche Anwendungen am Markt, von denen zwei wichtige hier vorgestellt werden. Der Standard **JSR-94** des Java Community Prozesses[13] definiert eine Schnittstelle zu einer Rule-Engine, die unabhängig von der konkreten Implementierung der Rule-Engine (also unabhängig vom eingesetzten Rule-Engine-Produkt) machen soll. Einige bekanntere Produkte wie z.B. auch die hier eingesetzte Rule-Engine Drools unterstützen diesen Standard. Ein späterer Austausch der Rule-Engine gegen eine andere (JSR-94 kompatible) Rule-Engine ist damit problemlos möglich. Der dadurch entstehende Overhead ist gering und auch die Einarbeitung kann aufgrund des sehr kleinen Umfangs dieses Standards als gering beschrieben werden. Allerdings müssen dabei zwei verschiedene Arten von Anwendungen unterschieden werden, um den Einsatz dieses Standards zu bewerten und zu rechtfertigen. Wird eine Anwendung entwickelt, die kompatibel zu möglichst vielen verschiedenen Rule-Engines sein soll, wird dadurch natürlich der potenzielle Kundenkreis ohne Mehraufwand beträchtlich erhöht. Wird die Rule-Engine aber in eine unternehmenseigene Anwendung integriert, bei der die Wahl der Rule-Engine nicht einem möglichen Kunden bzw. Anwender überlassen werden soll, wird die Auswahl eines konkreten Rule-Engine-Produktes meist schon in einem sehr frühen Stadium (z.B. nach einer anfänglichen Evaluierungsphase) fest getroffen und aufgrund der hohen Kosten einer späteren Umstellung (für Einarbeitung etc.) nicht mehr geändert. Außerdem sollte die konkrete Ansteuerung einer Rule-Engine in einer eigenen Komponente gekapselt werden, was per se schon zu einer gewissen Unabhängigkeit vom Rule-Engine-Produkt führt. In diesem Fall bringt eine Verwendung dieses Standards keine erheblichen Vorteile und es kann z.B. zugunsten einer besseren Ausnutzung spezifischer Eigenschaften der verwendeten Rule-Engine auf die Verwendung des Standards verzichetet werden. Dies ist auch der Grund, warum das hier behandelte Praxisbeispiel diesen Standard nicht verwendet.

Ein weiterer Standard ist **RuleML**[14]. Er stellt eine Initiative von verschiedenen Organisationen aus Industrie und Lehre dar, die einen künftigen Standard einer Beschreibungssprache für Regeln definieren will. Unter dem Gesichtspunkt, dass Regeln im System-technischen Sinne allgemein gültige Regeln darstellen sollen, kann ein solcher Standard zu einem wichtigen Werkzeug bei der Verwendung von regelbasierten Systemen werden. Dann könnte auch der Austausch von Rule-Sets zwischen verschiedenen Systemen verschiedener Hersteller greifbar

[11] Dieser Begriff ist hier im Sinne einer Service-orientierten Architektur (Stichwort SOA) gemeint. Dabei werden Komponenten nicht in eine Anwendung integriert, sondern deren Funktionalitäten werden von einem Server zur Verfügung gestellt. Dies ermöglicht eine Entkopplung vom Anbieter und von der jeweiligen Implementierung des Services. Siehe [Erl05].

[12] Verwendet wird eine eigene Syntax der Darstellung, die aber grob an die aus der UML bekannten Zustandsdiagramme angelehnt ist.

[13] Der Java Community Process (JCP) definiert in Kooperation mit Unternehmen, Hochschulen und Organisationen Standards für die Programmiersprache Java.

[14] Abk. für Rule Markup Language, *„Beschreibungssprache für Regeln“*, http://www.ruleml.org.

(da kostengünstig) werden. Leider ist die Arbeit dieser Initiative in einem noch sehr frühen Stadium und macht darüber hinaus trotz eines viel versprechenden Ziels noch keinen ausgereiften Eindruck. Daher kann als Fazit die weitere Beobachtung der Entwicklung dieses (und evtl. auftauchender ähnlicher Versuche) stark empfohlen werden, die konkrete Verwendung des Standards ist zum jetzigen Zeitpunkt in der Praxis aber noch nicht möglich.

Kapitel 4

Geschäftsregeln

Im bisherigen Verlauf der Arbeit wurden allgemeine Grundlagen zu regelbasierten Systemen sowie Eigenschaften von Rule-Engines erläutert. Dieses Kapitel widmet sich nun dem konkreten Einsatz von Regeln in geschäftlichen Anwendungen. Dazu wird zuerst die allgemeine Definition von Regeln auf den Einsatz in modernen Anwendungen hin erweitert. Anschließend werden wichtige Aspekte von Geschäftsregeln erläutert.

4.1 Definition

In Kapitel 2 wurde der Begriff der „Regel“ allgemein definiert. Es wurde auch erwähnt, dass regelbasierte Systeme in den letzten Jahren einen Boom im Bereich geschäftlicher Anwendungen erfahren haben. Dabei ist auch der Begriff „Business Rules“, zu deutsch „Geschäftsregeln“, geprägt worden. Obwohl nicht abzustreiten ist, dass sich dieser Begriff eher aus Gründen der besseren Vermarktbarkeit in diesem Segment etabliert hat, lässt sich mit ihm doch eine gewisse Abgrenzung zum allgemeinen Regelbegriff erreichen. Denn Regeln besitzen in geschäftlichen Anwendungen einige zusätzliche Anforderungen bzw. Eigenschaften. Diese tragen hauptsächlich der Anforderung Rechnung, dass Rule-Engines in bestehende geschäftliche Anwendungen integriert werden und mit dieser Anwendung interagieren müssen. Dies bedeutet zum einen, dass die Fakten, die in den Geschäftsregeln verwendet werden, einen Bezug zu Daten innerhalb der Anwendung besitzen, und zum anderen, dass die Aktionen der Regeln auch auf Daten, Methoden und Objekte innerhalb dieser Anwendung zugreifen müssen.

Somit lässt sich die formale Definition von Regeln dahingehend erweitern, dass Fakten durch Geschäftsobjekte repräsentiert werden, und dass Konsequenzen Geschäftsobjekte manipulieren und Geschäftsprozesse steuern. Dies betrifft hauptsächlich zwei Aspekte der Arbeit mit Rule-Engines:

- Die Formulierung der Regeln muss in einer Syntax erfolgen, die entweder direkt auf Geschäftsobjekte zugreift oder zumindest in einen eindeutigen Bezug gebracht werden kann. Dies wird in der Praxis meist dadurch erreicht, dass Regeln mit Hilfe von Quellcode der Zielplattform formuliert werden, oder dass durch eine sog. „Fachsprache“ (siehe Abschnitt 4.2.4) eine Syntax definiert wird, die eine Zuordnung zu Geschäftsobjekten ermöglicht.

- Die Rule-Engine selbst muss in eine Umgebung eingebettet werden, die diesen Zugriff auf die Anwendung erlaubt, in der sie eingebettet wird. Wie schon erwähnt, bestehen Rule-Engine-Produkte meist aus Klassenbibliotheken oder Frameworks, die wie „normale“ Software-Komponenten direkt in die Anwendung integriert werden.

Eine weitere, wenn auch etwas indirektere Eigenschaft von Geschäftsregeln ist die Tatsache, dass sie sich im Unternehmensalltag beweisen müssen. Dabei spielt außer der Wirtschaftlichkeit des kompletten Systems auch die Akzeptanz innerhalb des Unternehmens eine große Rolle. Da gerade mit Geschäftsregeln häufig der Versuch unternommen wird, die Implementierung der Geschäftslogik durch Fachabteilungen durchführen zu lassen, müssen Regeln auch in einer Form zu pflegen sein, die von IT-fernen Fachkräften schnell verstanden werden kann. Dies führte zum einen zu komfortablen Regeleditoren, wie z.B. dem von *JRules*, aber auch zu anderen Ansätzen der Regelpflege, wie z.B. Entscheidungstabellen und Fachsprachen.

Im Folgenden werden Geschäftsregeln aus den beiden wichtigen Perspektiven betrachtet: der technischen und der fachlichen.

4.2 Die technische Sicht: Syntax und Notation von Geschäftsregeln

4.2.1 Ein erster Ansatz

Nach allen Grundlagen folgt in diesem Abschnitt nun die erste Regeldefinition, die durch eine Rule-Engine verarbeitet werden kann. Im bisherigen Verlauf der Arbeit waren eigentlich nur umgangssprachliche Regeln gegeben, die immer die Frage offen gelassen haben, in welcher Form bzw. Syntax diese Regeln dann in der Praxis wirklich formuliert werden. Diese Lücke wird jetzt geschlossen werden. Dazu werden die ersten beiden Regeln dieser Arbeit nochmals aufgegriffen:

1. *„Wenn ich müde bin, dann gehe ich ins Bett!“*
2. *„Wenn ich müde bin und es vor 21 Uhr ist, dann lege ich mich auf die Couch!“*

Bevor mit der Definition einer Regelsyntax begonnen werden kann, muss geklärt werden, in welche Art von Anwendung diese Regeln integriert werden könnten. Da sie mit dem Verhalten eines Menschen zu tun haben, sollen sie in diesem Beispiel in eine Anwendung integriert werden, die das Verhalten bzw. den Tagesrhythmus eines Mensch simuliert.[1] Es gibt also einen fiktiven Menschen innerhalb dieser Simulation, der seinen Tag im Computer im Zeitraffer verbringt. Um sich die Mühen eines Echtzeitsystems zu ersparen, soll der Mensch rundenweise simuliert werden, in diesem Fall wird ein Simulationsschritt pro Runde angenommen. Damit läuft auch die Uhrzeit pro Runde um eine gewisse Zeit weiter, z.B. um jeweils 15 Minuten. Die Klasse `Mensch`, die den Testmenschen repräsentiert, besitzt einige Attribute (z.B. den Zustand, der unter anderem „müde“ oder „wach“ sein kann) und einige Methoden (wie z.B. `geheZu()`, um ihn anzuweisen, irgendwo hin zu gehen).

Dabei ist zu beachten, dass die Klasse `Mensch` nur die jeweiligen Attribute und Methoden festlegt. Sie legt aber nicht fest, *wann* der Mensch müde wird, *wann* er *wohin* geht etc., denn diese Logik soll durch die Regeln definiert werden. Die Anwendung liefert nur das Grundgerüst.

Mit Hilfe der beiden Regeln soll festgelegt werden, wie sich der Mensch verhält, wenn er müde wird. Fachlich scheinen die Regeln korrekt zu sein[2], jetzt müssen sie nur noch in einer

[1] Je nach Neigung des Lesers kann hier an eine eher wissenschaftliche Anwendung zur Untersuchung des Verhaltens von Menschen oder aber auch an ein Computerspiel wie „Die Sims“ gedacht werden. In eigenen Anwendungen wird sich dieses Problem nicht stellen, da dann der „normale“ und deutlich sinnvollere Weg beschritten wird: zu einer existierenden Anwendung werden bestehende Anforderungen in Regeln formuliert.

[2] Bei der Korrektheit der Regel kann hier natürlich nur die formale bzw. syntaktische Korrektheit geprüft werden. Die semantische Korrektheit muss auf eine andere Art und Weise verifiziert werden. Hier wird aber von vollständiger Korrektheit ausgegangen.

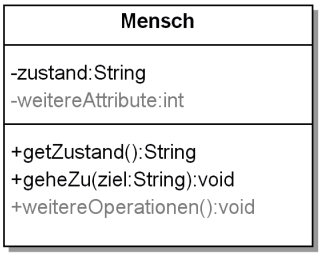


Abbildung 4.1: Klassendiagramm: „Mensch“

Form deklariert werden, in der sie von der Rule-Engine verstanden werden können. Da die umgebende Anwendung (die mit der Klasse „Mensch“) eine klassische Java-Applikation ist, sollen in einem ersten Versuch die Regeln als Pseudo-Quellcode programmiert werden:

```
// Regel 1:
if ( zustand == "muede" ) {
  geheZu( "Bett" );
}


// Regel 2:
if ( (zustand == "muede") && (uhrzeit < 21) ) {
  geheZu( "Couch" );
}
```

In dieser Form besitzen sie zwar nicht mehr den intuitiven Charakter der oben genannten natürlichsprachlichen Regeln, liegen dafür aber in einer Syntax vor, die sich fast schon direkt ausführen bzw. kompilieren lässt.

Außerdem kann hier noch ein zweiter wichtiger Punkt beobachtet werden: Die beiden umgangssprachlich formulierten Regeln sind, wie schon festgestellt, korrekt. Die beiden `if.. then..`-Anweisungen sind ganz offenbar auch eine korrekte Umsetzung der beiden Regeln.[3] Aber dennoch würde das Programm nicht korrekt funktionieren. Denn eine korrekte Umsetzung würde etwa folgendermaßen aussehen:

```
if ( zustand == "muede" ) {

  if (uhrzeit < 21) {
    geheZu( "Couch" );
  } else {
    geheZu( "Bett" );
  }

}
```

Allerdings lässt sich dies auch nicht mit Sicherheit sagen. Die hier gewählte Lösung richtet sich nach der Argumentation im Grundlagenkapitel, derzufolge die zweite Regel eine Ausnahme von der ersten ist, z.B. da sie exakter formuliert ist. Hier findet sich das Problem der

[3]Der Beweis bleibt zwar offen, aber es wird für dieses einfache Beispiel angenommen.

Konflikt-Menge und deren Lösung wieder![4] Werden andere Ansprüche an die Regeln gestellt, d.h. soll die Konfliktlösung auf eine andere Art und Weise betrieben werden, müssten die Regeln evtl. auch anders verschachtelt werden. Ungeachtet der exakten Lösung dieses speziellen Problems lässt sich aber erkennen, dass die Konflikt-Lösung durch die Art und Weise der Schachtelung der Abfragen im Quellcode vom Entwickler vorgenommen werden muss. Dies kann aus mehreren Gründen unerwünscht bzw. problematisch sein. Einer dieser Gründe ist so simpel wie offensichtlich: bei komplexeren Systemen stellt es eine große Fehlerquelle dar. Aber dieses Problem löst nun die Rule-Engine.

Im Gegensatz zur richtigen Verschachtelung der `if..then..`-Abfragen stellen die eigentlichen Prüfungen (zustand == „muede“) und Aktionen (geheZu(„Couch“)) kein Problem dar. Sie sind korrekt und dies vor allem völlig unabhängig von der Art der Abfragen, in die sie eingebettet sind.

Darauf aufbauend wird nun eine neue Form der Regeln erzeugt. Im Grundlagenkapitel wurde dazu erläutert, dass Regeln immer die Form *„Wenn .. dann ..“* besitzen. Somit könnten die `if..then..`-Anweisungen des Beispiels wieder durch abstraktere Anweisungen ersetzt werden, in die die oben erstellten Abfragen und Aktionen eingebettet werden. Dies würde folgendes Resultat liefern:

```
WENN
  zustand == "muede"
DANN
  geheZu("Bett");


WENN
  zustand == "muede"
UND
  uhrzeit < 21
DANN
  geheZu("Couch")
```

Damit wäre in einem ersten Ansatz ein Mittelweg erreicht, der

- die formale Syntax von Regeln einhält.
- die Möglichkeit bietet, die Regeln unabhängig voneinander beschreiben zu können, um die Konfliktlösung dem Interpreter zu überlassen.
- der Abfragen und Aktionen in einer Form enthält, die ausführbarem (und einem Entwickler vertrautem) Quellcode sehr ähneln.

Auch wenn dieser Ansatz noch nicht vollständig ist und bei genauerer Betrachtung noch einige Fragen aufwerfen würde, zeigt er dennoch eine mögliche Variante der Umsetzung auf. Die Rule-Engine Drools z.B. geht bei der Formulierung der Regeln einen ähnlichen Weg. Die „Wenn .. dann ..“ - Struktur wird durch eine XML-Struktur erreicht, die Abfragen und Aktionen werden als Java-Code in die XML-Tags eingebettet. Der Java-Code wird zur Laufzeit dann von einer speziellen Komponente[5] interpretiert und ausgeführt. Dadurch ist es möglich, die Syntax von Java auch zur Deklaration von Regel-Code zu verwenden. Vereinfacht dargestellt ergibt sich das Beispiel dann in Drools zu folgender Regeldeklaration:

[4] Siehe dazu die Erläuterung in Abschnitt 2.4.

[5] Drools benutzt dafür die Komponente *Janino*, *http://www.janino.net*.

```
<rule name="Regel 1">
  <condition>   zustand == "muede" </condition>
  <consequence> geheZu("Bett");    </consequence>
</rule>

<rule name="Regel 2">
  <condition>   zustand == "muede" </condition>
  <condition>   uhrzeit < 21       </condition>
  <consequence> geheZu("Couch");   </consequence>
</rule>
```

4.2.2 Ein weiterer Ansatz: Entscheidungstabellen

Da die Akzeptanz innerhalb eines Unternehmens sowie die Effektivität, mit der Regeln erfasst werden können, im Alltagseinsatz sehr wichtige Faktoren sind, gab und gibt es zahlreiche Bestrebungen zu Verbesserungen. Eine dieser Bestrebungen führte zu den sog. *„Entscheidungstabellen“*. Sie bieten die Möglichkeit, vor allem eine große Anzahl von gleichartigen Regeln komfortabler pflegen zu können. Gleichartig bedeutet dabei vor allem, dass sich viele Regeln einige Fakten teilen, auf die in ihren Prämissen getestet wird. In einer Entscheidungstabelle stellt eine Regel eine Reihe dar, die aus Prämissen in den Spalten bestehen. Dazu wird der zu prüfende Fakt (z.B. *„Uhrzeit“*) als Spaltenüberschrift verwendet, der erwartete Zielwert (z.B. *„> 21 Uhr“*) dann in den Schnittpunkt von Regel und Fakt gesetzt. Wenn sich ein Wert in einem solchen Schnittpunkt befindet, stellt dieser Wert zusammen mit dem Fakt in der Überschrift eine Prämisse dar (*„Uhrzeit > 21 Uhr“*). Wenn sich kein Wert darin befindet, wird dieser Fakt ignoriert, es entsteht folglich auch keine Prämisse.[6] In Tabelle 4.1 sind die beiden Regeln der Simulationsanwendung in einer Entscheidungstabelle dargestellt.

Entscheidungstabellen führen aber keine neuen Möglichkeiten ein, sondern bieten nur einen Formalismus, mit dem größere Mengen an Regeln übersichtlich dargestellt werden können. Der grafische Regeleditor von *JRules* kann z.B. Regeln auch in beiden Darstellungsformen anzeigen. Und je nach Situation und Einsatzzweck ist ein solcher Wechsel auch hilfreich und sinnvoll. Aber Entscheidungstabellen sind auch bei der Verwendung von Rule-Engines sehr populär, für die kein komfortabler grafischer Regeleditor von Haus aus existiert. Denn sie können leicht auch in Tabellensoftware wie z.B. Microsoft Excel modelliert und dann automatisiert in eine vorgegebene Notation transformiert werden. Eine solche Möglichkeit kann dann fast schon als „Fachsprache“ bezeichnet werden (siehe Abschnitt 4.2.4).

Intensiver sollen Entscheidungstabellen in dieser Arbeit allerdings nicht betrachtet werden. Hier sollte lediglich auf dieses mächtige Instrument aufmerksam gemacht und für die Vielzahl an Möglichkeiten sensibilisiert werden. Für weitere Informationen werden [Ros03] und [vH02] empfohlen.

4.2.3 Produktspezifische Notationsvarianten in der Praxis

Es lassen sich viele verschiedene Anforderungen an eine Regelnotation formulieren. Sie sollte auf der einen Seite möglichst einfach und intuitiv sein, auf der anderen Seite aber auch möglichst flexibel und erweiterungsfähig. In der Praxis besitzt jede Rule-Engine eine eigene Syntax zur Beschreibung der Regeln. Dies hat zur Folge, dass ein Wechsel der verwendeten Rule-Engine auch eine Neu-Deklaration der Regeln zur Folge hat. Zwischen Rule-Engines sind

[6] In den Entscheidungstabellen der Softwaretechnik werden solche leeren Felder auch als „don’t care“ bezeichnet, da ihr Wert keinen Einfluss auf die gesamte Entscheidung hat.

