

INSTITUT FÜR INFORMATIK

DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN



Diplomarbeit

Analyse und Entwurf von Business Rules auf Basis einer Implementierung für die eBusiness-Plattform hybric

Michael Buschmann

Aufgabensteller: Univ.-Prof. Dr. Hans Jürgen Ohlbach

Betreuer: Dr. Bernhard Lorenz

Abgabetermin: 24. Juli 2007

In einer globalisierten Welt ist der Konkurrenzdruck zwischen Online-Shops so hoch wie nie zuvor. Wenn die Konkurrenz nur einen Mausklick entfernt ist, ist es schwierig den potentiellen Käufer zum Abschluss des Kaufvorgangs zu bewegen. Deshalb setzen große Online-Shops vermehrt auf so genannte verkaufsfördernde Maßnahmen. Zu diesen verkaufsfördernden Maßnahmen gehören auch die Promotions. Unter Promotions versteht man allgemein Preisnachlässe oder andere Vorteile wie Geschenke oder kostenlose Beigaben, die unter bestimmten Voraussetzungen gewährt werden.

Anbieter von eCommerce-Werkzeugen stehen daher vor der Herausforderung, ihre Produkte mit der Fähigkeit, Promotions anzubieten, auszustatten. Eins der führenden Produkte auf diesem Sektor ist die hybris eCommerce Plattform der Münchner Firma hybris GmbH. Auch dort gibt es im Rahmen eines aktuellen Großprojektes die Anforderung, Promotions-Funktionalitäten in das Produkt zu integrieren. In der vorliegenden Arbeit werden zunächst die Grundlagen vorgestellt. Es wird dargestellt, welche Ausprägungen von Promotions existieren, das Prinzip und die Technik von Business Rules bzw. Business Rule Engines erläutert und die hybris eCommerce Plattform vorgestellt.

Im zweiten Teil der Arbeit wird die projektspezifische Umsetzung der Promotion-Funktionalitäten für den neuen Online-Shop der Firma Virgin Megastores analysiert. Es wird erläutert, welche Anforderungen seitens der Shop-Betreiber an die Promotion-Lösung gestellt werden und wie diese umgesetzt werden.

Danach wird untersucht, ob und inwiefern sich eine ähnlich funktionale Lösung unter Einsatz von Business Rules abbilden lässt. Es wird die Integration der Business Rule Engine in die hybris eCommerce Plattform und Regeln, mit denen Promotions konkret definiert werden, vorgestellt.

Am Ende der Arbeit werden die beiden Lösungen verglichen. Es werden Stärken und Schwachstellen der regelbasierten Promotion-Lösung beschrieben und Ansatzpunkte zur Verbesserung und Erweiterung des regelbasierten Ansatzes dargelegt.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 24. Juli 2007

Unterschrift des Kandidaten

Inhaltsverzeichnis

Inhaltsverzeichnis	i
Abbildungsverzeichnis	iii
Tabellenverzeichnis	iv
1 Einleitung	1
1.1 Motivation	1
1.2 Aufgabenstellung und Zielsetzung	1
1.3 Aufbau und Inhalt	2
1.4 Kooperationen	2
2 Hauptteil	3
2.1 Grundlagen	3
2.1.1 Promotions	3
2.1.2 Business Rules	5
2.1.3 Business Rule Engines	7
2.1.4 Hybris	18
2.2 Projektspezifische Implementierung der Promotions	28
2.2.1 Anforderungen	28
2.2.2 Implementierung	31
2.2.3 Fazit	49
2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine	51
2.3.1 Motivation	51
2.3.2 Einbindung einer Rule Engine	52
2.3.3 Abbildung von Promotions	93
2.3.4 Fazit	135

3 Zusammenfassung	140
3.1 Beurteilung	141
3.2 Ausblick	142
A Anhang	143
A.1 Promotion-Extension	143
A.1.1 Itemdefinition	143
A.1.2 Java Klassen	154
A.2 Rules-Extension	165
A.2.1 Itemdefinition	165
A.2.2 Java Klassen	168
Literaturverzeichnis	177

Abbildungsverzeichnis

2.1	Diagramm eines Rete-Netz (ohne Wurzelknoten)	12
2.2	Optimierung eines Rete-Netz	13
2.3	Hybris Item-Hierarchie (aus [4])	20
2.4	Darstellung der Attribute description und unit	26
2.5	UML Diagramm Virgin Promotions	34
2.6	UML Diagramm Restrictions	35
2.7	UML Diagramm Results und Actions	36
2.8	Aktivitätsdiagramm Auswertung von Promotions	41
2.9	Klassendiagramm der angelegten hybris Typen	64
2.10	Administrationskonsole mit eingebundener Business Rules-Gruppe	68
2.11	DefaultReference für den Typ Rule	73
2.12	Suchanfrage für den Typ Rule	74
2.13	Suchergebnis für den Typ Rule als editview	75
2.14	Editor für den Typ Rule	76
2.15	vereinfachtes Aktivitätsdiagramm Auswertung von Promotions mit Business Rules	133

Tabellenverzeichnis

2.1	Beispiel-Tabelle in MySQL	25
2.2	Schema der Tabelle Rules	67
2.3	Buy 2 Get 1 Free Promotion Teaser	100
2.4	DSL-Mapping für Teaser #1	103
2.5	Buy 2 Get 1 Free Promotion Teaser mit DSL	103
2.6	Spend 50€and get free shipping Promotion Teaser	105
2.7	DSL-Mapping für Teaser #2	106
2.8	Spend 50€and get free shipping Promotion Teaser mit DSL	106
2.9	OrderThreshold Promotion	120
2.10	DSL-Mapping für Promotion #1	121
2.11	OrderThreshold Promotion mit DSL	122
2.12	Multibuy Promotion 2 Produkte / 5€	123
2.13	Multibuy Promotion 5 Produkte / 20€	124
2.14	DSL-Mapping für Promotion #2	125
2.15	Multibuy Promotion 2 Produkte / 5€mit DSL	126
2.16	Buy2get1freeTshirt Promotion	127
2.17	DSL-Mapping für Promotion #3	128
2.18	Buy2get1freeTshirt Promotion mit DSL	129
2.19	doDiscountValues Regel	131
2.20	doProducts Regel	132
2.21	undoProducts Regel	134
2.22	undoDiscountValues Regel	135

1 Einleitung

1.1 Motivation

Der Konkurrenzdruck im heutigen B2C¹ Umfeld von Online-Shops ist durch die Globalisierung einerseits und die hohe Verfügbarkeit von günstigen Online-Shop-Systemen, die schon mit geringem Startkapital das Betreiben eines Online-Shops ermöglichen, andererseits, enorm.

Die Betreiber von Online-Shops suchen deshalb verstärkt nach Alleinstellungsmerkmalen bzw. Kaufanreizen für potentielle Kunden um in diesem Umfeld zu bestehen. Einen solchen Kaufanreiz bieten die, bereits aus dem „real-life“-Handel bekannten, Promotions, zu Deutsch „Verkaufsförderungsmaßnahmen“. Promotions dienen unter anderem dazu eine höhere Kundenbindung und gesteigerten Umsatz zu erreichen. Bekanntestes Beispiel hierfür sind so genannte *Multibuy*²- oder *Linksave*³-Promotions. Insbesondere im angloamerikanischen Raum, der u.a. aufgrund der fehlenden Sprachbarriere einem noch höheren Konkurrenzdruck ausgesetzt ist als beispielsweise der deutschsprachige Online-Markt, werden diese Promotions zunehmend auch im eCommerce eingesetzt. Hersteller von eCommerce-Frameworks sehen sich deshalb vor der Herausforderung Promotionfunktionalitäten irgendeiner Art in ihre Produkte zu integrieren, um den Anforderungen ihrer Kunden gerecht zu werden.

1.2 Aufgabenstellung und Zielsetzung

Im Rahmen dieser Arbeit soll untersucht werden wie Promotionfunktionalitäten in die eBusiness-Plattform der Firma hybris integriert werden können. Speziell soll hierbei analysiert werden inwiefern sich Business Rules, die besonders geeignet erscheinen die grundsätzliche Wenn-Dann-Struktur von Promotions - „wenn der Kunde x macht, dann bekommt er y“ - abzubilden, dafür eignen.

¹Business-To-Customer - Handel zwischen Firmen und Endkunden

²Staffelrabatte wie „2 zum Preis von 1“

³„Beim Kauf eines Buchs gibt es ein Lesezeichen geschenkt“

1 Einleitung

Während dieser Untersuchung wird idealerweise ein auf Business Rules basierendes, in die eBusiness-Plattform von hybris integriertes Framework entstehen, welches es ermöglicht Promotions anzuwenden. Dabei steht zunächst die reine Funktionalität im Fokus, aber auch die Benutzbarkeit und effiziente Verwaltung durch die späteren Shop-Betreiber soll betrachtet werden.

Es handelt sich bei der vorliegenden Arbeit nicht um eine genaue Untersuchung der Arbeitsweise von Business Rule Engines selbst, so dass dazu nur die notwendigen Grundlagen vorgestellt werden sollen.

1.3 Aufbau und Inhalt

Die Arbeit gliedert sich dabei grob in zwei Teile. Im ersten Teil werden die Grundlagen vorgestellt, auf denen die Arbeit später aufbauen wird. Zunächst werden allgemein Promotions definiert, welche Arten von Promotion es gibt und wie sich diese auswirken könnten. Danach wird auf Business Rules eingegangen, welche Arten von Business Rules es gibt, wie die Verarbeitung von Business Rules in der Theorie vonstatten geht und welche Implementierungen bzw. Standards es für Business Rules im Java-Umfeld gibt. Im zweiten Teil wird auf die Integration von Promotion-Funktionalitäten in die hybris Plattform eingegangen. Zunächst wird gezeigt wie die Forderung nach Promotions in einem aktuellen Großprojekt der Firma hybris projektspezifisch implementiert wurde. Danach wird gezeigt, wie eine vergleichbare Lösung unter Zuhilfenahme von Business Rules implementiert werden könnte.

1.4 Kooperationen

Diese Arbeit entsteht in Zusammenarbeit mit der Firma ARITHNEA GmbH, einem Implementierungspartner der hybris GmbH. Die projektspezifische Implementierung der Promotion-Funktionalität für Virgin Megastores entsteht in Kooperation mit der Firma neoworks Ltd. in Großbritannien, einem weiteren Partner der hybris GmbH.

2 Hauptteil

2.1 Grundlagen

2.1.1 Promotions

Promotions, oder Verkaufsförderungsmaßnahmen, sollen dazu dienen, das Verkaufsergebnis beziehungsweise den Umsatz an einer bestimmten Ware zu erhöhen. Anders als Werbung, die prinzipiell das Kaufinteresse an einer Ware beim Kunden wecken soll, geben Promotions einen zusätzlichen Kaufanreiz, wenn das Interesse an der Ware bereits geweckt ist[34]. In dieser Arbeit werden Promotions immer als auf Konsumenten gerichtete Verkaufsförderungsmaßnahmen verstanden, und zwar sowohl Preis-Promotions, der aktive Preisnachlass bei Waren, als auch nicht Nicht-Preis-Promotions, also zusätzliche Angebote wie die Teilnahme an Gewinnspielen, kostenlose Warenproben o.ä. Grundsätzlich haben alle Promotions zum Ziel die Marktpräsenz einer Ware zu erhöhen, evtl. auch unter Inkaufnahme von sinkenden Gewinnmargen. Wie diese Steigerung der Marktpräsenz erreicht wird hängt von der Ausprägung der gewählten Promotion ab. Prinzipiell ist die Abgrenzung der unterschiedlichen Ausprägungen untereinander nicht immer problemlos möglich und es gibt Promotions, die Ähnlichkeiten zu mehreren Ausprägungen haben.

Ausprägungen

Cross-Selling Unter *Cross-Selling* versteht man das Anbieten von ergänzen Produkten zu einer betrachteten Ware[27]. Dabei müssen die durch *Cross-Selling* angebotenen Produkte nicht unbedingt funktional ergänzend zum Verwendungszweck des ursprünglichen Produktes sein, sondern können auch je nach Werbestrategie aus einem größer gefassten Produktbereich kommen. Ziel ist es einerseits, den Gesamtumsatz des Einkaufs zu erhöhen, andererseits können mittels *Cross-Selling* dem Kunden auch Alternativen angeboten werden bzw. das Interesse des Kunden auf neue Gebiete gelenkt werden um dem Kunden ein breiteres Warenspektrum zugänglich zu machen. Beispiel: Ein Kunde interessiert

2 Hauptteil

sich für einen Videorekorder, so sind ergänzend Videokassetten für ihn interessant. Alternativ könnte man ihm aber auch Fernseher oder Videokameras anbieten.

Up-Selling Unter *Up-Selling* versteht man den Versuch gleichartige, aber höherwertige, bzw. mit höheren Gewinnmargen versehene, Produkte zu einem Ausgangsprodukt anzubieten[33]. Ziel ist eine Steigerung des Umsatzes und/oder das Erzielen von höheren Gewinnen. Beispiel: Interessiert sich ein Kunde für einen günstigen Videorekorder, kann man ihm alternativ einen höherwertigen Videorekorder oder sogar DVD-Player, jeweils mit höherem Preis oder höheren Gewinnmargen, anbieten.

Linksave *Linksave*, auch Bedingungskauf genannt, ist eine kostenlose oder im Preis reduzierte Zusatzleistung[23]. Typisch für *Linksave*-Promotions ist die Dreingabe von minderwertigen Waren beim Kauf von hochpreisigen Produkten. Häufig wird der *Linksave* auch nicht von einzelnen Produkten, sondern von anderen Faktoren wie dem gesamten Einkaufswert o.ä. abhängig gemacht. Dabei nennt man das ursprünglich erworbene Produkt den Anker. Ein typisches Beispiel für eine *Linksave*-Promotion wäre die Zugabe eines kostenlosen T-Shirts zu jeder Spielkonsole.

Multibuy Bei *Multibuy* handelt es sich um klassische Staffelrabatte, beim Kauf einer gewissen Anzahl von gleichartigen Produkten erhält der Käufer eine Vergünstigung[21]. Diese Vergünstigung kann entweder ein weiteres kostenloses oder preislich reduziertes Exemplar des promoteten Produktes sein (klassischer *Multibuy*), oder ein anderes Produkt bzw. eine andere Leistung sein (Kombination zwischen *Multibuy* und *Linksave*).

Beispiel für klassischen *Multibuy*: „Nimm 3, zahle nur 2“.

Beispiel für Kombination *Multibuy* und *Linksave*: „Kaufe 3 CDs und du bekommst ein T-Shirt gratis“.

zeitversetzte Promotions Genau genommen keine wirklich eigenständige Ausprägung von Promotions sind zeitversetzte Promotions. Das interessante an diesen Promotions ist nicht die eigentliche Vergünstigung für den Kunden, sondern der Zeitpunkt, an dem diese wirksam wird. Bei allen oben genannten Ausprägungen von Promotions wird die eigentliche Vergünstigung sofort bei Eintreffen der Vorbedingungen aktiviert, es existieren aber auch Promotions die den Kunden längerfristig an den Händler binden sollen. Beispiel: „Beim Einkauf mit einem Gesamtwert über 50 € ist der nächste Einkauf um 5 € günstiger“.

2.1.2 Business Rules

Unter Business Rules oder Geschäftsregeln versteht man allgemein Regeln die innerhalb einer geschäftlichen Organisation angewendet werden, wie „Kunden mit mehr als 10000€ Jahresumsatz sind Premiumkunden“[26]. Im Umfeld der Informatik müssen solche Business Rules aber nicht zwangsläufig auch abstrakte geschäftsrelevante Aussagen treffen, sondern können durchaus auch grundlegendere technische Gegebenheiten beschreiben[29]. Hauptziel beim Einsatz von Business Rules in technischen Systemen ist, eine Trennung von Programmlogik, was zumeist in einer Programmiersprache wie Java verfasster Quellcode ist, und Businesslogik, die die eigentlichen Geschäftsregeln darstellt, zu erreichen. Durch diese Trennung soll es möglich sein, dass sich die beteiligten Expertengruppen¹ jeweils nur auf ihr Fachgebiet konzentrieren können. Programmierer entwickeln ihren Code unabhängig von den konkreten Geschäftsregeln, die zumeist auch anderen, in der Regel wesentlich kürzeren Entwicklungszyklen unterworfen sind. Validiert werden die Daten auf den, den Programmierern unbekannt, Geschäftsregeln. Die Experten für Geschäftsregeln hingegen können ihre Vorgaben einfach angeben, ohne sich um die technische Umsetzung oder Programmierung kümmern zu müssen. Dadurch kann man „Reibungsverluste“ verringern und insgesamt höhere Qualität auf beiden Seiten erreichen.

Aufbau

Business Rules sind eng verwandt mit der Prädikatenlogik, welche gewissermaßen die Grundlage für die Business Rules darstellt. Eine Business Rule zerfällt immer in zwei Teile, den Bedingungsteil² und die Konsequenz³. Der Bedingungsteil wird, weil er bei Prädikatenschreibweise zumeist links steht, auch „linke Seite“ oder kurz *LHS*⁴, der Konsequenzteil auch „rechte Seite“ oder *RHS*⁵ genannt. Darüber hinaus verfügen Business Rules auch über die aus der Prädikatenlogik bekannten Verknüpfungen⁶. Sobald die Bedingung der Business Rule wahr wird - das heißt die Prämisse erfüllt ist - folgt daraus die Konsequenz. Man sagt die Business Rule „feuert“.

¹Programmierer und Manager bzw. Sachbearbeiter

²oft auch Prämisse genannt

³Konklusion

⁴engl. left-hand-side

⁵engl. right-hand-side

⁶UND, ODER, NICHT

Deklarativer Ansatz

In der prozeduralen Programmierung [31] entspricht dieser Aufbau grundlegend dem bekannten „if-then-else“-Konstrukt. Bei der prozeduralen Programmierung wird versucht, eine Anforderung in kleine Teile zu zerlegen und diese Schritt für Schritt mit Anweisungen zu erledigen. Die eigentlich Problembeschreibung („was muss getan werden?“) ist also vermischt mit dem Lösungsweg („wie muss es getan werden?“). Problematisch hierbei ist, dass der daraus entstehende Code nicht mehr die reine Anforderung sondern auch die Umsetzung enthält und nicht mehr ohne Kenntnisse, die über die eigentliche Problemstellung hinausgehen, verstanden werden kann. Im Gegensatz dazu verwenden Business Rules einen deklarativen Ansatz. Beim deklarativen Paradigma[28] wird strikt zwischen der eigentlichen Anforderung und dem Weg dorthin unterschieden. Die Deklaration sagt lediglich aus, was gemacht werden soll, die Interpretation, also wie es gemacht wird, ist davon getrennt. Dadurch wird ein höherer Abstraktionsgrad erreicht, der viel mehr der „menschlichen“ Denkstruktur entspricht und es Sachbearbeitern ohne tieferes Verständnis von Programmierung erlauben soll, ihre Anforderungen deklarativ, also beschreibend, umzusetzen. Die Auswertung der Regeln, also die Interpretation, wird von einer Software automatisch ausgeführt, dem so genannten *Interpreter*.

Ausprägungen

Prinzipiell unterscheidet man grob drei verschiedene Arten von Business Rules[25]:

- Konsistenzregeln⁷ stellen Rahmenbedingungen für Daten dar. Solange Daten diese Konsistenzregeln erfüllen, werden sie als wahr bzw. gültig eingestuft. Diese Regeln bieten also innerhalb ganzer Systeme die Möglichkeit, Daten nach einheitlichen Maßstäben zu validieren. Innerhalb von Geschäftsanwendungen müssen Daten nicht nur nach technischen Gesichtspunkten (bspw. Integer != null), sondern auch nach geschäftsrelevanten Vorgaben (bspw. „Kontostand darf nicht unter 0 sinken“) validiert werden. Mit Hilfe von Konsistenzregeln können Daten also flexibel validiert werden und die Gültigkeitskriterien ohne Anpassung des Quellcodes geändert werden.
- Produktionsregeln⁸ generieren aus vorhandenen Daten neue Daten. Dabei kann die Komplexität dieser Regeln stark variieren von einfachen Berechnungsvorschriften wie „1+1=2“ (*computati-*

⁷engl. meist consistency rules

⁸engl. meist production rules

on) bis hin zu komplexen Schlussfolgerungen wie „Wenn ein Kunde pro Jahr mehr als 10000 € Umsatz hat, in Europa wohnt, immer mit Kreditkarte bezahlt und mindestens vier Kinder hat, dann ist er ein Premiumkunde“ (*inference*).

- Aktionsregeln⁹ definieren Aktionen, die unter bestimmten Vorbedingungen durchgeführt werden. Ein einfaches Beispiel für eine Aktionsregel ist „Alle Kunden die länger als ein Jahr nichts gekauft haben, bekommen eine Werbemail geschickt“. Man nennt diese Regeln auch ereignisbasierte Geschäftsregeln und kann mit ihrer Hilfe auch komplette Aktionsketten bzw. Workflows abbilden. Dadurch dass Aktionsregeln direkt Aktionen auslösen, also mit ihrer Umgebung interagieren müssen, sind sie zumeist stärker mit ihrer Laufzeitumgebung, also dem Programm, verbunden.

2.1.3 Business Rule Engines

Wie bereits erwähnt, wird zur Auswertung von deklarativ angegeben Business Rules ein *Interpreter* benötigt. Dieser *Interpreter* wird Business Rules Engine genannt. Im Prinzip bestehen alle herkömmlichen Business Rule Engines aus drei Komponenten:

Regelbasis In der Regelbasis¹⁰ genannt sind alle erstellten Regeln gespeichert. Die Regelbasis stellt also gewissermaßen das gesammelte Expertenwissen dar. Dieses Wissen ist sinnvollerweise persistent gespeichert, je nach verwendeter Rule Engine oder Anwendungsfall zumeist in einer Datenbank oder im Filesystem.

Faktenbasis In der Faktenbasis¹¹ sind alle dem Auswertungssystem bekannten Fakten gespeichert. Diese Fakten sind prinzipiell Datentypen einer Programmiersprache, im Fall von objektorientierten Programmiersprachen wie Java also Objekte. Die Datenbasis stellt eine Laufzeitumgebung für diese Objekte dar, sie können modifiziert, hinzugefügt und entfernt werden. Im Gegensatz zur Regelbasis wird die Faktenbasis nicht persistent gespeichert, sondern ist sitzungsabhängig.

Regel-Interpreter Der *Regel-Interpreter*¹² ist das Herzstück jeder Business Rule Engine. Sie

⁹engl. meist action rules

¹⁰engl. oft auch *Production Memory* oder *Rule-Base*

¹¹engl. Working Memory

¹²engl. Inference Engine

2 Hauptteil

übernimmt die eigentliche Auswertung der Regeln, d.h. sie prüft die in der Regelbasis enthaltenen Regeln mit den in der Faktenbasis enthaltenen Fakten. Die Prozess der Auswertung erfolgt im Prinzip über Pattern Matching¹³, wofür es verschiedene Algorithmen gibt:

- Linear
- Leaps
- Treat
- Rete

Die meisten Business Rules Engines verwenden standardmäßig den *Rete*-Algorithmus zur Auswertung der Regeln, deswegen wird im Rahmen dieser Arbeit auch nur dieser Algorithmus später kurz vorgestellt.

Arbeitsweise

Inferenz Die theoretische Grundlage zur Arbeit von Business Rule Engines ist die so genannte Inferenz¹⁴[18].

Beispiel: „Wenn ich Hunger habe, dann esse ich etwas“.

Unabhängig von etwaigen Vorkenntnissen über die Person ist der obige Satz nur als einfache Regel zu verstehen. Sobald man aber die Information erhält, dass die Person hungrig ist kann man daraus ableiten, dass die Person bald etwas essen wird, es folgt also eine Aktion. Dies entspricht in der Prädikatenlogik dem sog. „Modus Ponens“:

$$(X \wedge (X \rightarrow Y)) \rightarrow Y$$

Beim praktischen Einsatz von Rule Engines bleibt es aber nicht bei der Auswertung einer einzelnen Regel, sondern es werden viele Regeln ausgewertet, wobei die Auswertung jeder Regel neue Informationen bzw. Fakten ergeben kann, die dann wiederum andere Regeln „erfüllen“ können usw. Man spricht hierbei von verketteter Inferenz. Für die in dieser Arbeit untersuchten Promotions könnte an der exemplarischen Promotion „Neukunden erhalten kostenfreie Lieferung“ die Schlussfolgerung folgendermaßen aussehen: „Wenn der bestellende Kunde noch nie bestellt hat, ist er ein Neukunde“ und

¹³dt. Musterabgleich

¹⁴dt. Schlussfolgerung

„Wenn der bestellende Kunde ein Neukunde ist, dann bekommt er kostenfreie Lieferung“.

Ereignisbasierte Inferenz Die ereignisbasierte Inferenz geschieht die Auswertung nach dem bekannten „wenn-dann“-Muster. Sobald eine neue Information bekannt wird, wird überprüft ob mit dieser neuen Information weitere Regeln aktiviert werden können, die dann wiederum selbst neue Informationen produzieren können. Auf diese Weise wird ein Ausgangszustand durch die verkettete Ausführung von Regeln in einen anderen Zustand überführt. Man nennt diese ereignisbasierte Inferenz deshalb auch vorwärtsverkettete Inferenz.

Beispiel:

Regel A: Wenn es fliegen kann, dann ist es ein Vogel.

Regel B: Ein Vogel der schwarz ist, ist ein Rabe.

Zu Beginn ist lediglich bekannt dass das Objekt fliegen kann und schwarz ist.

Aus Regel B kann zu Beginn nichts geschlossen werden, da das Objekt zwar schwarz ist, aber es ist nicht bekannt dass es sich um einen Vogel handelt.

Aus Regel A kann geschlossen werden, dass es sich um einen Vogel handeln muss, weil das Objekt fliegen kann.

Sobald die Information, dass es sich um einen Vogel handelt bekannt wird, kann aus Regel B geschlossen werden, dass es sich um einen Raben handelt.

Für die in dieser Arbeit betrachtete Anwendung der Business Rules zur Abbildung von Promotion-Funktionalitäten wird ereignisbasierte Inferenz verwendet.

Zielgerichtete Inferenz Im Gegensatz zur ereignisbasierten Inferenz geht die zielgerichtete Inferenz den umgekehrten Weg. Mit der zielgerichteten Inferenz ist es möglich, eine Annahme auf ihren Wahrheitsgehalt hin zu untersuchen. Man nimmt die Annahme und prüft danach alle Regeln, die „erfüllt“ sein müssen, um die Annahme wahr zu machen. Im Zuge der Überprüfung dieser Regeln müssen evtl. wiederum weitere Regeln geprüft werden, weshalb man die zielgerichtete Inferenz auch rückwärtsverkettete Inferenz nennt.

Beispiel:

Regel A: Wenn es fliegen kann, dann ist es ein Vogel.

Regel B: Ein Vogel der schwarz ist, ist ein Rabe.

Annahme: Es handelt sich um einen Raben.

2 Hauptteil

Da das Objekt angeblich ein Rabe ist, müsste es gemäß Regel B schwarz sein und ein Vogel sein.

Gemäß Regel A müsste es, um ein Vogel zu sein, fliegen können.

Wenn das Objekt also schwarz ist und fliegen kann, so ist die Annahme, dass es sich bei dem Objekt um einen Raben handelt, wahr.

Ein Anwendungsgebiet von zielgerichteter Inferenz sind Expertensysteme zur Krankheitsdiagnose. Dort werden mittels Regeln die Symptome von Krankheiten beschrieben, beispielsweise „wenn der Patient Fieber, Gliederschmerzen und Schnupfen hat, dann hat er eine Grippe“. Mittels der zielgerichteten Inferenz kann der behandelnde Arzt seine eigene Diagnose („Der Patient hat wahrscheinlich Grippe“) auf ihren Wahrheitsgehalt hin überprüfen („Wenn er wirklich Grippe hat, dann muss er Fieber, Gliederschmerzen und Schnupfen haben; wenn er eins der Symptome nicht zeigt, kann es keine Grippe sein“).

In dieser Arbeit wird die zielgerichtete Inferenz zur Abbildung der Promotions nicht verwendet.

Auswertung durch Pattern Matching[30] Die Auswertung der Regeln erfolgt durch Musterabgleich¹⁵. Dabei wird versucht ein vordefiniertes Muster in einem beschränkten Suchbereich mittels Vergleich von Werten aufzufinden. Bei den Business Rules entspricht jede Prämisse, d.h. Bedingung einer Regel, einem Muster, welches durch Durchsuchen des Wertebereichs, also der Objektmenge innerhalb der Faktenbasis, versucht wird zu verifizieren. Wenn ein Objekt gefunden wird, was dem in der Prämisse angegebenen Muster entspricht, so ist die Prämisse erfüllt.

Rete-Algorithmus Ein grundlegendes Problem eines einfachen *Pattern Matching* Algorithmus ist die sehr schlechte Performance bei einer großen Anzahl von Regeln und eines großen Suchbereichs. Sobald eine Information (bzw. Objekt) aus der Faktenbasis entfernt, hinzugefügt oder verändert wird, müssen alle Prämissen neu überprüft werden. Wenn viele Regeln - mit entsprechend vielen Prämissen - innerhalb eines großen Suchraums - also auf vielen Objekten - evaluiert werden müssen, ist ein einfacher *Pattern Matching* Algorithmus nicht mehr ausreichend performant.

Deshalb entwickelte in den Jahren 1978-1979 der Amerikaner Dr. Charles Forgy den so genannten *Rete*-Algorithmus¹⁶[19]. Es handelt sich dabei um einen effizienten Algorithmus zur Lösung des oben geschilderten Performanceproblems einfacherer *Pattern Matching*-Algorithmen. Der *Rete*-

¹⁵engl. Pattern Matching

¹⁶lat. für „Netz“ oder „Netzwerk“

Algorithmus findet zunächst in zwei Phasen statt, der Kompilierphase und der Ausführungsphase. In der Kompilierphase werden die Prämissen aller Regeln zu einem Entscheidungsnetzwerk bzw. Datenflußdiagramm in maschinennahem Code umgewandelt. Diese Kompilierphase findet, ohne Änderung der Regeln selbst, nur einmal statt und das entstandene Netzwerk wird im Speicher der Rule Engine gehalten. In der darauf folgenden Auswertungsphase findet die eigentlich Auswertung der Regeln, also das Prüfen der in der Datenbasis enthaltenen Fakten gegenüber den Regeln statt. Dieses Netzwerk besteht im Grunde aus vier verschiedenen Knoten (siehe Abbildung 2.1):

Wurzelknoten Alle Fakten, d.h. Objekte, der Faktenbasis treten am Wurzelknoten in das Netzwerk ein und werden automatisch an den oder die Kindknoten weitergereicht.

α -Knoten Unäre Knoten zum Testen der Fakten heißen α -Knoten. Diese Tests können entweder den Typ eines Objektes¹⁷, oder Eigenschaften, d.h. Attribute, der Objekte überprüfen. Wenn der Test erfolgreich verläuft, also den Boolean *true* zurück liefert, wird das entsprechende Objekt im Netzwerk an den oder die Kindknoten weitergegeben.

β -Knoten Im Gegensatz zu den α -Knoten sind die β -Knoten binäre Knoten, die zwei Objekte entgegennehmen können. Auf diesen Objekten können dann Vergleichsoperationen durchgeführt werden. Wenn diese Vergleichsoperation erfolgreich ist, werden die Objekte an die Kindknoten weitergereicht.

Terminalknoten Nachdem die Objekte das *Rete*-Netzwerk erfolgreich durchlaufen haben kommen sie an einem Terminalknoten an, d.h. die Prämissen der jeweiligen Regel wurden vollständig erfüllt, die Regel kann also „feuern“.

Die Knoten in einem *Rete*-Netzwerk sind zustandsbehaftet, haben also gewissermaßen ein Gedächtnis und wissen, welche Objekte von ihnen bereits geprüft wurden. Das heißt sobald ein Objekt einmal auf eine bestimmte Eigenschaft hin überprüft wurde ist das Ergebnis bereits bekannt, wodurch sehr viele Tests eingespart werden können¹⁸. Darüber hinaus können im *Rete*-Netzwerk Knoten von mehreren Regeln gleichzeitig verwendet werden, wenn beide Regeln eine gleiche Prämisse haben. Dies spart ebenfalls Tests ein, siehe Abbildung 2.2.

¹⁷instanceof-Schlüsselwort

¹⁸im Vergleich zum einfaches Pattern Matching Algorithmus

2 Hauptteil

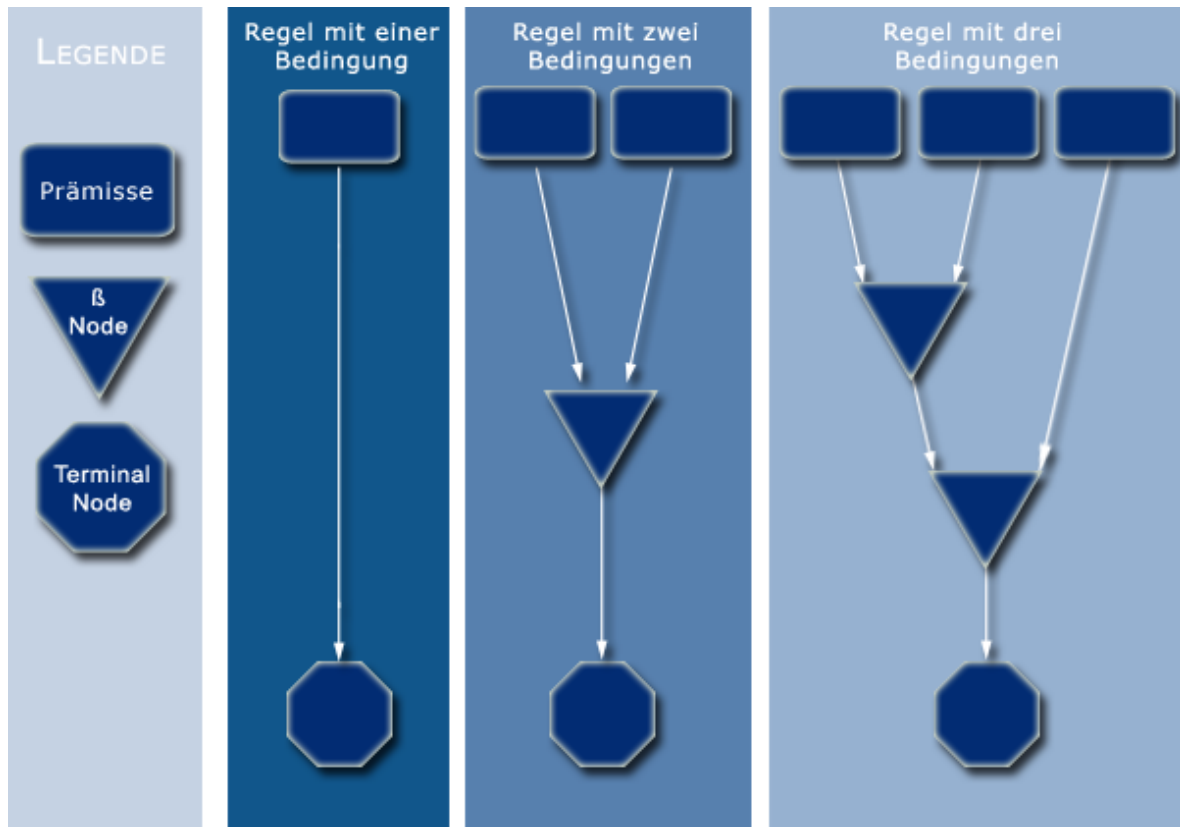


Abbildung 2.1: Diagramm eines Rete-Netz (ohne Wurzelknoten)

Sobald ein Objekt zum *Rete*-Netzwerk hinzugefügt wird, startet es am Wurzelknoten und „wandert“ im *Rete*-Netzwerk durch α - und β -Knoten nach unten in Richtung der Terminalknoten.

Im Vergleich zu einem einfachen *Pattern Matching* Algorithmus ist die Komplexität des *Rete*-Algorithmus wesentlich geringer. Beim einfachen *Pattern Matching* beträgt die Komplexität:

$$O((\text{Anzahl der Regeln} \times \text{Anzahl der Fakten})^{\text{Anzahl Prämissen pro Regel}})$$

Das bedeutet, dass selbst bei einer kleinen Regelbasis von 10 Regeln mit durchschnittlich 3 Prämissen und einer kleinen Faktenbasis von 50 Objekten, $(10 * 50)^3 = 125.000.000$ Vergleichsoperationen für jede Auswertung nötig sind.

Der *Rete*-Algorithmus hingegen hat allgemein lediglich eine Komplexität von:

$$O(\text{Anzahl der Regeln} \times \text{Anzahl Prämissen pro Regel} \times \text{Anzahl der Fakten})$$

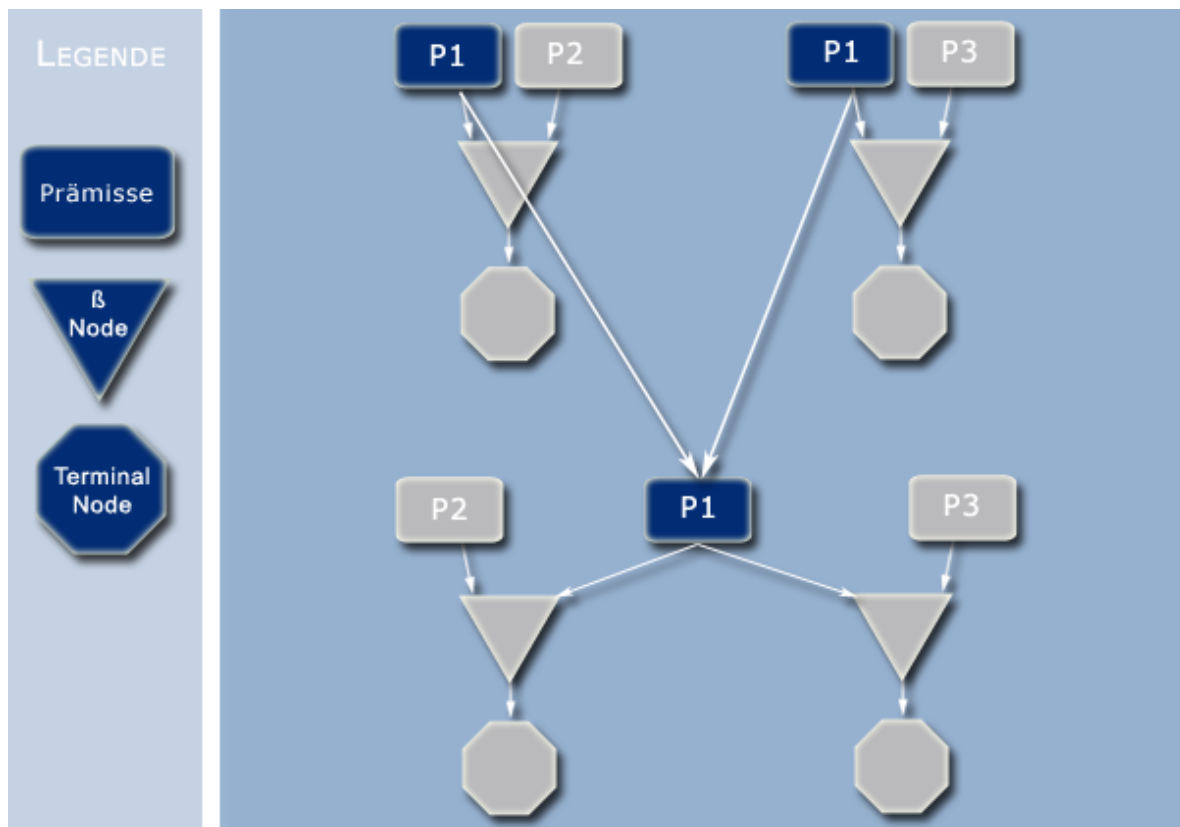


Abbildung 2.2: Optimierung eines Rete-Netz

Generell ist es aber schwierig die Leistungsfähigkeit des *Rete*-Algorithmus zu beurteilen. Beim ersten Lauf des Algorithmus, wenn also die Objekte zum ersten mal das *Rete*-Netz durchlaufen, ist die Anzahl der benötigten Tests sehr hoch, im schlimmsten Fall, wenn alle Prämissen aller Regeln unterschiedlich sind, sogar genauso hoch wie bei dem einfachen *Pattern Matching* Algorithmus. Ab der zweiten Auswertung des *Rete*-Algorithmus jedoch müssen nur noch die seit dem letzten Durchlauf veränderten Objekte betrachtet werden. Die besondere Stärke des *Rete*-Algorithmus tritt bei einer großen Regelbasis¹⁹ mit einer großen Faktenbasis und einer hohen Lebensdauer, also zumindest mehr als einem Durchlauf, zu Tage. Wie bereits erwähnt verwenden die meisten heutigen Business Rule Engines den *Rete*-Algorithmus. Dabei wird der ursprüngliche Algorithmus zwar erweitert und für die jeweilige Engine individuell angepasst - *JBoss Rules* verwendet beispielsweise einen speziell für objektorientierte Programmiersprachen optimierten *Rete*-Algorithmus namens *ReteOO*[7] - die

¹⁹je mehr Regeln, desto mehr Knoten können wahrscheinlich geteilt werden

2 Hauptteil

grundsätzliche Arbeitsweise bleibt jedoch immer dieselbe. In [15] wird genauer auf die Performance und Anforderungen von regelbasierten Systemen eingegangen, was hier allerdings nicht näher verfolgt werden soll.

Konfliktbehandlung Wie bereits erwähnt werden bei der Auswertung von Regeln zunächst die Prämissen jeder Regel überprüft. Sobald alle Prämissen erfüllt sind, wird die Regel beim *Rete*-Algorithmus nicht sofort ausgeführt, sie „feuert“ also nicht sofort. Es wird eine so genannte „Aktivierung“ erzeugt, die auf die auszuführende Regel und die Fakten, die die Prämissen der Regel erfüllen, verweisen. Diese Aktivierung wird auf die so genannte „Agenda“, eine zunächst ungeordnete Liste von Aktivierungen, gesetzt.

Sobald die erste Phase der Regelauswertung, also die Überprüfung des Wahrheitsgehalts der Prämissen, beendet ist, werden alle Aktivierungen der Agenda ausgeführt. Da die Möglichkeit besteht, dass die Agenda selbst bei der Ausführung einer Regel verändert wird - eine Regel verändert, entfernt oder fügt eine neue Information in den *Working Memory* ein, die Regeln werden daraufhin ausgewertet und eine neue Aktivierung wird erzeugt oder eine bereits auf der Agenda stehende Aktivierung ist nicht mehr zulässig - ist es, sobald mehr als eine Aktivierung innerhalb der Agenda vorliegt wichtig, in welcher Reihenfolge die Aktivierungen abgearbeitet werden.

Das Vorgehen einer Business Rule Engine bei der Festlegung der Reihenfolge der Aktivierungen nennt man Konfliktbehandlung. Dabei gibt es eine Reihe von Strategien:

statische Priorität Es besteht die Möglichkeit, jeder Regel bereits zum Definitionszeitpunkt eine vorgegebene Priorität zuzuweisen. Die Aktivierungen können dann innerhalb der Agenda aufgrund der Priorität der zugrunde liegenden Regeln sortiert und ausgeführt werden. Das ermöglicht die Erstellung einer exakten Ausführungsreihenfolge aller Regeln vom Benutzer. Nachteilig bei diesem Verfahren ist der, bei einer großen Anzahl von Regeln, hohe Wartungsaufwand: Vor allem bei sehr dynamischen Regelsets mit vielen Veränderungen kann es schnell passieren, dass der Überblick verloren geht oder eine Regel von der Priorität her zwischen zwei anderen eingefügt werden muss, was eine Änderung der Priorität vieler anderer Regeln nach sich ziehen kann. Darüber hinaus nimmt man der Rule Engine durch die Vergabe von Prioritäten die Möglichkeit, die Abarbeitungsreihenfolge zu optimieren.

fachspezifisch Eine weitere Möglichkeit ist es, den Regeln abhängig von den Personen, die sie

erstellt haben, Prioritäten zuzuweisen. So könnten Regeln, die von Abteilungsleitern erstellt wurden, eine höhere Priorität zugewiesen werden, als Regeln, die bspw. von einfachen Sachbearbeitern erstellt wurden. Dieses Verfahren bringt im wesentlichen aber auch die selben Nachteile wie der statische Vergabe von Prioritäten mit sich.

exakt Ein anderer Ansatz zur Konfliktauflösung ist die Priorisierung von exakteren Regeln über allgemeinere Regeln. Dabei wird die „Exaktheit“ einer Regel anhand der Anzahl und Genauigkeit der einzelnen Prämissen bestimmt. Grundlegend für diese Strategie ist die Annahme, dass genauer definierte Regeln im Konfliktfall eine Präzisierung oder vielleicht auch Ausnahme von den allgemeineren Regeln darstellen und deshalb bevorzugt abgearbeitet sind.

Reihenfolge Regeln können auch anhand der Reihenfolge, in der sie in die Rule Engine geladen wurden, priorisiert werden. Gebräuchliche Strategien hierbei sind *FIFO*²⁰, also die Priorisierung von „älteren“ Regeln gegenüber „neueren“, oder *LIFO*²¹, was den gegenteiligen Effekt wie die *FIFO*-Strategie erzielt.

zufällig Eigentlich keine „Strategie“ im herkömmlichen Sinne ist das Vorgehen die Regeln ohne Einsatz einer Agenda sofort bei der Erfüllung aller Prämissen auszuführen. Dennoch ist auch die Option möglich.

Generell ist es sinnvoll alle Regeln so zu formulieren, dass sie für eine korrekte Funktionsweise nicht auf eine bestimmte Abarbeitungsreihenfolge der anderen Regeln angewiesen sind. Zu Konfliktvermeidung und Konfliktbehandlung siehe auch [7] und [35].

Marktübersicht Bis heute ist die Nutzung von Business Rule Engines eher in kleineren Nischen als umfassend zu beobachten. Dennoch existieren auf dem Markt viele verschiedene Lösungen, teils aus dem wissenschaftlichen und universitärem Umfeld, teils als ambitionierte Open-Source-Projekte und natürlich auch als ausgereifte, kostenpflichtige Produkte. Diese Lösungen unterscheiden sich hinsichtlich ihrer Marktreife, Dokumentationsqualität, Unterstützung innerhalb der Community und natürlich auch der Leistungsfähigkeit und dem Funktionsumfang der Engines und ergänzenden Entwicklungstools teilweise erheblich. Aus der Vielzahl der vorhandenen Lösungen sollen hier nur exemplarisch einige interessante Systeme präsentiert werden, um einen Überblick über den Facetten-

²⁰First In First Out

²¹Last In First Out

2 Hauptteil

reichtum des Marktes zu erhalten.

Jess *Jess*²²[13] ist eine der ältesten Business Rule Engines für Java. Es wurde 1995 von Ernest Friedman-Hill an den Sandia National Laboratories in Kalifornien(USA) entwickelt. Ursprünglich war es als reine Java-Portierung des wahrscheinlich ersten allgemein einsetzbaren Experten-Systems *CLIPS*²³ angelegt, hat sich im Laufe der Jahre aber weiterentwickelt und verändert. *Jess* ist für akademische Zwecke kostenlos einsetzbar, muss für den kommerziellen Einsatz aber lizenziert werden. Mittlerweile liegt *Jess* in der Version 7.1 vor, was auf einen hohe Reifegrad der Software schließen lässt. *Jess* bietet einen großen Funktionsumfang. So wird sowohl vorwärts- als auch rückwärtsverkettete Inferenz geboten, ein Eclipse-Plugin zur grafischen Entwicklung von Regeln ist vorhanden und die Java-Rule-Engine-API *JSR 94* wird unterstützt. Dennoch wird *Jess* im kommerziellen Umfeld nur selten eingesetzt, was in erster Linie an der äußerst gewöhnungsbedürftigen Regel-Definitionssprache *Jess rule language* liegen dürfte. Diese Definitionssprache ist an die Programmiersprache *Lisp*²⁴ angelegt und für Sachbearbeiter ohne tiefere Programmierkenntnisse ungeeignet.

ILog JRules Das kommerzielle Produkt *JRules* der Firma *ILog* [12] ist eigentlich mehr als eine reine Rule Engine, sondern ein integriertes „Business Rule Management System“ (*BRMS*). Es bietet viele für den kommerziellen Einsatz in Großprojekten wichtige Features wie eine ausgeklügelte Infrastruktur zur Verwaltung der Regeln. Darüber hinaus sind herausragende Tools zur Definition von Regeln erhältlich. Regeln können sowohl in einer Java-ähnlichen Definitionssprache *ILog Rules Language* definiert werden, als auch in der, der natürlichen Sprache sehr ähnlichen, Definitionssprache *Business Action Language*, was die Entwicklung von Regeln für Sachbearbeiter vereinfacht. Insgesamt stellt *JRules* wohl das zur Zeit umfassendste und komfortabelste System zur Ausführung von Business Rules dar, der Einsatz von *JRules* geht jedoch mit relativ hohen Lizenzkosten einher.

JBoss Rules Die Business Rule Engine *JBoss Rules*²⁵ entstand aus dem Projekt *Drools*, was im Grunde genommen eine für objektorientierte Programmiersprachen optimierte Implementierung des *Rete* Algorithmus namens *ReteOO*[7] ist. Herausragend an *JBoss Rules* ist die GNU GPL²⁶ Lizenz,

²²Java Expert System Shell

²³aus dem Jahre 1985

²⁴entwickelt 1959

²⁵<http://www.jboss.com/products/rules>

²⁶GNU General Public License - siehe auch http://www.redhat.com/licenses/jboss_eula.html

die den kostenlosen Einsatz von *JBoss Rules* auch für kommerzielle Zwecke ermöglicht. Darüber hinaus existiert für *JBoss Rules* eine sehr große Community, die hilft das Projekt voranzutreiben. *JBoss Rules* bieten neben *JSR 94*-Unterstützung und einem Eclipse-Plugin zur grafischen Regelentwicklung auch Unterstützung für so genannte *Domain Specific Languages (DSL)*, die es, ähnlich wie die *Business Action Language* von ILog JRules, ermöglicht, Regeln in natürlicher Sprache zu formulieren.

Zur Konfliktlösung verwendet *JBoss Rules* standardmäßig eine LIFO-Strategie. Jedes Regel-Set verfügt über einen inkrementellen Zähler, der jeder dem Set hinzugefügten Regel eine eindeutige Nummer gibt. Je später die Regel dem Regel-Set hinzugefügt wurde, desto höher ist ihre Priorität und desto „schneller“ wird sie von der Agenda abgearbeitet. Darüber hinaus bietet *JBoss Rules* die Möglichkeit eine manuelle Priorität zu vergeben, die so genannte „salience“.

Gegen Ende der Bearbeitungszeit der vorliegenden Arbeit erschien *JBoss Rules* in der Version 4.0.

JSR 94 Ab dem Jahr 2000 wird innerhalb der Java-Community unter dem Namen *JSR*²⁷ 94 [24] eine einheitliche API für Java-Business Rule Engines erstellt. Ziel dieser API ist, eine einheitliche Schnittstelle für alle grundlegenden Funktionen einer Business Rule Engine zu erstellen. Dabei wird angenommen, dass jede Business Rule Engine ein minimales Set von Operationen unterstützen muss für die zyklische Auswertung von Regeln, also z.B. die Fähigkeit Regeln zu parsen, Objekte innerhalb des *Working Memory* zu manipulieren, Regeln auszuführen und Ergebnis-Objekte zu erhalten. Nicht Teil der Spezifikation ist eine einheitliche Regel-Definitionssprache. Deshalb ermöglicht die *JSR 94* API zwar den Austausch der Business Rule Engine ohne umfangreiche Modifikation des Quellcodes, die Regeln selbst müssen in diesem Fall aber komplett neu erstellt bzw. portiert werden.

Hybris Integration Für die vorliegende Arbeit wurde entschieden die Implementierung auf Basis der Schnittstelle *JSR 94* zu realisieren, um für die Zukunft nicht auf eine Business Rule Engine festgelegt zu sein. Als Business Rule Engine kommen *JBoss Rules* zum Einsatz, da diese mit den *Domain Specific Languages* ein simples aber effektives Mittel zur nutzerfreundlichen Pflege von Regeln bieten. Außerdem ist die Nutzung von *JBoss Rules* auch für den kommerziellen Einsatz kostenlos, und durch den Einsatz der *JSR 94* API können spezielle Features umfangreicherer Business Rule Engines ohnehin kaum genutzt werden.

Die verwendete Version von *JBoss Rules* war die zu Beginn der Arbeit aktuelle Version 3.0.5.

²⁷Java Specification Request

2 Hauptteil

2.1.4 Hybris

Die hybris eCommerce Platform hat ihre Ursprünge in einer studentischen Projektgruppe, die im Jahre 1996 mit dem Ziel der Entwicklung einer standardisierten eCommerce-Lösung gegründet wurde. Ziel des Projekts war damals eine einfache Out-Of-The-Box eCommerce-Lösung zur schnellen und einfachen Inbetriebnahme eines Online-Shops, basierend auf Microsoft-Technologie. Später wurde die hybris GmbH gegründet, mit dem Ziel der kommerziellen Nutzung des Frameworks. Im Laufe der Zeit veränderte sich sowohl die grundlegende Technologie (Migration von Microsoft zu Java J2EE) als auch der konzeptionelle Fokus des Unternehmens (weg von Out-Of-The-Box-Shops hin zu anspruchsvollen Lösungen). Heute zählen viele namhafte Kunden zu den Referenzen von hybris, u.a. die Firmen Puma, Blaupunkt, Bechtle, Nokia oder Virgin Megastores.