Regelname	**Zustand**	**Uhrzeit**	**Aktion**
„Ins Bett“	„muede“		`geheZu("‘Bett"’)`
„Auf die Couch“	„muede“	$<$ 21 Uhr	`geheZu("‘Couch"’)`
...			
...			
...			

Tabelle 4.1: Beispielhafte Entscheidungstabelle

Regeln also im Normalfall nicht portierbar. In letzter Zeit sind Versuche gestartet worden, eine einheitliche Beschreibungssprache zu definieren, z.B. das in Abschnitt 3.4.5 vorgestellte Standardisierungsprojekt *RuleML*. Allerdings hat eine solche Standardisierung noch keinen Einzug in existierende Produkte gehalten. Im Folgenden wird deshalb noch eine kurze Übersicht über Notationen dreier Rule-Engines gegeben.

Die Entwicklergruppe rund um das Open-Source Projekt **Drools** rechtfertigt ihre Notation - XML mit eingebettetem Java-Code - in der guten Kombination beider Technologien im Hinblick auf die Zielgruppe. Als relativ junges Projekt wird es zum größten Teil von Entwicklern zum Einsatz in eigenen Java-Projekten bzw. zur Evaluierung der Möglichkeiten genutzt. Diese Entwickler sind sehr vertraut mit Java, das darüber hinaus auch den kompletten Sprachumfang zur Formulierung von Bedingungen und Aktionen bereit stellt. Für XML gelten dabei ähnliche Argumente. Die Vorteile auf Seiten der Entwickler liegen ebenfalls auf der Hand: Sowohl die Interpretation von Java-Code als auch der Umgang mit XML-Dateien kann über fertige Komponenten abgewickelt werden. Damit entfällt die Entwicklung eigener Komponenten zur Erfassung und Verarbeitung der Regeln.

Beispiel der Regelbeschreibung von Drools mittels XML

```
<rule name="Is customer a teenager?">

  <parameter identifier="customer">
    <class>com.camunda.Customer</class>
  </parameter>

  <condition>
    customer.getAge() < 18
  </condition>

  <consequence>
    System.out.println( "Der Kunde ist ein Jugendlicher!" );
  </consequence>

</rule>
```

Einen anderen Weg geht die kommerzielle Rule-Engine **JRules**. Sie bietet eine sehr komfortable Oberfläche, die dem Erfasser der Regeln ermöglicht, grafisch unterstützt Regeln zu definieren, die zudem der englischen Umgangssprache sehr ähneln (siehe Abbildung 4.2). Weiterhin wird die Pflege unterstützt durch Drop-Down-Menüs u.ä., so dass Vorkenntnisse in einer Programmiersprache nicht notwendig sind. Der Aufwand zur Entwicklung dieser Fachsprache zur Abstraktion von der eigentlichen Regelsprache schlägt sich aber sicherlich auch im hohen Preis nieder. Dafür kommt JRules von allen momentan verfügbaren Produkten der

Vorstellung am nächsten, dass die Fachabteilungen zukünftig selbst ihre Regeln definieren. Von der komfortablen Oberfläche abgesehen, geht *JRules* aber dennoch einen ähnlichen Weg wie Drools: Abfragen und Aktionen werden intern auf Java- oder .NET-Klassen und -Objekte abgebildet, die von einer bestehenden Anwendung bereit gestellt werden.

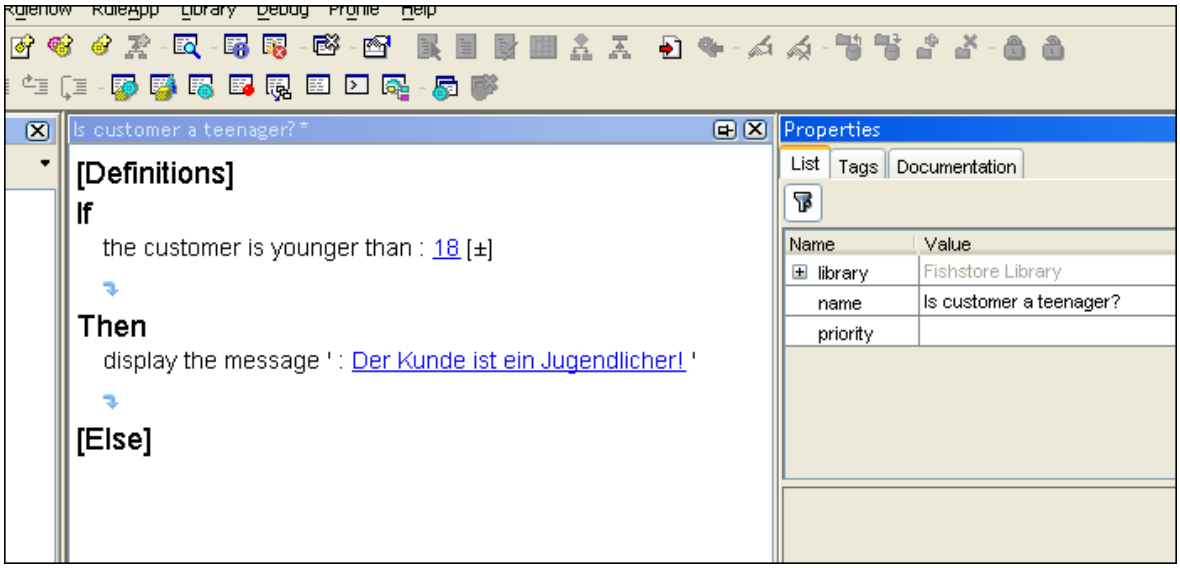


Abbildung 4.2: Screenshot Regeleditor von JRules

Gerade was den letztgenannten Punkt angeht, geht z.B. **Jess** einen anderen Weg. Hier ist es möglich, innerhalb der gleichen Deklaration sowohl Regeln als auch Datenstrukturen zu definieren, auf die diese Regeln zugreifen. Diese Eigenschaft hat Jess von funktionalen Sprachen wie z.B. Lisp geerbt, an dessen Syntax auch die Deklarationssyntax von Jess immer noch erinnert. Damit besteht die Möglichkeit, nicht nur die Komponente der regelbasierten Logik mittels der Regeldeklarationen umsetzen zu können, sondern auch die Datenstrukturen der Fakten, die innerhalb der Regeln verwendet werden sollen. Diese Möglichkeit kann in der Praxis einen erheblichen Flexibilitätsgewinn bedeuten, da zur Verwendung nicht vorhandener Strukturen nicht unbedingt die einbettende Anwendung geändert werden muss. Aber Jess bietet natürlich auch die Möglichkeit, auf vorhandene Klassen zuzugreifen.[7]

Beispiel der Regelbeschreibungssprache CLIPS von Jess

```
(defrule is-customer-a-teenager (customer-age ?a)
  (if (< ?a 18) then
  (display-message "Der Kunde ist ein Jugendlicher!"))
)
```

Deklarative Erstellung von Datenstrukturen in Jess

```
(deftemplate person
  (slot name)
  (slot age)
  (slot gender))
```

[7] Näheres zu *Jess*, insbesondere in diesem neuen Zusammenhang, kann unter *http://herzberg.ca.sandia.gov* oder in [FH03] nachgelesen werden.

```
(person (name Hendrik) (age 26) (gender male))
```

4.2.4 Ausblick: Fachsprachen

Die gezeigten Notationsvarianten orientierten sich mehr oder weniger an der formalen Struktur von Regeln sowie an den technischen Voraussetzungen der jeweiligen Produkte und Plattformen. Und obwohl Hilfsmittel wie Entscheidungstabellen durchaus eine große Steigerung der Produktivität ermöglichen, werden je nach Einsatzgebiet speziellere Lösungen benötigt. Zum Beispiel, wenn etwa eine besonders einfache Erfassung der Regeln ermöglicht werden soll, oder wenn eine Vielzahl an Informationen existiert, die nicht in einer Form vorliegen, die sich zur regelbasierten Erfassung eignen, um nur einige Beispiele zu nennen. Diese Probleme bzw. Aufgaben werden häufig durch eine Technik gelöst, die allgemein als „Fachsprache“[8] bezeichnet wird.

Eine Fachsprache ist ein Formalismus zur Darstellung von Wissen bzw. Regeln, der auf eine bestimmte Anforderung hin optimiert ist und das Wissen in eine Form transformieren kann, in der es z.B. von einer Rule-Engine gelesen werden kann. Damit kann die Erfassung in einer anderen Form stattfinden als die Verarbeitung. Da die Erfassung häufig die vom Menschen direkt durchgeführte Tätigkeit ist, können durch geeignete Formalismen oft große Optimierungen vorgenommen werden. Diese rechtfertigen oft auch einen hohen initialen Aufwand zur Entwicklung der Fachsprache.

Obwohl unter einer Fachsprache meist Varianten von Beschreibungssprachen verstanden werden, sind hier aber durchaus auch andere Techniken denkbar. Wie schon angesprochen, bietet *JRules* z.B. einen sehr komfortablen grafischen Regeleditor, der das Erfassen von natürlichsprachlichen Regeln ermöglicht. Diese vom Endergebnis her auch mit einer Fachsprache zu vergleichende Technik wird hier durch eine aufwändige grafische Oberfläche erreicht.

Die Thematik der Fachsprachen soll in dieser Arbeit aber ebenfalls nicht intensiver betrachtet werden, da sie zum Gesamtverständnis hier nicht erforderlich ist.

4.3 Die fachliche Sicht: Kategorien von Geschäftsregeln

Nachdem mit der Notation im vorherigen Abschnitt eher technische Gesichtspunkte von Geschäftsregeln in der Praxis betrachtet wurden, rückt nun der fachliche bzw. semantische Gesichtspunkt in den Fokus. Hier soll erläutert werden, welche Einsatzmöglichkeiten Geschäftsregeln bieten und welche Probleme damit gelöst werden können. Für diesen Zweck wird zunächst eine Einteilung von Geschäftsregeln in drei Kategorien vorgenommen und deren Eigenschaften erläutert.

Die Literatur schlägt viele Formen der Kategorisierung von Geschäftsregeln vor. Werden die Bewertungskriterien dieser Kategorien analysiert, so lässt sich ein Großteil der Vorschläge auf drei Kategorien zurückführen, die auch in dieser Arbeit vorgestellt werden. Teilweise werden diese Kategorien noch weiter verfeinert. Soweit nicht weiter benannt, beziehen sich die Ausführungen auf Kategorisierungsvorschläge in [Ros03], [Dat00], [vH02] und [Via97].

[8] Im Englischen meist *domain specific language* oder *domain language*.

4.3.1 Konsistenzregeln

Mit *Konsistenzregeln* werden Rahmenbedingungen definiert, innerhalb derer Daten als konsistent bzw. gültig angesehen werden.[9] Damit wird versucht, das in nahezu allen Anwendungen existierende Problem der Validierung zu lösen. Dieses Problem existiert in zwei verschiedenen Varianten, die meistens auch auf zwei verschiedenen Wegen gelöst werden:

- **Validierung von Eingaben direkt auf der Oberfläche**, wie z.B. einer Eingabemaske. Da Werte oft von Benutzern eingegeben werden, und falsche Werte somit auch oft auf ungültigen Eingaben beruhen, wird die Prüfung oft von der jeweiligen Oberflächenkomponente durchgeführt.
- **Validierung von Werten bzw. auch größeren Objektbeziehungen** oder ähnlichem innerhalb der Geschäftslogik. Dabei wird häufig versucht, die Gültigkeitsprüfungen innerhalb desjenigen Quellcodes zu implementieren, der die Werte oder Objekte manipuliert. Dabei soll bei einem Verstoß gegen eine Gültigkeitsregel also schon die Fehler verursachende Werteänderung vermieden werden.

Beide Ansätze versuchen also, ungültige Werte und Inkonsistenzen an der Quelle der Entstehung durch Prüfungen und Abfragen direkt im Quellcode zu bekämpfen. In Kapitel 2 wurde erläutert, dass der bei Rule-Engines verfolgte deklarative Ansatz eine Trennung von Geschäftslogik und Quellcode erlaubt. Mit Regeln liegt damit eine Möglichkeit vor, dieses Problem, das eine Umsetzung fachlicher Anforderungen darstellt, ebenfalls getrennt vom Quellcode zu lösen. Beispiele für solche Regeln sind:

- *„Eine Postleitzahl eines Ortes in Deutschland muss aus fünf Ziffern bestehen!“*
- *„Die Gesamtsumme einer Bestellung muss kleiner als das verbleibende Kreditlimit des Kunden sein!“*
- *„Bei den Stammdaten eines Kunden muss entweder Vor- und Nachname oder ein Firmenname hinterlegt sein!“*

Grundsätzlich sind Konsistenzregeln dadurch gekennzeichnet, dass sie verhindern sollen, dass das System ungültige Werte bzw. Zustände annimmt. Dies wird meist dadurch erreicht, dass ungültige Werte vor dem Speichern durch die Rule-Engine geprüft werden, und das Feuern einer Konsistenzregel die endgültige Speicherung verhindert.[10] Somit ähneln sich Konsistenzregeln sehr stark in Ihrem Aktionsteil.

Eine verwendete Variation der Konsistenzregeln sind *Richtlinienregeln*[11]. Diese definieren keine harte Anforderung, die nicht verletzt werden darf, sondern eher einen Hinweis, welche Richtlinie im Moment nicht mehr eingehalten wird. Solche Richtlinien werden häufig eingesetzt, um einem Benutzer bei Entscheidungen zu helfen. Wenn z.B. die Entscheidung darüber, ob ein Kunde auch bestellen darf, wenn sein Kreditlimit mit der aktuellen Bestellung überschritten werden würde, in die Hand der Mitarbeiter des Vertriebs gelegt werden sollte, könnte die „harte“ Konsistenzregel in eine „weiche“ Richtlinienregel geändert werden: *„Die Gesamtsumme einer Bestellung SOLLTE kleiner als das verbleibende Kreditlimit des Kunden sein!“* Dies hat Auswirkungen auf die Konsequenz der Regel bzw. auf die Art und Weise, wie die Konsequenz interpretiert wird. Das Prinzip ist hinter beiden Regelarten allerdings das gleiche, weshalb sie auch in die gleiche Kategorie eingeordnet werden.

[9] engl. *constraint rules* (*constraint* = Rahmenbedingung); da diese Regeln ungültige Werte „zurückweisen“ sollen, werden sie teilweise auch als *rejector rules* bezeichnet.

[10] Siehe dazu das Einsatzszenario „Validierung von Geschäftsobjekten“ in Abschnitt 5.2.2.

[11] engl. *guideline rules*.

4.3.2 Produktionsregeln

Produktionsregeln[12] produzieren neue Werte aus alten Werten. In der Literatur wird hierbei oft zwischen der einfachen Berechnung von Werten (*computation*) und der Inferenz neuer Fakten (*inference*) unterschieden. Die einfache Berechnung von Daten stellt dabei eine allgemein gültige Formel dar, deren Ergebnis einen Wert darstellt. Bei der Inferenz dagegen ist die Berechnung des Wertes abhängig vom aktuellen Kontext. Beispiele für solche Regeln sind:

- *„Die Gesamtsumme einer Bestellung errechnet sich aus der Summe der einzelnen Positionen plus Mehrwertsteuer plus Versandkosten!“* (Berechnungsvorschrift)
- *„Wenn die Gesamtsumme der einzelnen Positionen einer Bestellung größer als 100 Euro sind, dann sind die Versandkosten gleich 0 Euro!“* (Inferenz)

Hierbei muss beachtet werden, dass Berechnungsregeln keine Regeln sind, die der ursprünglichen Definition von Regeln (*„Wenn ... dann ...“*) entsprechen. Wie innerhalb der Definition von Geschäftsregeln zu Beginn des Kapitels angedeutet, müssen für solche Regeln geeignete Formulierungsmöglichkeiten gefunden werden.

Auch solche Berechnungen kommen in großer Anzahl innerhalb der Anwendungslogik geschäftlicher Anwendungen vor. Die beiden genannten Beispiele könnten z.B. aus einem System stammen, das Bestellungen von Kunden über das Internet abwickelt. Auch hier lässt sich wieder erkennen, dass sich Regeln, zumindest in dieser noch sehr umgangssprachlich und abstrakt formulierten Form, auch in Anforderungsdokumenten für Softwaresysteme wieder finden.

Die schon in Abschnitt 2.2.2 erwähnten *Produktionssysteme* setzen überwiegend bzw. ausschließlich Produktionsregeln ein. Damit nutzen sie die die regelbasierte Technik hauptsächlich dazu, um neues Wissen zu erzeugen und nicht z.B. zur Steuerung von Geschäftsprozessen, wie die im folgenden Abschnitt erläuterten *Aktionsregeln*.

4.3.3 Aktionsregeln

Aktionsregeln[13] definieren, auf welche Art und Weise, d.h. mit welcher Aktion, auf das Eintreffen bestimmter Ereignisse reagiert werden soll. Aus diesem Grund können sie auch als *“Ereignisregeln“* bezeichnet werden. Die Bedingung dieser Regeln beschreibt das eintreffende Ereignis, die Konsequenz beschreibt die Aktionen, die dann ausgeführt werden sollen. Beispiele für solche Regeln sind:

- *„Sobald alle Artikel einer noch nicht ausgelieferten Bestellung auf Lager sind, setze den Status der Bestellung auf 'versandfertig' und veranlasse die Auslieferung!“*
- *„Sobald eine Bestellung ausgeliefert worden ist, schicke eine Benachrichtigungsmail an den Kunden!“*

Mit Hilfe von Ereignisregeln (hier ist diese Bezeichnung besser geeignet!) ist es auch möglich, komplette Workflows innerhalb eines Systems zu modellieren. Aus diesem Grund werden Rule-Engines immer häufiger im Zusammenhang mit Systemen zum Geschäftsprozessmanagement[14] eingesetzt. Ein offensichtlicher Anwendungsbereich wäre es z.B., Zustandsübergänge zwischen Zuständen in der Prozesskette durch Regeln steuern zu lassen, da diese Übergänge

[12] Im Englischen meist *derivation rules*, *computation rules*, *producer rules* oder *inference rules*.

[13] Diese Regeln besitzen in der Literatur sehr unterschiedliche Bezeichnungen, die wichtigsten sind *action enabler rules* ([vH02]), *projector rules* ([Ros03], S.79), *action/event rules* und *behavior rules*.

[14] engl. *business process management*, *BPM*.

stets vom aktuellen Kontext abhängig sind. Diese Möglichkeiten sind aber nicht Teil dieser Arbeit.[15]

Aktionsregeln und Konsistenzregeln werden teilweise als Gegenspieler betrachtet. Während Konsistenzregeln bei Erkennung einer Inkonsistenz den aktuellen Vorgang abbrechen, starten Aktionsregeln bei Eintreffen bestimmter Voraussetzungen einen neuen Vorgang.[16]

Unter einem anderen Gesichtspunkt können auch Produktions- und Aktionsregeln als Gegenspieler betrachtet werden. Produktionsregeln haben keinen direkten Einfluss auf den Programmablauf. Erst dadurch, dass Werte durch sie geändert werden, kann ein indirekter Einfluss entstehen. Im Gegensatz dazu beeinflussen Aktionsregeln immer den Programmablauf direkt, da die Regel explizit eine Aktion auslöst. Damit sind Aktionsregeln meist auch stärker an den Kontext bzw. die Anwendung gekoppelt.

4.4 Die praktische Sicht: das Beispiel „Game of Life“

4.4.1 Einführung

Als Beispiel wird nun die bekannte Simulationsanwendung „Game of Life“ des Mathematikers John Horton Conway verwendet.[17] Es simuliert das Verhalten von Zellen auf einem zweidimensionalen, beliebig großen „Spielfeld“. Jedes Feld bietet dabei den Lebensraum für genau eine Zelle, die entweder lebendig oder tot sein kann. Pro Runde wird entschieden, ob eine lebende Zelle in der nächsten Runde weiterleben darf oder stirbt, und ob eine tote Zelle wieder zum Leben erweckt wird. Diese Entscheidung wird im Game of Life anhand der Anzahl ihrer Nachbarn entschieden. Sind es zu viele, stirbt eine Zelle an Überbevölkerung. Sind es zu wenige, stirbt sie an Einsamkeit. Nur wenn die Zahl der Nachbarn stimmt, überlebt eine Zelle bzw. darf eine tote Zelle wieder lebendig werden. Diese Logik soll nun anhand von Regeln formuliert und in einer realen Notation deklariert werden.

1. *„Wenn eine Zelle lebendig ist und weniger als drei Nachbarn hat, stirbt sie!“*
2. *„Wenn eine Zelle lebendig ist und mehr als vier Nachbarn hat, stirbt sie!“*
3. *„Wenn eine Zelle tot ist und genau drei Nachbarn hat, wird sie lebendig!“*

Anmerkung: Genau betrachtet, muss diese Logik eigentlich nicht anhand von Regeln formuliert werden, was eigentlich eine Transformation in die Syntax von Regeln andeuten würde. Denn ursprünglich wurden die Regeln dieses Spiels von Conway schon in dieser regelbasierten Form beschrieben, weil sie die natürliche, umgangssprachliche und intuitive Lösung dafür darstellen. Genau hier liegt auch ein großer Vorteil von Regeln: sie sind eine sehr genaue formale Abbildung natürlicher Denkweisen.

Die Anwendung, in die diese Logik eingebettet werden soll, stellt nur die Umgebung des Spiels bereit: das Spielfeld und die Zellen. Wie den Regeln zu entnehmen ist, benötigt die Zelle ein Attribut, das den aktuellen Lebenszustand enthält sowie eine Methode, die die Anzahl der Nachbarn zurück liefert. Damit könnte die Klasse Zelle folgendermaßen aussehen:

Als Notation wird nun die der Rule-Engine Drools verwendet. Die Wahl der Notation spielt eigentlich keine Rolle, da in diesem Abschnitt nur ein knapper Einblick gegeben werden kann. Da Drools aber auch innerhalb der Praxisbeispiels verwendet wird, erschien es hier am geeignetsten.

[15] Als weiterführende Literatur zum Thema Geschäftsprozessmanagement wird [Ses04] empfohlen. Es bietet einen guten Einstieg sowohl aus technischer als auch aus betriebswirtschaftlicher Sicht.

[16] Siehe [vH02].

[17] Inspiriert durch ein Tutorial der Rule-Engine Drools. Näheres dazu unter [Dro05].

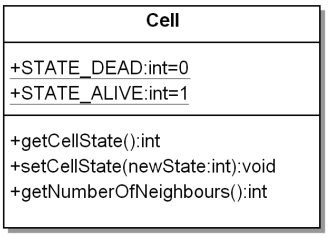


Abbildung 4.3: Klassendiagramm „Game of Life“

4.4.2 Die Regeln

Nun werden die Regeln in der von der Rule-Engine vorgeschriebenen Syntax erfasst. Dazu wurden im vorherigen Abschnitt die Regeln in umgangssprachlicher Form dargestellt sowie die Schnittstelle der relevanten Klasse `Cell` erläutert. Somit kann nun mit der Formulierung der Regeln in der technischen Notation begonnen werden:

„Wenn eine Zelle lebendig ist und weniger als drei Nachbarn hat, stirbt sie!“

Ob eine Zelle lebendig oder tot ist, wird von der Methode `getCellState()` zurück geliefert, die Anzahl der Nachbarn von der Methode `getNumberOfNeigbours()`. Ob sie lebendig wird, lebendig bleibt oder stirbt, wird durch die Methode `setCellState( String newCellState )`[18] gesteuert. Damit kann die Regeln nun in Java-Code formuliert werden:

```
if ( ( cell.getCellState() == Cell.STATE_ALIVE ) &&
     ( cell.getNumberOfNeighbours() < 3 ) ) {

     cell.setCellState( Cell.STATE_DEAD );

}
```

Wie zu Beginn dieses Kapitels erläutert wurde, wird die in Java vorhandene `if.. then..`-Struktur durch eine entsprechende XML-Struktur ersetzt. Damit sieht die Deklaration der Regeln nun folgendermaßen aus:

```
<java:condition>   cell.getCellState() == Cell.STATE_ALIVE </java:condition>
<java:condition>   cell.getNumberOfNeighbours() < 3        </java:condition>

<java:consequence> cell.setCellState( Cell.STATE_DEAD ); </java:consequence>
```

Analog dazu kann mit den beiden nächsten Regeln verfahren werden. Da sie ebenfalls nur auf die hier verwendeten Methoden zugreifen, ähneln sie der ersten Regel sehr.

„Wenn eine Zelle lebendig ist und mehr als vier Nachbarn hat, stirbt sie!“

```
<java:condition>   cell.getCellState() == Cell.STATE_ALIVE </java:condition>
<java:condition>   cell.getNumberOfNeighbours() > 4        </java:condition>

<java:consequence> cell.setCellState( Cell.STATE_DEAD ); </java:consequence>
```

[18] Diese Vorgehensweise würde eigentlich einen Seiteneffekt haben: wenn der Zustand der Zelle sofort geändert wird, hat das sofortige Auswirkungen auf die Berechnung der Nachbarzellen. Aus diesem Grund müsste mit der eigentlichen Zustandsänderung so lange gewartet werden, bis die Berechnungen der aktuellen Runde abgeschlossen sind. Dies soll hier aber nicht berücksichtigt werden.

„Wenn eine Zelle tot ist und genau drei Nachbarn hat, wird sie lebendig!“

```
<java:condition>   cell.getCellState() == Cell.STATE_DEAD  </java:condition>
<java:condition>   cell.getNumberOfNeighbours() == 3       </java:condition>

<java:consequence> cell.setCellState( Cell.STATE_ALIVE );</java:consequence>
```

4.4.3 Das Spiel in Aktion

Durch die deklarierten Regeln ist die Rule-Engine nun im Stande, in jeder Runde die Population der Zellen zu bestimmen. Dabei muss die natürlich in die Anwendung integriert werden, die die Klasse `Cell` bereitstellt sowie die grundsätzlichen Aufgaben der Anwendung übernimmt, die nicht regelbasiert gelöst werden, z.B. die Anzeige des Spielfelds. Diese Anwendung muss in jeder Runde die Rule-Engine aufrufen, damit die neuen Zellzustände berechnet werden. Diese Integration ist ein Thema des nächsten Kapitels.

Kapitel 5

Integration einer Rule-Engine in bestehende Anwendungen

Nachdem in den vorangegangenen Kapiteln alle notwendigen Grundlagen erarbeitet worden sind, kann nun der erste Schritt in die Praxis erfolgen. Dazu beschreibt dieser Abschnitt, auf welche Art und Weise eine Rule-Engine in eine bestehende Anwendung integriert werden kann, um diese so um die Funktionalitäten regelbasierter Systeme zu erweitern.

Von einem gewissen Standpunkt aus handelt dieses Kapitel von der Integration von Software-Komponenten im allgemeinen Sinne. Ein für diese Beschreibung wichtiger Punkt aber, der Rule-Engines von den meisten Software-Komponenten unterscheidet, ist die Bestimmung des fachlichen Einsatzzweckes. Dieser Einsatzzweck entscheidet oft maßgebend darüber, zu welchen anderen Komponenten Schnittstellen bestehen müssen und auf welche Art und Weise die Komponente integriert werden kann. Bei Rule-Engines existiert dieser Einsatzzweck allerdings nicht von Anfang an. Der eigentliche Zweck ergibt sich erst durch die Semantik der Regeln, die von ihr verarbeitet werden.[1]

Somit ist diese Aufgabenstellung pauschal und ohne Kenntnisse von Einsatzzweck, Regeln und bestehender Infrastruktur nur schwer zu erörtern, weshalb sie in der Praxis meist nach individuellen Gegebenheiten evaluiert wird. Um aber dennoch ein Gefühl für praktikable Möglichkeiten vermitteln zu können, werden hier beispielhafte Integrationsmöglichkeiten anhand einiger Einsatzszenarien beschrieben. Diese Szenarien sind motiviert durch Realisierungsmöglichkeiten des Praxisbeispiels im nächsten Kapitel und sollen als Motivation und Inspiration für individuelle Problemstellungen dienen.

Anmerkung des Autors: Aufgrund fehlender Literatur für diesen Bereich wurde überlegt, diese Thematik nicht in die Arbeit aufzunehmen. Die Problemstellung wird vom Autor aber in der Praxis als wichtiger und sensibler Punkt betrachtet. Außerdem wurden im Laufe der Arbeit einige Szenarien prototypisch erarbeitet. Daher wurde beschlossen, es basierend auf eigenen Erfahrungen in der hier dargestellten Form zu beschreiben. Die Ähnlichkeit mit Beschreibungen von *Mustern* der Softwaretechnik ist beabsichtigt. Aufgrund der fehlenden Absicherung der beschriebenen Szenarien in realen Projekten und produktiven Systemen wurde allerdings bewusst ein anderer Begriff zur Differenzierung gewählt.

[1] Diese Eigenschaft war Teil der Definition einer Rule-Engine.

5.1 Integration der Beispielanwendung „Game of Life“

Bei der Vorstellung des Beispielprogramms am Ende des letzten Kapitels ist eine Frage offen geblieben: wie wird die Rule-Engine in die „Game of Life“-Anwendung integriert, damit die Rule-Engine die Zellen leben und sterben lassen kann? Um diese Frage zu beantworten, wird zuerst kurz der Programmablauf der Simulation betrachtet.

Wie schon erläutert läuft das Spiel rundenweise ab. Dies bedeutet, dass in jeder Runde das aktuelle Spielfeld angezeigt wird und danach die neuen Zellzustände für die nächste Runde ermittelt werden. Diese Iteration endet, wenn sich durch die aktuelle Konstellation keine Änderung zur nächsten Runde hin mehr ergeben würde, wenn also das System stabil wird. In Pseudo-Code sieht diese einfache Anwendung also folgendermaßen aus:

```
do
{

  spielfeldAnzeigen();
  neueZustaendeBerechnen();

}
while ( zustaendeNichtStabil() );
```

Die im Falle prozeduraler Programmierung als Funktion implementierte Logik zur Berechnung der neuen Zustände (`neueZustaendeBerechnen()`) kann nun ersetzt werden durch einen Aufruf der Rule-Engine, die diese Logik mittels der Regeln durchführen wird. Somit sieht die Version der Anwendung mit Rule-Engine folgendermaßen aus:

```
initialisiereRuleEngine(); // Lade Regeln, erzeuge Working Memory

do
{

  spielfeldAnzeigen();
  workingMemory.fireAllRules();

}
while ( zustaendeNichtStabil() );
```

Somit ist die komplette Logik, auf der die Lebensbedingungen der Zellen basieren, in die Deklaration der Regeln ausgelagert. Das Programm selbst enthält nur noch den notwendigen Rahmen (das Spielfeld, die ständig weiterlaufende Schleife etc.). Daran ist auch wieder sehr gut das deklarative Prinzip zu erkennen, das hinter der regelbasierten Steuerung der Programmlogik steht: die Teile des Quellcodes, die stark von Entscheidungen abhängig sind, werden in Regeln ausgelagert und zur Laufzeit von einem Interpreter (der Rule-Engine) verarbeitet. Und diese Teile sind eben nicht die Schleife oder die Anzeige des Spielfeldes, sondern die Regeln des Spiels.[2]

Obwohl es sich hierbei natürlich nur um Pseudo-Code handelt, der in dieser Form noch nicht ausführbar ist, sollte eines noch betrachtet werden: Bevor die Rule-Engine (bzw. der Working Memory) auf diese Art und Weise aufgerufen werden kann, muss sie natürlich noch

[2] Vergleiche nochmals Abschnitt 2.5 zur Theorie der deklarativen Programmierung.

erzeugt und initialisiert werden. Dabei ist vor allem wichtig, dass zu Beginn des Programms Zellen für das komplette Spielfeld erzeugt und im Working Memory abgelegt werden. Nur dann befinden sich die Zellen auch als Faktenbasis im Working Memory, damit die Regeln auf sie angewendet werden können. Dieses Erzeugen der Zellen würde etwa folgendermaßen aussehen:

```
for (int x=0; x < breiteDesSpielfeldes; x++)
{
  for (int y=0; y < hoeheDesSpielfeldes; y++)
  {
    Cell cell = new Cell( x, y );
    workingMemory.assertObject( cell );
  }
}
```

Damit befindet sich nun jede Zelle des Spielfeldes im Working Memory und bei jedem `fireAllRules()` - Aufruf werden die Regeln auch auf jede Zelle angewendet. Abbildung 5.1 zeigt den Programmablauf in einem Sequenzdiagramm. Die Einsatzszenarien im folgenden Abschnitt werden dann ebenfalls durch Sequenzdiagramme illustriert.

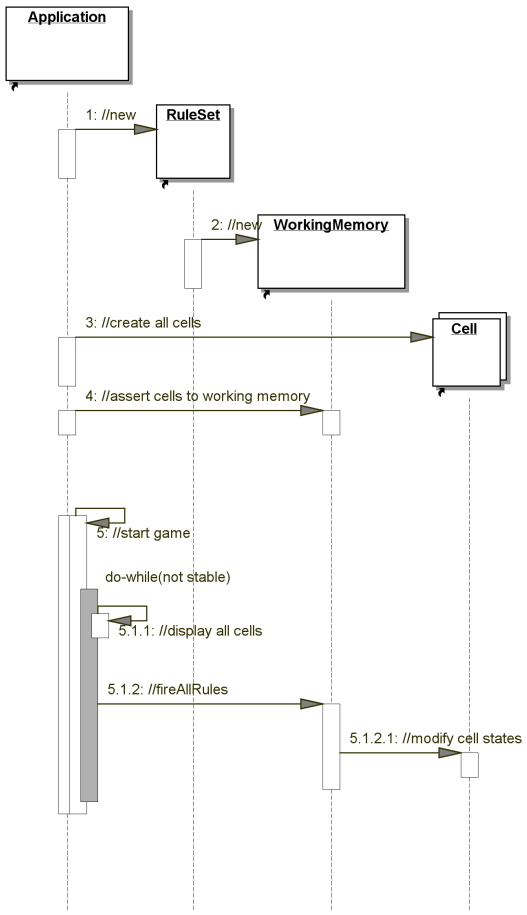


Abbildung 5.1: Sequenz-Diagramm „Game of Life“

5.2 Einsatzszenarien

5.2.1 Personalisierte Weboberfläche

Anwendungsbereich

Die Rule-Engine soll eingesetzt werden, um die Inhalte einer Weboberfläche zu erzeugen bzw. zu manipulieren. Die Besonderheit besteht in diesem Beispiel darin, dass individuelle Daten des Benutzers zur Faktenbasis hinzugefügt werden, um die Regeln auf personalisierte Daten anwenden zu können. Woher diese Daten stammen, soll in diesem Beispiel nicht berücksichtigt werden. Denkbar wären z.B. gespeicherte Kundendaten bei eingeloggten Kunden oder Eingaben während des Besuchs über Formulare.

Wichtig dabei ist, dass für jede neue Session[3] auch ein neuer Working Memory angelegt und nur mit Daten dieser Session gefüllt wird (vgl. „Permanenter Working Memory“, Abschnitt 5.2.3). Zu beachten ist auch, dass hier der Rule-Engine-Aufruf durch den Seitenaufruf des Benutzers angestoßen wird.

Sequenz-Diagramm: Siehe Abbildung 5.2.

Beteiligte Klassen

- **PresentationTierApplication** Die Anwendung, die Webseiten-Anfragen entgegennimmt, Seiten generiert und zurückliefert. In der Praxis besteht diese Anwendung oft aus mehreren Komponenten, z.B. einem Webserver und einem Content Management System. Aus Gründen der Übersichtlichkeit soll diese Anwendung hier aber etwas abstrakter betrachtet werden.
- **Session** In dieser Container-Klasse sind die aktuellen persönlichen Daten jedes einzelnen Benutzers abgelegt.
- **Working Memory** Für jede neue Session wird ein individueller Working Memory angelegt. Dieser bleibt aber über die Dauer einer Session erhalten.
- **BusinessTierInterface** Stellt abstrakt die Schnittstelle dar, über die die Oberflächenanwendung Daten der Geschäftslogik zur Generierung einer Webseite abfragen kann.

Übertragung auf andere Anwendungsbereiche

- Sollen individuelle Daten als Fakten in der Regelverarbeitung verwendet werden, muss für jedes individuelle Set an Daten ein eigener Working Memory verwendet werden.
- Wenn definierte Ereignisse auftreten, zu denen eine erneute Regelverarbeitung notwendig wird (hier der Seitenaufruf), so kann dieses für den `fireAllRules()`-Aufruf verwendet werden. Ansonsten muss auf andere Art und Weise sichergestellt werden, dass dieser Aufruf zum richtigen Zeitpunkt stattfindet.

[3]Der Begriff *Session* wird hier im Sinne des Internets verwendet, die den Zeitraum bezeichnet, über den sich ein und derselbe Benutzer über eine Internetpräsenz bewegt.

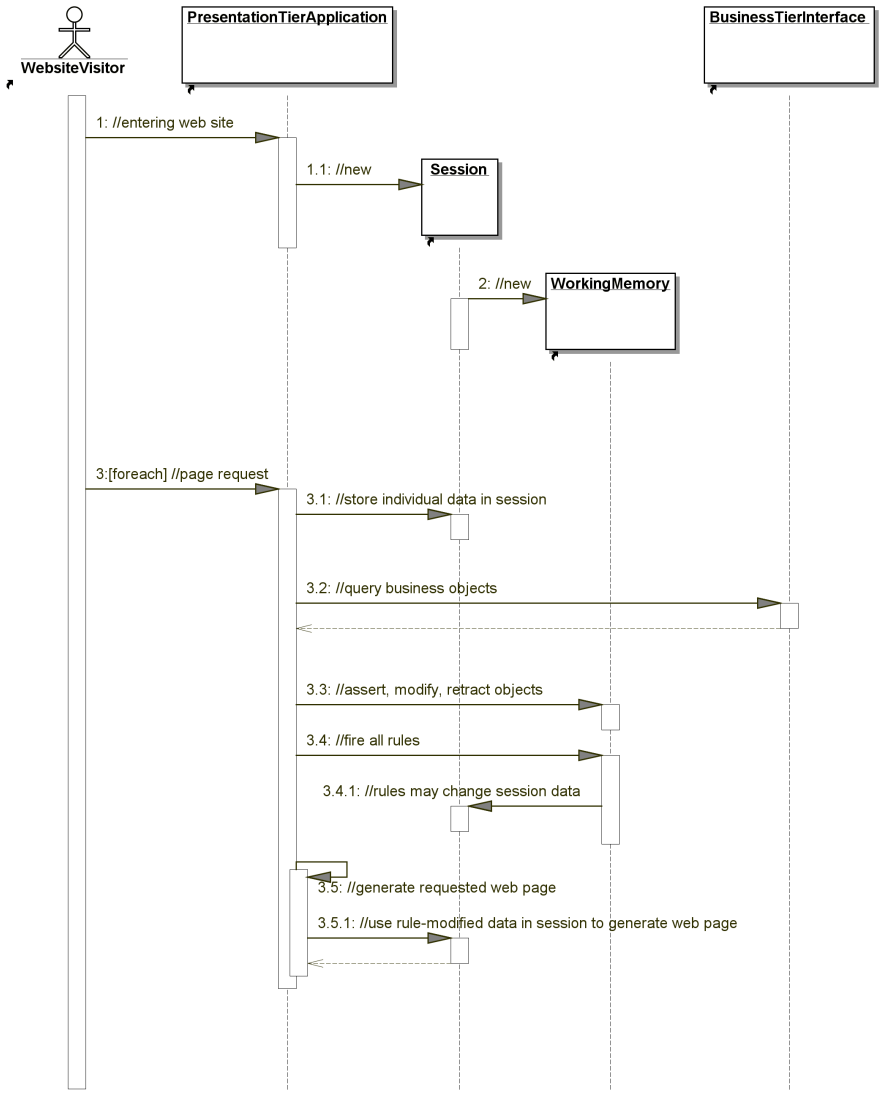


Abbildung 5.2: Sequenz-Diagramm Szenario „Personalisierte Weboberfläche“

5.2.2 Validierung von Geschäftsobjekten

Anwendungsbereich

Die Rule-Engine soll eingesetzt werden, um Geschäftsobjekte zu validieren und einen konsistenten Datenbestand zu wahren. Im Gegensatz zu Validierungsmechanismen, die direkt an die Oberfläche gebunden sind, kann dieses Szenario unabhängig von der Oberfläche eingesetzt werden und darüber hinaus auch zur Validierung von Daten eingesetzt werden, die in der Geschäftslogik entstehen. Existiert eine allgemeine Schnittstelle zur Datenhaltungsschicht, so kann diese Schnittstelle als Ort der Integration verwendet werden. Damit existieren auch explizite Zeitpunkte für den Aufruf der Rule-Engine und somit für den Zeitpunkt des Validierens. Validierungsfehler und Inkonsistenzen verhindern dann das endgültige Speichern der Daten, so dass der persistente Datenbestand stets konsistent und gültig bleibt.