Technologie

Wie bereits erwähnt, basiert das hybris Framework, in der aktuellsten für hybris-Partner verfügbaren Version 3.0 Milestone 1, auf Java J2EE Technologie. Im Unterschied zu früheren Versionen wird zur Auslieferung des Web-Contents kein vollständiger Applicationserver²⁸ mehr benötigt, sondern nur noch ein einfacher Servlet-Container (beispielsweise Apache Tomcat [9]). Die Gründe für die Migration von Applicationservern hin zu einfachen Servlet-Containern waren einerseits die schlechte Performance (u.a. durch hohen Ressourcenbedarf) der Applicationservern, andererseits die häufig kritisierte EJB-Persistenzschicht, welche durch ein selbst entwickeltes Persistenzframework²⁹ abgelöst wurde, und somit die Verwendung von Applicationservern überflüssig machte. Für die persistente Speicherung der Shop-Daten kommen gewöhnliche relationale Datenbanksysteme wie Oracle oder MySQL zum Einsatz. Grundsätzlich werden alle JDBC-kompatiblen Datenbanksysteme unterstützt, solange sie eine Reihe von Anforderungen (u.a. Unterstützung von Sub-Queries) erfüllen. Als Laufzeitumgebung kommt Java ab Version 5.0 zum Einsatz.

²⁸wie der JBoss Application Server oder andere J2EE-kompatible Applicationserver

²⁹grundsätzlich ähnlich zu Hibernate [10]

Modularität

Die hybris Plattform ist keine monolithische Gesamtanwendung, sondern besteht aus vielen Einzelsystemen, den so genannten Extensions[4]. Extensions sind thematisch abgegrenzte Einheiten, die Anforderungen einzelner Aufgaben kapseln. Die Extensions untereinander stehen in Abhängigkeitsbeziehungen. So sind im Anwendungskern, ebenfalls eine Extension namens *core*, alle wichtigen Basistypen definiert, worauf alle Extensions zugreifen. Weitere wichtige Extensions, die im Basisumfang von hybris enthalten sind, sind *hmc* (Administrationsoberfläche), *Mediaweb* (für die Verwaltung von Medien aller Art, vor allem Bildern), *europe1* (Abbildung des Preis-Modells für europäische Rahmenbedingungen) oder *storefoundation* (Shop-Frontend inkl. Beispielshop). Dabei greift *Storefoundation* sowohl auf *core*, *Mediaweb* als auch *europe1* zu und ist ohne das Vorhandensein dieser Extensions nicht ohne Anpassungen lauffähig. Sowohl die im Rahmen dieser Diplomarbeit entstehende, projektspezifische Promotion-Implementierung als auch die für den Business-Rule basierten Ansatz benötigte Business Rules-Engine werden als eigene Extension konzipiert. Andere Extensions, vor allem *Storefoundation* werden auf diese Extensions zugreifen, die neu anzulegenden Extensions selber werden in die Administrationsoberfläche *hmc* integriert.

Typsystem

Zur Abbildung von Business Objekten³⁰ verwendet hybris ein proprietäres Typsystem[4]. Die verwendeten Objekte werden grundsätzlich in XML-Dateien angegeben, wo die Namen der Objekte, ihre Beziehung zu anderen Objekten (beispielsweise innerhalb einer Vererbungshierarchie) und die Attribute, über die diese Objekte verfügen, definiert werden können. Wichtige Business Objekte sind beispielsweise *User* oder *Employee* (welche beide von *Principal* erben), Warenkorb und Bestellung (Subtypen von *AbstractOrder*) oder *Product*. Für jedes Business Objekt existieren drei wichtige Komponenten:

Java-Objekte Beim Kompilieren der Anwendung werden zu allen in XML definierten *Items* Java-Klassen erzeugt.

Persistenz-Schicht Beim Initialisieren bzw. Updaten der Anwendung werden für die in XML definierten *Items* Tabellen in der Datenbank erzeugt.

³⁰wie Produkte, Bestellungen etc.

2 Hauptteil

Administrationsschicht In einer separaten XML-Datei kann angegeben werden, wie das jeweilige Objekt später in der Administrationskonsole angezeigt wird.

Hierarchie Prinzipiell müssen alle definierten Objekte immer von bereits existierenden *Items* erben, die einige notwendige Attribute - wie den Primary Key, der grundsätzlich für alle Objekte innerhalb der hybris Platform einen eindeutigen Identifier darstellt - definieren. Im ursprünglichen Lieferumfang der hybris Platform sind bereits die wichtigsten Objekte für den Betrieb von Online-Shops enthalten, wie auf Abbildung 2.3 zu sehen ist.

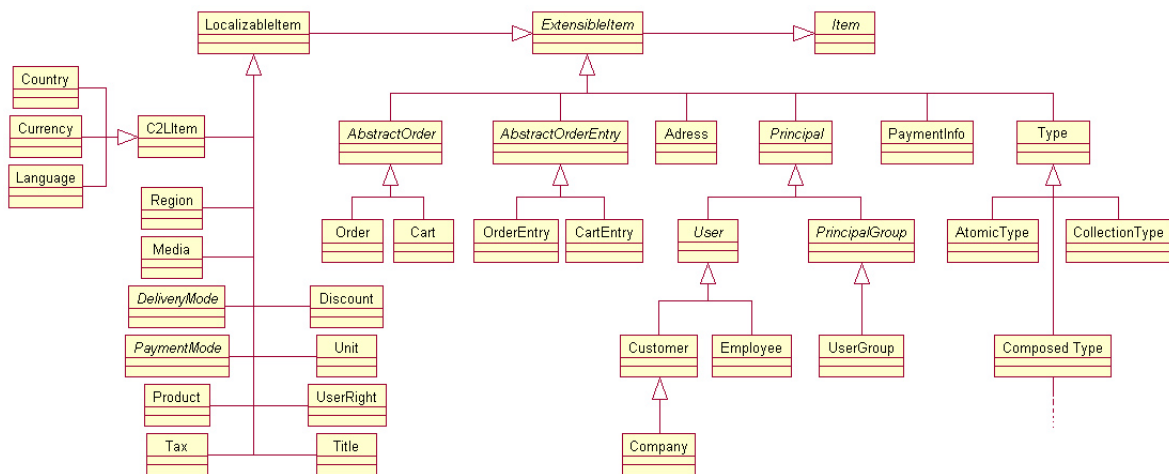


Abbildung 2.3: Hybris Item-Hierarchie (aus [4])

Jedes Business Objekt der hybris Platform erbt immer vom obersten Basisitem *Item*. Dieses *Item* bringt bereits die grundlegendsten Eigenschaften für alle davon erbenden *Items* mit, wie die Fähigkeit, in der Datenbank abgespeichert zu werden, und vom hybris Caching-Framework verarbeitet zu werden. Die, direkt von *Item* erbende, Klasse *ExtensibleItem* erweitert die *Items* um die Fähigkeit beliebige Attribute ohne Änderung des Quellcodes zu speichern bzw. auszulesen, was auch ermöglicht, dass alle von *ExtensibleItem* erbenden Instanzen zur Laufzeit um Attribute erweitert werden können. Wiederum von *ExtensibleItem* erbt die Klasse *LocalizableItem* die die Internationalisierung der Platform ermöglicht. *LocalizableItems* ermöglichen es, einem Attribut mehrere Werte³¹ zuzuweisen. Während

³¹genauer: für jede in der Platform bekannte Sprache einen eigenen Wert

also bei *ExtensibleItems* nur ein Wert pro Attribut möglich ist (bspw. „name=nice product“) bieten *LocalizableItems* die Möglichkeit, unabhängige Werte für alle Sprachen zu definieren (bspw. „name(en)=nice product, name(de)=schönes Produkt“ etc). Für alle Erweiterungen des Typkonzepts existiert die Basisklasse *GenericItem*. *GenericItem* erbt von *LocalizableItem* ohne dessen Funktionalitäten zu erweitern und dient lediglich als Markierung für neu definierte Objekte, es wird jedoch empfohlen, dass alle selbst definierten *Items* von *GenericItem* erben.

Manager-Klassen Wie bereits erwähnt, ist die Hybris-Plattform ein modulares System aus weitgehend unabhängigen Einzelkomponenten (Extensions). Jede dieser Komponenten verfügt über eine Manager-Klasse, die alle Funktionalitäten, die für andere Komponenten von Nutzen sein könnten, kapseln. Durch diese - dem bekannten Façade-Pattern [20] entsprechende - Architektur, wird eine einheitliche und kompakte API zur Verfügung gestellt, die die Komplexität der Gesamtkomponente verbirgt. Wichtige Manager-Klassen sind *OrderManager* (zur Verwaltung von Warenkörben und Bestellungen), *UserManager* oder *ProductManager*.

JaloSession Die Klasse *JaloSession* ist der zentrale Zugangspunkt zur Hybris Plattform. In der *JaloSession* werden alle für eine Nutzersitzung relevanten Daten wie Warenkorb oder der angemeldete Benutzer gespeichert. Darüber hinaus enthält die Klasse *JaloSession* Methoden um Zugang zu allen verfügbaren Managerklassen sowie zu den hybris-eigenen Suchen³² zu erhalten. Technisch gesehen handelt es sich bei der Klasse *JaloSession* um eine Wrapper-Klasse zum eigentlichen *HttpSession*³³-Objekt, das zur Sitzungsverfolgung³⁴ verwendet wird.

AbstractOrder *AbstractOrder* ist die Superklasse von *Cart* (Warenkorb) und *Order* (Bestellung). Die *AbstractOrder*-Klasse übernimmt die Verwaltung von einzelnen „Zeilen“ (*AbstractOrderEntry*) des Warenkorbs (*CartEntry*) bzw. der Bestellung (*OrderEntry*) sowie (mit Hilfe der angegebenen *PriceFactory*, also z.B. *Europe1* für den europäischen Raum) die Berechnung der Einzelpositionen und der Summe, der Steuern und Rabatte. Das Hinzufügen oder Entfernen neuer Bestellpositionen oder Rabatte sind Methoden der *AbstractOrder*.

Principal Die abstrakte Klasse *Principal* stellt eine „Userentität“ dar. Subklassen von *Principal* sind sowohl *User* (für den einzelnen Benutzer) als auch *UserGroup*. Eine *UserGroup* ist generell eine Ansammlung von weiteren *Principals*, kann also sowohl User als auch User-Gruppen ent-

³²Generic- und FlexibleSearch, die hier aber nicht eingehend vorgestellt werden sollen

³³Java Interface javax.servlet.http.HttpSession

³⁴engl. Session Tracking

2 Hauptteil

halten. Jeder User kann Mitglied in mehreren User-Gruppen sein. Dadurch entsteht eine hierarchische Nutzerstruktur. Alle Zugriffsrechte werden immer auf *Principals* vergeben, so dass Rechte sowohl für einzelne Nutzer wie auch auf Nutzer-Gruppen, die die Rechte dann an die einzelnen Mitglieder „vererben“, definiert werden können. Einzelne Nutzer verfügen in der Regel über eine Nutzerkennung und ein Passwort und können sich damit im Shop-Frontend oder auch in der Administrationskonsole authentifizieren³⁵.

Product Eine wichtige Kernkomponente der Shopfunktionalität stellt die Klasse *Product* dar. Sie verwaltet Produkte und Produktvarianten. Ein Produkt kann über mehrere Varianten verfügen, beispielsweise kann ein T-Shirt (Produkt) die Varianten rotes T-Shirt und blaues T-Shirt haben. Wichtige Attribute des Produktes sind der Produktcode und mehrere Preiszeilen, z.B. für unterschiedliche User-Gruppen.

Media Die *Media*-Klasse ist eine Wrapper-Klasse für alle Arten von Dateien (meistens Grafiken), die innerhalb der hybris Plattform verwaltet werden können. Diese Medias verfügen über einen Code und eine URL, mit der sie im Shop-Frontend angezeigt werden können.

Java Wie bereits erwähnt, wird aus allen definierten Typen beim Kompilieren der Plattform Java-Code erzeugt. Im folgenden soll exemplarisch dargestellt werden welche Klassen erzeugt werden und wie diese aussehen. Als Beispiel wird hier ein einfaches *ExampleItem* definiert, welches von *GenericItem* erbt und als einziges Attribut einen Namen vom Typ *java.lang.String* enthält:

```
<itemtype code="ExampleItem" extends="GenericItem"
  deployment="com.arithnea.rules.entity.Rule" autocreate="true"
  generate="true">
  <attributes>
    <attribute qualifier="name" type="java.lang.String" >
      <defaultvalue>' just a string'</defaultvalue>
      <persistence type="property"/>
    </attribute>
  </attributes>
</itemtype>
```

Listing 2.1: Typdefinition in XML

³⁵Single Sign On

Jedes Business Objekt wird durch das XML-Tag *itemtype* definiert, woraus dann in hybris Java-Klassen erzeugt werden. Das Attribut *code* definiert die Bezeichnung, unter der der Typ später in der hybris Plattform zu finden ist. Darüber hinaus entspricht, sofern wie in o.g. Beispiel nicht explizit angegeben, der Klassenname für das definierte Objekt dem angegebenen Code. Mit dem Attribut *extends* wird ähnlich wie in der Java-Syntax angegeben, von welchem Typ der neu definierte Typ erben soll. In o.g. Beispiel erbt das *ExampleItem* von *GenericItem*, was die Standardeinstellung für *Items* ohne tatsächliche Vererbungsbeziehung ist. Das *deployment*-Attribut gibt den hybris-internen Deployment-Code an, dessen Bedeutung später noch detaillierter erklärt wird. Die Attribute *auto-create* und *generate* besagen ob das jeweilige *Item* beim Initialisieren der Plattform bereits angelegt wird (*autocreate*) und ob Java-Code produziert wird. Normalerweise werden beide Attribute standardmäßig auf *true* gestellt. Innerhalb des *Attributes*-Tags werden dann die jeweiligen Attribute, über die der neu definierte Typ verfügen soll, festgelegt. In o.g. Beispiel wird nur ein einziges Attribut namens *name*, welches vom Typ *java.lang.String* ist, definiert. Zusätzlich wird definiert, dass der Standardwert dieses Attributes „just a string“ ist. Das heißt, sobald eine Instanz des *ExampleItems* angelegt wird, wird das *name*-Attribut auf „just a string“ vor belegt. Würde man hier einen lokalisierbaren String benötigen, so müsste das *type*-Attribut des *Attribute*-Tags statt auf *java.lang.String* auf *localized:java.lang.String* gesetzt werden. Das *Persistence*-Tag innerhalb des *Attribute*-Tags gibt an wie die Persistenzschicht das Attribut behandeln soll. Gebräuchliche Attributwerte sind hier entweder „property“, was bedeutet, dass das Attribut persistent in der Datenbank gespeichert wird oder „jalo“, was bedeutet, dass das Attribut nicht persistent gespeichert wird, sondern der Wert für dieses Attribut zur Laufzeit ermittelt wird. Sofern man das Attribut *generate*=“true“ gesetzt hat, werden beim Kompilieren der hybris Plattform für jeden Typen zwei Java-Klassen erzeugt. Die Klassennamen für diese neu erzeugten Klassen entsprechen *Generated*<Itemcode> bzw. nur <Itemcode>, wobei <Itemcode> von *Generated*<Itemcode> erbt. Für das o.g. Beispiel würden also die Klassen *GeneratedExampleItem* und *ExampleItem* (*extends GeneratedExampleItem*) erzeugt. Innerhalb der *Generated*-Klasse werden unter anderem für alle Attribute des neu geschaffenen *Items* *getter*- und *setter*-Methoden gemäß der Java-Bean-Konvention³⁶ erzeugt. Diese Methoden greifen dann auf das Cache- und Persistenzframework von hybris zu, um den Attributwert aus dem Cache oder der Datenbank zu ermitteln bzw. abzuspeichern. Eigener Code kann dann in die davon erbende *Item*-Klasse (im oberen Beispiel also *ExampleItem*) implementiert werden. Um beispielsweise jedes mal wenn das

³⁶für das Attribut *name* die Methoden *getName()* und *setName(String name)* etc.

2 Hauptteil

Attribut *name* eines *ExampleItems* gelesen wird ein Log-Statement zu erzeugen, müsste man in die Klasse *ExampleItem* folgenden Code einfügen:

```
public String getName()
{
    log.info(" `Es wird gerade das Attribut `Name` angefragt aus dem
        Item "` +this);
    return super.getName();
}
```

Listing 2.2: Beispiel für modifizierte getter-Methode

Für Attribute deren Persistenz auf *jalo* eingestellt ist wird innerhalb der *Generated*-Klasse lediglich eine abstrakte *getter*- und *setter*-Methode implementiert, so dass man den Code der für die Auswertung der *jalo*-Attribute zuständig ist auf jeden Fall von Hand implementieren muss.

Persistenzschicht Beim Initialisieren der Platform werden durch automatisch ablaufende Skripte aus diesen XML-Daten entsprechende Tabellen in der Datenbank erzeugt (siehe Tabelle 2.1). Für jeden Typen wird dabei, sofern gewünscht, eine eigene Tabelle entsprechend dem definierten Deployment erzeugt. Für alle Attribute des jeweiligen Typen, auch für von Supertypen geerbte Attribute, wird dabei eine Spalte innerhalb der Tabelle erzeugt. Der atomare Primärschlüssel der Tabelle ist immer die so genannte PK³⁷, ein innerhalb der hybris Platform eindeutiger Objekt-Identifizier.

Falls für ein *Item* kein eigenes Datenbank-Deployment angegeben wird, so werden alle Objekte dieses Typs in die Tabelle des Supertyps, sofern dieser über ein eigenes Deployment verfügt, geschrieben und diese Tabelle um die entsprechenden Attribute bzw. Spalten erweitert. Die Suche nach einem Supertyp mit eigenem Deployment und damit eigener Datenbank-Tabelle wird rekursiv weitergereicht, bis eine passende Tabelle gefunden wird, was spätestens bei dem Typ *GenericItem*, von dem alle selbst definierten *Items* erben sollen, der Fall ist. Im Extremfall, wenn überhaupt keine eigenen Deployments für alle definierten Typen angegeben werden, verfügt also die Tabelle *genericitem* über so viele Spalten wie es Attributdefinitionen gibt. Nachdem die Auswertung von Attributen grundsätzlich *lazy* erfolgt, d.h. Attributwerte werden erst beim direkten Zugriff auf das jeweilige Attribut aus der Datenbank geladen, entstehen bei der normalen Arbeit mit der hybris Platform (aus Usersicht: Anzeigen

³⁷Primary Key

Field	Type	Null	Key	Default	Extra
hjmpTS	bigint(20)	YES		NULL	
PK	bigint(20)	NO	PRI		
createdTS	datetime	NO			
modifiedTS	datetime	YES		NULL	
aCLTS	bigint(20)	YES		0	
TypePkString	bigint(20)	NO	MUL		
OwnerPkString	bigint(20)	YES	MUL	NULL	
propTS	bigint(20)	YES		0	
p_name	varchar(255)	YES		just a string	

Tabelle 2.1: Beispiel-Tabelle in MySQL

einer Produktseite im Online-Shop, aus Administrationssicht: Bearbeiten eines Typen in der Administrationskonsole), extrem viele einzelne Anfragen, was zu hoher Last auf den Datenbanksystemen führt. Darum verfügt die hybris Plattform über einen Caching-Layer, wo häufig angefragte Objekte bzw. Attributwerte zwischengespeichert werden. Die genaue Funktionsweise der Caching-Schicht ist an dieser Stelle aber nicht relevant.

Exkurs zur Clusterfähigkeit: Die hybris Plattform ist für aufwändige Shopsysteme mit vielen gleichzeitigen Benutzern und damit verbunden vielen Page Impressions pro Sekunde ausgelegt. Da die Leistungsfähigkeit eines einzelnen Servers begrenzt ist, ist es notwendig die Plattform auf mehreren verteilten Servern laufen lassen zu können³⁸. So ist es denkbar und üblich, dass mehrere Applicationserver, die hierbei den Flaschenhals darstellen[4], gemeinsam auf einen Datenbankserver zugreifen. Wenn einer der Applicationserver eine Änderung an der Datenbank vornimmt (bspw. ein Produkt wird bestellt und der Lagerbestand des Produktes um 1 verringert), so werden alle anderen Applicationserver mittels UDP-Multicast von dieser Änderung unterrichtet. Sobald ein Applicationserver die Nachricht bekommt, dass ein *Item* in der Datenbank geändert wurde, so wird jede von dem *Item* gecachte Information aus dem Cache des Applicationserver entfernt (invalidiert) und beim nächsten Zugriff aus der Datenbank neu geladen. Dieser Vorgang geschieht für den Entwickler der hybris-Anwendung vollkommen transparent, d.h. er weiß nicht ob ein angefragter Attributwert aus dem Cache geladen wird oder aus der Datenbank, die Kenntnis davon ist aber wichtig bei der Integration einer Business Rule

³⁸Clustering

2 Hauptteil

Engine, wie später noch genauer dargelegt werden wird.

Administrationskonsole hmc

Die Administrationskonsole *hmc*³⁹ bietet ein Administrationsbackend für die Betreiber der hybris-Plattform. Es handelt sich um einen webbasierten Thin-Client, der mit allen gängigen Browsern genutzt werden kann. In der *hmc* können alle innerhalb der hybris Plattform definierten Business-Objekte angelegt, gelöscht und manipuliert werden. In früheren Versionen der hybris Plattform wurde die Administrationsoberfläche noch *webmc* genannt, so dass man vor allem in Konfigurationsdateien etc. noch oft auf diese Bezeichnung stößt. Jede Extension, deren Business Objekte in der *hmc* zugänglich sein sollen, enthält eine XML-Datei, in der die Darstellung der Business Objekte definiert werden kann. Dadurch ist die Darstellung der jeweiligen Objekte innerhalb der definierenden Extension gekapselt.

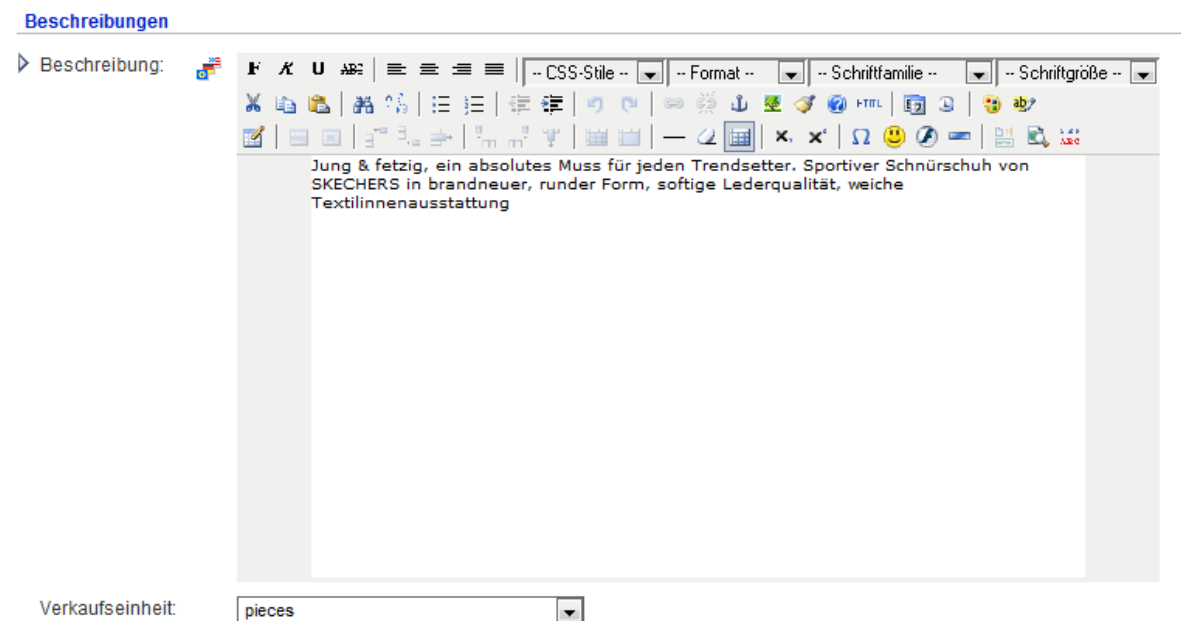


Abbildung 2.4: Darstellung der Attribute description und unit

Der XML-Code, um beispielsweise die Sektion „Beschreibungen“ auf Abbildung 2.4 zu definieren, sieht folgendermaßen aus:

³⁹hybris Management Console

```

<section name="product.descriptions">
  <listlayout>
    <attribute name="description">
      <wysiwygeditor/>
    </attribute>
    <attribute name="unit"/>
  </listlayout>
</section>

```

Listing 2.3: Ausschnitt WebMC.xml

Er beschreibt lediglich dass für das Attribut *description* des angezeigten Business Objekts (in diesem Fall *Product*) ein WYSIWYG⁴⁰-Editor verwendet werden soll, für das Attribut *unit* wird der standardmäßig definierte Editor (in diesem Fall eine einfache SelectBox) verwendet.

Shop-Frontend

Für die spätere Darstellung der Online-Shops existiert eine eigene Extension namens *Storefoundation*. Sie bildet das Web-Interface, welches Kunden zu Gesicht bekommen. Im Wesentlichen handelt es sich um eine Java-basierte Webanwendung, die fortschrittliche Technologien wie JSP[5] (Java-ServerPages), JSF[6] (JavaServerFaces) und Facelets[2] verwendet. Es ist nicht Ziel der Arbeit diese Technologien detailliert darzustellen, deswegen sei hier nur eine kleine Einführung in die relevanten Funktionsweisen gegeben: Einer der Kritikpunkte an der JSP-Technologie ist seit jeher die mangelnde Trennung der Darstellung der Webseite (durch HTML und CSS definiert) von der dahinter stehenden Business-Logik. Durch den Einsatz von JSF (welches eines auf JSP basierendes Framework ist) erreicht man eine stärkere Trennung der beiden Komponenten. Die Webseiten selbst enthalten wie statische HTML-Seiten keinerlei Java-Code mehr, dieser wird in Java-Objekte, so genannte Backing Beans, ausgelagert. Für die hier behandelte Promotion-Funktionalität ist es notwendig, solche Backing Beans zu entwerfen und in die im Beispielshop vorhandenen XHTML-Seiten einzubinden, um Promotions für User kenntlich zu machen.

⁴⁰What You See Is What You Get - Grafischer Editor

2.2 Projektspezifische Implementierung der Promotions

Im Folgenden wird dargelegt, wie eine projektspezifische Implementierung von Promotions innerhalb der hybris Platform in einem aktuellen Großprojekt durchgeführt wurde.

Das Projekt umfasst den kompletten Relaunch des Internet Shops des englischen Unternehmens Virgin Megastores, einer Tochtergesellschaft der bekannten Virgin Group. Virgin Megastores handelt in erster Linie mit Musik-CDs, DVDs und Computerspielen aller Art, aber auch mit elektronischem Zubehör, Zeitschriften etc.