Bei dieser Lösung läuft die Validierung sehr generisch ab, sowohl die ConsistencyChecker-Komponente wie auch die Rule-Engine können beliebige Objekte entgegennehmen, da der

Typ bzw. die Klasse von Objekten erst im Working Memory bei der Verarbeitung durch die Regeln relevant wird. Dabei besteht dann das Problem der Rückmeldung von Fehlern. Die hier gewählte Lösung besteht darin, dass der ConsistencyChecker jegliche Fehlermeldungen durch die Regeln aufnimmt und im Fall eines Fehlers einen Fehlercode o.ä. an das aufrufende Objekt zurück sendet. Innerhalb der Regeln wird dann nur auf den ConsistencyChecker zugegriffen:

```
<java:condition> postleitzahl.length != 5 </java:condition>

<java:consequence>
  consistencyChecker.addError( "Postleitzahl muss 5 Stellen besitzen!" );
</java:consequence>
```

Dieser Ansatz zur Rückmeldung von Regelergebnissen wird bei der Beschreibung des Feedback-Mechanismus' in Abschnitt 6.5.1 detailliert erläutert.

Sequenz-Diagramm: Siehe Abbildung 5.3.

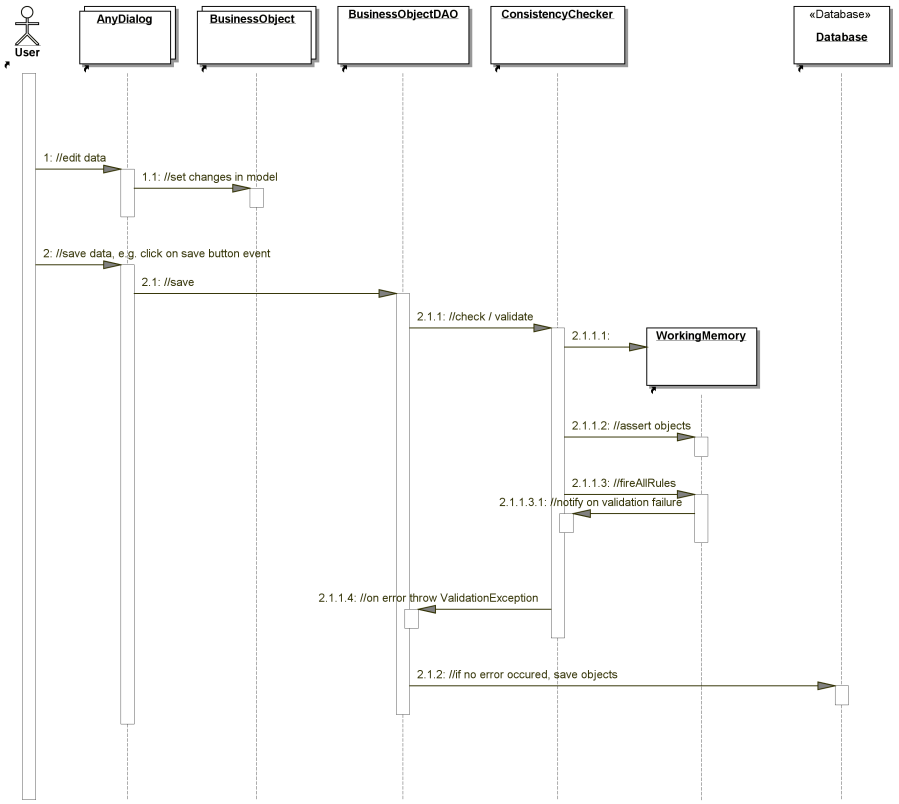


Abbildung 5.3: Sequenz-Diagramm Szenario „Konsistenzprüfungen"

Beteiligte Klassen

- **AnyDialog** Hier werden die Daten eines Geschäftsobjektes geändert. Änderungen werden in das jeweilige Geschäftsobjekt (*BusinessObject*) übertragen. Beim Klick auf dem „Speichern"-Button wird das jeweilige DAO des Geschäftsobjektes aufgerufen. Diese

Klasse steht stellvertretend für jeden Dialog der Anwendung. Zusätzlich soll sie in diesem Beispiel die Komplexität realer geschäftlicher Anwendungen verbergen. Hier ist der Zeitpunkt interessant, an dem ein Geschäftsobjekt persistenziert werden soll.[4]

- **BusinessObject** Das veränderte Geschäftsobjekt, z.B. „Kunde“ oder „Bestellung“.
- **BusinessObjectDAO** Das DAO[5] kapselt Persistenzmechanismen für das betreffende Geschäftsobjekt. DAO's sind eine oft eingesetzte Realisierung der Persistenz.
- **ConsistencyChecker** Diese Klasse kapselt die komplette Validierung einschließlich der Rule-Engine-Ansteuerung. Sie nimmt Geschäftsobjekte entgegennehmen und reicht sie zur Prüfung auf Konsistenz und Gültigkeit an die Rule-Engine weiter.
- **WorkingMemory** Der WorkingMemory, in den die Geschäftsobjekte eingefügt werden, die validiert werden sollen.

Anmerkungen

- Zur Validierung werden hier ausschließlich Konsistenzregeln (siehe Abschnitt 4.3) verwendet. Diese haben bei der hier vorgestellten Lösung gemeinsam, dass ihre Konsequenzen jeweils Methoden der `ConsistencyChecker`-Komponente aufrufen.

Übertragung auf andere Anwendungsbereiche

- Funktionalitäten, die unabhängig von der Oberfläche sind, sollten innerhalb der Geschäftslogikschicht integriert werden. Die Oberflächenkomponente wird schlanker und leichter austauschbar. Trotzdem können zusätzliche Validierungsmaßnahmen auf der Oberfläche integriert werden. Ein ganz deutlicher Nachteil der hier gezeigten Methode ist z.B., dass Fehler erst im Moment des Speicherns gemeldet werden, was nicht unbedingt eine optimale Bedienbarkeit der Oberfläche bedeutet. Der richtige Kompromiss aus regelbasierter und klassischer Validierung sowie aus der Integration in Oberfläche und Geschäftslogikschicht stellt dabei eine der Herausforderungen eines realen Projektes dar.
- Bei der Integration innerhalb der Geschäftslogik können Regeln zentral verwaltet werden. Vor allem in Verbindung mit dem Anforderungsmanagement können große Synergieeffekte entstehen.
- Die Validierung kann von Geschäftsobjekten entkoppelt werden. Dadurch werden auch Geschäftsobjekte leichtgewichtiger, fehlerfreier und wartungsfreundlicher.

5.2.3 Permanenter Working Memory

Anwendungsbereich

Die Rule-Engine soll eingesetzt werden, um große Mengen von Fakten zu unregelmäßigen Zeitpunkten zu verarbeiten. Dann ist es sinnvoll, allen Objekten einen einzelnen, zentralen Working Memory zur Verfügung zu stellen. Evaluierungsläufe werden dann nicht mehr bei jeder Änderung der Fakten, sondern zu bestimmten Zeitpunkten von einer verantwortlichen Komponente durchgeführt. Vorteil eines permanenten Working Memory ist es, dass sich schon

[4] Das Beispiel eines Dialogs erschien als die intuitivste Möglichkeit zur Änderung eines Objekts.
[5] Data Access Object, übersetzt etwa „Datenhaltungs-Zugriffs-Objekt“. Siehe [Mal03].

alle Objekte darin befinden, wenn ein Evaluierungsdurchlauf gestartet werden muss. Ein bei großen Mengen an Objekten aufwändiges Hinzufügen aller Objekte entfällt somit. Außerdem kann die Eigenschaft des Rete-Algorithmus' ausgenutzt werden, der ab dem zweiten Durchlauf sehr viel schneller Resultate liefert (siehe Abschnitt 2.3.3).

Sequenz-Diagramm: Siehe Abbildung 5.4.

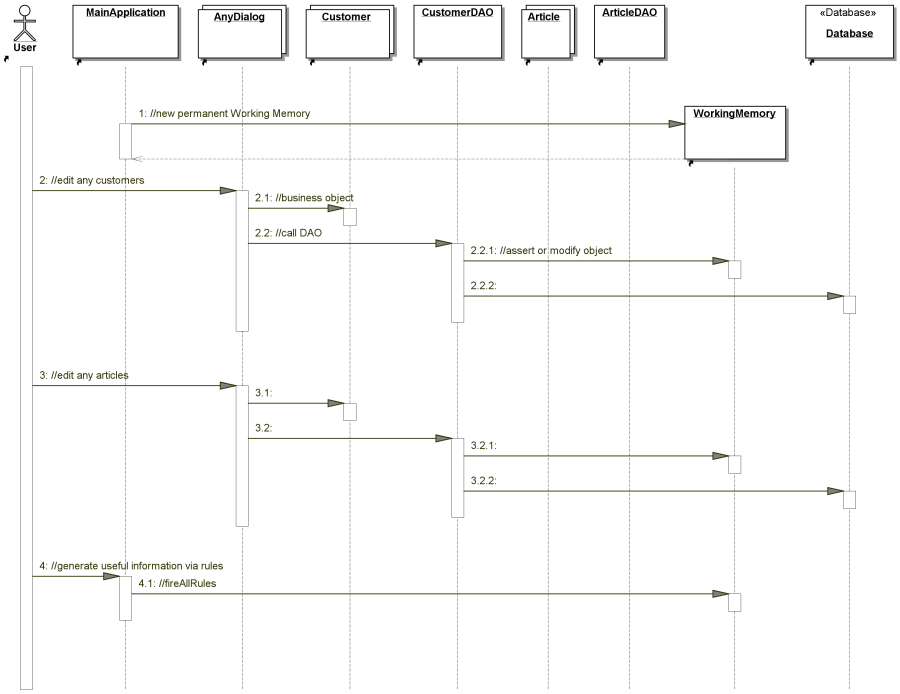


Abbildung 5.4: Sequenz-Diagramm Szenario „Permanenter Working Memory"

Beteiligte Klassen

- **MainApplication** Erzeugt zu Beginn den gemeinsamen Working Memory. Startet außerdem bei Bedarf Evaluierungsläufe der Rule-Engine unabhängig von Modifizierungen des Working Memory (z.B. dem Hinzufügen von Objekten).
- **AnyDialog, Customer, Article, DAOs** Diese Klassen bringen zum Ausdruck, dass alle Änderungen an allen Geschäftsobjekten im Moment des Speicherns auch Änderungen am selben Working Memory hervor rufen.
- **WorkingMemory** Der Working Memory wird zu Beginn erzeugt, ist von allen relevanten Änderungen betroffen und bleibt über einen langen Zeitraum hinweg bestehen.

Übertragung auf andere Anwendungsbereiche

- Der permanente Working Memory kann z.B. zur Erstellung von Kennzahlen oder für Auswertungen verwendet werden, deren Berechnung häufig viele tausend Objekte zu Grunde liegen, wobei die Berechnung aber nicht bei jeder Änderung eines Objektes durchgeführt werden soll.

5.3 Zusammenfassung

Diese drei Einsatzszenarien sollen stellvertretend für die jeweiligen individuellen Einsatzzwecke sein. Aus ihnen können aber die wichtigsten Fragestellungen abgeleitet werden, die bei fast jedem Einsatz beachtet werden müssen:

- Der Ort (bzw. die Schicht) innerhalb der Architektur, in der die Rule-Engine integriert wird.
- Zeitpunkt des Aufrufs der Rule-Engine. Damit hängt auch die Fragestellung zusammen, ob äußere Ereignisse existieren, die diesen Aufruf implizieren können.
- Die Lebensdauer des Working Memorys, d.h. die Fragestellung, ob über einen längeren Zeitraum Objekte in den selben Working Memory eingefügt werden oder ob bei jeder Aktion ein neuer Working Memory erzeugt wird.

Es ist auch möglich und in der Praxis üblich, mehere dieser Einsatzszenarien zu kombinieren. Es muss keine verbindliche, anwendungsweit einzuhaltende Entscheidung für eines dieser Szenarien bzw. eine Variante davon getroffen werden. Meist existieren verschiedene Einsatzzwecke, aus denen dann auch individuelle Anforderungen an die Integration resultieren.

Auf jeden Fall bleibt der Hinweis bestehen, dass diese Entscheidungen teilweise große Auswirkungen auf Architektur- und Designentscheidungen haben, weshalb die verschiedenen Möglichkeiten mit Sorgfalt evaluiert werden sollten.

Kapitel 6

Praxisbeispiel

6.1 Umfeld „mad-moxx“

Der Geschäftsbereich *mad-moxx* des Unternehmens *Computation* hat sich auf den Vertrieb von Grafikkarten und Grafikkartenzubehör über das Internet spezialisiert.[1] Die Spezialisierung auf die kleine Nische im Hardwaresegment hat unter anderem zur Folge, dass sehr großes Know-How in diesem kleinen Bereich vorhanden ist und ständig erweitert wird. Dieses Know-How besteht dabei z.B. aus dem Wissen, welche Hardware-Kombinationen am schnellsten und stabilsten betrieben werden können oder welche Produkte am besten mit vorhandenen Hardware-Konfigurationen des Kunden kombiniert werden können. Von Natur aus ist diese Problematik sehr komplex. Sie hängt von sehr vielen verschiedenen, individuellen Faktoren ab, z.B. von Technik und Einsatzgebieten des Kunden abhängen. Dadurch sind Regeln als entscheidungsbasierter Formalismus prinzipiell zu ihrer Beschreibung geeignet sind.

6.2 Anforderungen

Nun sollte ein System entwickelt werden, mit dem dieses Wissen erfasst und verarbeitet werden kann. Die Verarbeitung sollte dabei sowohl für interne Zwecke, als auch für die Aufbereitung im Internet genutzt werden, um in möglichst großem Maße davon profitieren zu können. An das System werden folgende Anforderungen gestellt:

1. Flexible und vollständige Erfassung des vorhandenen Expertenwissens muss möglich sein.

2. Möglichst vielfältige Verwendungsmöglichkeiten der erfassten Wissensbasis in verschiedenen Anwendungsbereichen.

3. Einbeziehung von Informationen des Kunden in den Wissensverarbeitungsprozess (z.B. über einen Dialog auf der Homepage o.ä.), um eine individuelle Auswertung des Wissens zu ermöglichen.

4. Erarbeitung eines Konzeptes und prototypische Lösung aller auftauchenden kritischen Probleme als *„proof of concept“*.

5. Prototypische Implementierung eines Dialogsystems zur Interaktion mit dem Benutzer. Dieses Dialogsystem soll später in der Internetpräsenz eingebettet werden.

[1] *http://www.mad-moxx.de*, *http://www.computation.de*.

Die Entscheidung, dieses Projekt mittels regelbasierter Systeme und Rule-Engines durchzuführen, wird folgendermaßen begründet:[2]

- Das Wissen ist zu einem großen Teil stark abhängig vom Kontext (siehe Kapitel 2)
- Da sich Wissen in sehr kurzen Abständen ändert bzw. erweitert und von den Fachabteilungen selbst hinzugefügt werden soll, eignet sich der deklarative Ansatz regelbasierter Systeme (siehe Abschnitt 2.5)
- Das Wissen ist teilweise sehr komplex und wird von sehr vielen Ausnahmen und Einzelfällen beherrscht. Dies lässt sich mit herkömmlichen Methoden nur schwer formalisieren. Auch hier eignen sich Regeln von Natur aus sehr gut
- Die Gesamtanzahl an erfassten Regeln wird bereits innerhalb der ersten zwölf Monate auf mindestens 1000 geschätzt. Bei einer solchen Größenordnung machen sich die Vorteile von Rule-Engines bei der Erfassung von Regeln, sowie bei Pflege und Fehlersuche bemerkbar. Andere Möglichkeiten, wie z.B. die Implementierung direkt im Quellcode, scheiden dadurch in der Praxis aus
- Die von Rule-Engines verwendeten Algorithmen zur Auswertung der Regeln sind sehr ausgereift und effizient, was sich ebenfalls bei größer werdenden Regelsets in der Laufzeit der Auswertungen stark bemerkbar macht (siehe Abschnitt 2.3.5)
- Mit Rule-Engines existieren ausgereifte Produkte am Markt, die die Verarbeitung von erstellen Regeln durchführen. Viele Aufgaben, die die Rule-Engine übernimmt, müssten ansonsten selbst entwickelt werden (siehe Kapitel 3)

Als Rule-Engine wurde das in Abschnitt 3.4 vorgestellte Produkt Drools gewählt. Dies wird folgendermaßen begründet:

- Da das bestehende System in Java entwickelt wurde und im Unternehmen hauptsächlich Java-Know-How vorhanden ist, sollte auch die eingesetzte Rule-Engine auf Java basieren
- Da große Anteile des Wissens durch Produktionsregeln erzeugt werden, muss das System die Vorwärtsverkettung beherrschen.
- Drools ist als Open-Source-Projekt kostenlos. Trotzdem konnte aufgrund des Bekanntheitsgrades sowie der großen Nutzergemeinde genug Vertrauen geschlossen werden. Die kürzliche Fusion von Drools mit „JBoss Inc.“ stützte diese Wahl, da schon einige Produkte dieses Unternehmens eingesetzt werden

Im Folgenden wird das prototypisch entwickelte Konzept vorgestellt und die herausgearbeiteten Probleme dargelegt.

6.3 Das Gesamtsystem in der Übersicht

Das System soll es ermöglichen, eine angelegte Regelbasis gegen eine Menge von vorhandenen Fakten zu prüfen, deren Ergebnis dann in verschiedenen Anwendungen zur Verfügung stehen soll. Damit Art und Anzahl dieser Anwendungen nicht beschränkt sind, soll innerhalb der regelverarbeitenden Komponente eine Schnittstelle zur Verfügung gestellt werden, auf die von

[2] Einige Richtlinien zur Entscheidungshilfe für oder gegen den Einsatz von Rule-Engines werden z.B. in [Rud05] genannt.

Anwendungen zugegriffen werden kann. Sowohl die Regeln, wie auch die regelverarbeitende Komponente sollen keine Kenntnisse über die Anwendungen besitzen. Abbildung 6.1 zeigt eine Skizze dieses Modells.