Wie bereits erwähnt, sind die Praxis der Promotions vor allem im angloamerikanischen Raum sehr bewährt und wird durchgängig eingesetzt, weshalb für das Projekt von Anfang an die Anforderung zur technischen Umsetzung innerhalb der hybris Platform existierte. Im Rahmen dieser Arbeit wurde bei der Entwicklung einer speziell auf die Anforderungen von Virgin Megastores zugeschnittenen Lösung mitgewirkt.

2.2.1 Anforderungen

Zu den Anforderungen von Virgin Megastores gehörte eine Liste von Promotions, die umgesetzt werden sollten. Alle hier genannten Promotions sollten hinsichtlich eines Zeitraums („nur vom 01.12.2007-31.12.2007“) und eines Nutzerkreises („nur für Neukunden“) eingeschränkt werden können.

Multibuy Eine klassische Multibuy-Promotion nach dem Muster „Kaufe x Produkte für y €“ sollte zur Verfügung stehen. Dabei sollten die qualifizierenden (also die Promotion auslösenden) Produkte in einer Liste gepflegt werden können. Die Promotion sollte mehrfach anwendbar sein (sofern ausreichend Produkte im Warenkorb vorhanden sind) und die qualifizierenden Produkte bei der Anwendung der Promotion „verbrauchen“. Diese Produkte sollten also für andere Promotions nicht mehr verfügbar sein.

BOGOF Eine Abart der oben genannten Multibuy-Promotion ist die so genannte BOGOF (Buy One Get One Free) Promotion. Dabei ist der Name nicht wörtlich zu nehmen, denn unter dem Sammelbegriff sind auch andere als die klassische „Kaufe eins und du bekommst ein weiteres umsonst“-

2.2 Projektspezifische Implementierung der Promotions

Promotion zu verstehen. Unter anderem sollte es hierbei möglich sein die Anzahl der qualifizierenden Produkte direkt anzugeben, also „Kaufe x Produkte für den Preis von x-1“. Genau wie in der oben genannten Multibuy Promotion werden die qualifizierenden Produkte hierbei verbraucht.

Stepped Promotion Eine weitere Verfeinerung der Multibuy-Promotion sind Staffelrabatte. Dabei wird nicht nur eine Anzahl und Gesamtpreis für die gegebene Anzahl von Produkten, sondern mehrere angegeben. Je größer die Anzahl wird, desto größer wird auch der Rabatt, der gewährt wird. Ein Beispiel wäre „Kaufe 2 für 20€, kaufe 4 für 35€“. Nachdem auch diese Promotion mehrfach anwendbar sein soll (und die qualifizierenden Produkte dabei verbraucht) ist es wichtig, dass auch immer die höchste verfügbare Promotionsstufe angewendet wird, anstatt mehrfach eine geringere anzuwenden. Im Beispiel würde das bedeuten: Wenn 6 qualifizierende Produkte im Warenkorb sind, so soll nicht 3 mal die „kaufe 2 für 20€“-Promotion (insgesamt 60€), sondern 1 mal die „kaufe 4 für 35€“-Promotion und 1 mal die „kaufe 2 für 20€“-Promotion (insgesamt 55€) angewendet werden.

Perfect Partner - Spend x€ get product y for z€ Diese Promotion ist ein Fall einer klassischen Linksave-Promotion. Dabei muss ein bestimmter Schwellwert beim Gesamtwert des Warenkorbs überschritten werden um die Promotion verfügbar zu machen. Sobald dieser Schwellwert überschritten ist, soll ein vordefiniertes Produkt zu einem reduzierten Preis angeboten werden. Sobald der Schwellwert wieder unterschritten wird (bspw. durch das Entfernen eines Produktes aus dem Warenkorb) soll auch die Promotion wieder rückgängig gemacht werden und das Produkt wieder zum normalen Preis verfügbar sein.

Perfect Partner - Buy product x, get product y for z€ Ähnliche der oben genannten Perfect Partner-Promotion ist auch diese Promotion eine klassische Linksave-Promotion. Im Gegensatz zur oben genannten ist aber zur Qualifizierung dieser Promotion kein absoluter Warenkorbwert erforderlich, sondern ein Produkt aus einer speziellen Liste von Produkten. Sobald sich eins dieser Produkte im Warenkorb befindet, kann ein anderes zu definierendes Produkt zu einem reduzierten Preis erworben werden.

Bundle - Buy product a, product b, product c for z€ Diese Promotion ist im Grunde eine Cross-Selling-Promotion. Für eine fest definierte Zusammenstellung von Produkten wird ein fester

2 Hauptteil

Endpreis angegeben, der geringer ist als die Summe der Einzelpreise der Produkte. Idealerweise sollen die Zusammenstellungen ergänzende oder artverwandte Produkte beinhalten.

Spend over x€ and get y Die letzte von Virgin Megastores angeforderte Promotion ist artverwandt mit der ersten Perfect Partner-Promotion. Es soll ein fester Warenkorbwert hinterlegt werden können, ab dem gewisse Vergünstigungen „freigeschaltet“ werden können. Als mögliche Vergünstigungen waren kostenfreie (oder kostenreduzierte) Lieferung, Geschenke, Gutscheine für zukünftige Einkäufe etc geplant. Mit den Gutscheinen für zukünftige Einkäufe ist auch eine klassische zeitversetzte Promotion gegeben.

Frontend-Integration

Die Integration in das Shop-Frontend teilt sich in zwei unterschiedliche Bereiche, das Bewerben (An teasern) von verfügbaren Promotions auf Produktdetail- oder Übersichtsseiten im normalen Shopbereich und das Anzeigen von vollständig oder teilweise qualifizierten Promotions im Warenkorb und der Bestellübersicht (Checkout-Bereich).

Teaser Auf den Kategorienseiten sollten Produkte, für die Promotions existieren, hervorgehoben präsentiert werden. Dabei sollte es eine Möglichkeit zur Einflussnahme von Produktmanagern o.ä. geben, so dass bspw. die Anzahl von hervorgehoben präsentierten Produkten begrenzt werden kann. Auf den Produktdetailseiten sollte es spezielle Bereiche geben, wo die für das jeweilige Produkt verfügbaren Promotions mit Text oder einem Bild beworben werden können.

Checkout Im Checkout-Bereich (also Warenkorb und Bestellübersichtsseiten) sollten sowohl die bereits erreichten und eingelösten Promotions, als auch teilweise erreichte Promotions übersichtlich angezeigt werden. Es sollte ersichtlich sein, welche Produkte welche Promotions qualifiziert haben, welche Produkte gegebenenfalls umsonst und/oder automatisch hinzugefügt wurden, welche Rabatte gewährt wurden etc.

Darüber hinaus sollte auch für teilweise erfüllte Promotions angezeigt werden, was zu tun ist, um diese vollständig zu erfüllen.

Usability

Nachdem die Promotions in erster Linie von Produktmanagern gepflegt werden, sollten diese durch ein einfaches Benutzerinterface dazu befähigt werden. Die gesamte Promotion-Funktionalität sollte über die hybris-Administrationskonsole *hmc* gepflegt werden können.

Insbesondere ist es erforderlich, dass keine tiefer gehenden Verständnisse der hybris Plattform oder von Programmierung erforderlich sind, und ein Produktmanager nach kurzer Einarbeitungsphase autonom mit den Promotion-Funktionalitäten zurecht kommt.

Nicht nur bei der Pflege der Daten, sondern auch bei der Eingabe von Bestellungen in der *hmc* muss (für Promotions) ein hohes Maß an Nutzerfreundlichkeit gegeben sein. Notwendig ist dies z.B. in Call-Centern, wo Bestellungen per Telefon abgegeben werden und direkt in der *hmc* angelegt werden.

Persistenz

Aus Gründen der Nachvollziehbarkeit ist es notwendig, dass alle Promotions bzw. deren Auswirkungen auf eine erfolgte Bestellung dauerhaft gespeichert werden. So kann auch nach einigen Wochen, wenn die ursprünglichen Promotions vielleicht gar nicht mehr aktiv sind, noch nachvollzogen werden, wie ein bestimmter Preis o.ä. zustande gekommen ist.

Für die bereits erwähnte Call-Center-Fähigkeit kann es auch notwendig sein, eine Bestellung so aufzunehmen, als wäre sie an einem anderen Tag erfolgt. In diesem Fall müssen die Promotions auch für den eingegebenen Tag berechnet werden können (bspw. im Falle von Promotions mit eingeschränkter Gültigkeitsdauer).

2.2.2 Implementierung

Extension

Die Promotion-Funktionalität für das Virgin Megastores Projekt wurde von Anfang an als projektspezifische, speziell auf den Kunden zugeschnittene Lösung entwickelt. Daher hätte die Möglichkeit bestanden die Promotion-Funktionalität direkt in die Frontend-Extension zu integrieren. Nach kurzer Überlegung wurde jedoch entschieden die Promotion-Funktionalität in eine eigene Extension auszulagern. Dadurch wird eine bessere Kapselung des Codes erreicht, was der Wartbarkeit des Systems

2 Hauptteil

zu Gute kommt. Darüber hinaus wird auf diese Weise die Weiterentwicklung zu einem allgemeineren Konzept begünstigt.

Im Gegensatz zu herkömmlichen Extensions, die im Normalfall ohne Anpassungen anderer Extensions lauffähig sind und ohne „Installation“ in das System integriert werden können, wurde jedoch diese strikte Trennung bei der Entwicklung nicht konsequent durchgehalten. Deshalb muss die Promotion-Extension mit einem eigenen ANT-Skript⁴¹ installiert werden, wobei Teile der *Storefoundation*-Extension modifiziert werden. Aus diesem Grund ist die *Promotion*-Extension, im Gegensatz zu anderen Extensions, auch auf eine bestimmte Versionsnummer der hybris-Plattform (Version 3 Milestone 1) festgelegt.

Architektur

Typisierung In der Promotion-Lösung für Virgin Megastores gibt es für jede Promotion einen eigenen Typen. Insgesamt wurden nach genauer Analyse der Anforderungen zehn verschiedene Arten von Promotions identifiziert, die jeweils einem hybris Typen mit eigenen Java-Klassen entsprechen. Daraus resultiert ein relativ starres Framework mit klar untereinander abgegrenzten Promotions. Diese starre Architektur bringt es mit sich, dass die Erweiterung der Promotion-Funktionalität nur sehr schwierig möglich und aufwändig ist. Sobald beispielsweise eine neue Promotion, die vom Wesen her nicht den bereits existierenden Promotion-Typen entspricht, hinzugefügt werden soll, so muss das gesamte Datenmodell angepasst und neuer Java-Code geschrieben werden. Deshalb ist es bei diesem Modell nicht möglich, dass zur Laufzeit von Sachbearbeitern ohne Programmierkenntnisse neuartige Promotions angelegt werden.

Dennoch wurde aus folgenden Gründen beschlossen diese Architektur zu verwenden:

- Durch die Verwendung von hart-codierten, vordefinierten Promotions ist die Entwicklungszeit wesentlich kürzer als für den Entwurf eines generischen Frameworks. Einzelne Promotions können hierbei schnell fertig gestellt und einzeln getestet werden, ohne zuvor einen aufwändigen Rahmen entwickeln zu müssen.
- Da jede Promotion einen eigenen hybris Typen darstellt, ist die Pflege in der Administrationskonsole sehr einfach. Die Darstellung der Promotion bzw. der Attribute kann genau auf die jeweilige Promotion angepasst werden, so dass es auch für Sachbearbeiter ohne tieferes

⁴¹Another Neat Tool - Werkzeug zum automatisierten Erzeugen von Programmen ähnlichen make

2.2 Projektspezifische Implementierung der Promotions

Verständnis der hybris Plattform möglich ist, die Promotions zu pflegen.

- Nachdem jede einzelne Promotion über eigene Java-Sourcen verfügt, ist es einfach möglich für jede Promotion genau zugeschnittenen Code zu schreiben. Dadurch hat man beliebigen Zugriff auf die gesamte hybris-Programmierschnittstelle und verfügt praktisch über unbegrenzte Möglichkeiten zur Gestaltung von Promotions.

Generell sind alle Promotions Subtypen von *AbstractPromotion*. Dieser Typ bringt grundlegende Funktionalitäten, die alle Promotions benötigen, mit. Neben anderen, die später noch eingehender betrachtet werden, sind das der Start- und Endzeitpunkt, wann die Promotion verfügbar ist, ein Code, Titel und eine Beschreibung sowie einen Schalter, ob die Promotion überhaupt aktiv (enabled) ist, oder nicht. Darüber hinaus beinhaltet die *AbstractPromotion* eine URL, die auf eine Seite im Shop verweisen kann, wo die Promotion genauer vorgestellt werden kann.

Unterscheidung Product- und Order-Level-Promotion Die einzelnen Promotions werden jedoch nicht direkt von *AbstractPromotion* abgeleitet, denn es gibt noch eine zwischen-geschaltete Abstraktionsstufe. Alle Promotions werden eingeteilt in Product- und Order-Level-Promotions.

Alle Promotions, die lediglich von der gesamten Bestellung an sich abhängig sind bzw. auch nur auf diese angewandt werden, sind Order-Level-Promotions. Dabei enthält der Typ *OrderPromotion* selbst keinerlei weitere Attribute, sondern dient lediglich als „Markierung“ für die ableitenden Promotions. Im Gegensatz dazu sind Promotions, die von einzelnen Produkten oder Bestellzeilen abhängig sind Product-Level-Promotions. Der Typ *ProductPromotion* enthält unter anderem ein *ProductBanner*, ein hybris *Media*-Objekt, welches auf Produktseiten die von dieser *ProductPromotion* betroffen sind, angezeigt werden kann. Damit wird der Anforderung von Teasern im Shop-Frontend Rechnung getragen.

Vorbedingungen und Filterlogik Damit eine Promotion gültig werden kann, müssen zunächst gewisse Vorbedingungen erfüllt sein. Je nach Promotion können diese Vorbedingungen ganz unterschiedlicher Natur sein, es gibt jedoch auch allgemeinere Vorbedingungen, die für ganz unterschiedliche Promotion dennoch gleich sein können.

Beispiel: Bei einer Promotion vom Typ *OrderThresholdChangeDeliveryModePromotion* ist eine

2 Hauptteil

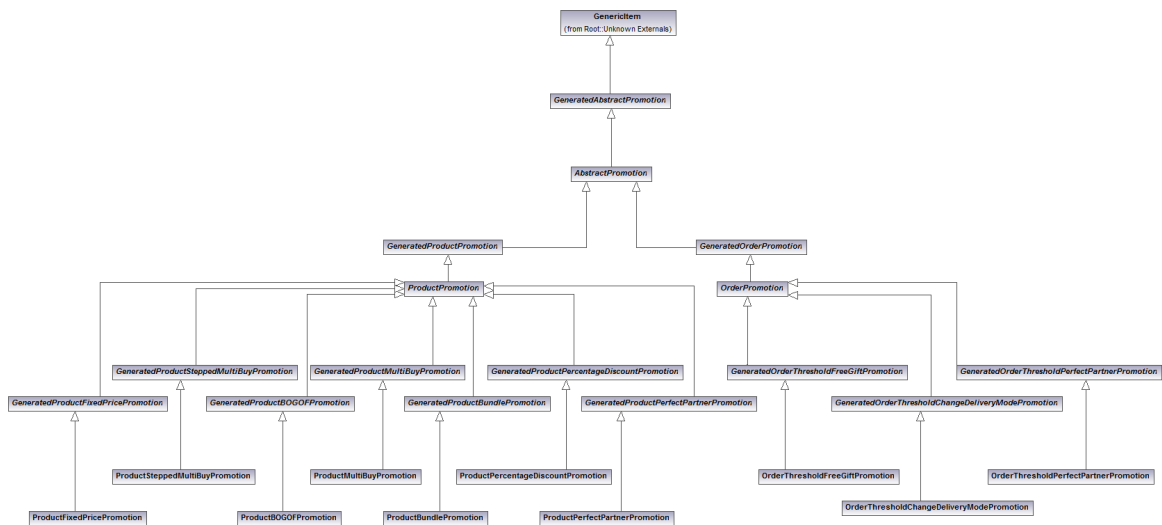


Abbildung 2.5: UML Diagramm Virgin Promotions

Vorbedingung, dass der Gesamtwert des Warenkorbs über einem gewissen Schwellwert x liegt. Diese Vorbedingung gibt es beispielsweise bei einer Promotion vom Typ *ProductBOGOFPromotion* nicht. Es könnte jedoch sein, dass bei beiden Promotions gleichermaßen nur Neukunden, also eine bestimmte Usergruppe, diese Promotion erhalten.

Zu Beginn der Entwicklung der Promotion-Extension war geplant, jede Vorbedingung oder Einschränkung als eigenes Attribut in jedem Promotion-Typen, der die jeweilige Einschränkung benötigt, abzubilden. Die beispielhaft genannte *OrderThresholdChangeDeliveryModePromotion* hätte demnach ein Double-Attribut um den Mindestbestellungs-Schwellwert zu definieren. Falls bei dieser Promotion auch die Usergruppe einschränkbar sein sollte, so müsste ein weiteres Attribut mit der erlaubten Usergruppe vorhanden sein.

Nachdem jeder Promotion-Typ eine eigene Java-Klasse hat, führt dieses Konzept schnell zu redundantem Code, da alle Vorbedingungen die prinzipiell gleich sind (z.B. Usergruppen-Einschränkung), dennoch für jede einzelne Promotion in die Java-Klasse aufgenommen werden müsste. Darüber hinaus ist das ursprünglich angedachte Konzept sehr starr und kaum erweiterbar, denn sobald eine Promotion um eine neue Vorbedingung (beispielsweise soll eine Promotion in Zukunft auch auf Usergruppen eingeschränkt werden können, wo das vorher nicht möglich war) erweitert werden würde, müsste sowohl das Typsystem erweitert werden (ein zusätzliches Attribut eingefügt) und die

2.2 Projektspezifische Implementierung der Promotions

Java-Klasse des Typs angepasst werden.

Aus diesen Gründen wurde während der Entwicklung dieses unflexible Konzept, inspiriert durch die teilweise parallel entwickelte und später in dieser Arbeit präsentierte Lösung mit Business Rules, durch eine Art Filtermechanismus aufgeweicht.

Die Idee des Filtermechanismus war, alle Vorbedingungen die von mehreren unterschiedlichen Promotions gleich genutzt werden, in einen eigenen Typ auszulagern, unter anderem auch um Redundanzen im Java-Code zu verringern. Es wurde ein hybride Typ namens *AbstractPromotionRestriction* eingeführt, von dem alle gemeinsam genutzten Vorbedingungen erben. Jede Promotion (d.h. jeder Subtyp von *AbstractPromotion*) kann über eine beliebig lange Liste von *AbstractPromotionRestrictions* verfügen, die einen Filter für die jeweilige Promotion darstellen.

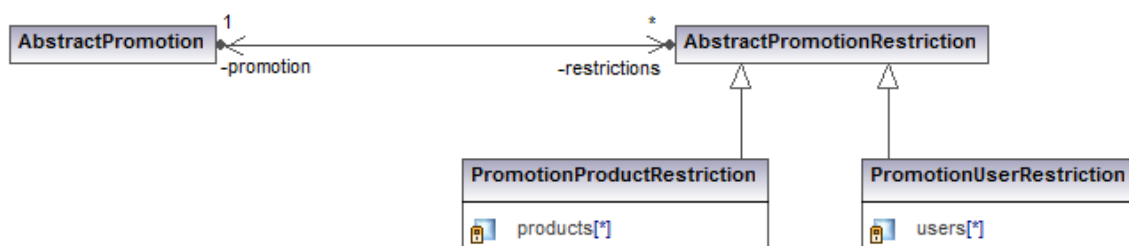


Abbildung 2.6: UML Diagramm Restrictions

Beim aktuellen Stand des Projektes gibt es bisher lediglich zwei konkrete Restrictions, nämlich den Typ *PromotionUserRestriction* und den Typ *PromotionProductRestriction*. Einer *PromotionUserRestriction* kann eine Liste von Usern (genauer: *Principal*, also auch Usergruppen) übergeben werden, die von der Promotion ausgeschlossen sind. Sobald der aktuelle User in der Liste der Restriction enthalten ist, kann die Promotion nicht mehr angewendet werden.

Eine *PromotionProductRestriction* enthält eine Liste von Produkten, die aus der Berechnung von Promotions ausgeschlossen werden. Jedes Produkt der Liste wird also, bevor die eigentliche Promotion ausgewertet wird, aus der Produktliste entfernt und steht für die Vorbedingungen der Promotion selbst nicht mehr zur Verfügung.

2 Hauptteil

Ergebnisse Als Resultat der Auswertung eines Warenkorbs bzw. einer Bestellung von einer Promotion ist eine Liste von Auswertungs-Ergebnissen vom Typ *PromotionResult*. Dabei kann die Auswertung einer Promotion prinzipiell beliebig viele (also auch 0) *PromotionResults* erzeugen.

Jedem *PromotionResult* ist eine Liste von Auswirkungen bzw. Aktionen auf den Warenkorb zugeordnet. Diese Auswirkungen vom Typ *AbstractPromotionAction* modellieren die eigentlichen Schritte die ausgeführt werden, sobald eine Promotion tatsächlich angewendet wird. Darüber hinaus enthält auch jede *AbstractPromotionAction* eine UNDO-Funktionalität, um die Änderungen am Warenkorb bzw. der Bestellung, die sie selbst eventuell durchgeführt hat, rückgängig zu machen. Dafür enthält die Klasse *AbstractPromotionAction* die abstrakten Methoden *apply* zum Anwenden und *undo* zum Rückgängigmachen der Effekte der jeweiligen Aktion.

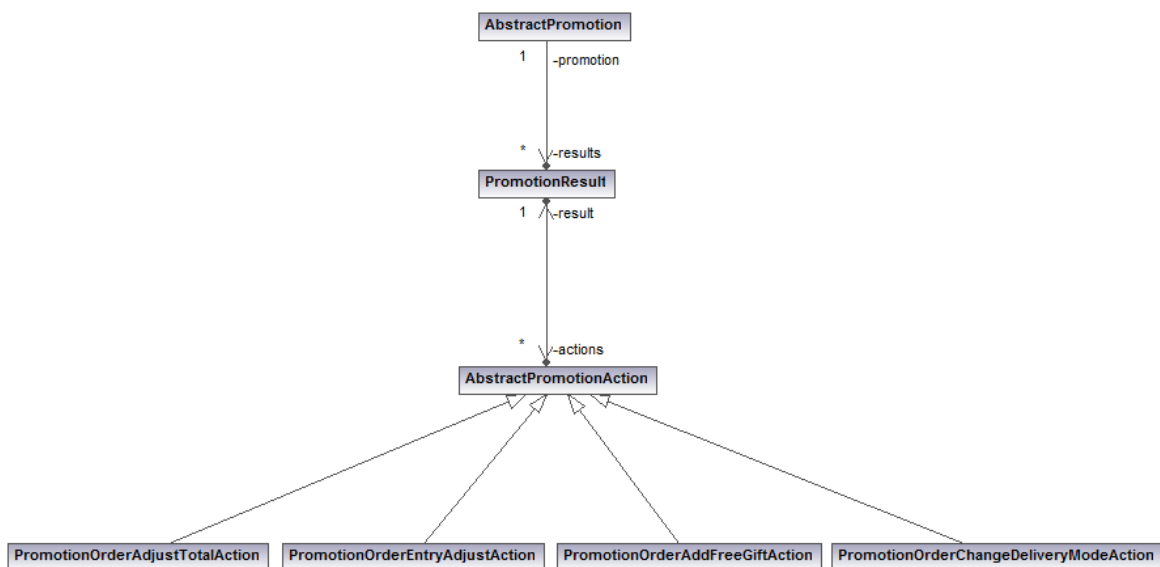


Abbildung 2.7: UML Diagramm Results und Actions

Um der Anforderung des Bewerbens von teilweise, aber noch nicht vollständig, erfüllten Promotions gerecht zu werden, hat jedes *PromotionResult* eine „Sicherheit“, ein Attribut *certainty* vom Typ *float*, was gewissermaßen eine Art Erfüllungsgrad des Auswertungs-Ergebnisses angibt. Diese *certainty* hat eine Wertebereich zwischen 0.0F und 1.0F, wobei 1.0F einen Erfüllungsgrad von 100% darstellt, das *PromotionResult* ist also sicher oder wurde bereits angewendet. Je geringer die *certainty* ist, desto größer der Abstand vom aktuellen Warenkorb zur aktuellen Promotion. Zum Anteaern von nicht

2.2 Projektspezifische Implementierung der Promotions

vollständig erfüllten Promotions kann man also die *PromotionResults* mit $certainty < 1.0F$ nach der *certainty* absteigend sortieren und jeweils nur die „nächstliegende“ Promotion bewerben.

Sowohl *PromotionResult* als auch *AbstractPromotionAction* (bzw. die davon ererbenden Subtypen) sind hybrid-Typen und damit persistent in der Datenbank gespeichert. So sind die einer Bestellung zugeordneten *PromotionResults* dauerhaft mit der Bestellung selbst verbunden und können auch später (im Falle von Reklamationen etc.) genau nachvollzogen werden. Dem *PromotionResult* selbst wiederum ist die Promotion, die das vorliegende Ergebnis „erzeugt“ hat, sowie alle *AbstractPromotionActions*, die ausgeführt wurden, zugeordnet.

Bei der Auswertung und Anwendung von produktabhängigen Promotions werden unter anderem einzelne Produkte im Warenkorb betrachtet. Dabei ist es im Normalfall so, dass jedes Produkt nur für jeweils eine Promotion verwendet werden kann, von dieser also „verbraucht“ wird. Da das verbrauchte Produkt nicht aus dem Warenkorb entfernt werden kann (es muss ja nach wie vor angezeigt und berechnet werden), müssen die von Promotions verbrauchten Produkte gesondert vermerkt werden. Dazu wurde der mit dem *PromotionResult* verknüpfte Typ *PromotionOrderEntryConsumed* eingeführt. Objekte dieses Typ verweisen auf eine Zeile des Warenkorbs bzw. der Bestellung und enthalten einen geänderten Preis oder eine geänderte Anzahl von Produkten, die die jeweilige Eigenschaft der ursprünglichen Bestellzeile überschreiben. Sofern das zugehörige *PromotionResult* bereits ausgelöst wurde ($certainty = 1.0F$) wird die jeweilige *PromotionOrderEntryConsumed* auch aktiv und „verbraucht“ die jeweiligen Produkte, andernfalls dient es lediglich als „Markierung“ für den Fall, dass die zugrunde liegende Promotion wirklich ausgelöst wird.

Konfliktbehandlung Bei steigender Anzahl von Promotions innerhalb eines Systems wird die Wahrscheinlichkeit, dass zwei Promotions gegenseitig in Konflikt geraten, immer größer. Unter einem Konflikt versteht man hierbei, dass bei mindestens zwei unterschiedlichen Promotions die Anwendung der einen Promotion eine Auswirkung auf die Anwendung der anderen Promotion hat, so dass die Reihenfolge der Anwendung entscheidend wird⁴².

Beispiel:

Promotion 1: Kaufe 3 CDs für nur 14 € (bei 6 € Einzelpreis)

Promotion 2: Beim Kauf über 15 € keine Versandkosten (statt 5 € normalerweise)

Sobald der Kunde tatsächlich 3 CDs kauft, so ist der reguläre Preis (ohne Anwendung einer Promo-

⁴²Race Condition

2 Hauptteil

tion) $3 \times 6 \text{ €} + 5 \text{ €} = 23 \text{ €}$. Bei der Anwendung der beiden Promotions kommt hingegen zur Race Condition. Wird Promotion 1 zuerst ausgeführt, so sinkt der Preis zunächst auf 14 € , und Promotion 2 wird nicht mehr ausgeführt. Der Kunde muss dann $14 \text{ €} + 5 \text{ €}$ Versandkosten = 19 € bezahlen. Wird hingegen Promotions 2 zuerst ausgeführt, sinkt der Preis zunächst auf $3 \times 6 \text{ €} = 18 \text{ €}$, danach wird Promotion 1 angewendet und der Preis sinkt auf 14 € . Möglicherweise müssten nach Anwendung von Promotion 1 alle anderen Promotions nochmal überprüft werden, was in obigem Beispiel dazu führen würde, dass die zuerst angewendete Promotion 2 wieder rückgängig gemacht würde. Darüber hinaus würde eine solche erneute Überprüfung bei einer Vielzahl von Promotions sehr aufwändig werden. Um diesen Konflikte, deren Auftreten bereits ab weniger Promotions wahrscheinlich ist, begegnen zu können, wurden mehrere Maßnahmen umgesetzt.

- Eine Promotion, die anwendbar ist, wird sofort angewendet. Es findet keine Berechnung aller möglichen Kombinationen von erreichbaren Promotions statt. Das heißt, sobald Promotion 1 überprüft wird und als anwendbar erkannt wird, wird diese sofort angewendet, auch wenn durch das Anwenden der Promotion nur ein relativ kleiner Kundenvorteil entsteht und vielleicht mehrere andere Promotions dadurch nicht mehr ausgelöst werden können. Darüber hinaus ist es nicht möglich, Promotion-Effekte auszuschlagen. Wenn beispielsweise durch eine Promotion ein Geschenkartikel kostenlos zum Warenkorb hinzugefügt wird, kann der Kunde diesen nicht mehr entfernen, egal ob er das Geschenk haben möchte oder nicht.
- Der Typ *AbstractPromotion* enthält ein Prioritäts-Attribut (*priority*), welches von dem Ersteller der jeweiligen Promotion angegeben werden muss. Bei der Überprüfung aller möglicher Promotions für einen Warenkorb oder eine Bestellung werden die Promotions immer nach ihrer Priorität absteigend bearbeitet. Dadurch (und durch die Regel, dass eine anwendbare Promotion auch sofort angewendet wird) erhalten die Promotions eine absolute Reihenfolge untereinander, und das Ergebnis der Überprüfung ist immer dasselbe. Das Anwenden von Promotions ist demnach deterministisch, sofern es keine zwei Promotions, die u.U. in Konflikt geraten könnten, mit der selben Priorität gibt.
- Eine weitere Vorgabe bei der Anwendung der Promotions ist die bevorzugte Anwendung von Product-Promotions gegenüber Order-Promotions. Das wird dadurch erreicht, dass alle Product-Promotions standardmäßig eine höhere Priorität als Order-Promotions bekommen. Das ermöglicht eine für den Menschen intuitivere Auswertung und Anwendung der Promotions. Es

2.2 Projektspezifische Implementierung der Promotions

hat sich gezeigt, dass die meisten Versuchskunden die Anwendung zuerst auf Produktebene (z.B. Buy 1 Get 1 Free) und danach auf Bestellungsebene (z.B. kostenloser Versand bei Bestellungen über x €) erwarten. Hierbei handelt es sich aber lediglich um eine Konvention und Empfehlung, durch das Prioritäts-Attribut ist es möglich - wenn auch nicht ratsam - dieses Verhalten zu ändern.

Da bei der Berechnung der Promotions zunächst alle vorherigen Promotions gelöscht und komplett neu berechnet werden, ist im Zusammenspiel mit den o.g. Maßnahmen ein deterministischer Ablauf gegeben. Das heißt, für zwei identische Warenkörbe werden dieselben Promotions gewährt, egal wie diese Warenkörbe zustande gekommen sind. Es wird aber nicht verhindert, dass Kunden für ihren jeweiligen Warenkorb unter Umständen nicht das optimale Promotion-Ergebnis erzielen. Deswegen wird empfohlen, beim Anlegen von Promotions genau auf andere, möglicherweise in Konflikt stehende Promotions, zu achten.

Manager-Klasse Bereits in den Grundlagen wurden die Manager-Klassen der Extensions vorgestellt. Auch die *Promotion*-Extension verfügt über eine solche Klasse, den *PromotionManager*. Diese Klasse stellt eine übersichtliche API für die Verwendung der Promotions dar. Von anderen Extensions aus - beispielsweise aus der Frontend-Extension *Storefoundation* - sollte nur der *PromotionManager* verwendet werden, um die gesamte Promotion-Funktionalität zu nutzen.

Einige vom *PromotionManager* bereitgestellte Methoden:

getProductPromotions Die *getProductPromotions*-Methode dient zum Bewerben der Promotion auf Produkt(übersichts)seiten. Mit einem gegebenen Produkt kann eine nach Priorität geordnete Liste von relevanten Promotions ausgegeben werden, die dann entsprechend beworben werden können.

updatePromotions Diese Methode berechnet alle Promotions für einen angegebenen Warenkorb neu. Dabei werden alle Product-Level-Promotions automatisch angewendet. Sollten bereits früher Promotions auf diesen Warenkorb angewendet worden sein, so wird überprüft ob diese nach wie vor anwendbar sind auf den aktuellen Warenkorb und werden, sofern sie nicht mehr valide sind, gelöscht. Die *updatePromotions*-Methode wird aus der *Storefoundation*-Extension jedes mal aufgerufen, sobald Warenkorb-Interaktion (also Hinzufügen von Produkten, Entfernen von Produkten oder verändern der Anzahl von Produkten) stattgefunden hat.

2 Hauptteil

getPromotionResults Mit dieser Methode können alle *PromotionResult*-Objekte für einen gegebenen Warenkorb oder eine gegebene Bestellung angezeigt werden. Damit erhält man die Promotions, die auf das angegebene *AbstractOrder*-Objekt bereits angewendet wurden.

transferPromotionsToOrder Sobald die Bestellung zum Abschluss kommt, wird der Warenkorb(*Cart*) in eine Bestellung(*Order*) umgewandelt. Dabei wird aus der *Storefoundation*-Extension die *transferPromotionsToOrder*-Methode aufgerufen, die dafür sorgt, dass alle Promotions, die zuvor auf den Warenkorb angewendet waren, auch auf die daraus entstehende Bestellung angewendet werden.

Diese vier Methoden sind praktisch gesehen die einzigen Methoden des *PromotionManagers*, die von außerhalb (in der Regel der *Storefoundation*-Extension) aufgerufen werden. Natürlich bietet der *PromotionManager* noch wesentlich mehr als die hier vorgestellten Methoden, die hier aber nicht alle präsentiert werden können, und für das Verständnis der Promotion-Lösung von Virgin Megastores nicht notwendig sind.

Auswertung

Startpunkt der Auswertung von Promotions ist im Normalfall immer die *Storefoundation*-Extension. Sobald der Kunde eine Warenkorb-Interaktion durchführt (Produkte in den Warenkorb legen, Anzahl verändern, Produkte aus dem Warenkorb löschen), wird innerhalb der für den Warenkorb zuständigen JSF-Bean (*CartJSFBean*) die Methode *invalidateModel()* aufgerufen. Diese sorgt dafür, dass der Warenkorb neu berechnet wird. Von dort aus wird der *PromotionManager* aufgerufen, der die Verwaltung der Promotions übernimmt. Zur Berechnung und Anwendung der Promotions ist nur dieser eine Methodenaufruf nötig, wodurch die Promotion-Funktionalität sauber von der Frontend-Extension getrennt ist.

Die weitere Auswertung der Promotions wird danach vom *PromotionManager* gesteuert. Falls der Warenkorb nicht bereits vorher schon (ohne Einbeziehung der Promotions) berechnet wurde, so wird dies nachgeholt, damit alle späteren Berechnungen auf einem aktuellen Stand durchgeführt werden. Wie bereits erwähnt, soll es (z.B. für Call-Center-Mitarbeiter) möglich sein, eine Art „was wäre wenn“-Berechnung für jedes beliebige Datum durchzuführen. Daher kann bei der Berechnung der

2.2 Projektspezifische Implementierung der Promotions

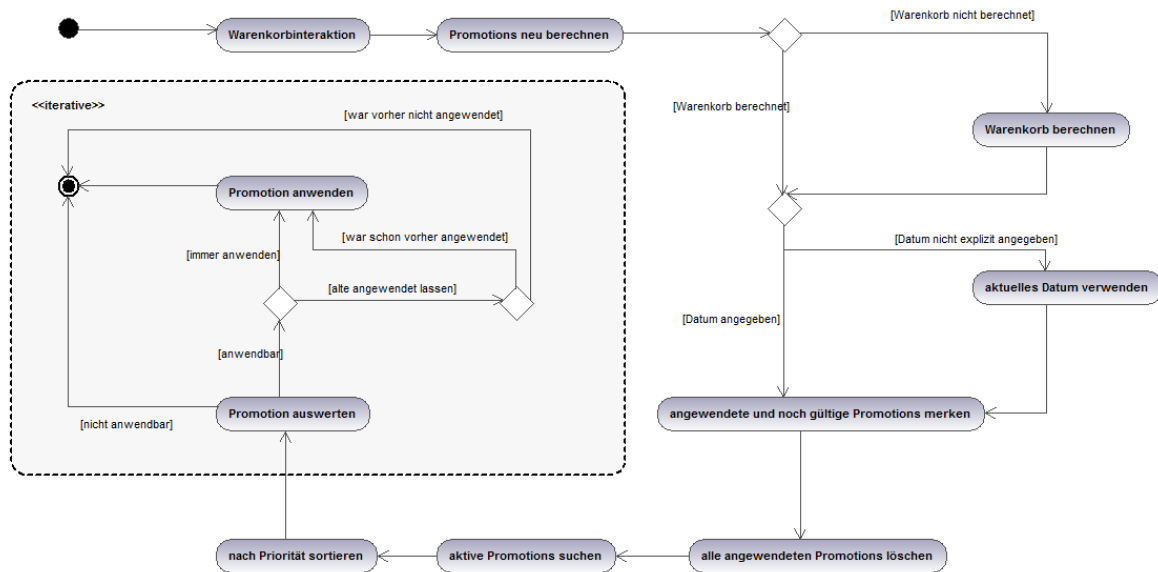


Abbildung 2.8: Aktivitätsdiagramm Auswertung von Promotions

Promotions ein Datum mit angegeben werden. Wenn kein Datum angegeben wird - was der Regelfall ist - wird immer das aktuelle Datum verwendet.

Danach beginnt die eigentlich Arbeit der Promotion-Extension. In einem ersten Schritt werden alle in früheren Berechnungsdurchläufen auf den Warenkorb angewendeten Promotions ermittelt. Das ist möglich, da alle Ergebnisse dieser Berechnungen vom Typ *PromotionResult* in der Datenbank gespeichert sind. Diese Ergebnisse werden auf dem aktualisieren Warenkorb nochmals validiert, da aufgrund der vorangegangenen Warenkorbinteraktion die Möglichkeit besteht, dass den zuvor angewendeten Promotions die Grundlage entzogen wurde, und diese nicht mehr gültig sind. Von allen immer noch gültigen angewendeten Promotions wird ein eindeutiger Schlüssel⁴³ erzeugt. Der Schlüssel enthält unter anderem die Klasse und den Code der jeweiligen Promotion, bei *ProductPromotions* auch die qualifizierenden Produkte. Dieser Schlüssel ist für zwei nicht absolute identische *PromotionResults* niemals gleich. Diese Schlüssel werden in einer Liste gespeichert.

Nun wird ein kompletter Reset aller Promotions des Warenkorbs durchgeführt. Das bedeutet, dass alle *PromotionResults* aus vorherigen Berechnungsdurchläufen rückgängig gemacht und aus der Datenbank gelöscht werden. Danach ist der Warenkorb in seinem Ursprungszustand, genau so als wäre

⁴³unique data key

2 Hauptteil

die Promotion-Extension nicht installiert. Hintergrund dieser Aktion ist, die Berechnung immer von vorne zu beginnen, damit für Warenkörbe mit demselben Inhalt immer dasselbe Ergebnis zustande kommt. Würde man die vorhandenen *PromotionResults* beibehalten, so würde auch die Reihenfolge, in der die Produkte in den Warenkorb gelegt wurden, entscheiden, welche Promotions angewendet werden, denn bereits angewendete Promotions könnten dadurch mit höherer Priorität versehene, aber erst zu einem späteren Zeitpunkt verfügbar gewordene Promotions, blockieren.

Im darauf folgenden Schritt werden alle Promotions, die für die Anwendung auf den Warenkorb möglicherweise in Frage kommen, ermittelt. Dazu wird eine Anfrage auf Datenbank-Ebene durchgeführt, die sowohl Order- als auch Product-Level-Promotions anfragt. Einzige Einschränkungen für Order-Level-Promotions an dieser Stelle sind die Abfrage ob die Promotion überhaupt noch aktiv (enabled) ist, und ob das angefragte Datum innerhalb des Gültigkeitszeitraums für die Promotion liegt.

Bei *ProductPromotions* wird zusätzlich abgefragt, ob eines der Produkte, die Bestandteil der *ProductPromotion* sind, bereits im Warenkorb liegt. Ist das nicht der Fall, so kann die *ProductPromotion* keinesfalls ausgelöst werden, und kann in diesem ersten Schritt bereits ausgefiltert werden. Da bei *ProductPromotions* u.U. auch ganze Kategorien statt einzelnen Produkten angegeben werden können (bspw. „ein Produkt aus der Kategorie xyz“), können alle Produkte, die in dieser Kategorie enthalten sind, die Promotion auslösen und müssen deshalb überprüft werden. Das entspricht auf Datenbank-Ebene aber lediglich einer einfachen Join-Operation.

Das Ergebnis dieser Datenbank-Anfrage sind demnach alle Promotions, die unter Umständen auf einen bestimmten Warenkorb angewendet werden könnten. Auf Order-Promotion-Ebene müssen immer alle zum angefragten Zeitpunkt aktiven Promotions genauer überprüft werden, auf *ProductPromotion*-Ebene können zusätzlich noch alle außen vor gelassen werden, deren Produkt-Bedingungen nicht mit dem Warenkorb zusammenpassen können. Dadurch wird die Anzahl der Promotions vor einer aufwändigeren Evaluierung bereits frühzeitig begrenzt.

Der logisch nächste Schritt ist die Sortierung der ermittelten Promotions anhand ihrer Priorität. Sofern später die Promotions in der sortierten Reihenfolge evaluiert und angewendet werden, ist so sichergestellt, dass höher priorisierte Promotions auch wirklich vor anderen angewendet werden. Aus technischer Sicht erfolgt die Sortierung bereits im Rahmen der vorherigen Datenbank-Abfrage mittels einer „order by“-Operation, aus logischer Sicht entspricht sie aber eher einem separaten Schritt, denn prinzipiell sind auch andere Sortierung- bzw. Priorisierungsmechanismen, als eine einfache „order

2.2 Projektspezifische Implementierung der Promotions

by“-Operation über das Prioritäts-Attribut, denkbar. Die letzte ausstehende und wichtigste Aufgabe ist die Evaluierung, und gegebenenfalls Anwendung, der bisher nur grob gefilterten Promotions.

Evaluierung einer Promotion Die Evaluierung einer Promotion ist ganz individuell und von Promotion zu Promotion unterschiedlich. Der Supertyp *AbstractPromotion* stellt dafür eine abstrakte *evaluate*-Methode zur Verfügung, die von der konkreten Implementierung jeder einzelnen Promotion überschrieben wird bzw. werden muss. Bei der Evaluierung einer Promotion wird eine Liste von *PromotionResults* erzeugt, mit jeweils unterschiedlichem Erfüllungsgrad (*certainty*). Sobald die *certainty* den Wert 1.0F hat, gilt das Ergebnis als erfüllt und der Warenkorb kann entsprechend den Vorgaben der Promotion angepasst werden. Im Folgenden werden einige interessante Promotions bezüglich ihrer Evaluierungslogik vorgestellt:

OrderThresholdFreeGiftPromotion Dieser Order-Promotion-Typ ermöglicht es, dem Kunden bei Überschreiten eines gewissen Bestellwerts (*threshold*), ein Produkt zu schenken. Bei der Auswertung dieser (sehr einfachen) Promotion ist es lediglich notwendig zu überprüfen, ob der „geforderte“ Bestellwert bereits überschritten ist. Berechnungsgrundlage ist hierbei der gesamte Inhalt des Warenkorbs, abzüglich aller bereits angewendeter Rabatte. Das heißt, wenn (z.B. durch einen Gutschein oder andere Rabattsysteme, die nichts mit den Promotions selbst zu tun haben) die Bestellwert durch Rabatte unter den Schwellwert gefallen ist, ohne diese Rabatte aber über dem Schwellwert liegen würde, wird diese Promotion dennoch nicht ausgeführt. In diesem Fall gibt die Evaluierung ein *PromotionResult* ohne *PromotionAction* zurück mit einem Erfüllungsgrad < 1.0F:

$$\text{certainty} = \text{Bestellwert} / \text{Schwellwert}$$

Das heißt, wenn der Bestellwert aktuell bei 50 € liegt, der Schwellwert der Promotion jedoch bei 100 €, so erhält das zurückgegebene *PromotionResult* einen Erfüllungsgrad von 0.5F.

Ist der Schwellwert hingegen überschritten, das heißt die Promotion kann „feuern“, so wird ein *PromotionResult* mit Erfüllungsgrad 1.0F (also ausgelöst) zurückgegeben, verknüpft mit einer *PromotionOrderAddFreeGiftAction*, die im Anwendungsschritt dafür sorgt, dass ein Geschenk zum Warenkorb hinzugefügt wird.

Von der Evaluierungslogik gesehen, gehört diese Promotion zu den „einfachsten“ Promotions. Es wird lediglich ein einzelner Wert überprüft werden. Darüber hinaus wird die von den Business Rules inspirierte zusätzliche Filterlogik (*AbstractPromotionFilter*) nicht verwendet.

ProductBOGOFPromotion Für diesen Typ einer Product-Promotion hingegen ist die Auswertung etwas komplizierter und findet zweistufig statt. Für den ersten Schritt ist im wesentlichen der so genannte *PromotionEvaluationContext* verantwortlich. Dieser Kontext stellt gewissermaßen einen virtuellen Container dar, in dem die Berechnung der Promotions stattfindet. Hier können alle Zwischenergebnisse, insbesondere die bereits „verbrauchten“ Produkte des Warenkorbs, gespeichert werden. Zunächst werden, mit Hilfe dieses Context, die für die jeweilige Promotion „interessanten“, d.h. qualifizierende, Produkte aus dem Warenkorb berechnet. Dazu wird die Schnittmenge der qualifizierenden Produkte mit den Produkten des Warenkorbs berechnet:

relevante Produkte = qualifizierende Produkte \cap im Warenkorb enthaltene Produkte

Diese Berechnung findet mit Hilfe der hybris Flexible Search auf Datenbank-Ebene statt. Bei der Evaluierung innerhalb des *PromotionEvaluationContext* werden zusätzlich noch die *AbstractPromotionFilter* evaluiert, d.h. die abstrakte *evaluate*-Methode wird auf den mit der Promotion verknüpften Subtypen von *AbstractPromotionFilter* aufgerufen.

Aus dieser Schnittmenge werden die bereits von anderen Promotions verbrauchten Produkte abgezogen, so dass am Ende eine Sicht (view) der Promotions auf den Warenkorb entsteht, in der nur für Promotions noch verfügbare und für die jeweilige Promotion qualifizierende Produkte enthalten sind.

Nach diesem ersten Vorbereitungsschritt findet dann die eigentliche Evaluierung der Promotion selbst statt. Diese besteht im Prinzip aus einer Iteration über die in der Produkt-Sicht enthaltenen Produkte. Bei diesem Typ von Promotion ist eine Anzahl qualifizierender Produkte und eine Anzahl kostenloser Produkte definierbar. So sind mit demselben Typ Promotions wie „buy 1 get 1 free“, „buy 3 get 2 free“ oder auch „buy 2 get 5 free“ abbildbar. Die Iteration läuft demnach so lange, wie die Anzahl der in der Sicht verfügbaren Produkte größer als die Anzahl der qualifizierenden Produkte (bei „buy 3 get 1 free“ demnach 3) ist. Innerhalb der Sicht werden die Produkte nach Preisen aufsteigend sortiert. Sobald die Promotion qualifiziert wird, werden die qualifizierenden Produkte vom unteren Ende (also teuerste zuerst) verbraucht und aus der Sicht entfernt. Dann wird die definierte Anzahl von kostenlosen Produkten vom oberen Ende (also billigste zuerst) der Sicht entfernt und dementsprechend viele *PromotionOrderAdjustTotalAction* mit dem *PromotionResult* verknüpft.

Falls im letzten Iterationsschritt noch qualifizierende Produkte übrig sind, die Anzahl aber zur vollständigen Qualifizierung der Promotion nicht mehr ausreicht, so wird ein *PromotionResult*

2.2 Projektspezifische Implementierung der Promotions

mit einem Erfüllungsgrad $< 1.0F$, genauer

Erfüllungsgrad = verbleibende qualifizierende Produkte / benötigte qualifizierende Produkte erzeugt. Bei dieser Art von Promotion können immer (d.h. nicht konfigurierbar) nur die billigsten Produkte geschenkt werden, während die teuersten immer bezahlt werden müssen.

ProductSteppedMultiBuyPromotion Die *ProductSteppedMultiBuyPromotion* dient dazu Staffelpromotionen abzubilden. Alle Promotions der Art „buy 2 for 20€, 4 for 35€, 8 for 60€“ etc. können hiermit dargestellt werden. Ähnlich wie die vorangegangene Promotion, wird auch die *ProductSteppedMultiBuyPromotion* zweistufig ausgewertet. Im ersten Schritt werden analog zur *ProductBOGOFPromotion* zunächst die Bedingungen der *AbstractPromotionFilter* ausgewertet. Danach wird ebenfalls eine Sicht auf den Warenkorb erzeugt, in der nur die für die Promotion qualifizierenden Produkte, die noch nicht von anderen Promotions verbraucht wurden, enthalten sind.

Im zweiten Schritt wird dann die eigentliche Promotion selbst evaluiert. Zu diesem Zweck verfügt die *ProductSteppedMultiBuyPromotion* über eine Liste von *PromotionQuantityAndPricesRows*, mit denen die Staffelung der Preise dargestellt werden können. Eine *PromotionQuantityAndPricesRow* stellt eine Preisstufe der Staffelung dar, und besteht aus einer Anzahl qualifizierender Produkte und einer Liste von Preisen (für jede angebotene Währung höchstens einen). Bei der Evaluierung der Promotion wird diese Liste von *PromotionQuantityAndPricesRows* nach der Anzahl qualifizierender Produkte absteigend sortiert und der Reihe nach betrachtet, so dass immer die größtmögliche Stufe der Preisstaffelung überprüft wird. Ist die Anzahl der in der Sicht auf den Warenkorb enthaltenen Produkte kleiner als die aktuell untersuchte Staffelung, so wird die nächstkleinere Staffelung überprüft. Das geschieht so lange, bis eine zutreffende Stufe der Staffelung erreicht wird. Für diese Staffelung wird dann ein *PromotionResult* mit Erfüllungsgrad $1.0F$ erzeugt, wobei die qualifizierenden Produkte aus der Sicht entfernt werden. Mit dem *PromotionResult* wird eine *PromotionOrderAdjustTotalAction* verknüpft, die bei der Anwendung des *PromotionResults* den Bestellwert genau um die Differenz zwischen dem Standardpreis der Produkte und dem Staffelpreis senkt. Danach wird die Überprüfung der Staffelung auf derselben Staffelungs-Stufe fortgesetzt, so dass dieselbe oder niedrigere Stufen des Staffelpromotions auf denselben Warenkorb eventuell mehrfach angewendet werden können. Falls am Ende der Auswertung noch qualifizierende Produkte in der Sicht auf den Warenkorb übrig sind, die jedoch nicht ausreichen um die niedrigste Staffelpromotionsstufe vollständig zu erfüllen, so

2 Hauptteil

wird ein *PromotionResult* mit Erfüllungsgrad $<1.0F$ erzeugt, so dass zumindest die mögliche Erfüllung dieser Promotion beworben werden kann („wenn Sie noch zwei Packungen mehr kaufen, erhalten Sie 5 € Rabatt“).

Für Promotions, deren Evaluierung keine *PromotionResults* bzw. nur *PromotionResults* mit Erfüllungsgrad $<1.0F$ ergeben hat, ist die Auswertung an dieser Stelle beendet.

Anwendung von PromotionResults Ähnlich zur Evaluierung ist auch die Anwendung einer Promotion von Promotion zu Promotion verschieden. Dazu ist jedem *PromotionResult* eine Reihe von *AbstractPromotionActions* zugeordnet. Wie bereits erwähnt können diese *Actions* ausgelöst (und der Warenkorb damit manipuliert) werden, sobald das Result erfüllt ist. Die Klasse *AbstractPromotionAction* enthält eine abstrakte *apply*-Methode, die von der jeweiligen Action (z.B. *PromotionOrderAdjustTotalAction*) überschrieben wird. Sobald ein *PromotionResult* auf den Warenkorb angewendet wird, wird diese *apply*-Methode auf jeder dem *PromotionResult* zugeordneten *PromotionAction* aufgerufen.

Hier einige Beispiele von interessanten *Actions*, die auf den Warenkorb angewendet werden können:

PromotionOrderAdjustTotalAction Diese Action ist dafür zuständig, den Gesamtbestellwert eines Warenkorbs bzw. einer Bestellung um einen angegebenen Betrag zu senken. Dazu verfügt der *hybris*-Typ lediglich über ein *Double*-Attribut, was den Betrag festlegt. Sobald die *apply*-Methode dieser Action aufgerufen wird, wird der Bestellung ein Rabatt in Höhe des festgelegten Betrags der Action hinzugefügt. Dieser Rabatt (*Discount*) erhält einen speziellen Code, so dass er später eindeutig als von einer *PromotionOrderAdjustTotalAction* angewendet identifiziert werden kann.

```
final DiscountValue dv = new DiscountValue(code, this.getAmount(ctx) * -1, true, order.getCurrency(ctx).getIsoCode(ctx));
order.addGlobalDiscountValue(ctx, dv);
```

Dadurch kann die verwendete *PriceFactory* bei späterer Berechnung des Warenkorbs ohne Kenntnis der Promotion selbst den Preis korrekt ermitteln. Bei Neuberechnung der Promotions (nach Warenkorbinteraktion) werden zunächst alle angewendeten Action rückgängig gemacht. Dazu wird die *undo*-Methode aufgerufen. Innerhalb dieser Methode wird der zuvor auf den

2.2 Projektspezifische Implementierung der Promotions

Warenkorb angewendete Rabatt entfernt. Dazu wird über die Liste aller angewendeter Rabatte iteriert und der Rabatt, der aufgrund seines Codes zu dieser *PromotionOrderAdjustTotalAction* gehört, wird entfernt.