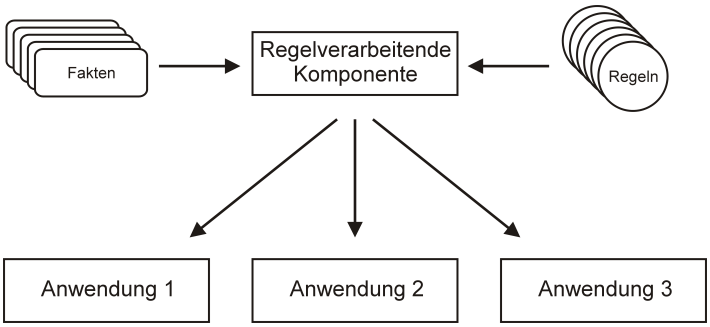


Abbildung 6.1: Modell des Gesamtsystems

Innerhalb der Evaluierungsphase des Projektes sollte beispielhaft mit zwei verschiedenen Anwendungen geplant werden: einer Integration in die Internetpräsenz sowie die Verwendung innerhalb einer internen Desktop-Anwendung. Entwurf und Realisierung der Anwendungen standen aber im Hintergrund.

Die Hauptquelle für die Fakten soll das existierende Warenwirtschafts-System „CCS“[3] sein. In bestimmten Anwendungen, wie z.B. im Web, soll allerdings noch eine zusätzliche Faktenquelle verwendet werden: individuelle Informationen des Kunden. D.h. es sollen, z.B. über Formulare im Web, Informationen des Kunden eingeholt und zur Faktenbasis hinzugefügt werden können.

Da die Regelbasis von Mitarbeitern bzw. den Fachabteilungen innerhalb des Unternehmens erfasst werden soll, können diese als verantwortliche Personen bzw. Akteure definiert werden.

Mit diesen zusätzlichen Informationen kann das Gesamtsystem zu dem in Abbildung 6.2 dargestellten Modell verfeinert werden. Im folgenden Abschnitt werden Ausschnitte dieses Modells detailliert erläutert.

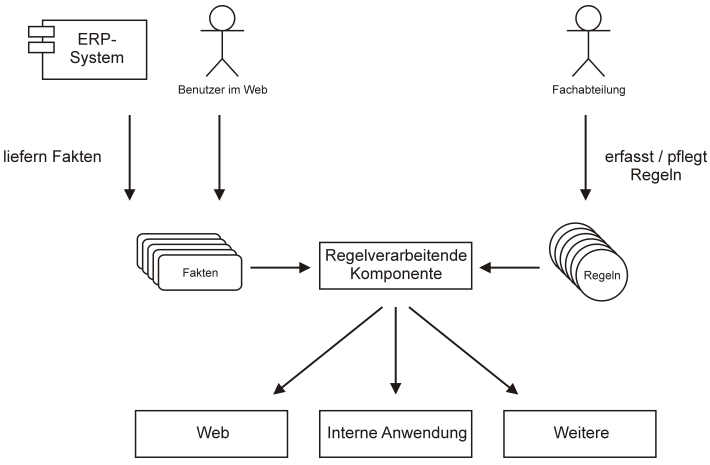


Abbildung 6.2: Verfeinertes Modell des Gesamtsystems

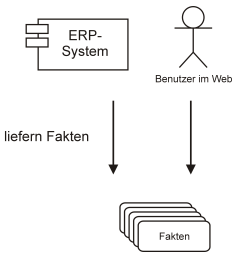
[3]CCS steht für *„Computation Combined Services“* und ist eine Eigenentwicklung der Schwesterfirma *camunda* (*http://www.camunda.com*). Es wird im folgenden auch abkürzend als das „ERP-System“ bezeichnet.

6.4 Detaillierte Betrachtungen

6.4.1 Die Regelverarbeitung

Die Regelverarbeitung an sich stellt ein Problem dar, das mit den zum aktuellen Zeitpunkt vorhandenen Technologien, Produkten und Erfahrungen recht einfach gelöst werden kann. Die gewählte Rule-Engine Drools kann Fakten in Form von Java-Objekten problemlos verarbeiten. In dieser Komponente konnten keine kritischen Probleme identifiziert werden.

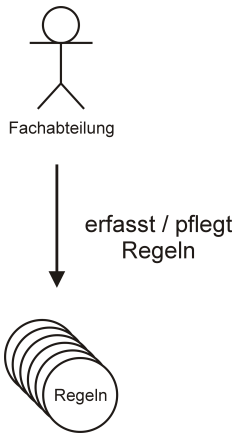
6.4.2 Die Faktenquellen



Die Bereitstellung von Fakten aus dem in Java programmierten ERP-System kann ebenfalls als Standardproblem bezeichnet werden. Es existieren Schnittstellen zur Abfrage aller benötigten Daten aus dem System (im ersten Schritt werden hauptsächlich Artikeldaten verarbeitet) und die Integration von ERP-System und Rule-Engine kann mit Hilfe der in der Arbeit vorgestellten Grundlagenm, sowie der in Kapitel 5 dargestellten Einsatzszenarien, durchgeführt werden. Allerdings wurde der wirkliche Zugriff auf Artikeldaten des ERP-Systems innerhalb des Prototypen nicht umgesetzt. Da das System schon seit längerem produktiv eingesetzt wird, musste die Machbarkeit dieses Zugriffs nicht mehr sicher gestellt werden. In den Prototypen wird mit Dummy-Artikelobjekten gearbeitet, die allerdings auf den realen Geschäftsobjekten basieren.

Die Einbindung von Benutzerinformationen in die Faktenbasis wird mit Hilfe der in den Abschnitten 6.5.2 und 6.5.3 vorgestellten Methoden *„UserFact“* und *„Sub-Goaling“* durchgeführt.

6.4.3 Die Regelerfassung

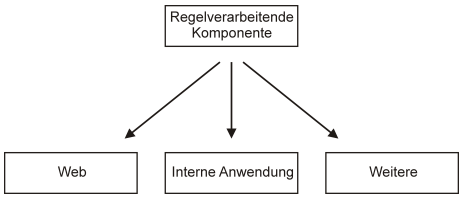


Auch bei der Regelerfassung konnten keine kritischen Probleme identifiziert werden. Zum momentanen Zeitpunkt wurde auf die Entwicklung einer eigenen Fachsprache verzichtet, da die reine Umsetzbarkeit aller geforderten Probleme im Vordergrund stand. Daher werden beim Prototyp die Regeln im Drools-Standard mittels XML-Dateien gepflegt. Auf Fakten, die aus dem ERP-System stammen, wird direkt über die jeweiligen Geschäftsobjekte zugegriffen. Da hier gute Dokumentationen (UML-Modelle und JavaDocs) vorhanden sind, ist das Deklarieren dieser Zugriffe problemlos möglich.

Allerdings betreffen folgende Punkte indirekt auch die Regelerfassung: Die Schnittstelle zwischen Regeln und Anwendungen wird über den in Abschnitt 6.5.1 erläuterten „Feedback-Mechanismus"durchgeführt. Dieser muss bei der Deklaration der Regeln berücksichtigt werden. Die Benutzung der *UserFacts* sowie des *Sub-Goalings* muss ebenfalls bei der Regelerfassung berücksichtigt werden.

6.4.4 Die Anwendungsschnittstellen

Durch die Anforderung der anwendungsunabhängigen Regelverarbeitung wurde der *„Feedback-Mechanismus“* (siehe Abschnitt 6.5.1) entwickelt, der es ermöglicht, Schnittstellen zwischen den Regeln und den Anwendungen einzuführen.



6.4.5 Zusammenfassung

Bei der Analyse des Gesamtsystems sind zwei kritische Probleme identifiziert worden:

- Schnittstelle zwischen Regeln und Anwendungen.
- Verwendung von Benutzerinformationen bei der Regelverarbeitung.

Die Lösung dieser Probleme wird innerhalb des nächsten Abschnittes erläutert. Bei der Analyse der Komponente „UserFact“, die das Problem der Einbeziehung von Benutzerinformationen löst, ist ein weiteres Problem identifiziert worden, das mittels des sog. „Sub-Goaling“ (siehe Abschnitt 6.5.3) gelöst wird. Diese Technik wird ebenfalls erläutert. Die Lösungen dieser drei Probleme wird in diesem Kapitel in den Vordergrund gestellt, da sie innovative Lösungen für aufgetretene Probleme darstellen. Da die restlichen Anforderungen relativ problemlos umgesetzt werden können, werden sie nur kurz bei der Beschreibung des zweiten Prototyps (siehe Abschnitt 6.6.2) erläutert.

6.5 Eigene Implementierungen

In diesem Abschnitt werden die Lösungen für die drei identifizierten Probleme vorgestellt. Deren Einsatz wird innerhalb der beiden Prototypen an praktischen Beispielen demonstriert.

6.5.1 Feedback-Mechanismus zur allgemeinen Verwendung der Regeln

Das Problem aus Sicht der Regeln

Gegeben sei z.B. eine Regel, die besagt, dass eine Grafikkarte X und ein Grafikkartenlüfter Y nicht kompatibel sind. Wenn diese Regel nur für eine Anwendung deklariert wird, ist die konkrete Aktion für diese Anwendung in der Regel enthalten. Im Beispiel soll die Regel dafür sorgen, dass auf der Homepage eine Warnmeldung angezeigt wird (z.B. sobald beide Produkte im Warenkorb liegen). Dann würde die Regel etwa folgendermaßen aussehen:

„Wenn Grafikkarte = 'X' und Lüfter = 'Y' ist, dann zeige auf der Homepage die Warnmeldung 'Diese beiden Produkte sind nicht kompatibel!' an!“

Wenn nun diese Regel in einem anderen Kontext eine andere Aktion ausführen soll, ergeben sich zwei Möglichkeiten:

1. Eine zweite Regel mit den gleichen Bedingungen wird erstellt, die aber eine andere Aktion ausführt. In diesem Fall müssten aber an sich gleiche Regeln mehrfach gepflegt werden. Durch die entstandene Redundanz innerhalb der Regelbasis würden weitere Schwierigkeiten auftauchen.
2. Die Regel wird so modifiziert, dass sie noch keine konkrete Aktion ausführt, sondern nur den Sachverhalt feststellt, der dann je nach Kontext zur Ausführung der jeweiligen Aktion führen kann. Diese Möglichkeit soll im Folgenden erläutert werden.

Die oben genannte Regel wird zwischen Bedingung und Konsequenz durch eine Indirektion erweitert, die beide Teile der Regel voneinander entkoppelt. Umgangssprachlich könnte diese Maßnahme wie folgt ausgedrückt werden:

1. *„Wenn Grafikkarte = 'X' und Lüfter = 'Y' ist, dann gilt: „X und Y sind nicht kompatibel!“*

2. *„Wenn gilt, dass X und Y nicht kompatibel sind, dann zeige die Warnmeldung 'Diese beiden Produkte sind nicht kompatibel!' an!“*

Somit führt Regel 1 noch keine direkte Aktion aus, sondern erzeugt eine neue Information bzw. einen neuen Fakt. Dieser Fakt kann dann als Eingabe für die zweite Regel verwendet werden, die diese Aktion ausführt. Somit könnten nun beliebig viele weitere Regeln den Fakt *„X und Y sind nicht kompatibel“* als Eingabe verwenden. Die erste Regel ist damit unabhängig von Ihrem Verwendungszweck und die erzeugte Information kann beliebig eingesetzt werden.

Es lässt sich noch eine weitere Feststellung machen: Die ursprüngliche Regel war nach der in Kapitel 4 vorgestellten Kategorisierung eine *Aktionsregel*. Nach der Entkopplung der beiden Regeln stellt die erste Regel nun eine *Produktionsregel* dar, während erst die zweite Regel eine *Aktionsregel* ist. Die Produktionsregeln enthalten also das Expertenwissen in einer reinen Form, während die Aktionsregeln das Kontrollwissen für den speziellen Anwendungsbereich enthalten.

Die Schnittstelle zwischen der eigentlichen wissenserzeugenden Regelbasis und der Anwendung befindet sich also genau in dieser neu erzeugten Information. Zu diesem Zweck wird das Problem im nächsten Abschnitt nochmals aus architektonischer Sicht betrachtet und ein Datencontainer entwickelt, der die Zwischenspeicherung der Information und gleichzeitig auch die Rolle der Schnittstelle übernimmt.

Das Problem aus Sicht der Architektur

Werden klassische Software-Komponenten betrachtet, lässt sich die Strategie der Komponentenbildung auf einen wesentlichen Bestandteil reduzieren: Eine Komponente ist eine Sammlung von Klassen in einer bestimmten Programmiersprache - hier Java - die nach außen hin eine definierte Schnittstelle anbietet. Diese Schnittstelle besteht ebenfalls aus einer oder mehreren Klassen, die, gegeben durch die Prinzipien der Objektorientierung, Schnittstellen in Form von öffentlichen Methoden anbieten können. So können, unter Beachtung bestimmter Richtlinien, Software-Komponenten mit sauberen Schnittstellen entworfen werden.

Allerdings sollen im Projekt keine Software-Komponenten entwickelt werden: es sollen anderen Anwendungen keine Java-Klassen bzw. -Komponenten zur Verfügung gestellt werden, sondern ein Set von Regeln. Diese Regeln sind in einer durch die Rule-Engine bestimmten Syntax deklariert und werden durch die Rule-Engine verarbeitet. Die Rule-Engine selbst führt diese Verarbeitung generisch durch und kennt dabei keine fachlichen Zusammenhänge. Somit ist die Rule-Engine auch nicht in der Lage, eine fachliche Schnittstelle für andere Anwendungen zur Verfügung stellen zu können. Die Regeln selbst können dies auch nicht, da sie, wie schon erwähnt, keine Java-Klassen sind.

Die Lösung dieses Problem wird dadurch erreicht, dass ein Datencontainer modelliert wird, der die generierten Informationen aufnimmt. Dieser Container wird nach fachlichen Anforderungen standardisiert und als Java-Interface definiert. Hier ist zwar die Erstellung von Java-Code notwendig, allerdings nur während der Defintion von Schnittstellen. Diese Schnittstellen ändern sich im produktiven Einsatz nur noch sehr selten im Vergleich zu den Regeln, die auf die Schnittstellen zugreifen.

Die Regeln generieren neue Informationen dadurch, dass sie auf dieses Interface zugreifen. Dafür muss sich das Interface, genauer gesagt eine Implementierung dieses Interfaces, im Working Memory befinden.

Da die Verarbeitung der Regeln nicht zentral abläuft, sondern von jeder Anwendung selbst geregelt wird, muss die Anwendung selbst eine Implementierung bereit stellen und dem Working Memory hinzufügen. Dadurch wird erreicht, dass jede Anwendung die erzeugten Informationen individuell verarbeiten kann.

Entfernt erinnert dieses Verhalten an das *Observer*-Muster der Softwaretechnik. Bei diesem Muster registriert sich ein Client bei einer Klasse, von der sie Informationen erhalten möchte, dem Observer, und implementiert für diesen Zweck ein standardisiertes Interface, über das der Observer auf den Client zugreift. Der Client greift also nicht auf den Observer zu, sondern umgekehrt.[4] Beim hier realisierten Feedback-Mechanismus registriert sich die Anwendung (als Client) bei den Regeln (dem Observer) dadurch, dass sie die Implementierung des Interfaces im Working Memory bereit stellt.

Dieses Vorgehen erscheint auch unter einem weiteren Gesichtspunkt als logisch: Auf die Regeln selbst kann von außen nicht zugegriffen werden, da sie keine Java-Klassen, sondern „nur“ Entscheidungslogik in abstrakter Form darstellen, die von der Rule-Engine interpretiert wird. Regeln können allerdings, wenn sie feuern, durch Aktionen selbst auf andere Objekte zugreifen. Somit kann der Informationsfluss zwischen Regeln und Anwendung nur von den Regeln in Richtung der Anwendungen laufen. Dies erinnert wieder an den Informationsfluss unter Einsatz des Observer-Musters.

Durch diesen Mechanismus kann das Interface als wohl definierte fachliche Schnittstelle zwischen Regeln und Anwendungen verwendet werden. Dabei ist es sinnvoll, Interfaces auch auf fachlicher Ebene zu unterscheiden und eigene Interfaces für jeden fachlichen Bereich, hier Problembereich genannt, zu definieren.

Während des Projektes wurde prototypisch das Interface für Kompatibilitätsdaten, das sog. `DreamCompatibilityOutput`-Interface[5], definiert (siehe Abbildung 6.3) und innerhalb der Prototypen implementiert. Zusätzlich implementieren alle Output-Interfaces das Marker-Interface `DreamOutput`.

Einsatz in der Praxis - Entwicklung eines neuen Problembereichs

Jeder Problembereich besteht aus einem Regelset und einem zugehörigen Output-Interface. Innerhalb der Konsequenzen der Regeln wird nun lediglich auf das eigene Output-Interface des Problembereichs zugegriffen. Eine solche Regel sieht dann z.B. folgendermaßen aus:

```
<application-data identifier="compatibilityOutput">
  com.camunda.rules.prototype.DreamCompatibilityOutput
</application-data>

<rule name="Beispielregel">

  <parameter identifier="graphicscard">
      <class>DreamGraphicscard</class>
  </parameter>
```

[4] Hier ist nur die Tatsache des Registrierens beim Observer und das Prinzip des umgekehrten Informationsflusses relevant. Andere Eigenschaften des Musters sollen nicht berücksichtigt werden, vor allem, dass sich mehrere Client gleichzeitig beim selben Observer registrieren können. Für mehr Informationen zum Observer-Pattern siehe [Vli96].

[5] Das Präfix *„Dream“* entstammt der unternehmensinternen Projektbezeichnung *„mad-moxx dream“*.

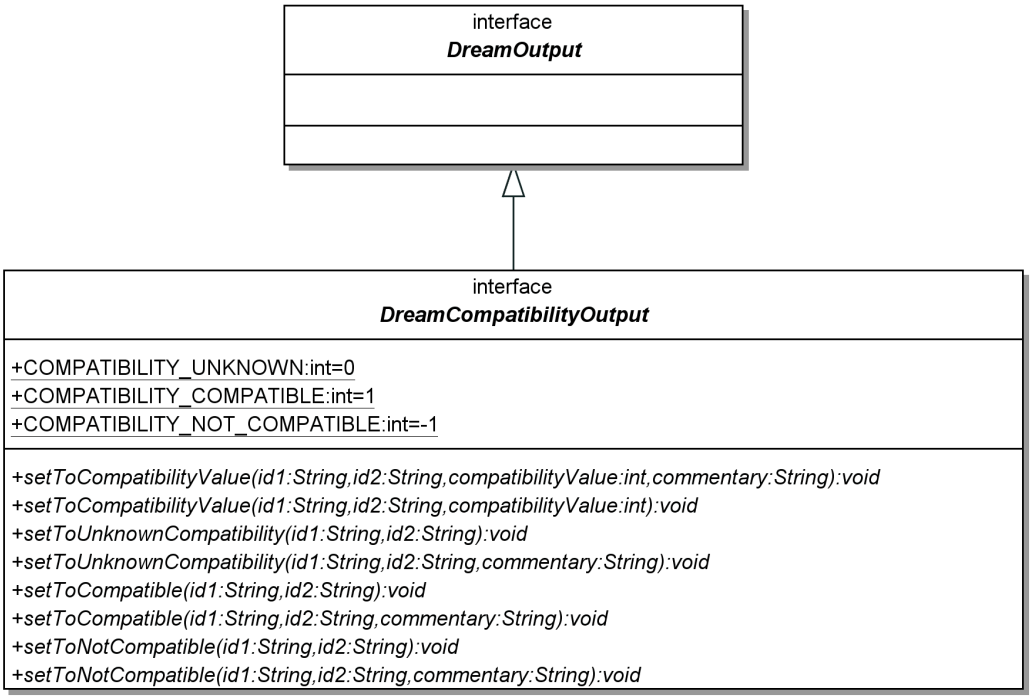


Abbildung 6.3: Klassendiagramm DreamOutput

```
  <parameter identifier="cooler">
      <class>DreamCooler</class>
  </parameter>


  <java:condition>
    graphicscard.getArticle().getId().equals("1005.10179")
  </java:condition>

  <java:condition>
    cooler.getArticle().getId().equals("1005.10065")
  </java:condition>

  <java:consequence>
    compatibilityOutput.setToCompatible(
      graphicscard.getArticle().getId(),
      cooler.getArticle().getId(),
      "Arctic Cooling ATI-Silencer 3 ist kompatibel zu Sapphire 9800 XT AGP"
    );
  </java:consequence>

</rule>
```

Hier wird z.B. eine explizite Kompatibilität der beiden Produkte über die angegebene Produkt-ID definiert und durch die Konsequenz der Regel an das *CompatibilityOutput*-Objekt übergeben.