PromotionOrderAddFreeGiftAction Durch die *PromotionOrderAddFreeGiftAction* wird dem Warenkorb ein kostenloses Produkte als Geschenk hinzugefügt. Dazu erhält die Action bei ihrer Erzeugung ein Produkt, welches dann bei Anwendung der Action (mittels der *apply*-Methode) dem Kunden geschenkt wird. Sobald die *apply*-Methode aufgerufen wird, wird der Bestellung eine neue Bestellzeile, bestehend aus diesem Produkt mit Anzahl 1 hinzugefügt. Diese Bestellzeile wird als Geschenkzeile markiert (give-away). Diese Geschenkfunktionalität ist nicht Bestandteil der Promotion-Extension, sondern ist Teil der Standard-Preisberechnungsfunktionalität von hybris. Sobald die Bestellzeile derart markiert wurde, wird der Preis dieser Zeile automatisch auf 0 gesetzt.

Danach wird ein Objekt vom Typ *PromotionOrderEntryConsumed* angelegt und mit dem verantwortlichen *PromotionResult* verknüpft. Mit dieser *PromotionOrderEntryConsumed*-Objekt wird gespeichert, dass die neu angelegte Bestellzeile Bestandteil einer Promotion ist, und nicht zur Berechnung weiterer Promotions zu Verfügung steht. Das ist notwendig, da sonst das geschenkte Produkt selber Grundlage für neue Promotions werden könnte, die dem Kunden zugute kommen, und im schlimmsten Fall könnte auf diese Weise sogar eine Endlosschleife - bei der jedes Geschenk selbst wieder eine Geschenkpromotion auslöst - entstehen.

Beim UNDO dieser *PromotionAction* wird im Prinzip die neu angelegte Bestellzeile samt zugehörigem *PromotionOrderEntryConsumed*-Objekt wieder gelöscht. Falls die Anzahl der Produkte der Bestellzeile jedoch größer als 1 ist, so muss hier eine gesonderte Behandlung erfolgen. Dies ist der Fall, wenn der Kunde im Warenkorb die Anzahl der geschenkten Produkte erhöht hat. Dies kann einerseits durch den Versuch, sich mehr als ein Geschenk zu sichern, andererseits auch durch die Absicht, ein Produkt zusätzlich zu kaufen, entstehen. In diesem Fall werden aus der Bestellzeile nur die geschenkte Anzahl von Produkten entfernt, und die Bestellzeile danach als normale Bestellzeile (also kein give-away mehr) markiert. So werden die verbleibenden Produkte als normale Bestellung weitergeführt und auch mit dem richtigen Preis kalkuliert.

PromotionOrderEntryAdjustAction Die *PromotionOrderEntryAdjustAction* dient dazu, den Preis einer Bestellzeile anzupassen. Verwendet wird diese Action unter anderem bei der *ProductFixedPricePromotion*, bei der z.B. einzelne Produkte zu einem fest definierten Preis an-

2 Hauptteil

geboten werden. In der Administrationskonsole wird beim Anlegen der Promotion neben den qualifizierenden Produkten eine Liste von Preisen (ein Preis für jede Währung) angegeben, was dann den neuen Verkaufswert der qualifizierenden Produkte darstellt. Dabei wird aus Gründen der Kapselung von Funktionalitäten bei der Berechnung des Warenkorbs nicht einfach der neu definierte Preis verwendet, sondern es werden auf den alten Preis absolute Rabatte angewendet. Diese Rabatte können in der hybris Datenstruktur nicht feingranularer als auf Bestellzeilen-Ebene angewendet werden. Prinzipiell wäre es unproblematisch, diese Rabatte dann einfach für entsprechende Bestellzeilen anzuwenden, weil in einer Bestellzeile jeweils nur ein Produkt mit einer bestimmten Anzahl enthalten ist. Problematisch daran ist, dass von einer Bestellzeile mit 10 Produkten bereits 5 von anderen Promotions „verbraucht“ sein können, so dass nur noch 5 der 10 Produkte rabattiert werden dürfen. Deshalb muss zunächst ermittelt werden, welche Bestellzeilen aus dem Warenkorb für die Promotion in Frage kommen und wie viele Produkte der Bestellzeile noch nicht „verbraucht“ sind.

Beispiel: Qualifizierende Produkte sollen zum Festpreis von 5 € angeboten werden. In einer Bestellzeile sind 10 qualifizierende Produkte zum Einzelpreis von 10 € enthalten. 5 der 10 Produkten sind bereits von anderen Promotions verwendet worden. Der anzuwendende Rabatt liegt demnach bei $(10 - 5) \times (10 € - 5 €) = 25 €$. Der Gesamtwert der Bestellzeile liegt also bei $(10 \times 10 €) - 25 € = 75 €$.

Das UNDO dieser Action ist vergleichsweise einfach, da hier ähnlich wie bei der *PromotionOrderAdjustTotalAction* lediglich der (aufgrund seines Codes) eindeutig identifizierbare Rabatt wieder entfernt werden muss, ohne dass zusätzliche Berechnungen angestellt werden müssen.

Wie bereits erwähnt bieten die Subtypen von *AbstractPromotionAction* mittels der *apply*-Methode die Möglichkeit den Warenkorb zu verändern und die Promotions tatsächlich anzuwenden. Ob diese Action bei der Auswertung der Promotion dann auch wirklich angewendet werden oder nicht, hängt von den Parametern ab, mit denen die Auswertung im *PromotionManager* aufgerufen wird. Dort kann man die Anwendungseinstellungen separat für Product-Level-Promotions und Order-Level-Promotions einstellen. Als Parameter wird jeweils ein *EnumerationValue* übergeben:

APPLY NONE Keine *PromotionAction* wird angewendet, egal welche Ergebnisse die Auswertung liefert. Tatsächlich entspricht ein Aufruf der Auswertung der Promotion mit der *APPLY NONE*-Einstellung für Product- und Order-Level-Promotions effektiv einem Entfernen aller auf den Warenkorb bzw. die Bestellung angewendeter Promotions. Im Lauf der Auswertung wer-

2.2 Projektspezifische Implementierung der Promotions

den, wie bereits erwähnt, zunächst alle vorhandenen Promotions gelöscht. Nachdem keine neue Promotion angewendet werden, bleibt der Warenkorb bzw. die Bestellung ohne jede Promotion.

APPLY ALL Alle Actions von qualifizierten *PromotionResults* (mit Erfüllungsgrad = 1.0F) werden angewendet. Diese Einstellung ist die Standardeinstellung für das Shop-Frontend, da man davon ausgehen kann, dass alle verfügbaren Promotions auch wirklich angewendet werden sollen.

KEEP APPLIED Bei der Auswertung der Promotions werden wie bereits erwähnt zunächst alle bereits angewendeten Promotions mit Hilfe eines eindeutigen Schlüssels zwischengespeichert. Danach werden alle Promotions gelöscht und eine neue Berechnung durchgeführt. Bei der Einstellung *KEEP APPLIED* werden dann im Anwendungsschritt nur diejenigen Promotions (bzw. die zugehörigen Actions) angewendet, die qualifiziert wurden und zusätzlich bereits vorher angewendet waren. Es reicht also nicht aus, dass die Vorbedingungen für eine Promotion alle erfüllt sind, sondern auch, dass die Promotion vor der Berechnung bereits angewendet war.

Sobald die Anwendung aller erfüllten *PromotionResults* bzw. deren Actions abgeschlossen ist, ist die Berechnung der Promotions abgeschlossen. Der Warenkorb befindet sich nun in einem komplett berechneten Zustand und alle Promotions wurden vollständig angewendet, sofern gewünscht.

2.2.3 Fazit

Erfüllung der Anforderungen

Dadurch, dass jede Promotion durch einen eigenen hybris-Typen mit eigener Java-Klasse abgebildet wurde, können problemlos alle Anforderungen für Promotions umgesetzt werden, sofern sie sich mit der hybris-API realisieren lassen.

Die Nutzerfreundlichkeit ist gegeben, da für jeden existierenden Typen die Darstellung in der Administrationskonsole *hmc* separat definiert werden kann. So kann speziell auf besondere Gegebenheiten einzelne Promotions eingegangen werden, ohne Kompromisse bei gemeinsam genutzten Komponenten eingehen zu müssen. Darüber hinaus sind zum Zeitpunkt dieser Arbeit bereits so genannte Wizards angedacht, die den Sachbearbeitern bei der Erstellung von Promotions unter die Arme greifen sollen. Diese Wizards stellen modale Dialoge dar, mit deren Hilfe in einem mehrstufigen Prozess Promotions erzeugt werden können.

Einige Anforderungen von Virgin Megastores sind in der aktuellen Implementierung noch nicht ent-

2 Hauptteil

halten. Aufgrund von zeitlichen Einschränkungen wurde beschlossen, Funktionalitäten wie bspw. die Call-Center-Fähigkeit oder zeitversetzte Promotions noch nicht zu implementieren.

Vorbedingungen

Die Vorbedingungen der Promotions sind zweistufig. Zunächst werden (zumindest bei den meisten Promotions) die Filter⁴⁴ ausgewertet, danach die für die jeweilige Promotion fest codierten Vorbedingungen. Das Konzept der fest codierten, für jede Promotion speziell implementierten Vorbedingungen ist dabei sehr unflexibel und starr. Darüber hinaus führt es teilweise zu redundantem Code, da mehrere Promotions prinzipiell gleichartige Vorbedingungen haben, der Code dafür aber nicht wiederverwendbar ausgelagert wurde.

Durch die zweistufige Vorbedingungslogik entsteht eine inkonsequente, teilweise auch unintuitive Bedienung. Auch für den Ersteller der Promotions (also Sachbearbeiter) ist es nicht unbedingt eingängig, weshalb manche Vorbedingungen in einfachen Textfeldern gepflegt werden, andere hingegen separat anzulegende Objekte sind. Hier wäre eine einheitliche Lösung sinnvoller gewesen, wobei die flexiblere Filterlogik geeigneter erscheint.

Konfliktbehandlung

Zur Konfliktbehandlung müssen bei der Promotion-Extension mehrere Kompromisse eingegangen werden, die teils gravierende Nachteile mit sich bringen. So wurde mit der Priorisierung der Promotions eine weitreichende Quelle für Fehler und Missverständnisse geschaffen. Sehr schnell kann es passieren, dass bei der Vergabe der Prioritäten gewisse Überschneidungen entstehen. Darüber hinaus kommt es zu Schwierigkeiten, wenn zwischen zwei Promotions mit Priorität 1 und 2 eine dritte eingefügt werden soll. Erhält die neue Promotion dann die Priorität 2, so muss unter Umständen bei allen folgenden Promotions die Priorität angepasst werden.

Verschärft wird das Problem noch dadurch, dass im späteren Einsatz höchstwahrscheinlich mehr als eine Person die Promotions pflegen wird. Das bedeutet insbesondere bei der Vergabe von Prioritäten, dass es hier zu Missverständnissen und Fehlkonfigurationen kommen kann, da nicht jeder Bearbeiter über alle Promotions und deren Prioritäten Bescheid wissen kann.

Vermieden werden könnten diese Konflikte jedoch nur durch die Bedingung, dass jedes Produkt nur in

⁴⁴AbstractPromotionFilter

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

höchstens einer Promotion gleichzeitig vorkommen darf und zusätzlich die Product-Level-Promotions immer vor den Order-Level-Promotions (oder umgekehrt) ausgewertet würden. Das entspräche allerdings auch wieder einer impliziten Form der Prioritätsvergabe. Darüber hinaus käme es, nachdem Promotions nicht nur auf einzelne Produkte sondern auch auf Produkt-Kategorien angewendet werden können, sehr schnell zu Konflikten bei der Erstellung der Promotions. Angenommen Mitarbeiter A definiert eine Promotion für alle CDs der gesamten Sortiments, und Mitarbeiter B (der evtl. in einer anderen Abteilung arbeitet und von der bereits definierten Promotion nichts weiß) versucht, eine Promotion für alle Beatles-CDs zu definieren. Hier würde zum Erstellungszeitpunkt ein Konflikt auftreten, der erst aufgelöst werden müsste („zu was steht die Promotion in Konflikt?“, „wer ist dafür verantwortlich?“, „wie kann der Konflikt beseitigt werden?“ etc). In Hinblick auf diese Situation wurde von Seiten Virgin Megastores entschieden, eine explizite Priorisierung von Promotions durchzuführen und Konflikte bei der Pflege bzw. beim Anlegen in Kauf zu nehmen.

Flexibilität

Durch das Konzept von einem eigenen Typen für jede Promotion muss für jede neu zu definierende Promotion ein neuer hybrider Typ mit eigener Java-Klasse angelegt werden. Das bedeutet, dass keine neuartige Promotion zur Laufzeit hinzugefügt werden kann. Lediglich durch die Verwendung der von Business Rules inspirierten Filterlogik kann eine gewisse Steigerung der Flexibilität erreicht werden. Darüber hinaus muss für jede neuartige Promotion auch neuer Java-Code geschrieben werden, so dass nur erfahrene Programmierer und keine Sachbearbeiter in der Lage sind, die Promotion-Funktionalität zu erweitern.

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

2.3.1 Motivation

Obwohl die projektspezifische Implementierung der Promotion-Funktionalität den Anforderungen von Virgin Megastores gerecht wird (und auch speziell auf diese Anforderungen entwickelt wurde), so gibt es doch Punkte die verbesserungswürdig sind. Dazu gehört vor allem das Konzept der 1:1 auf hybriden Typen abgebildeten Promotions. Dieses Konzept ist ohne Programmierkenntnisse überhaupt nicht und auch für erfahrene Programmierer nur aufwändig um neue Promotions zu erweitern. Darüber

2 Hauptteil

hinaus sind die vorhandenen Promotions klar definiert und können, mit der an die Business Rules erinnernden Filterlogik, nur minimal verändert werden.

In dieser Arbeit wird versucht, ein wesentlich flexibleres Konzept zur Anwendung und Pflege dieser Promotions zu entwickeln. Es soll ein möglichst flexibles, durch Sachbearbeiter ohne Programmierkenntnisse pfleg- und erweiterbares Framework entstehen.

Die „wenn-dann“-Struktur der Promotions („Wenn der Kunde ein Neukunde ist, dann erhält er 5 € Rabatt“) scheint mit Business Rules verwandt zu sein. Business Rules bestehen im Prinzip auch aus einem Bedingungsteil⁴⁵ und einem Ausführungsteil⁴⁶. Diese Artverwandtschaft der beiden Konzepte macht die Business Rules theoretisch zu einer hervorragenden Grundlage um diese Promotions umzusetzen.

Zur Anwendung und Pflege von Promotions mit Hilfe der Business Rules muss zunächst eine Business Rule Engine in die hybris Plattform integriert werden und danach die promotionspezifischen Anforderungen untersucht und implementiert werden.

2.3.2 Einbindung einer Rule Engine

Auswahl

Für die folgende Arbeit und damit für die Integration in hybris wird die einheitliche Java-API *JSR 94*[24] in der Version 1.1 verwendet. Das heißt, dass sämtlicher entstehender Quellcode lediglich die von dieser API definierten Klassen und Interfaces verwendet und keinerlei Funktionalitäten der jeweiligen Implementierung benutzt. Aufgrund dieser Einschränkung können herausragende Funktionen einiger Business Rule Engines nicht verwendet werden, da die *JSR 94* API lediglich den kleinsten gemeinsamen Nenner der notwendigen Funktionen darstellt. Nichtsdestotrotz wird dadurch eine Abhängigkeit von einem speziellen Implementierungsprodukt vermieden, so dass auch später noch theoretisch eine andere Implementierung verwendet werden könnte.

Als konkrete Implementierung wird *JBoss Rules*[7] in der zum Startzeitpunkt der vorliegenden Arbeit aktuellen Version 3.0.5 verwendet, die bereits vollständig *JSR 94* kompatibel ist. Begründet ist die Entscheidung für *JBoss Rules* einerseits durch den durch das JBoss-Lizenzmodell auch für kommerzi-

⁴⁵linke Seite - „wenn“

⁴⁶rechte Seite - „dann“

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

elle Zwecke kostenlosen Einsatz⁴⁷. Andererseits bietet *JBoss Rules* (vor allem im Vergleich zu vielen vergleichbaren Produkten) eine relativ übersichtliche und leicht verständliche Regeldefinitionsprache und es gibt eine sehr große Community die kostenfreien Support leisten kann.

Die Verwendung der *JSR 94* API bedeutet aber nicht gleichzeitig den Verzicht auf besondere Features der jeweiligen Rule Engine. So wird auch in der vorliegenden Arbeit eine Besonderheit der *JBoss Rules*, die *Domain Specific Languages*[7], verwendet. Diese erlauben es Regeln in natürlicher Sprache zu formulieren.

Extension

Die Business Rule Engine wird in hybris als separate Extension eingebunden. Diese Extension wird *rules* genannt. Die gesamte Extension soll komplett unabhängig von anderen Extensions lauffähig sein. Die *rules*-Extension selbst hat zunächst mit den Promotions bzw. den Promotion-Funktionalitäten als solches nichts zu tun. Sie bietet lediglich die Möglichkeit zur Erzeugung, Pflege und Ausführung von Business Rules an. Dadurch kann die Business Rules Funktionalität auch zu anderen Zwecken als Promotions eingesetzt werden.

Exkurs: Mittlerweile wird die Business Rule Extension erfolgreich in kommerziellen Projekten der Firma ARITHNEA GmbH[1] eingesetzt. Beim Online-Shop der Firma Fritz Berger GmbH[3], wo ebenfalls die hybris Plattform verwendet wird, wird die Business Rule Extension produktiv eingesetzt. Dort wird, unter anderem, die Verfügbarkeit von Liefer- und Zahlungsarten und Produkten durch Business Rules gesteuert. Der Einsatz der Business Rule Extension hat sich hier sowohl aus Performance- als auch aus Flexibilitätssicht bewährt.

Erzeugung des Extension-Gerüsts Um in der hybris Plattform eine neue Extension zu erzeugen, wird in der Regel der im Lieferumfang enthaltene Extension-Generator verwendet[4]. Dabei handelt es sich im Prinzip um ein ANT-Skript, was die Erzeugung aller notwendiger Verzeichnisse und Dateien der neuen Extension übernimmt. Die Struktur und Inhalte der Verzeichnisse und Dateien werden aus einer Vorlage, dem so genannten Template, entnommen. Hierbei kann jede beliebige bereits vorhandene Extension als Vorlage für die neue Extension benutzt werden.

⁴⁷GNU General Public License - siehe auch http://www.redhat.com/licenses/jboss_eula.html

2 Hauptteil

Für die *rules*-Extension wird inhaltlich keine Vorlage benötigt, da keine der vorhandenen Extensions verwandt mit der zu erzeugenden *rules*-Extension ist, so dass vom Quellcode oder den Bibliotheken nichts übernommen werden soll⁴⁸. Für diese Zwecke existiert in der hybris Platform eine „leere“ Extension namens *yempty*, die als Basisvorlage ohne Inhalt für alle komplett neu entwickelten Extensions dient. Die *yempty*-Extension enthält lediglich die Verzeichnisstruktur und grundlegende Dateien. Zu diesen Dateien gehören sowohl leere bzw. mit Standardwerten versehene Konfigurationsdateien als auch XML-Schema-Dateien zur Validierung z.B. von XML-Konfigurationsdateien.

Zur Konfiguration des Extension-Generators existiert lediglich eine übersichtliche Propertydatei. Dort werden folgende Einstellungen vorgenommen:

```
extension.name=rules
extension.package=com.arithnea.rules
extension.abstractclassprefix=Generated
extension.managername=RulesManager

extgen.directory.source=../ext/yempty
extgen.directory.target=../ext-commerce/rules
```

Listing 2.4: project.properties vom Extension-Generator

Ein Lauf des Extension-Generators mit den oben genannten Einstellungen erzeugt eine neue Extension namens *rules* im Verzeichnis *hybris-Wurzelverzeichnis/ext-commerce/rules*. Als Vorlage dient die Extension *yempty*. Alle erzeugten Klassen kommen in das package *com.arithnea.rules* und damit in das Verzeichnis *src/com/arithnea/rules*. Die Manager-Klasse für die Extension heißt *RulesManager*. Die für jeden Typen erzeugten abstrakten „Zwischenklassen“ bekommen das Präfix *Generated*, was die Standardeinstellung innerhalb der gesamten hybris Platform ist.

Einbindung der Bibliotheken Sobald das Gerüst der Extension fertig erzeugt ist, muss die Business Rule Engine selbst installiert werden. Hierzu ist es notwendig, die Bibliotheken der Rule Engine (im vorliegenden Fall *JBoss Rules*) in die hybris Platform zu integrieren. JBoss stellt eine fertige Distribution bereit⁴⁹, die nicht nur die Bibliotheken der Business Rule Engine *JBoss Rules*, sondern auch die Bibliotheken für die Java-API *JSR 94* und alle sonstigen Bibliotheken, zu denen Abhängigkeiten

⁴⁸der Extension-Generator kopiert alle in der Vorlage enthaltenen Klassen und Bibliotheken

⁴⁹<http://labs.jboss.com/jbossrules/downloads>

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

bestehen, enthält.

Alle Bibliotheken müssen in das *lib*-Verzeichnis der jeweiligen Extension kopiert werden. Zusätzlich müssen sie innerhalb der hybris Platform registriert werden. Dazu werden die Bibliotheken einzeln in die Datei *extensioninfo.xml*, eine Konfigurationsdatei auf extension-Ebene, in der die Einstellungen für jede Extension vorgenommen werden, eingetragen. Sobald die Bibliotheken registriert sind, sind sie der hybris Platform bekannt und können im Source-Code verwendet werden.

Anmeldung der Extension Damit die Extension bei jedem Kompilieren und Deployment der hybris Platform einbezogen wird, muss sie als aktive Extension angemeldet werden. Dazu muss sie in die Liste der aktiven Extensions aufgenommen werden, welche in der Datei *extensions.xml* im hybris-Hauptverzeichnis gespeichert ist.

```
<hybrisconfig>
  <extensions>
    ...
    <extension dir="../../ext-commerce/rules"/>
  </extensions>
</hybrisconfig>
```

Listing 2.5: extension.xml

Sobald die Extension angemeldet ist, gilt sie als aktiv. Sie wird automatisch in den Kompilier- und Deploymentvorgang der Gesamtanwendung einbezogen. Das bedeutet, dass ab diesem Zeitpunkt andere Extensions auf die neue Extension zugreifen dürfen, und ihre Funktionalitäten verwenden. Darüber hinaus führt ein Fehler, bspw. im Source-Code, ab der Anmeldung der Extension zum Scheitern des Kompilier- bzw. Deploymentvorgangs der Gesamtanwendung.

Um eine Extension vorübergehend zu entfernen, genügt es demnach, die jeweilige Extension aus der Konfigurationsdatei *extensions.xml* zu entfernen.

Eigene Promotion-Extension Wie bereits erwähnt, ist in der *rules*-Extension lediglich die Funktionalität für die Erzeugung, Wartung und Ausführung von Business Rules enthalten. Die für diese Arbeit zu untersuchende Promotion-Funktionalität ist in dieser Extension noch nicht vorhanden. Generell gibt es zwei Möglichkeiten:

- Die Erzeugung einer eigenen Extension für die reine Promotion-Funktionalität

2 Hauptteil

- Das Hinzufügen der Promotion-Funktionalität zur Business Rule-Extension

Für die vorliegende Arbeit wurde entschieden die zweite Möglichkeit zu realisieren. Einerseits ist es aus Gründen der Übersichtlichkeit vorteilhaft, den gesamten zu bearbeitenden Quellcode innerhalb einer Extension zu pflegen. So reicht es bei Änderungen nur diese eine Extension zu kompilieren und zu deployen. Darüber hinaus ist auf den Entwicklungssystemen bereits die für Virgin Megastores entwickelte Promotion-Extension installiert, was u.U. zu Konflikten mit einer weiteren Promotion-Extension führen könnte. Andererseits ist ein Ziel der auf Business Rules basierenden Lösung möglichst ohne bzw. mit möglichst wenig Java-Code auszukommen. Durch die Verwendung von Java-Code geht die Flexibilität der Business Rules, zur Laufzeit durch Sachbearbeiter ohne Programmierkenntnisse Änderungen vorzunehmen, teilweise verloren.

Eventuell zu erstellende Klassen werden demnach in die *rules*-Extension integriert und erhalten dort ein eigenes Verzeichnis bzw. package namens *com.arithnea.rules.promotions*.

Darüber hinaus muss auch in der User-Frontend-Extension *storefoundation* der Code (ähnlich wie bei der Lösung für Virgin Megastores) angepasst werden, damit die auf Business Rules basierende Promotion-Lösung bei Warenkorb-Interaktion überhaupt aufgerufen wird.

Typsystem

Eine der Kernkomponenten jeder Extension ist das jeweilige Typsystem. Die Typen jeder Extension werden jeweils in der Datei *resources/items.xml* definiert. Dabei muss für jedes verwendete Business Objekt ein Typ innerhalb einer Vererbungshierarchie mit eigenen Attributen definiert werden.

Eine *JSR 94* kompatible Business Rule Engine lädt Regeln als so genannte *Rule Execution Sets* aus diversen serialisierbaren oder nicht serialisierbaren Objekten:

- `org.w3c.dom.Element` (auslesen eines XML-Elements)
- `java.io.InputStream` (auslesen eines binären InputStream)
- `java.lang.Object` (auslesen eines herstellerabhängigen abstrakten Syntaxbaums)
- `java.io.Reader` (auslesen von Zeichenketten)
- `java.lang.String` (auslesen von einer URI)

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Nachdem Regeln als Zeichenketten angegeben werden, werden bei der hybrisi Business Rule Extensions die Regeln immer aus einem *java.io.Reader*-Objekt ausgelesen. Wie dieser Reader erzeugt wird, ist dabei abhängig von der konkreten Implementierung. Prinzipiell ist es möglich, ein *Rule Execution Set* als einfachen String zu behandeln und auch nur jeweils in einem String zu speichern. Das hätte allerdings den Nachteil, dass z.B. bei der Verwendung derselben Regel in mehreren Regel-Sets redundanter Code entstehen würde und Änderungen an der Regel in mehreren Regel-Sets vorgenommen werden müssten. Deswegen ist es ratsam, ein modulares System zum Speichern der Regeln zu verwenden. Bei der Abbildung dieses Systems auf das hybrisi Typsystem und bei der Erzeugung des *java.io.Reader* zur Serialisierung der Regel-Sets muss, um eine komfortable Bedienung des Systems zu gewährleisten, auf die konkret verwendete Business Rule Engine eingegangen werden. Das bedeutet, dass, wenn ein Austausch der Business Rule Engine notwendig ist, auch das Typsystem und die Serialisierung der Regel-Sets verändert werden muss. Das ist aber mit relativ geringem Aufwand möglich. Alternativ wäre es auch möglich, das Typsystem von hybrisi so allgemein anzulegen (bspw. als Typ mit nur einem String, der das gesamte Regel-Set enthält), dass der Austausch der Rule Engine ohne Änderung von Typen oder Quellcode möglich ist. Im Hinblick auf die Usability ist dies allerdings nicht zu empfehlen.

Für die persistente Speicherung der Regel-Sets werden in der vorliegenden Arbeit lediglich drei neue hybrisi-Typen benötigt.

Rule Der Typ *Rule* speichert eine einzelne Regel. Die Regel an sich ist für die Business Rule Engine nicht ausführbar. Eine Regel verfügt im Wesentlichen über einen Bedingungsteil (left hand side, *LHS*) und einen Ausführungsteil (right hand side, *RHS*). Darüber hinaus besitzt jede Regel einen Namen. Mit diesen drei Attributen ist es prinzipiell schon möglich eine Regel für *JBoss Rules* abzubilden. Zusätzlich sollen optionale Attribute gepflegt werden können. Diese optionalen Attribute stellen zusätzliche Angaben da, die *JBoss Rules* auswerten kann [7]. Wichtige optionale Attribute sind beispielsweise *no-loop* oder *salience*. Wird das Attribut *no-loop* auf „true“ gesetzt (Standardwert ist „false“), so wird verhindert, dass durch die Ausführung einer Regel die Regel selbst wieder aktiviert wird, was zu einer Endlosschleife führen kann. Das Attribute *salience* entspricht einer manuellen Prioritätsvergabe. Wie bereits erwähnt, ist es empfehlenswert auf explizite Angabe einer Priorität zu verzichten und die Regeln von vornherein so zu anzulegen, dass sie nicht von einer vorgegebene Reihenfolge der Aktivierung abhängig sind.

2 Hauptteil

Als Attribute benötigt der Typ *Rule* demnach einen Namen vom Typ *java.lang.String*. Für die optionalen Attribute erhält er einen Map-Typen, der jeweils einen Schlüssel (Attributnamen) und einen Wert (Attributwert) als *java.lang.String* speichert. Der Bedingungs- und Ausführungsteil werden ebenfalls als Attribute vom Typ *java.lang.String* abgebildet. Aus Gründen der Benutzbarkeit erhält jede Regel zusätzlich noch eine Beschreibung. Diese wird für die eigentliche Erzeugung der Regel bzw. des Regel-Sets nicht benötigt, hilft aber bei der Wartung und Pflege. Die Beschreibung wird als lokalisierter String *localized:java.lang.String* definiert, so dass die Beschreibung auch mehrsprachig, z.B. beim Einsatz in international tätigen Unternehmen, angelegt werden kann. Zuletzt wird noch ein Attribut für die Java-*import*-Statements benötigt. Um der Business Rule Engine die in den Regeln verwendeten Java-Klassen bekannt zu machen, müssen analog zu herkömmlichen *import*-Statements in Java-Klassen auch bei den Rule Engines die Klassen importiert werden. Das geschieht eigentlich nicht für jede Regel separat, sondern auf *Rule Execution Set*-Ebene. Da im hybriden Typ-Modell die Regeln jedoch getrennt von den Regel-Sets gepflegt werden ist es sinnvoll die *imports* der Klassen auf Regel-Ebene zu pflegen, da auf Regel-Set-Ebene die benötigten Java-Klassen nicht sofort ersichtlich sind. Das anzulegende Attribut ist eine Sammlung⁵⁰ mit dem Element-Typ *java.lang.String*.

```
<maptypes>
  <maptype code="RulesAttributeMap" generate="true"
    argumenttype="java.lang.String" autocreate="true"
    returntype="java.lang.String" />
</maptypes>

<itemtype code="Rule"
  extends="GenericItem"
  autocreate="true"
  generate="true"
  deployment="com.arithnea.rules.entity.Rule">
<attributes>
  <attribute qualifier="name" autocreate="true"
    type="java.lang.String">
    <persistence type="property"/>
    <modifiers read="true" write="true" search="true"
      optional="false" unique="true"/>
  </attribute>
</attributes>
```

⁵⁰Java-Interface *java.util.Collection*

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
</attribute>
<attribute qualifier="description" autocreate="true"
  type="localized:java.lang.String">
  <persistence type="property"/>
  <modifiers read="true" write="true" search="true"
    optional="true"/>
</attribute>
<attribute qualifier="lhs" autocreate="true"
  type="java.lang.String">
  <persistence type="property">
    <columnntype database="mysql">
      <value>text</value>
    </columnntype>
  </persistence>
  <modifiers read="true" write="true" search="true"
    optional="false"/>
</attribute>
<attribute qualifier="rhs" autocreate="true"
  type="java.lang.String">
  <persistence type="property">
    <columnntype database="mysql">
      <value>text</value>
    </columnntype>
  </persistence>
  <modifiers read="true" write="true" search="true"
    optional="false"/>
</attribute>
<attribute qualifier="optionalAttributes" autocreate="true"
  type="RulesAttributeMap">
  <persistence type="property"/>
  <modifiers read="true" write="true" search="true"
    optional="true"/>
</attribute>
<attribute qualifier="imports" autocreate="true"
  type="StringCollection">
```

2 Hauptteil

```
<persistence type="property"/>
  <modifiers read="true" write="true" search="true"
    optional="true"/>
</attribute>
</attributes>
</itemtype>
```

Listing 2.6: Itemdefinition für den Typ Rule

Zunächst wird der im Typ *Rule* verwendete Map-Typ⁵¹ *RulesAttributeMap* definiert. Der Typ *RulesAttributeMap* verwendet sowohl für den Schlüssel als auch für den Wert den Typ *java.lang.String*. Der Typ *Rule* selbst wird entsprechend den oben genannten Vorgaben definiert. Wichtig ist hierbei, dass die Attribute *lhs* und *rhs* eine Sonderrolle einnehmen. Standardmäßig werden Attribute vom Typ *java.lang.String* bei MySQL-Datenbanken als Spalten vom Typ *varchar(255)*, also als Zeichenkette mit maximaler Länge von 255 Zeichen, angelegt. Nachdem es wahrscheinlich ist, dass der Bedingungs- bzw. der Ausführungsteil einer Regel länger als 255 Zeichen ist, muss man für diese Attribute einen anderen Spaltentyp angeben. Das geschieht mit einer Reihe von *columnntype*-Tags, jeweils eines pro unterstützter Datenbank. Im oben angegebenen Quelltext wird der Spaltentyp bei MySQL-Datenbanken auf den Typ *text* gesetzt, welcher genug Platz bieten sollte. Selbstverständlich wird auch für alle anderen unterstützten Datenbanken ein eigener *columnntype* angegeben (bei Oracle-Datenbanken bspw. *CLOB*⁵²). Aus Übersichtlichkeitsgründen wurde in der obigen Darstellung aber darauf verzichtet.

Für den neu definierten *Rule*-Typ werden die Klassen *GeneratedRule.java* und die davon ererbende Klasse *Rule.java* erzeugt. In der Klasse *GeneratedRule.java* werden unter anderem für die definierten Attribute entsprechend der Java-Bean-Konvention *getter*- und *setter*-Methoden erzeugt:

- *getName()* / *setName(String)*
- *getDescription()* / *setDescription(String)*
- *getImports* / *setImports(Collection)*
- *getAllOptionalAttributes* / *setAllOptionalAttributes(Map)*
- *getLhs* / *setLhs(String)*

⁵¹Java-Interface *java.util.Map*

⁵²character large object - Zeichenketten bis 4 GB

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

- getRhs / setRhs(String)

RuleSet Regeln werden bei der *JSR 94* API immer im Rahmen von *Rule Execution Sets* ausgeführt. Diese Regel-Sets umfassen einen bestimmten Themenbereich (bspw. Promotion) von Regeln, die zusammengehören. Aus einem Regel-Set wird beim *Rete*-Algorithmus das *Rete*-Netz erzeugt. Unterschiedliche Regel-Sets verfügen über unterschiedliche Working-Memories und teilen sich somit keine Objekte, werden also vollkommen unabhängig voneinander ausgeführt.

In *hybris* werden diese *Rule Execution Sets* durch den Typen *RuleSet* abgebildet. Hauptbestandteil eines *RuleSet* sind die einzelnen Regeln vom Typ *Rule*. Darüber hinaus benötigt jedes *RuleSet* einen eindeutigen Namen, unter dem das *Rule Execution Set* später dann auch geladen und aufgerufen werden kann. Neben dem Namen benötigt jedes Regel-Set einen so genannten Header. Dort können wichtige Verarbeitungsinformationen angegeben werden. Im Header eines *Rule Execution Set* wird bei den *JBoss Rules* ein Package angegeben, analog zur Package-Angabe in herkömmlichen Java-Klassen. Darüber hinaus werden die *Imports* angegeben, die bei der *hybris* Implementierung automatisch aus den jeweils vom Regel-Set verwendeten Rules übernommen werden. Zusätzlich können so genannte *Functions* definiert werden, die sich ähnlich wie herkömmliche Methoden in Java verhalten. Mit diesen *Functions* können oft benutzte Funktionalitäten oder Berechnungen aus den Regeln ausgelagert werden.

Exkurs: Je nachdem wie stark die *Functions* genutzt werden, kann es sinnvoll sein, diese *Functions* als eigenen Typen auszulagern und mit den *RuleSet*-Typen lediglich zu verknüpfen. So könnten die *Functions* separat von den Regel-Sets gepflegt werden und auch innerhalb mehrerer Regel-Sets frei von Redundanzen genutzt werden.

Ähnlich wie der Typ *Rule* erhält auch der Typ *RuleSet* eine Beschreibung, die zum Ausführen der Rule Engine selbst eigentlich nicht benötigt wird, aber für die Wartung und Pflege der *RuleSets* nützlich ist. Optional kann jedem *RuleSet* ein Objekt vom Typ *DSL* zugeordnet werden, was im folgenden Absatz vorgestellt werden wird.

```
<collectiontypes>
  <collectiontype code="RulesCollection" elementtype="Rule"
    autocreate="true" generate="true" type="list"/>
</collectiontypes>

<itemtype code="RuleSet"
```

2 Hauptteil

```
        extends="GenericItem"
        autocreate="true"
        generate="true">
<attributes>
    <attribute qualifier="name" autocreate="true"
        type="java.lang.String">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="false" unique="true"/>
    </attribute>
    <attribute qualifier="description" autocreate="true"
        type="localized:java.lang.String">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="true"/>
    </attribute>
    <attribute qualifier="header" autocreate="true"
        type="java.lang.String">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="false"/>
    </attribute>
    <attribute qualifier="dsl" autocreate="true" type="DSL">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="true"/>
    </attribute>
    <attribute qualifier="rules" autocreate="true"
        type="RulesCollection">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="true"/>
    </attribute>
</attributes>
```

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

</itemtype>

Listing 2.7: Itemdefinition für den Typ RuleSet

Um eine Liste von Rules speichern zu können, muss zunächst der Collection-Typ *RulesCollection* angelegt werden. Dieser Typ entspricht einer Sammlung⁵³ von Rules-Objekten. Für das Attribut Rules des *RuleSet*-Typen wird dieser *RulesCollection*-Typ verwendet.

DSL Wie bereits erwähnt, wird aus Gründen der Usability die Möglichkeit von *JBoss Rules* der Definition von *Domain Specific Languages* verwendet. Technisch gesehen bestehen die *Domain Specific Languages* prinzipiell lediglich aus einer Übersetzungstabelle. Eine angegebene Zeichenkette kann dabei in eine andere Zeichenkette überführt werden. Dabei können Platzhalter bzw. Variablen definiert werden, um dynamische Daten übergeben zu können.

```
Bedingungsteil einer Regel ohne DSL:  
Cart (total > 50.00)  
mögliche DSL:  
Warenkorbwert über '{0}'=Cart (total > {0})  
Bedingungsteil mit DSL:  
Warenkorbwert über '50.00'
```

Listing 2.8: Beispiel für eine DSL

Sobald eine *DSL* verwendet wird, werden beim Kompilieren des Regel-Sets durch *JBoss Rules* zeilenweise der Bedingungs- und Ausführungsteil jeder Regel übersetzt. Das bedeutet, dass sobald eine *DSL* verwendet wird auch jede Zeile in der *DSL* enthalten sein muss. Durch Voranstellen des Zeichens > wird die jeweilige Zeile von der Übersetzung ausgenommen (escaped) und es können herkömmliche Regeldefinitions-Sprachkonstrukte verwendet werden.

Der hybride Typ *DSL* benötigt für eine korrekte Funktionsweise eigentlich nur ein Textfeld, welches das eigentliche Mapping der Zeichenketten enthält. Dazu wird ein einfaches Attribut vom Typ *java.lang.String* verwendet. Daneben erhält der *DSL*-Typ (analog zu den Typen *Rule* und *RuleSet*) auch die Attribute *name* und *description*, die beide nur zur besseren Verwaltung und Übersichtlichkeit beitragen und eigentlich nicht notwendig sind.

⁵³Java-Interface *java.util.Collection*

2 Hauptteil

```
<itemtype code="DSL"
  extends="GenericItem"
  autocreate="true"
  generate="true">
  <attributes>
    <attribute qualifier="name" autocreate="true"
      type="java.lang.String">
      <persistence type="property"/>
      <modifiers read="true" write="true" search="true"
        optional="false" unique="true"/>
    </attribute>
    <attribute qualifier="description" autocreate="true"
      type="localized:java.lang.String">
      <persistence type="property"/>
      <modifiers read="true" write="true" search="true"
        optional="true"/>
    </attribute>
    <attribute qualifier="mapping" autocreate="true"
      type="java.lang.String">
      <persistence type="property"/>
      <modifiers read="true" write="true" search="true"
        optional="false"/>
    </attribute>
  </attributes>
</itemtype>
```

Listing 2.9: Itemdefinition für den Typ DSL

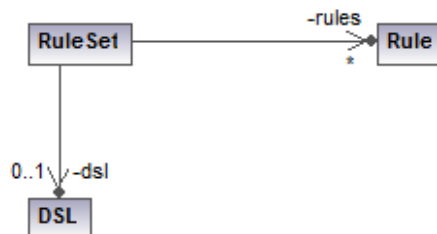


Abbildung 2.9: Klassendiagramm der angelegten hybrid Typen

Datenbankebene

Zur Sicherung der Daten in der Datenbank gibt es, wie bereits im Grundlagen-Kapitel erläutert, zwei Möglichkeiten.

- Es wird explizit ein Datenbankmapping angegeben, d.h. es wird eine eigene Tabelle in der Datenbank für einen neu definierten Typen angelegt. Diese Tabelle enthält dann die Attribute des Typen als Spalten.
- Die Verwaltung der Daten auf Datenbankebene wird von hybris automatisch gesteuert. In diesem Fall wird der neu definierte Typ in der Tabelle des nächstverwandten Supertypen mit eigener Tabelle gespeichert. Diese Tabelle wird dann um die den neu definierten Attributen entsprechenden Spalten erweitert.

Für die persistente Sicherung der Typen der Business Rule Engine werden für die vorliegende Arbeit sowohl die erste als auch die zweite Möglichkeit verwendet, um die Mechanismen zu demonstrieren. Generell ist aus Performance-Gründen die erste Variante vorzuziehen, da dort die *Items* über mehrere Tabellen verteilt werden und Tabellen von Typen mit vielen Subtypen ohne eigenes Deployment nicht so viele (hauptsächlich leere) Spalten enthalten.

Bei der Business Rules-Funktionalität ist allerdings davon auszugehen, dass nur relativ wenige *Items* angelegt werden müssen (ein *Item* pro *RuleSet*, ein *Item* pro *Rule*, ein *Item* pro *DSL*). Wenn man davon ausgeht, dass zehn *RuleSets* mit eigener *Domain Specific Language* und jeweils zehn Rules angelegt werden, so werden insgesamt lediglich 120 *Items*, d.h. Zeilen in Tabellen, angelegt. Für diese geringe Anzahl von *Items* sind eigene Tabellen eigentlich nicht notwendig. Hinzu kommt, dass auf diese *Items* nur relativ selten zugegriffen werden muss. Sobald ein *RuleSet* beim ersten mal aufgerufen wird, werden aus der Datenbank die zugehörigen *Items* geladen. Danach wird die Regel vom *Rete*-Algorithmus in ein *Rete*-Netz transformiert und in dieser Form im Speicher gehalten. Für die Auswertung des *RuleSets* müssen also nicht mehr jedes mal die *Items* aus der Datenbank geladen werden.

Für die Typen *RuleSet* und *DSL* wird keine eigenes Deployment angegeben. Das heißt, sie werden in der Tabelle *genericitems* (der nächsthöhere Supertyp mit eigenem Deployment) gespeichert. Diese Tabelle wird dann um die in diesen beiden Typen neu definierten Attribute (z.B. *p_dsl*, *p_rules*, *p_header* etc. erweitert.

Für den von der Anzahl der *Items* umfangreichsten Typen *Rule* wird ein eigenes Datenbank-

2 Hauptteil

Deployment angegeben. Dazu wird in der Definition des Typen das zusätzliche Attribut *deployment* angegeben:

```
<itemtype code="Rule"
    extends="GenericItem"
    autocreate="true"
    generate="true"
    deployment="com.arithnea.rules.entity.Rule">
    <attributes>
    ...
    </attributes>
</itemtype>
```

Listing 2.10: items.xml mit eigenem Deployment

Dieses Deployment-Attribut dient lediglich als Identifier, der auf ein, in der Datei *advanced-deployment.xml* definiertes, so genanntes Bean verweist.

```
<package name="com.arithnea.rules.entity">
    <bean name="Rule" generic="true" abstract="false"
        type="Entity" typecode="6200" reentrant="True"
        cmp-type="Container">
        <extends target="GenericItem"/>
        <bean-mapping persistence-name="Rules"/>
    </bean>
</package>
```

Listing 2.11: Ausschnitt aus advanced-deployment.xml

Der hier vorgestellte Code enthält noch viele aus der Version 2 der hybris Plattform übernommenen Bestandteile. Dort wurden wie anfangs erwähnt noch J2EE Enterprise Beans, in diesem Fall Entity Beans, zur Abbildung der Daten verwendet. In der Version 3 der hybris Plattform wurde die Entity Beans durch ein proprietäres O/R-Mapping-Framework ersetzt. Wichtig ist lediglich der Name des Beans, der zusammen mit dem Namen des enthaltenden Packages den referenzierten Identifier enthält und der Name der Tabelle, in der das *Item* gespeichert werden soll. Dieser Name wird innerhalb des Tags *bean-mapping* im Attribut *persistence-name* definiert. Durch den hier präsentierten Code wird in der Datenbank eine Tabelle mit dem Namen *Rules* erzeugt, die für die Speicherung der Objekte des

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Typs *Rule* verantwortlich ist.

Field	Type	Null	Key	Default	Extra
hjmpTS	bigint(20)	YES		NULL	
PK	bigint(20)	NO	PRI		
createdTS	datetime	NO			
modifiedTS	datetime	YES		NULL	
aCLTS	bigint(20)	YES		0	
TypePkString	bigint(20)	NO	MUL		
OwnerPkString	bigint(20)	YES	MUL	NULL	
propTS	bigint(20)	YES		0	
p_optionalattributes	longblob	YES		NULL	
p_rhs	text	YES		NULL	
p_lhs	text	YES		NULL	
p_imports	longblob	YES		NULL	
p_name	varchar(255)	YES		NULL	

Tabelle 2.2: Schema der Tabelle Rules

Administrationskonsole

Die gesamte hybris Plattform wird von den Betreibern des Online-Shops in der Administrationskonsole *hmc* gepflegt. Dort werden neue Objekte angelegt und vorhandene Objekte gepflegt oder gelöscht. Darüber hinaus werden dort auch bestimmte Aktionen (z.B. Ausführen so genannter Jobs, bspw. Generierung von Suchindizes oder Berechnung von Produktkorrelationen) gesteuert. Erklärtes Ziel von hybris ist es, mit der Administrationskonsole, die in jedem herkömmlichen Webbrowser läuft, ein zentrales, überall verfügbares Administrationswerkzeug bereit zu stellen. Aus diesem Grund müssen die Business Rules ausschließlich in der Administrationskonsole pflegbar sein, ein weiteres externes Werkzeug soll nicht verwendet werden.

Wie bereits erwähnt wird die Oberfläche der Administrationskonsole über xml-Dateien konfiguriert. Eine solche xml-Konfigurationsdatei ist in jeder Extension vorhanden, so dass die Darstellung der neu definierten Typen in der Administrationskonsole auch in der Extension selbst definiert werden

2 Hauptteil

kann. Dadurch bleibt die Portabilität der einzelnen Extensions erhalten, da die Konfiguration der Darstellung der Extension in der Administrationskonsole Teil der Extension selbst und nicht der Administrationskonsole ist.

Die Darstellung der Business Rule Extension umfasst die Darstellung der drei neu angelegten Typen *Rule*, *RuleSet* und *DSL* selbst sowie deren Integration in die Navigationshierarchie. Nachdem die Business Rule-Funktionalität nicht mit einer anderen bereits vorhandenen Gruppe (bspw. Katalog oder Multimedia) verwandt ist, wird dafür eine eigene Gruppe definiert.

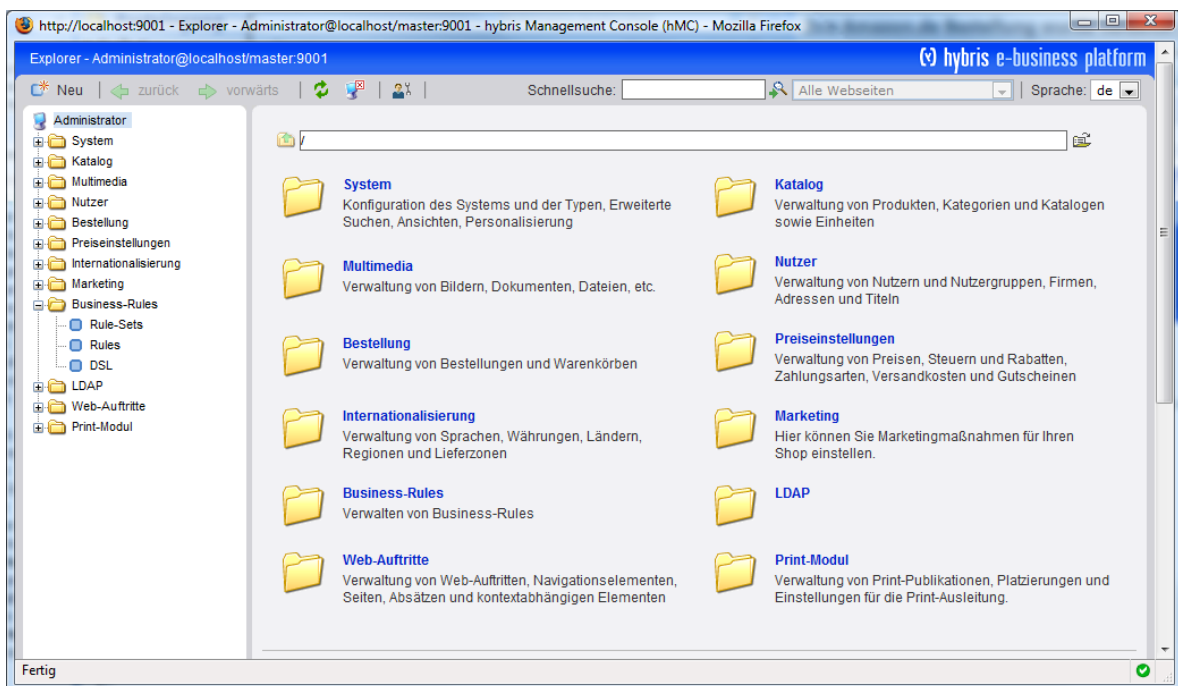


Abbildung 2.10: Administrationskonsole mit eingebundener Business Rules-Gruppe

Um die Business Rules-Gruppe wie auf dem Screenshot dargestellt einzubinden, muss in der Datei `webmc.xml` im Verzeichnis `webmc/resources` der `rules`-Extension folgender Code eingefügt werden:

```
<explorertree>
  <group name="rules" description="group.rules.description">
    <typeref type="RuleSet" description="typeref.ruleset.description"/>
    <typeref type="Rule" description="typeref.rule.description"/>
    <typeref type="DSL" description="typeref.dsl.description"/>
  </group>
</explorertree>
```

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
</group>  
</explorertree>
```

Listing 2.12: XML zur Darstellung der Business Rules-Gruppe

Sobald die hybris Platform aktualisiert worden ist, wird die Gruppe angezeigt. Die Lokalisierung der einzelnen Elemente wird für jede Sprache nach dem Muster

```
group.rules.description=Verwalten von Business Rules  
type_tree_ruleset=Rule-Sets  
type_tree_rule=Rules  
type_tree_dsl=DSL
```

Listing 2.13: Lokalisierung der Administrationskonsole

in eine property-Datei innerhalb der Extension geschrieben. Dadurch ist die Integration der neuen Typen in die Navigationsstruktur der Administrationskonsole erledigt.

Zusätzlich muss für jeden der drei neu definierten Typen selbst die Darstellung definiert werden. Hier soll aus Platzgründen lediglich die Definition der Darstellung für den Typen *Rule* präsentiert werden, die beiden anderen Typen werden analog definiert. Für die Darstellung jedes Typen gibt es im wesentlichen vier Ebenen:

Suchanfrage Wählt man in der Administrationskonsole einen Typen aus, so wird zunächst eine Suchmaske angezeigt. In dieser Suchmaske kann man Anfragekriterien über alle Attribute des Typen angeben. Beispielsweise kann man definieren, dass das Attribut *name* des Typen die Zeichenkette „xyz“ enthalten soll. Welche Attribute standardmäßig in der Suchmaske angezeigt werden, muss definiert werden.

Suchergebnis Sobald die Suche ausgeführt wird, wird ein Suchergebnis angezeigt. Dieses Suchergebnis enthält beliebig viele Objekte des gesuchten Typen, die den Anfragekriterien entsprechen. Wie dieses Suchergebnis angezeigt wird, muss definiert werden. Dabei gibt es bei der Darstellung der Suchergebnisse drei verschiedene Ansichten:

listview Die Ergebnisse werden als Liste von Objekten ausgegeben. Dabei werden von jedem gefundenen Objekt die definierten Attribute angezeigt.

editview Ähnlich wie bei listview werden auch hier die gefundenen Objekte mit den definierten Attributen angezeigt, allerdings können die Objekte direkt im Suchergebnis manipu-

2 Hauptteil

liert und verändert werden.

treeview Die gefundenen Objekte werden in einer Baumstruktur dargestellt, was insbesondere bei verschachtelten Strukturen (wie bspw. Kategoriebäumen) Sinn macht.