Innerhalb der Regeln wurde das allgemeine *CompatibilityOutput*-Interface durch den Tag `<application-data>` erreicht, da dies der von Drools vorgesehene Mechanismus zur Bekanntmachung solcher Objekte darstellt.[6]

Einsatz in der Praxis - Verwendung eines Problembereichs

Jede Anwendung, die diesen Problembereich integrieren möchte, muss eine geeignete Implementierung des `CompatibilityOutput`-Interfaces bereit stellen. Auf Rule-Engine-Ebene wird die konkrete Implementierung dann folgendermaßen übergeben:

```
workingMemory.setApplicationData(

  "compatibilityOutput",
  concreteDreamOutputImplementation

);
```

Zusammenfassung

Durch den hier beschriebenen Feedback-Mechanismus wird eine vollständige Entkopplung der Regeln von ihrem späteren Anwendungsbereich ermöglicht. Das bei Regeldeklarationen in der Praxis oft verwendete Paradigma der direkten Manipulation der Anwendungslogik wird dabei vollständig vermieden. Das Verhalten der dadurch entwickelten Problembereichs-Komponenten ist vollständig anwendungsneutral. Mit dieser Lösung können beliebige Java-Anwendungen auf die Resultate der Regelverarbeitung zugreifen, eine ebenfalls regelbasierte weitere Verarbeitung ist nicht notwendig. Die Anforderung, erfasstes Regelwissen möglichst universell einsetzen zu können, ist damit erfüllt.

Dennoch bleibt ein Problem offen: Um die Informationen des Problembereichs verwenden zu können, muss jede Anwendung eine Interface-Implementierung bereit stellen. Hier würde also ein Bruch mit dem deklarativen Ansatz entstehen, da doch wieder Java-Code geschrieben werden müsste. Allerdings liegt der Fokus in diesem Projekt an anderer Stelle: Hier wird davon ausgegangen, dass die Wissensbasis das Element ist, das sehr aufwändig zu erfassen für das Unternehmen enorm wichtig ist. Daher soll sie auch möglichst flexibel nutzbar sein, um die Einsatzmöglichkeiten nicht zu beschränken. Dennoch sind Möglichkeiten denkbar, den erwähnten Bruch mit dem deklarativen Ansatz zu vermeiden (siehe Ausblick in Abschnitt 7.2).

6.5.2 UserFact-Komponente zur deklarativen Pflege von Datenstrukturen

Beschreibung

In Abschnitt 4.2.3 wurden Regelnotationen verschiedener Produkte erläutert und verglichen. Dabei wurde die Eigenschaft der Notation von Jess hervorgehoben, die außer den Regeln auch die deklarative Beschreibung von Datenstrukturen ermöglicht. Im Gegensatz zu den meisten anderen Rule-Engines, die als Faktenbasis immer Objekte (bzw. allgemein: Daten) der einbettenden Anwendung benötigen, besteht hier also die Möglichkeit, auch bei der Definition von Datenstrukturen von der Vorteilen den deklarativen Programmierung zu profitieren.

Im Praxisbeispiel wurde nun das Problem identifiziert, dass außer den im Java-Geschäftsmodell vorhandenen Daten (z.B. Geschäftsobjekte des Artikelstamms) auch Informationen

[6] Solche syntaktischen Besonderheiten von Drools sollen hier nicht erläutert werden.

zur Faktenbasis hinzugefügt werden sollten, die nicht innerhalb der Anwendung verfügbar sind: individuelle Informationen des Kunden. So sollte es auch möglich sein, Regeln zu deklarieren, die Informationen des Kunden verwenden, um gewisse Aktionen durchzuführen. Wenn der Nutzer z.B. eingibt, welche Grafikschnittstelle sein Mainboard besitzt, könnte eine Regel veranlassen, ihm passende Grafikkarten zu empfehlen oder eine Warnung anzuzeigen, dass die aktuell ausgewählte Grafikkarte nicht kompatibel zu seinem Mainboard ist. Um eine Regel in Drools deklarieren zu können, die eine Information des Kunden innerhalb der Bedingung verwendet, muss auch eine entsprechende Repräsentation dieser Information als Java-Objekt vorhanden sein:

```
<java:condition>   userInformationObject.getValue() == xy </java:condition>
<java:consequence> ...                                    </java:consequence>
```

Diese hier abstrakt als `UserInformationObject` bezeichnete Repräsentation von Kundeninformationen wurde im Projekt durch eine Komponente realisiert, die als ***UserFact*** bezeichnet wird. Sie stellt Funktionalitäten bereit, um neue Informationen, die erfragt werden sollen, anzulegen und um sie innerhalb der Regeln verwenden zu können.

Ein *UserFact* besteht dabei hauptsächlich aus zwei Elementen: einem Namen und einem Wert:

- **Name** Der Name identifiziert die Informationen aus semantischer bzw. fachlicher Sicht, z.B. „Grafikschnittstelle des Mainboards“.
- **Wert** Der Wert repräsentiert die Eingabe des Kunden, z.B. „AGP“ oder „PCI-Express“. Wie später noch erläutert wird, kann der *UserFact* auch einen leeren Wert besitzen, z.B. wenn noch nichts eingegeben wurde.

Damit der Kunde Informationen angeben kann, müssen *UserFacts* auf der Oberfläche dargestellt werden. Die Art der Darstellung ist prinzipiell unabhängig von der Implementierung der Java-Komponente. Eine beispielhafte Darstellung auf einer Weboberfläche zeigt Abbildung 6.4.

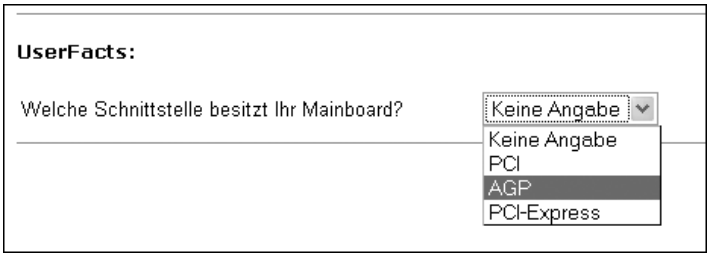


Abbildung 6.4: Screenshot UserFact im Web

Beispiel Es soll sichergestellt sein, dass die Grafik-Schnittstelle des Mainboards des Benutzers mit der Schnittstelle der gewünschten Grafikkarte übereinstimmt, ansonsten soll eine Warnmeldung angezeigt werden. Die entsprechende Regel lautet:

> *„Wenn die Schnittstelle der Grafikkarte ungleich der Schnittstelle des Mainboards des Kunden ist, dann zeige die Warnung 'Schnittstellen müssen übereinstimmen!' an!“*

Die Schnittstelle der Grafikkarte ist das Resultat einer Methode des Geschäftsobjektes *„Grafikkarte“* (z.B. `grafikkarte.getSchnittstellentyp()`), die Schnittstelle des Mainboards soll durch einen *UserFact* repräsentiert sein. Wenn es nun möglich sein soll, dass der Kunde die Eingabe der Information ignorieren kann, dann muss die Regel noch um eine weitere Bedingung erweitert werden:

„Wenn der UserFact 'Mainboard-Schnittstelle' bekannt ist und die Schnittstelle der Grafikkarte ungleich dem Wert des UserFacts 'Mainboard-Schnittstelle' ist, dann zeige die Warnung 'Schnittstellen müssen übereinstimmen!' an!“

Damit können *UserFacts* prinzipiell wie alle anderen Datentypen in Regeln verwendet werden. Im folgenden Abschnitt wird die Implementierung der UserFact-Komponente erläutert.

Implementierung

Da die Regeln in der Notation von Drools Java-basiert erfasst werden, müssen auch die *UserFacts* Java-Objekte sein. Dabei bestanden folgende Anforderungen an die Realisierung der *UserFacts*:

- Die Erstellung der *UserFacts* sollte nicht in der Anwendung, sondern innerhalb der Regeln erfolgen. Der sonst gewählte Weg, dass die Anwendung z.B. Artikelobjekte zum Working Memory hinzufügt, war hier also nicht möglich.
- Um die Erfassung der Regeln möglichst einfach zu gestalten, sollte über den Namen auf die einzelnen Facts zugegriffen werden.
- Es sollten verschiedene Typen von Facts existieren, die als Werte nur jeweils z.B. Zahlen, Zahlenbereiche, Strings oder eine Auswahl von Strings zulassen. Trotzdem sollte auch typübergreifend sicher gestellt sein, dass der Name des UserFacts eindeutig ist. Somit war auch die Modellierung der Facts als einzelne Objekte im Working Memory nicht mehr praktikabel, es musste eine Klasse entwickelt werden, die den Zugriff auf die einzelnen *UserFact*-Objekte regelt

Die schließlich entwickelte Komponente erfüllt die gestellten Anforderungen. Mit ihr können zur Laufzeit verschiedene Typen von *UserFacts* eingeführt und deren Werte abgefragt werden. Dazu existieren zwei wichtige Klassen (siehe auch Klassendiagramm in Abbildung 6.5):

- **UserFactManager** Der `UserFactManager` ist globaler Stellvertreter zur Erzeugung und Abfrage einzelner Facts. Er hält dazu eine Liste der UserFacts und stellt sicher, das Namen eindeutig vergeben werden. Über ihn wird auch innerhalb der Regeln auf die UserFacts zugegriffen.
- **FactObject** Die Klasse `FactObject` ist die abstrakte Superklasse für die eigentlichen, typisierten UserFacts. Ihre Schnittstelle enthält Methoden, um die Verfügbarkeit und den Namen Typ-unabhängig ermitteln zu können. Dies vereinfacht die Verwendung der UserFacts innerhalb der Regeln. Prototypisch wurden die folgenden typisierten UserFacts bisher implementiert: `StringFactObject`, `IntegerFactObject` und `String EnumerationFactObject`. Diese Objekte besitzen eine Methode `getValue()`, die den Wert des Facts im entsprechenden Typ zurückliefern. Über diese Methode wird auch in den Regeln zugegriffen.

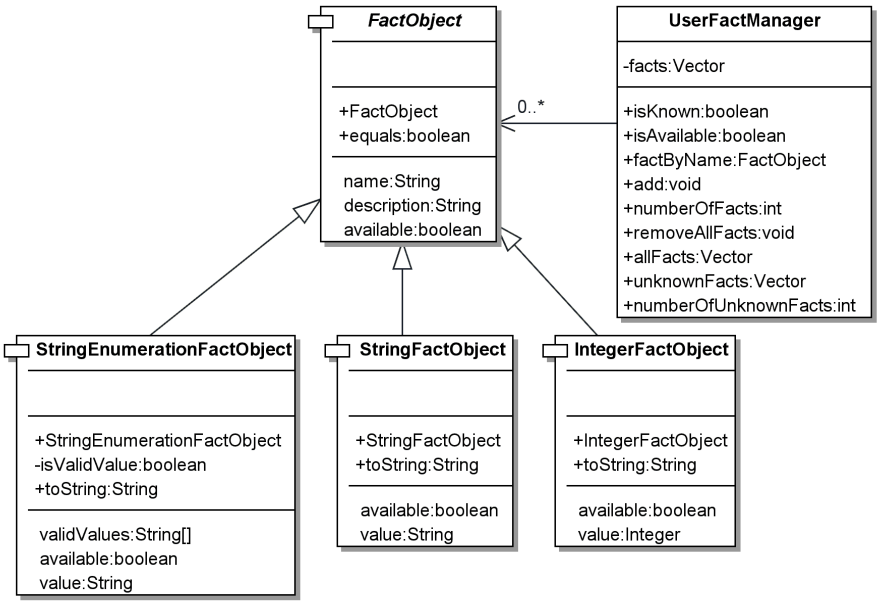


Abbildung 6.5: Klassendiagramm Komponente 'UserFact'

Die oben genannte Regeln kann nun mit Hilfe der UserFact-Komponente folgendermaßen formuliert werden:

```
<!-- "Wenn der UserFact 'Mainboard-Schnittstelle' bekannt ist..." -->

<java:condition>
  userFactManager.factByName("Mainboard-Schnittstelle").isAvailable()
</java:condition>



<!-- "...und die Schnittstelle der Grafikkarte ungleich dem Wert des
      UserFacts 'Mainboard-Schnittstelle' ist..." -->

<java:condition>
  graphicscard.getBusInterfaceType().equals(
    ((StringUserFact)userFactManager.
      factByName("Mainboard-Schnittstelle"))).getValue() ) == false
</java:condition>



<!-- "...zeige die Warnung 'Schnittstellen sollten übereinstimmen!' an!" -->

<java:consequence>
  output.addWarning( "Schnittstellen sollten übereinstimmen!" );
</java:consequence>
```

Typen von UserFacts

Die Klasse `FactObject` stellt die abstrakte Oberklasse für konkrete typisierte FactObject-Klassen dar. Prototypisch sind drei verschiedene Typen implementiert worden:

StringUserFact Bei diesem Typ sind freie Texteingaben möglich

IntegerUserFact Hier ist die Eingabe von Zahlen bzw. Ziffern möglich

StringEnumerationUserFact Hier ist die Angabe einer Menge von Werten möglich, die als gültige Werte anerkannt werden

Wie diese Typen auf einer Oberfläche angezeigt werden können und welche Unterschiede es bei der Anzeige zwischen den Typen gibt, wird bei der Beschreibung des zweiten Prototyps (siehe Abschnitt 6.6.2) erläutert.

Erzeugung der UserFacts in der Praxis

Damit zur Laufzeit die benötigten *UserFact*-Objekt bereit stehen, müssen sie wie jedes Java-Objekt erzeugt werden. Zusätzlich müssen sie dem `UserFactManager` übergeben werden, damit innerhalb der Regeln darauf zugegriffen werden kann. Dazu besitzt die Klasse eine Methode `add( FactObject )`, die neue Facts entgegennimmt:

```
userFactManager.add( new StringFactObject( "Mainboard-Schnittstelle" ) );

// Zugriff dann möglich per:
userFactManager.factByName( "Mainboard-Schnittstelle" ).doSomething();
```

Eine einfache Möglichkeit wäre, zu Beginn alle benötigten *UserFacts* auf diese Art und Weise zu erzeugen und hinzuzufügen. Dabei tauchen allerdings in der Praxis einige Probleme auf. Diese Probleme werden durch das *„Sub-Goaling“* gelöst, das im folgenden Abschnitt beschrieben wird.

6.5.3 Sub-Goaling zur Simulation der Rückwärtsverkettung

Erläuterung

Im vorherigen Abschnitt wurden die sog. *UserFacts* erläutert. Sie stellen Fakten dar, die vom Benutzer erfragt werden, um gewisse Sachverhalte klären zu können. Wenn z.B. geprüft werden soll, ob die Schnittstelle auf dem Mainboard des Kunden zur Schnittstelle der ausgewählten Grafikkarte kompatibel ist, muss diese Information natürlich vom Kunden erfragt werden, sie kann im System nicht bekannt sein. Je nach Einsatzgebiet und nach Größe des Regelsets können so recht schnell sehr viele Fakten vorhanden sein, die erfragt werden müssen.

Dies hätte zum einen den Nachteil, dass der Kunde sehr viele Informationen angeben muss, die evtl. aber gar nicht relevant werden, oder die er nur schwer beantworten kann. Deshalb wurden die Regeln schon so formuliert, dass sie nur feuern, wenn auch wirklich eine Eingabe erfolgt ist. Der Benutzer kann also *UserFacts* ignorieren und das System läuft dennoch weiter. Allerdings kann das System dann keine Aussage über den jeweiligen Sachverhalt treffen, die Information ist also weniger genau.

Doch auch wenn Angaben vom Benutzer ignoriert werden können, kann es als störend empfunden werden, wenn eine lange Liste von Informationen angezeigt wird, für die er Eingaben tätigen kann. Hier sollte vom System geprüft werden, zu welchem Zeitpunkt die Erzeugung

eines *UserFacts* sinnvoll sein könnte und nur dann sollte die Aufforderung zur Eingabe des UserFacts angezeigt werden. Die Erzeugung eines *UserFacts* wird im folgenden auch als „Einführung“ bezeichnet, da so der sehr technische Aspekt einer „Objekt-Erzeugung“ etwas in den Hintergrund gerückt wird.

Dieser Zeitpunkt tritt dann ein, wenn alle anderen Prämissen außer dem *UserFact* schon erfüllt sind. Dann fehlt zur vollständigen Erfüllung der Bedingungen einer Regel nur noch die Eingabe des *UserFacts*, um dann eine Aussage treffen zu können. *UserFacts* sollten also nur dann aktiv werden, wenn sie wirklich zur Prüfung von Regeln, deren andere Prämissen erfüllt sind, benötigt werden. Dieses Verhalten entspricht der rückwärtsverkettenden Inferenz, die bekanntermaßen prüft, welche Teilbedingungen erfüllt werden müssen, damit alle Bedingungen erfüllt sind.

Allerdings beherrscht die verwendete Rule-Engine Drools nur die Vorwärtsverkettung. Deshalb wurde dieses Problem mittels einer Technik gelöst, die an eine von Charles Forgy erstmals vorgeschlagene Technik angelehnt ist und von ihm als als „Sub-Goaling“ bezeichnet wurde. [7]. Er erläutert, dass durch das Definieren von sog. „Sub-Goals“, also Zwischen-Zielen, das Verhalten von Rückwärtsverkettung in vorwärtsverkettenden Inferenzmaschinen simuliert werden kann. Ein Zwischenziel stellt dabei eine Bedingung dar, bei deren Eintreffen das eigentliche Ziel (hier die Einführung des *UserFacts*) erreicht wird. Besonderheit dabei ist, dass diese Bedingungen manuell definiert werden und nicht von der Rule-Engine automatisch erzeugt werden. Dabei besteht im Gegensatz zur Rückwärtsverkettung der Vorteil, Zwischenziele zu beliebigen Zeitpunkten einführen zu können. Dieser Zeitpunkt und die Art der praktischen Umsatzung dieses Prinzips werden im weiteren Verlauf erläutert.

„Wenn ich müde bin und es nach 21 Uhr ist, dann gehe ich ins Bett!“

Es wird angenommen, dass die Frage nach der Müdigkeit eine Frage an den Benutzer mittels eines *UserFacts* wäre. Da die komplette Bedingung der Regel erst dann wahr wird, wenn alle einzelnen Prämissen wahr sind, muss die Frage an den Benutzer auch erst dann gestellt werden, wenn alle anderen Bedingungen bereits erfüllt sind. In diesem Beispiel könnte also so lange gewartet werden, bis es nach 21 Uhr ist, da ansonsten die Regel auf keinen Fall feuern wird.

Ansonsten könnte es sein, dass der Benutzer gefragt wird und dessen Antwort auch für eine Erfüllung der betreffenden Teilbedingung sorgt, die gesamte Bedingung aber aufgrund der äußeren Umstände niemals wahr wird. Dann wäre auch die Frage an den Benutzer überflüssig gewesen. Um nun also möglichst wenige Fragen an den Benutzer stellen zu müssen, sollte sicher gestellt werden, dass *UserFacts* erst dann aktiv werden, wenn alle anderen Teilbedingungen erfüllt sind. Mehr noch: Da *UserFacts* in mehreren Regeln vorkommen können, muss geprüft werden, wann in mindestens einer der Regeln alle anderen Teilbedingungen eintreffen. Ist dies erfüllt, soll der *UserFact* eingeführt werden.

Nun ist es durchaus möglich, dass zu einem gewissen Zeitpunkt alle Teilbedingungen eingetroffen sind und der *UserFact* eingeführt wird, zu einem späteren Zeitpunkt aber nicht mehr alle anderen Teilbedingungen wahr sind. Da der *UserFact* dann aber schon sichtbar war und evtl. auch schon eine Eingabe vom Benutzer stattgefunden hat, sollte er auch weiterhin aktiv bleiben.

Wie kann diese Prüfung nun in der Praxis möglichst einfach erreicht werden? Und wie könnte sie möglichst deklarativ geregelt werden, ohne diese Logik in den Java-Quellcode der *UserFact*-Komponente verlagern zu müssen?

[7] siehe [For05].

Für eine einzelne Regeln wurde festgestellt, dass die Erfüllung aller anderen Teilbedingungen die Voraussetzung dafür ist, dass der *UserFact* eingeführt werden soll. Allgemein lässt sich also sagen, dass das Eintreffen aller Teilbedingungen **mindestens einer Regel**, in der der *UserFact* vorkommt, die Bedingung dafür ist, dass er eingeführt werden soll.

Dies wird erreicht, indem für jede einzelne Regel, in der der *UserFact* vorkommt, eine zweite Regel angelegt wird. Diese enthält alle Teilbedingungen der ersten Regel in unveränderter Form, außer der Teilbedingung, die den *UserFact* enthält. Dafür wird in der zweiten Regel eine Bedingung eingefügt, die die nicht-Existenz des *UserFacts* prüft. Somit existiert nun eine Regel, die den oben erläuterten Anforderungen entspricht: Sie prüft alle übrigen Teilbedingungen, und sobald alle erfüllt sind und der *UserFact* noch nicht existiert, wird er eingeführt. Dies wird am folgenden Beispiel nochmals verdeutlicht.

Beispiel

Die eigentliche Regel *„Wenn ich müde bin und es nach 21 Uhr ist, dann gehe ich ins Bett!“* wird somit erweitert auf folgende zwei Regeln:

Regel 1:

```
WENN
  es nach 21 Uhr ist
UND
  der UserFact 'Müdigkeit'
  schon eingeführt wurde
UND
  er müde ist

DANN
  soll er ins Bett gehen
```

Regel 2:

```
WENN
  es nach 21 Uhr ist
UND
  der UserFact 'Müdigkeit' noch
  nicht eingeführt wurde

DANN
  Führe UserFact 'Müdigkeit' ein!
```

Es ist leicht zu erkennen, dass die Erzeugung der zusätzlichen Sub-Goaling-Regeln einem strengen Formalismus unterliegt. Daher ist es naheliegend, diese Regeln automatisiert zu generieren. Dies wurde innerhalb dieser Arbeit nicht realisiert.

Nun kann folgender simulierter Ablauf des Dialogs mit dem Benutzer anhand der beiden Regeln verfolgt werden:

1. Zu Beginn ist der *UserFact* 'Müdigkeit' noch nicht eingeführt worden, also ist noch nichts über die Müdigkeit des Benutzers bekannt. So lange der *UserFact* also nicht eingeführt wird, wird Regel 1 niemals feuern können, denn die Teilbedingungen 2 und 3 werden niemals wahr werden. Teilbedingung 2 von Regel 2 allerdings ist unter diesen Umständen immer wahr

2. So lange es vor 21 Uhr ist, ist Teilbedingung 1 von Regel 2 (*„WENN es nach 21 Uhr ist...“*) unwahr

3. Sobald es nach 21 Uhr ist, wird diese Teilbedingung wahr. Da die zweite Teilbedingung von Regel 2 immer noch wahr ist, kann Regel 2 nun feuern. Dadurch wird nun der *UserFact* eingeführt und Teilbedingung 2 von Regel 1 (*„WENN der UserFact 'Müdigkeit' noch nicht eingeführt wurde“*) wird ebenfalls wahr

4. Zu diesem Zeitpunkt (es ist immer noch nach 21 Uhr!) ist auch Teilbedingung 1 wahr, also fehlt zum Feuern von Regel 1 nur noch die Erfüllung von Teilbedingung 3 (*„WENN er müde ist...“*)

5. Durch das Einführen des *UserFacts* erscheint beim Benutzer eine Eingabeaufforderung zur Angabe seines Müdigkeitszustandes

6. Sobald er hier angibt, dass er müde ist, ist auch die letzte Teilbedingung von Regel 1 (*„WENN er müde ist...“*) erfüllt, und die Aktion kann ausgeführt werden. Dabei ist zu beachten, dass der *UserFact* nicht unbedingt eingegeben werden muss. Allerdings wird die Regel nicht feuern, wenn nicht explizit angegeben wurde, dass er müde ist.[8] Sie wird ebenfalls nicht feuern, wenn der Benutzer bis zum nächsten Tag wartet und erst dann angibt, dass er müde ist, denn dann ist es wieder vor 21 Uhr und die erste Teilbedingung ist nicht mehr erfüllt

6.6 Die Prototypen

Zur Verifizierung des Konzepts sowie der entwickelten Komponenten wurden zwei Prototypen erstellt. Der erste ist ein einfaches Programm, das lediglich die grundsätzliche Funktionalität der regelverarbeitenden Komponente bzw. den Einsatz einer Rule-Engine darstellen soll. Da der Feedback-Mechanismus eine anwendungsunabhängige Verwendung der Wissensbasis adressiert, wird er im ersten Prototypen auch schon eingesetzt. So kann die selbe Wissensbasis für beide Prototypen verwendet werden. Der zweite Prototyp ist eine Simulation eines Kunden-Dialogs auf Basis von Java Swing. In diesem Prototyp ist eine Interaktion mit dem Benutzer möglich. Dadurch können die beiden anderen Komponenten, das Sub-Goaling und die *UserFacts*, demonstriert und getestet werden.

Der Quellcode sowie die Regeldateien der Prototypen befinden sich auf der beiliegenden CD und werden hier daher nicht abgedruckt.

6.6.1 Prototyp 1

Beschreibung

Dieser erste Prototyp demonstriert die grundsätzliche Funktionalität der Rule-Engine Drools anhand einiger Regeln sowie den Feedback-Mechanismus mittels einer beispielhaften Implementierung des DreamCompatibilityOutput-Interfaces.

Die Klasse `DreamCompatibilityShellOutput` stellt diese Beispielimplementierung des Interfaces `DreamCompatibilityOutput` dar. Sie nimmt die durch die Regeln erzeugte Kompatibilitätsinformationen auf und gibt sie auf der Kommandozeile aus. Dies hat den Effekt, dass die Regelverarbeitung direkt verfolgt werden kann. Eine produktive Anwendung würde diesen Output sinnvoller verarbeiten, dies soll aber nicht Ziel des Prototyps sein.

Der Prototyp selbst ist implementiert in der Klasse `Prototype1` (siehe Abbildung 6.6) und kann über die Methode main() gestartet werden. Es werden folgende Schritte ausgeführt:

1. Es wird eine RuleBase erzeugt, deren Regeln aus der Datei `\bachelor_prototype1.xml` geladen werden

2. Es wird ein neuer Working Memory erzeugt

[8] Dieses Verhalten kann verglichen werden mit der Theorie der *Closed World Assumption*, die in Abschnitt 2.6.3 beschrieben wurde.

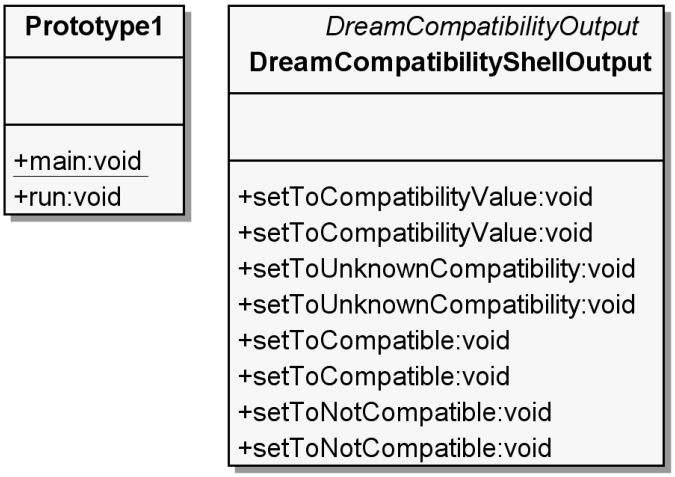


Abbildung 6.6: Klassendiagramm Prototyp 1

3. Es werden 17 Grafikkartenobjekte und 14 Grafikkartenlüfterobjekte zum Working Memory hinzugefügt
4. Es wird die erläuterte Beispielimplementierung `DreamCompatibilityShellOutput` hinzugefügt, die die Resultate der Regelverarbeitung aufnimmt
5. Schließlich wird die Regelverarbeitung gestartet

Die Ausgabe des Prototyps sieht dann etwa folgendermaßen aus: [9]

```
----- Running Prototype 1 -----

Loading rule base.

Firing all rules...

1005.10083 1005.10067 1   ATI-Silencer 1 ist kompatibel zu...
1005.10045 1005.10067 1   ATI-Silencer 1 ist kompatibel zu...
1005.10128 1005.10067 1   ATI-Silencer 1 ist kompatibel zu...
1005.10063 1005.10066 -1 ATI-Silencer 2 ist nicht kompatibel zu...

All rules fired.

Elapsed time: 0 seconds
```

Zusammenfassung

Mit diesem Prototyp konnte gezeigt werden, dass mittels der Technik der Regelverarbeitung auch komplexere Kompatibilitätsbeziehungen zwischen vielen verschiedenen Produkten abge-

[9] Aus Layoutgründen ist die Ausgabe gekürzt, sie ist allerdings in vollständiger Form auf der CD vorhanden.

bildet werden können. Außerdem konnte demonstriert werden, dass die Technik des Feedback-Mechanismus' das Ziel erreicht, Regeln unabhängig vom Verwendungszweck deklarieren zu können. Der Verwendungszweck wird erst durch die Anwendungs-spezifische Implementierung des entsprechenden Interfaces erreicht. Die weiteren Komponenten sowie eine komplexere Laufzeit-Demonstration bietet der nachfolgend beschriebene Prototyp 2.

6.6.2 Prototyp 2

Beschreibung

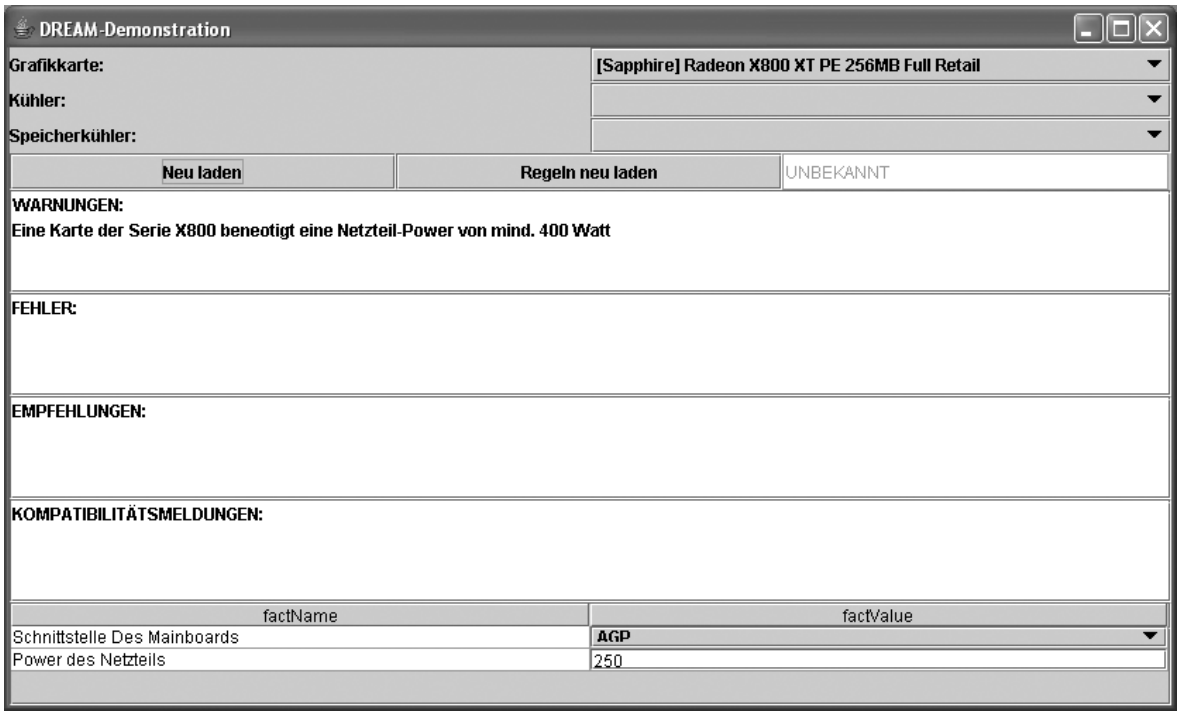


Abbildung 6.7: Screenshot Prototyp 2

Mit dem zweiten Prototyp sollte das Zusammenspiel aller Komponenten innerhalb einer realitätsnahen Anwendung demonstriert werden. Außer den schon im ersten Prototyp gezeigten Techniken sind hier noch die beiden Komponenten Sub-Goaling und *UserFact* integriert.

Der Prototyp stellt einen beispielhaften Dialog im Internet dar, in dem der Benutzer eine Grafikkarte und einen Grafikkartenlüfter aus einer Liste mehrerer Produkte auswählen kann, für die dann durch eine beispielhafte Regelbasis Informationen erzeugt und im Dialog angezeigt werden. Der Dialog soll einen Verkaufsberatungsdialog im Internet simulieren.

Die zentrale Rolle in der Anwendung spielt die Klasse `Dream`, die die Steuerung des Dialogablaufes übernimmt. Sie hält außerdem die ausgewählten Produkte, die Verbindungen zu den wissensverarbeitenden Komponenten, steuert den Dialog und übernimmt die Kommunikation mit Rule-Engine und Working Memory.

Der Dialog weist eine Besonderheit auf: Da Informationen nur zu den momentan ausgewählten Produkten angezeigt werden sollen, befinden sich auch nur diese Produkte im Working Memory. Bei jeder neuen Produktauswahl wird damit das bisherige Produkt des jeweiligen Typs durch das neu ausgewählte ausgetauscht. Außer den Artikel-Objekten befinden sich nur noch wenige Objekte im Working Memory, z.B. eine Instanz der Klasse `UserFactManager` oder eine Implementierung des `DreamCompatibilityOutput`-Interfaces.

In den folgenden Abschnitten zur weiteren Beschreibung des Prototyps liegt der Schwerpunkt auf dem konkreten Einsatz der in diesem Kapitel entwickelten Komponenten. Sonstige Gesichtspunkte des Dialogs, z.B. die Realisierung des Dialog mit Java Swing, werden nicht näher behandelt.

Ein UML-Diagramm des kompletten Prototyps befindet sich in Anhang B.

Der Feedback-Mechanismus in Aktion

Die Klasse `DreamOutput` verwaltet die kompletten Ausgaben, die im Dialog angezeigt werden. Zusätzlich implementiert sie das Interface `DreamCompatibilityOutput` und stellt somit die Schnittstelle zwischen der Regelbasis und dem Dialog dar. Eine Instanz dieser Klasse wird beim Start des Dialogs erzeugt. Ebenfalls beim Start wird in der Methode `createRuleBase()` die Rule-Engine initialisiert. Dort wird dem Working Memory auch die Interface-Implementierung hinzugefügt.

```
RuleBase rb =
  RuleBaseLoader.loadFromUrl(this.getClass().getResource( RULE_FILE ));

workingMemory = ruleBase.newWorkingMemory();

workingMemory.setApplicationData( "compatibilityOutput",
  (DreamCompatibilityOutput)dreamOutput );
```

Somit wird nun von allen feuernden Regeln dieses Problems das Objekt `dreamOutput` aufgerufen. Die dort implementierten Methoden des Interfaces regeln dann die Weiterverarbeitung der Informationen. Die hier gezeigte Methode wird z.B. von Regeln aufgerufen, die eine Inkompatibilität zwischen zwei Produkten feststellen.

```
public void setToNotCompatible( String id1,
                                String id2,
                                String commentary ){

  this.compatible = DreamCompatibilityOutput.COMPATIBILITY_NOT_COMPATIBLE;
  this.addCompatibility(commentary);
  publishToListeners();

}
```

Innerhalb dieser Methode wird folgendes ausgeführt:

- Sie setzt das interne Kompatibilitätsflag auf „Nicht kompatibel“
- Sie fügt den Kommentar, der durch die Regeln mit übergeben wird, in die Liste der aktuellen Meldungen ein. Diese Meldungen werden dann im Dialog angezeigt.
- Sie ruft die Methode `publishToListeners()` auf, die für eine Aktualisierung des entsprechenden GUI-Elementes sorgt. Diese Methode ist speziell für diesen Swing-Dialog notwendig und hat mit der eigentlichen fachlichen Informationsverarbeitung nichts zu tun. Sie zeigt aber, dass jede Anwendung auf eigene Art und Weise mit Resultaten der Regelverarbeitung umgeht bzw. auch wirklich umgehen kann.

Im Sinne des besseren Verständnisses soll hier noch ein Ausschnitt einer beispielhaften Regel gezeigt werden, die die oben dargestellte Methode aufruft, sobald sie feuert:

```
(...)

<java:consequence>

  compatibilityOutput.setToNotCompatible(
    graphicscard.getArticle().getId(),
    cooler.getArticle().getId(),
    "ATI-Silencer 1 ist nicht kompatibel zu kompatibel
                                    zu Variante XT der Serie 9800"
  );

</java:consequence>
```

Die UserFact-Komponente in Aktion

Die Klasse `Dream` erzeugt im Konstruktor einen `UserFactManager`, der die *UserFacts* des Dialogs verwaltet. Dies ist auch sinnvoll, da die *UserFacts* nur für die Dauer eines einzelnen Kunden gültig bleiben sollen. Beim nächsten Kunden sollen sie selbstverständlich wieder gelöscht werden. Dies wird auch hier erreicht, da für jeden (virtuellen) Kunden ein neues `Dream`-Objekt erzeugt wird.

Die grafische Realisierung im Dialog wurde durch eine Tabelle erreicht, die bei jedem neuen *UserFact* um eine Zeile erweitert wird (siehe Abbildung 6.8). Die Kopplung zwischen `UserFactManager` und Dialog wird durch die Klasse `UserFactsTableModel` erreicht, die eine Referenz auf den `UserFactManager` als *Data Model* hält.

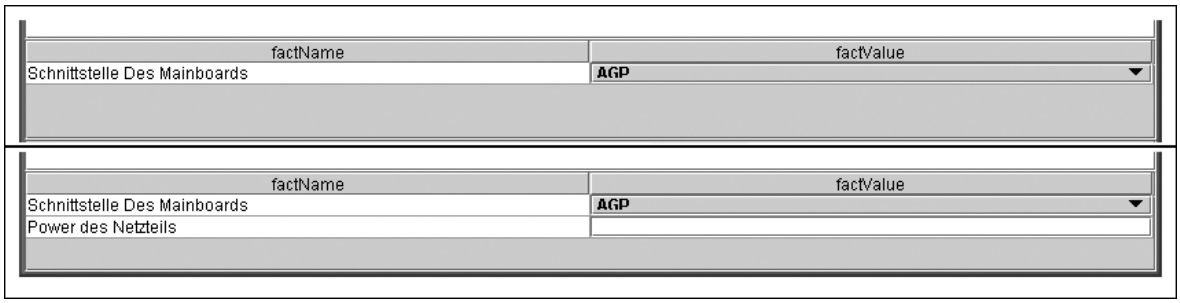


Abbildung 6.8: Screenshot UserFacts in Prototyp 2

Das Sub-Goaling in Aktion

Das Sub-Goaling wirkt sich auf die Regeln aus. In der Regelbasis des Prototyps wird anhand eines *UserFacts* die Leistung des Netzteils des Kunden abgefragt. Durch das Expertenwissen zu diesem *UserFact* können folgende umgangssprachliche Regeln aufgestellt werden:

- *„Für den Betrieb einer Grafikkarte mit dem Chipsatz „9800“ sollte ein Netzteil mit mindestens 350 Watt verwendet werden!“*
- *„Für den Betrieb einer Grafikkarte mit dem Chipsatz „X800“ sollte ein Netzteil mit mindestens 400 Watt verwendet werden!“*

- *„Bei Karten mit anderen Chipsätzen ist die Leistung des Netzteils nicht relevant!“*

Daraus kann gefolgert werden, dass der *UserFact „Leistung des Netzteils“* erst eingeführt werden muss, sobald die ausgewählte Grafikkarte entweder den Chipsatz „9800“ oder den Chipsatz „X800“ besitzt. Dementsprechend existieren auch diese zwei Sub-Goaling-Regeln:[10]

```
Regel 1:

<java:condition>
  userFactManager.isKnown("Leistung des Netzteils") == false
</java:condition>
<java:condition>
  graphicscard.(...).getChipsetFamilyKey().equals("60") == true
</java:condition>

<java:consequence>
  userFactManager.add( new IntegerFactObject("Leistung des Netzteils") );
</java:consequence>


Regel 2:

<java:condition>
  userFactManager.isKnown("Leistung des Netzteils") == false
</java:condition>
<java:condition>
  graphicscard.(...).getChipsetFamilyKey().equals("70") == true
</java:condition>

<java:consequence>
  userFactManager.add( new IntegerFactObject("Leistung des Netzteils") );
</java:consequence>
```

Durch diese Regeln wird zu gegebener Zeit der *UserFact* eingeführt. Da hier ein numerischer Wert, die Leistung des Netzteils in Watt, erfragt werden soll, wurde der Typ `IntegerFactObject` gewählt.

Zusammenfassung

Mit diesem Prototyp konnte gezeigt werden, dass die in diesem Kapitel entwickelten Komponenten auch im Kontext einer sinnvollen Anwendung funktionieren und verwertbare Resultate liefern. Dadurch, dass beide Prototypen die gleiche Regelbasis verwenden, kann auch gezeigt werden, dass die angestrebte Anwendungsunabhängigkeit tatsächlich erreicht wird. Das Sub-Goaling kann mit diesem Prototypen in einem realen Dialog getestet werden. So kann das Verhalten auch subjektiv beurteilt werden.

[10] Der Chipsatz wird über die Methode getChipsetFamilyKey() ermittelt. Ein Wert von „60“ entspricht dem Chipsatztyp „9800“, „70“ entspricht „X800. Die Regeln sind aus Gründen der Lesbarkeit gekürzt.

6.7 Fazit

Es hat sich gezeigt, dass mittels der in den ersten Kapiteln vermittelten Grundlagen, sowie einer der verfügbaren Rule-Engines, schnell und problemlos erste regelbasierte Testapplikationen entwickelt werden können. Sowohl die Erfassung der Regeln, als auch die Integration der Rule-Engine in eigene Anwendungen, stellen außer einer gewissen initialen Einarbeitungszeit keine nennenswerten Hürden dar.

Auch die Anforderungen des Projekts konnten mit Hilfe der vorgestellten Techniken und Komponenten prototypisch erfüllt werden.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Diese Arbeit gibt einen Überblick über die Grundlagen regelbasierter Systeme und deren Einbindung in moderne geschäftliche Anwendungen. Dabei werden vor allem die Regelinterpreter, die sog. „Rule-Engines“ sowie Geschäftsregeln erläutert. Schließlich wird anhand von verschiedenen Einsatzszenarien die Integration einer Rule-Engine in eigene Anwendungsarchitekturen demonstriert.

Zur Beantwortung der Frage, ob Rule-Engines zur Lösung eigener Probleme geeignet sind, muss geprüft werden, ob Anforderungen existieren, die sich zur Formalisierung mit Geschäftsregeln und damit zur Verarbeitung mit Rule-Engines eignen. Dies hängt im Wesentlichen davon ab, in welchem Maße die Anforderungen von Einflussfaktoren des aktuellen Kontexts abhängen, d.h. wie hoch der Anteil an Entscheidungen innerhalb der eigenen Geschäftslogik ist. Dabei hat sich gezeigt, dass viele Probleme und Anforderungen im Umfeld geschäftlicher Anwendungen regelbasiert ausgedrückt werden können. Mehr noch, häufig liegen Probleme von Natur aus in einer regelbasierten Form vor und können durch Regeln intuitiver und effizienter umgesetzt werden. Die alternative Realisierungsmöglichkeit der Umsetzung direkt im Quellcode verlangt einen erheblichen Mehraufwand während der Entwicklung zur Transformation der Anforderungen in stabile Softwarekomponenten. Mit Rule-Engines dagegen sind sehr viele ausgereifte Produkte am Markt erhältlich, mit denen die Verarbeitung der Regeln effizient durchgeführt werden kann. Eigene Entwicklungsleistung zur Interpretation der Regeln ist nicht notwendig. Dabei ist zu beachten, dass die Aufgaben der Rule-Engine, wie z.B. das Pattern-Matching oder die Konfliktlösung, keine speziellen Problemstellungen von regelbasierten Systemen sind, sondern in anderer Form bei der Realisierung im Quellcode nicht nur ebenfalls auftreten, sondern auch meist erheblich schwieriger zu lösen sind.

Aber auch bei der Erfassung und Pflege der Regeln haben Rule-Engines aufgrund des deklarativen Paradigmas erhebliche Vorteile. Änderungen an der Regelbasis sind ohne aufwändige Erstellungszyklen möglich, deren Aufwand für Kompilieren, Testen und Deployment auf die Zielplattform bzw. die Auslieferung an den Kunden in der Regel wesentlich größer ist.

Durch den Boom des Einsatzes von Rule-Engines im Bereich geschäftlicher Anwendungen in den letzten Jahren sind mittlerweile auch fundierte Erfahrungen am Markt vorhanden, die sich in ausgereiften Produkten, vielfältiger Literatur, eindrucksvollen Fallstudien und vorhandenem Beratungs-Know-How niederschlagen.

Im hier beschriebenen Projekt konnte nachgewiesen werden, dass sich Rule-Engines problemlos in eigene Anwendungen integrieren lassen. Da die Rule-Engine viele Probleme der Wissensverarbeitung löst, kann die Konzentration schnell auf die Definition der Regeln und somit auf die eigene Kernkompetenz im Unternehmen gerichtet werden. Auch die erweiter-

ten Anforderungen der anwendungsunabhängigen Wissensverarbeitung und der Einbeziehung individueller Informationen des Kunden in den Verarbeitungsprozess konnten durch die entwickelten Komponenten und Techniken erfüllt werden.

Es wird geschlossen, dass regelbasierte Systeme einen sehr guten Ansatz zur Erfüllung der im Projekt vorhandenen Anforderungen darstellen. Die flexiblen Möglichkeiten der Regeln erlauben es, das komplexe Expertenwissen effizient und komfortabel zu erfassen. Unterstützt durch die hier entwickelten eigenen Komponenten sowie die im folgenden Abschnitt beschriebenen weiteren Verbesserungsmöglichkeiten kann damit eine unternehmensweit nutzbare Wissensbasis aufgebaut und vielfältig verwendet werden.

7.2 Ausblick

Basierend auf den Evaluierungen dieser Arbeit soll in den nächsten Wochen das Projekt zur Umsetzung des geplanten Systems begonnen werden. Dazu wird nochmals eine detaillierte Analyse des im ersten Schritt zu erfassenden Wissens durchgeführt. Dabei wird auch eine Identifikation aller Problembereiche einschließlich der Definition ihrer fachlichen Schnittstellen in Form der Output-Interfaces stattfinden. Zusätzlich soll die Möglichkeit untersucht werden, Problembereiche durch eine zentrale Instanz über stringifizierte Namen zur Verfügung zu stellen. Diese Instanz wird soll folgende Aufgaben übernehmen:

- Sie kennt alle vorhandenen Problembereiche
- Sie kapselt den Zugriff auf die jeweiligen Regelsets und zugehörigen Output-Interfaces
- Sie stellt Standard-Implementierungen bereit
- Sie sorgt für eine zentrale Verwaltung der Problembereiche, z.B. um eine einzelne Datenhaltung für alle Anwendungen zu ermöglichen (Prinzip des „Single-Source“). Somit könnten z.B. auch Regeln zur Laufzeit geändert werden und stehen in allen verwendenden Anwendungen zur Verfügung

Um die Verwendung von Problembereichen in den Anwendungen einfacher zu gestalten, soll jeder Problembereich eine Standard-Implementierung des Interfaces bereit stellen. Diese Standard-Implementierung kann dann in den Working Memory der jeweiligen Anwendung eingefügt werden und so als Input für anwendungsspezifische Regeln dienen. Somit kann der deklarative Ansatz anwendungsübergreifend verfolgt werden. In der momentanen Version besteht hier ein Bruch, da für die Interface-Implementierung Java-Code erstellt werden muss.

Weiterhin soll evaluiert werden werden, mit welchen Mitteln eine Erleichterung der Regelerfassung erreicht werden kann. Aufgrund sehr vieler gleichartiger Regeln, z.B. bei Kompatibilitätsdaten, wird im ersten Schritt eine Umsetzung mit Entscheidungstabellen ins Auge gefasst. Zur Erleichterung des Sub-Goalings soll eine Komponente entwickelt werden, die basierend auf vorhandenen UserFact-Regeln die jeweils passenden Sub-Goaling-Regeln automatisch erstellt. Später soll die Entwicklung einer eigenen Fachsprache überlegt werden.

Im ersten Schritt ist die Umsetzung zwei konkreter Anwendungen geplant. Eine Anwendung soll die Mitarbeiter der verschiedenen Fachabteilungen des Unternehmens, vor allem Vertrieb, Support, Entwicklung und Service, durch die Bereitstellung und Aufbereitung des Fachwissens unterstützen. Durch die verbesserte Wissenstransparenz wird eine gesteigerte Produktivität sowie eine bessere abteilungsübergreifende Kommunikation erwartet. Als zweite Anwendung wird die Homepage (*www.mad-moxx.de*) um bestimmte Elemente zur Verbesserung des Kundenservices erweitert. So sollen z.B. zu jedem Artikel Informationen zu Kompatibilitäten und Hardwareempfehlungen angezeigt werden. Diese Funktionalität soll anschließend

erweitert werden um die Möglichkeit, während des Besuchs persönliche Informationen anzugeben, anhand derer die Zusatzinformationen personalisiert werden können.

Nach einer erfolgreichen Einführung soll schließlich ein Verkaufsberatungsdialog auf der Homepage integriert werden, der in seiner Grundstruktur dem zweiten Prototypen dieser Arbeit ähnelt. Um diesen Dialog möglichst kundenfreundlich zu gestalten, soll dabei die Technik des Sub-Goaling verwendet werden.

Auf mittlere Sicht ist geplant, zusätzlich Bewertungsstrategien für Produktempfehlungen sowie Techniken zum Umgang mit unsicherem Wissen (z.B. Fuzzy-Techniken) in die Wissensverarbeitung zu integrieren.

Ebenfalls fest geplant ist die Integration der Rule-Engine Drools in das unternehmenseigene ERP-System *CCS*. Hier soll der Fokus vor allem auf die Kombination der momentan bereits produktiv eingesetzten Techniken MDA und JBPM zusammen mit Business Rules gesetzt werden. Dies wird vom Autor zum momentanen Zeitpunkt als viel versprechender Ansatz eingeschätzt, erhebliche Fortschritte in der effizienten Modellierung und Implementierung der eigenen Geschäftsprozesse und Applikationen zu erreichen.

Anhang A

Links

Name	**URL**
Ilog JRules	http://www.ilog.de
Drools	http://www.drools.org
Mandarax	http://mandarax.sourceforge.net
Jess	http://herzberg.ca.sandia.gov
Visual Rules	http://www.visualrules.de
Java Community Process (JCP)	http://www.jcp.org
JCP-94 Rule-Engine API	http://www.jcp.org/en/jsr/detail?id=94
RuleML	http://www.ruleml.org

Tabelle A.1: Links

Anhang B

UML-Diagramm Prototyp 2

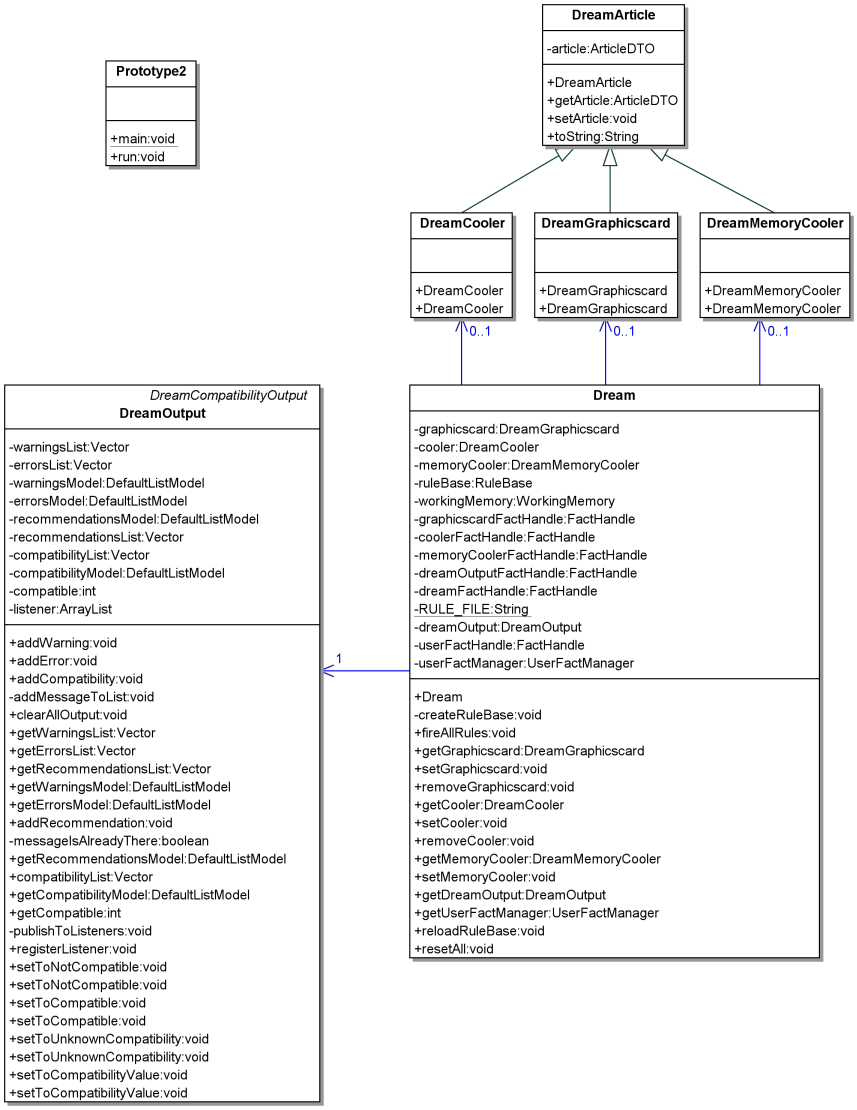


Abbildung B.1: Klassendiagramm Prototyp 2

Literaturverzeichnis

[And04] Andreas Andresen. *Komponentenbasierte Softwareentwicklung, 2.Auflage.* Hanser, 2004.

[AR01] Andrei Voronkov Alan Robinson. *Handbook of Automated Reasoning.* The MIT Press, 2001. ISBN ist für: Vol I + II.

[ARW04] Alberto Avritzer, Johannes P. Ros, and Elaine J. Weyuker. Estimating the cpu utilization of a rule-based system. *SIGSOFT Softw. Eng. Notes*, 29(1):1–12, 2004.

[Axm03] Michael Widenius; David Axmark. *MySQL.* mitp, 2003.

[Bal00] Helmut Balzert. *Lehrbuch der Software- Technik, Band 1 und 2.* Spektrum Akademischer Verlag, 2000.

[Bar82] E. Barr, A.; Feigenbaum. *Handbook of Artificial Intelligence, Vol. I.* Addison-Wesley, 1982.

[Bib93] Wolfgang Bibel. *Wissensrepräsentation und Inferenz.* Vieweg, 1993.

[Dat00] C. J. Date. *WHAT Not NOW.* Addison-Wesley, 2000.

[Dro05] Drools-tutorial: Conway's game of life example. http://drools.org, Oktober 2005. Besucht am 08.10.2005.

[E.H77] R.Davis; G.Buchanan; E.H.Shortcliffe. Production systems as a representation for a knowledge-based consultation program. *Artificial Intelligence*, 8:15–45, 1977.

[Erl05] Thomas Erl. *Service-Oriented Architecture.* Prentice Hall, 2005.

[FH03] Ernest Friedman-Hill. *Jess in Action.* Manning, 2003.

[For90] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. pages 324–341, 1990.

[For05] Forward and backward chaining: Part 2. http://www.rulespower.com, Oktober 2005. Besucht am 10.10.2005.

[Fra03] David S. Frankel. *Model Driven Architecture.* OMG Press, 2003.

[Ger87] Peter R. Gerke. *Wie denkt der Mensch?* J.F.Bergmann Verlag München, 1987.

[GG03] J. Schneeberger Günther Görz, C.-R. Rollinger. *Einführung in die künstliche Intelligenz, 4.Auflage.* Oldenbourg, 2003.

[Hel96] Joseph L. Hellerstein. An approach to selecting metrics for detecting performance problems in information systems. *SIGMETRICS Perform. Eval. Rev.*, 24(1):266–267, 1996.

[Her00] Dieter Herbst. *Erfolgsfaktor Wissensmanagement*. Cornelsen, 2000.

[LA87] Francois Fages Luc Albert. Average case complexity analysis of the rete multi-pattern match algorithm. 1987 1987.

[Lab05] Sandia National Laboratories. The rete algorithm. http://herzberg.ca.sandia.gov/jess/docs/52/rete.html, August 2005. Besucht am 10.08.2005.

[Mal03] Deepak Alur; John Crupi; Dan Malks. *Core J2EE Patterns*. Prentice Hall, 2003.

[Rei91] Ulrich Reimer. *Einführung in die Wissensrepräsentation*. B.G.Teubner Stuttgart, 1991.

[Ros03] Ronald G. Ross. *Principles of the Business Rule Approach*. Addison-Wesley, 2003.

[Rud05] George Rudolph. Some guidelines for deciding whether to use a rule engine. http://http://herzberg.ca.sandia.gov/jess/guidelines.shtml, Oktober 2005. Besucht am 12.10.2005.

[Rup04a] Chris Rupp. *Requirements-Engineering und -Management*. Hanser, 2004.

[Rup04b] N. Alex Rupp. Ruling out: Rule engines and declarative programming come to java. http://today.java.net/lpt/a/121, August 2004. Besucht am 08.08.2005.

[Ses04] Hermann J. Schmelzer; Wolfgang Sesselmann. *Geschäftsprozessmanagement in der Praxis*. Hanser, 2004.

[Soe74] Fritz Reinhardt; Heinrich Soeder. *dtv-Atlas zur Mathematik*. Deutscher Taschenbuch Verlag, 1974.

[vH02] Barbra von Halle. *Business Rules Applied*. Wiley, 2002.

[Via97] Victor Vianu. Rule-based languages. *Annals of Mathematics and Artificial Intelligence*, 19(1-2):215–259, 1997.

[Vli96] Erich Gamma; Richard Helm; Ralph Johnson; John Vlissdes. *Entwurfsmuster*. Addison-Wesley, 1996.

Abbildungsverzeichnis