Editor Sobald man ein Objekt öffnet, um es zu manipulieren, wird der Editor verwendet. Es muss definiert werden, welche Attribute an welcher Stelle wie angezeigt werden.

Reference Sobald ein Typ von einem anderen Typ referenziert wird (bspw. der Typ *Rule* vom Typ *RuleSet* aus, muss definiert werden, wie diese Objektreferenz angezeigt wird, bzw. welche Attribute das referenzierte Objekt möglichst gut beschreiben.

```
<type name="Rule">
  <organizer>
    <search>
      <condition attribute="name" />
      <condition attribute="description" />
    </search>
    <result defaultview="list">
      <listview>
        <itemlayout>
          <attribute name="name" width="100" />
          <attribute name="description" width="200" />
        </itemlayout>
      </listview>
      <editview>
        <itemlayout>
          <attribute name="name" width="100" />
          <attribute name="description" width="200" />
        </itemlayout>
      </editview>
      <treeview>
        <itemlayout>
          <attribute name="name" />
          <attribute name="description" />
        </itemlayout>
      </treeview>
    </result>
  </organizer>
</type>
```

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
</result>
<editor>
  <essentials>
    <listlayout>
      <attribute name="name" labelwidth="100" />
      <attribute name="description" labelwidth="100"
        width="400">
        <textareaeditor expanded="true" wrap="false" rows="3"
          />
      </attribute>
    </listlayout>
  </essentials>
  <tab name="tab.properties">
    <section name="common">
      <listlayout>
        <attribute name="imports" labelwidth="100" />
        <attribute name="optionalAttributes" labelwidth="100"
          />
        <attribute name="lhs" labelwidth="100" width="400">
          <textareaeditor expanded="true" wrap="false"
            rows="10"/>
        </attribute>
        <attribute name="rhs" labelwidth="100" width="400">
          <textareaeditor expanded="true" wrap="false"
            rows="10"/>
        </attribute>
      </listlayout>
    </section>
  </tab>
  <tabref idref="tab_administration" />
</editor>
</organizer>
<defaultreference>
  <itemlayout>
    <attribute name="name" />
```

2 Hauptteil

```
<attribute name="description" />
</itemlayout>
</defaultreference>
</type>
```

Listing 2.14: XML zur Darstellung des Typs Rule

DefaultReference Die Beschreibung der Darstellung des Typs *Rule* zerfällt in zwei Teile, dem *Organizer*-Tag und dem *DefaultReference*-Tag. Das *DefaultReference*-Tag beschreibt die letzte oben genannte Referenz-Ebene. Die hier angeführten Attribute werden angezeigt, sobald ein Objekt vom Typ *Rule* innerhalb eines referenzierenden Typen (bspw. *RuleSet*) angezeigt wird (siehe Abbildung 2.11). Mit der angegebenen Definition wird ein Objekt vom Typ *Rule* immer mit Name und Beschreibung referenziert, was für die meisten Anwendungsfälle ausreichen sollte. Wird die Referenz-Ebene nicht definiert, so wird standardmäßig die Referenz-Sicht des nächsten Supertypen angezeigt, was im Falle vom Typ *Rule* der Typ *GenericItem* ist, dessen *DefaultReference* lediglich aus dem (nicht aussagekräftigen) Primary Key des Objekts besteht.

Organizer Innerhalb des *Organizer*-Tags werden die drei anderen Ebenen, also Suchanfrage (*search*), Suchergebnis (*result*) und Editor (*editor*) definiert (siehe Abbildung 2.12). Die Suchanfrage ist beim Typ *Rule* so definiert, dass standardmäßig die Attribute *name* und *description* als Suchbedingungen zur Verfügung stehen. Selbstverständlich sind auch zur Laufzeit dynamisch neue Suchbedingungen hinzufügbare.

Für das Suchergebnis werden für die drei verfügbaren Sichten (*listview*, *editview*, *treeview*) die angezeigten Attribute jeweils separat definiert. Für den Typ *Rule* werden jedoch immer dieselben Attribute *name* und *description* verwendet (siehe Abbildung 2.13). Da die Sichten *listview* und *editview* eine Art tabellarische Darstellung sind kann hier noch die Spaltenbreite des jeweiligen Attributes (in Pixeln) definiert werden.

Die wichtigste Darstellungsebene in der Administrationskonsole ist die Bearbeitungssicht (*editor*). Im Editor können alle Attribute des jeweiligen Objekts verändert werden (siehe Abbildung 2.14). Der

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

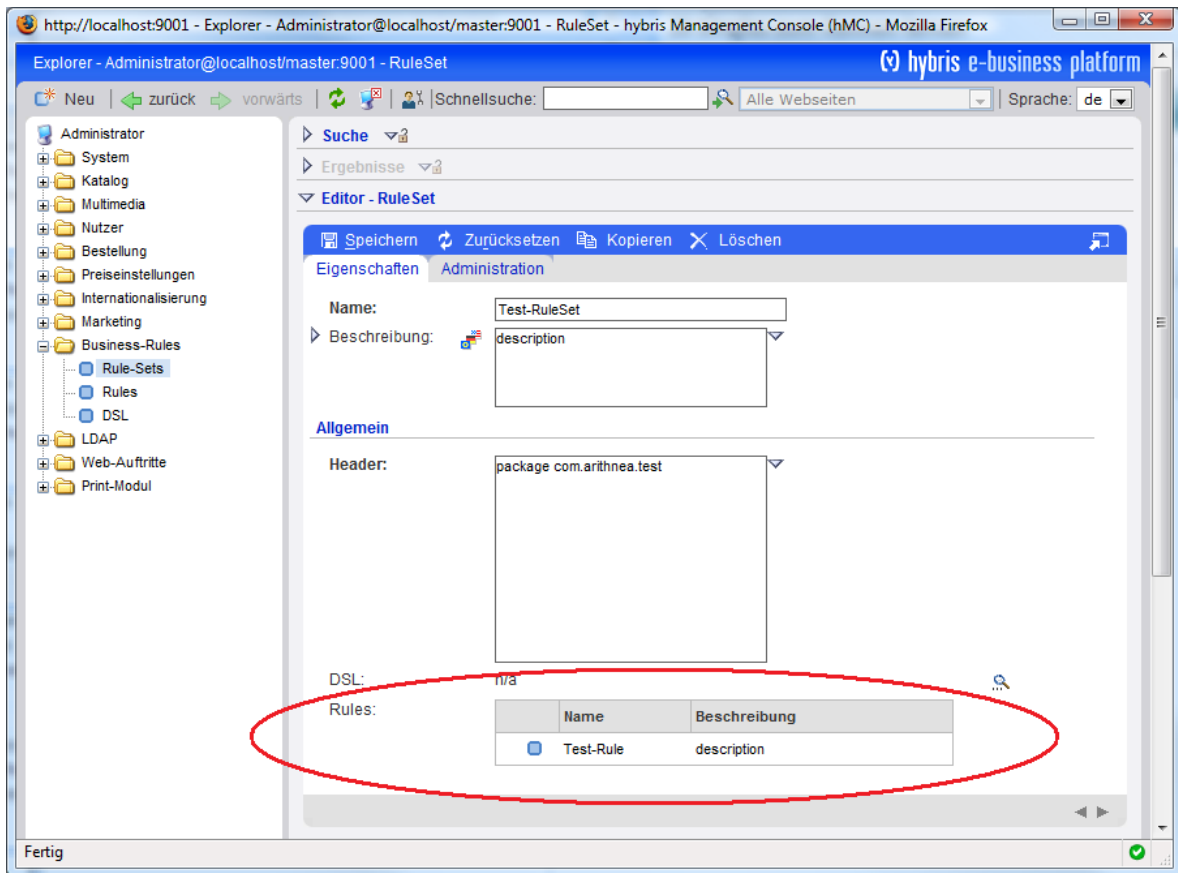


Abbildung 2.11: DefaultReference für den Typ Rule

Editor verfügt zur Verbesserung der Übersichtlichkeit über die Möglichkeit so genannte Karteireiter (tabs) zu definieren, so dass zusammenhängende Attribute auf einer Seite dargestellt werden. Für den Typ *Rule* werden zwei dieser Karteireiter definiert, *Properties* für die neu definierten Attribute des Typs selbst und ein Administrations-Tab, in dem alle von den Supertypen geerbten Attribute, die zur „normalen“ Bearbeitung der Objekte nicht geändert werden müssen, angezeigt werden. Das Administrations-Tab wird für den Typen *Rule* nicht extra definiert, es wird lediglich die an einer anderen Stelle (außerhalb der *rules*-Extension) vorgenommene Standarddefinition referenziert.

Die unter dem Tag *essentials* angegebenen Attribute werden auf jedem Karteireiter oben dargestellt, was hilfreich ist, um zu wissen welches Objekt gerade bearbeitet wird. Die dort angezeigten Attribute werden im Typ *Rule* als Liste, also untereinander, angezeigt. Damit die Felder einheitlich untereinander angezeigt werden, wird zusätzlich definiert, dass der angezeigte Attributname (label)

2 Hauptteil

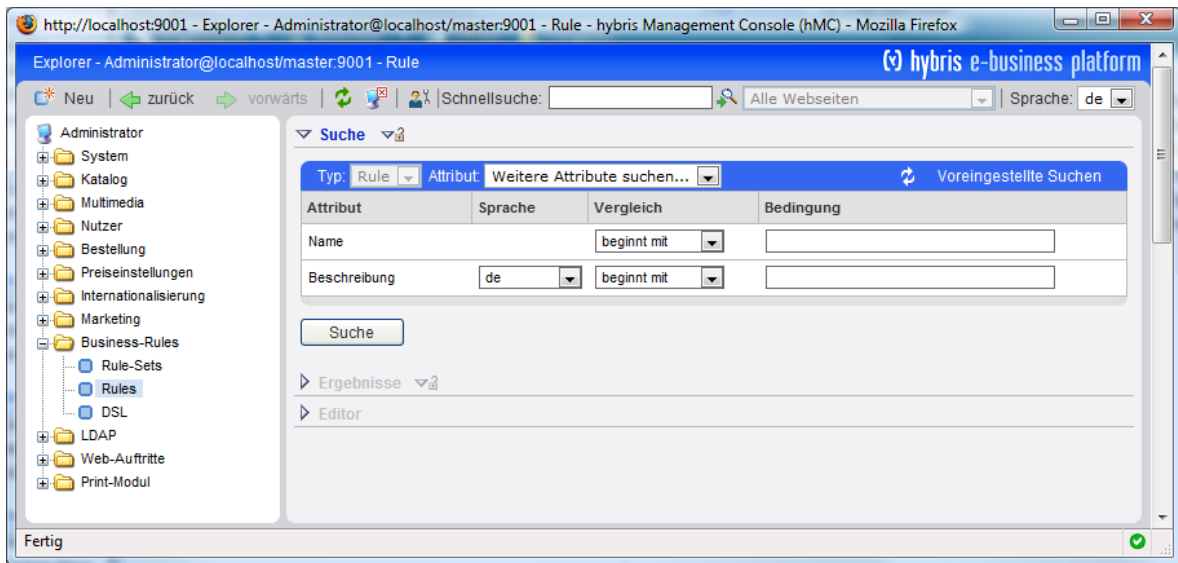


Abbildung 2.12: Suchanfrage für den Typ Rule

immer eine Breite von 100 Pixeln hat. Für das Attribut *description* wird extra definiert, wie das Attribut genau angezeigt werden soll. Im Gegensatz zum Attribut *name*, was entsprechend der Standardeinstellung für Attribute vom Typ *java.lang.String* als einzeiliges Textfeld dargestellt wird, verwendet das Attribut *description* eine andere Darstellungsvariante. Es wird als 3-zeiliges, 400 Pixel breites, von vornherein ausgeklapptes Textfeld ohne automatischen Zeilenumbruch dargestellt. Das dient bei Attributen, die ihrem Verwendungszweck entsprechend voraussichtlich viel Text enthalten, der Übersichtlichkeit.

Die verbleibenden Attribute werden auf dem Karteireiter für normale Eigenschaften (*properties*) dargestellt im Abschnitt für allgemeine (*common*) Attribute.

Implementierung

Mit der Abbildung der Regeln, Regel-Sets und *Domain Specific Languages* in das hybris Typsystem können diese in der Administrationskonsole bearbeitet und persistent in der Datenbank gespeichert werden. Um die Regeln auch tatsächlich ausführen zu können ist es notwendig, diese Regel-Sets aus der Datenbank zu laden und durch die Business Rule Engine auf die Ausführung vorzubereiten. Dazu

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

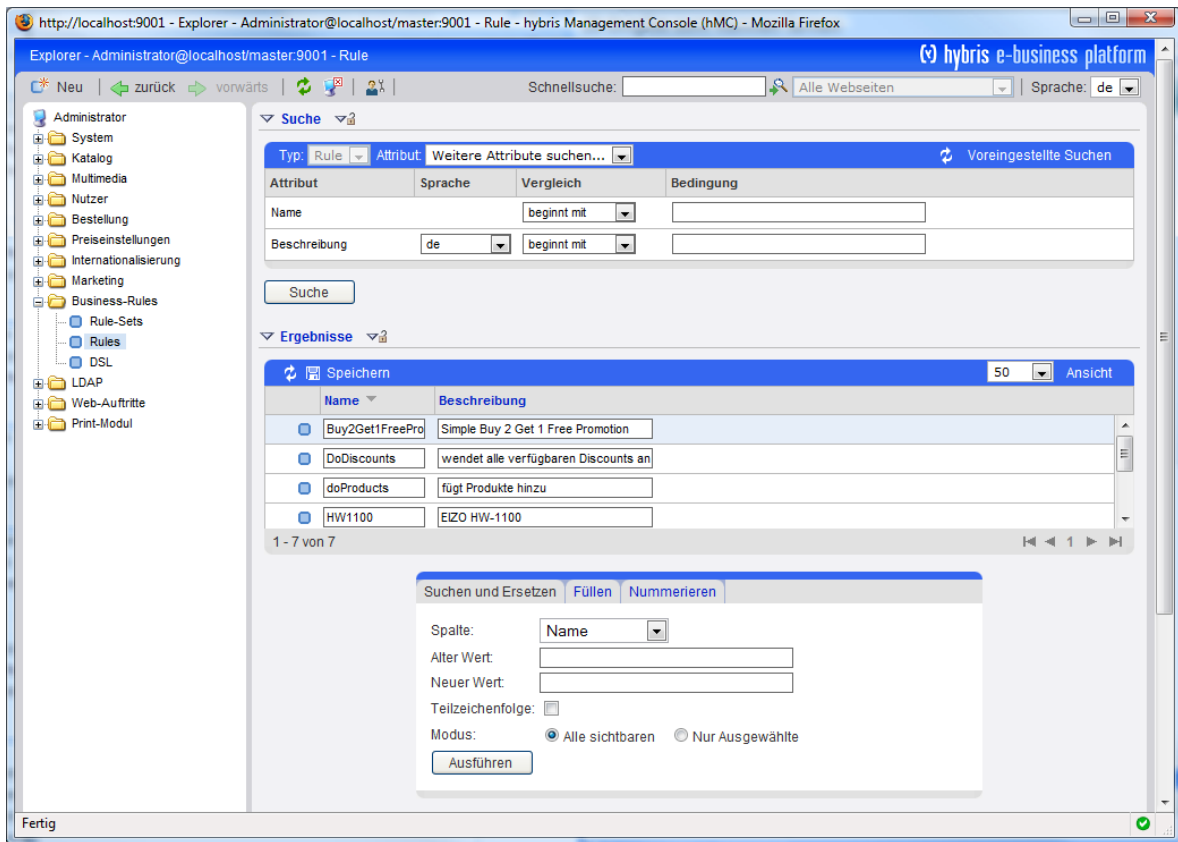


Abbildung 2.13: Suchergebnis für den Typ Rule als editview

ist es notwendig auf Quellcode-Ebene die erforderlichen Klassen und Methoden zu implementieren. Zentraler Anlaufpunkt gemäß dem Façade-Pattern[20] für die Funktionalität der Business Rules soll die Klasse *RulesManager* sein. Die gesamte Funktionalität der Extension soll demnach nur über diese Klasse gesteuert werden können.

Ausführung eines Regel-Sets Zum Ausführen eines bestimmten Regel-Sets wird die Methode *executeRules* definiert:

```
public List executeRules(String ruleset, List objects, ObjectFilter  
    filter) {...}
```

Listing 2.15: Signatur der executeRules-Methode

2 Hauptteil

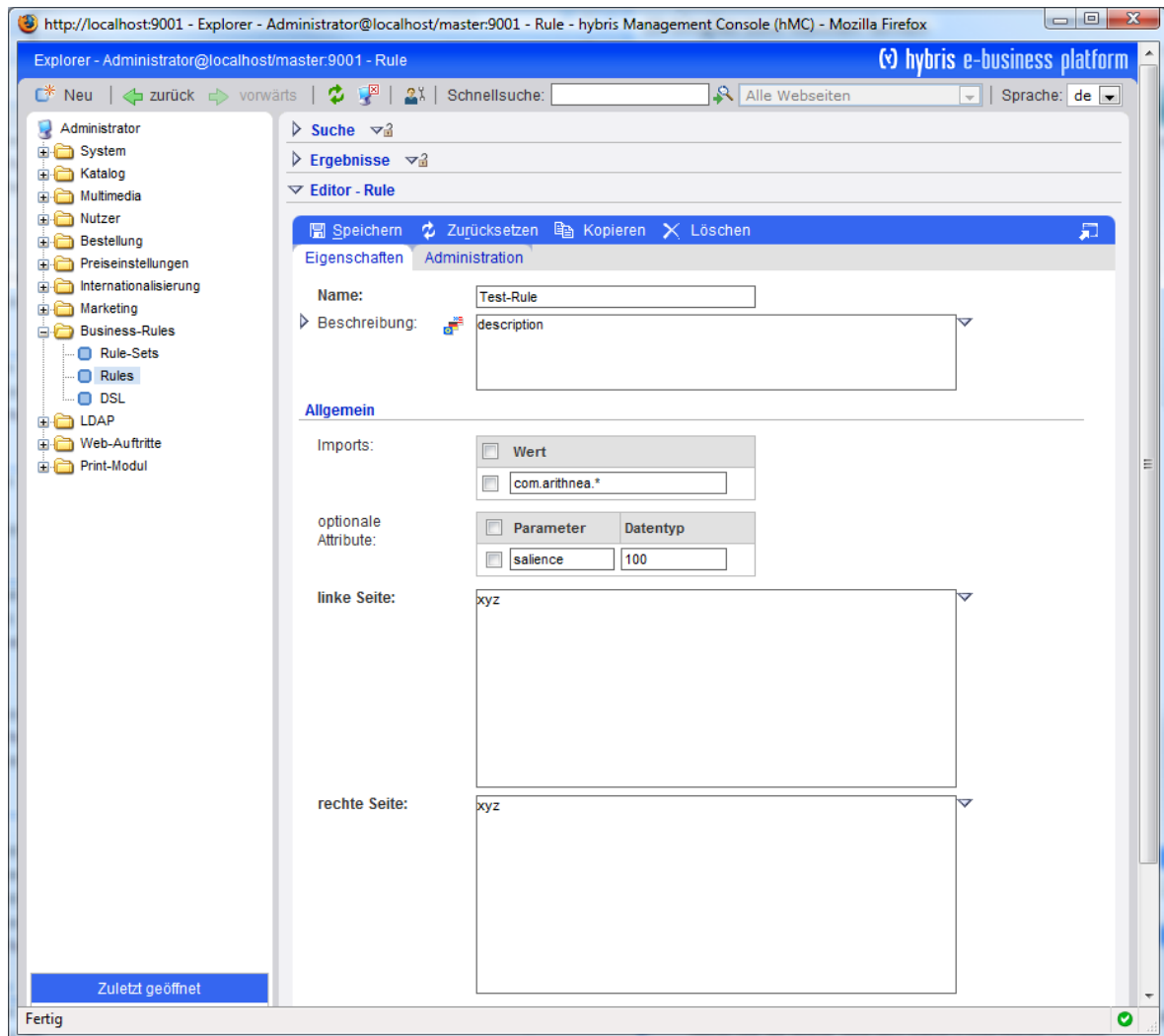


Abbildung 2.14: Editor für den Typ Rule

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Gemäß der *JSR 94*-Definition entspricht diese Methode dem Ausführen einer zustandslosen Rules-Sitzung. Es wird angegeben, welches Regel-Set ausgeführt werden soll und welche Objekte initial an das erzeugte *Rete*-Netzwerk übergeben werden. Als Ergebnis wird der gesamte *Working Memory* (d.h. alle enthaltenen Objekte) nach Auswertung der Regeln als Objekt-Liste zurückgegeben.

Objektfilter Da in der Regel nur wenige Objekte des *Working Memory* als Ergebnis interessant sind, kann zusätzlich ein ObjektFilter übergeben werden. Dieser ObjektFilter, der das von der *JSR 94*-API definierte Interface *javax.rules.ObjectFilter* implementieren muss, sorgt dafür, dass nur bestimmte Objekte als Ergebnis einer Auswertung zurückgegeben werden. Für die vorliegende Arbeit wird ein Filter benötigt, der nur für Instanzen einer bestimmten Klasse „durchlässig“ ist. Dieser wird implementiert als die Klasse *com.arithnea.rules.filter.RulesObjectFilter*.

```
package com.arithnea.rules.filter;

import javax.rules.ObjectFilter;

public class RulesObjectFilter implements ObjectFilter {

    private Class c;

    public RulesObjectFilter(Class c)
    {
        this.c = c;
    }

    public Object filter(Object object) {
        return (object.getClass().equals(c)) ? object : null;
    }

    public void reset() {
        // do nothing
    }
}
```

Listing 2.16: Die Klasse *com.arithnea.rules.filter.RulesObjectFilter*

2 Hauptteil

Dieser *RuleObjectFilter* kann mit der gewünschten Klasse erzeugt werden und ist nur durchlässig für Instanzen der jeweiligen Klasse. Beispiel: Ein mit der Klasse *String* instanzierter *RulesObjectFilter*⁵⁴ ist nur durchlässig für *String*-Objekte.

Wird beim Aufruf der *executeRules* kein *ObjectFilter* (null) übergeben, so werden alle im *Working Memory* enthaltenen Objekte zurückgegeben.

Serialisieren eines Regel-Sets Sobald ein Regel-Set zum ersten mal ausgeführt werden soll, muss es zunächst in den Speicher der Business Rule Engine geladen werden. Dort wird das *Rete*-Netzwerk erzeugt und dann im Arbeitsspeicher gehalten, so dass das Laden des Regel-Sets nur einmal erfolgen muss, egal wie oft das Regel-Set danach ausgeführt wird.

Wie bereits erwähnt, werden in der vorliegenden Arbeit die *Rule Execution Sets* von *JBoss Rules* immer aus einem Objekt der Klasse *java.io.Reader* geladen. Da ein *Rule Execution Set* im *hybris* Typsystem jedoch nicht als atomares Objekt, sondern als ein Objekt vom Typ *RuleSet*, einem (optionalen) Objekt vom Typ *DSL* und beliebig vielen Objekten vom Typ *Rule* besteht, ist eine komplexe Serialisierung notwendig. Diese serialisiert ein Objekt vom Typ *RuleSet* (mit allen zugehörigen Objekten) zu einem Objekt der Klasse *java.io.Reader*.

Dazu wird in der von *hybris* erzeugten Klasse *com.arithnea.rules.jalo.RuleSet* die Methode *getReader* hinzugefügt, die die persistenten *hybris*-Objekte in ein *Rule Execution Set* in dem von *JBoss Rules* verwendeten Format übersetzt:

```
public Reader getReader()
{
    StringBuilder sb = new StringBuilder();
    sb.append(this.getHeader() + "\n");
    if (this.getDsl() != null) sb.append("expander jsr94.dsl" + "\n");
    sb.append(getImports());
    sb.append("global org.apache.log4j.Logger log\n");
    sb.append("\n");
    for (Rule rule : (Collection<Rule>) this.getRules())
    {
        sb.append(rule.getRule());
        sb.append("\n");
    }
}
```

⁵⁴`new RulesObjectFilter(String.class)`

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
}  
  
log.info("Generated Ruleset:\n" +sb.toString());  
  
return new StringReader(sb.toString());  
}
```

Listing 2.17: getReader-Methode in der Klasse RuleSet

- Export des im *RuleSet* definierten Headers. Dieser enthält die package-Angabe und eventuell einige *Functions*.
- Falls eine *Domain Specific Language* für das zu exportierende *RuleSet* angegeben ist, wird eine feste Markierung („expander jsr94.dsl“) gesetzt, so dass die beim Laden des *RuleSet* separat anzugebende *DSL* auch verwendet wird.
- Export der *java-Imports*. Diese werden wie bereits erwähnt auf *Rule*-Ebene definiert. Beim Export werden die *Imports* von allen mit dem *RuleSet* verknüpften Rules verwendet. Hierbei kann es zu Duplikaten kommen, wenn zwei Rules denselben *Import* verwenden. Diese Duplikate werden nicht eliminiert, da mehrfach angegebene *Imports* sich nicht negativ auswirken.
- Export des Loggers. Dieser Schritt ist prinzipiell nicht notwendig, dennoch ist es sinnvoll in jeder Regel einen Logger verwenden zu können, ohne dass dies explizit angegeben werden muss. Der Logger wird als global definiert, und steht damit allen Regeln zur Verfügung. Die Instanz des Loggers selbst wird bei der Erzeugung des *Working Memory* als Parameter übergeben, allerdings nicht wie andere Objekte durch den *Rete*-Algorithmus ausgewertet.
- Export der Regeln selbst. Der Export einer Regel besteht aus Namen, optionalen Attributen, dem Bedingungs- und dem Ausführungsteil.

In allgemeiner Form sieht der Export eines *RuleSet* demnach folgendermaßen aus:

```
<header des RuleSet>  
expander jsr94.dsl #nur wenn eine DSL für das RuleSet definiert ist  
<imports der Rules>  
global org.apache.log4j.Logger log  
<Name der ersten Regel>  
<optionale Attribute der ersten Regel>
```

2 Hauptteil

```
when
<Bedingungsteil der ersten Regel>
then
<Ausführungsteil der ersten Regel>
end
```

Nachdem das gesamte *RuleSet* in einen String exportiert wurde, wird ein Objekt der Klasse *java.io.StringReader* daraus erzeugt, die dann zum Kompilieren des *Rete*-Netzes verwendet wird.

Laden eines Regel-Sets Sobald das in hybris gespeicherte *RuleSet*-Objekt serialisiert und der *Reader* erzeugt wurde, kann die Regel geladen werden.

Zunächst muss die Rule Engine selbst initialisiert werden. Dies geschieht beim ersten Aufruf des *RulesManager* automatisch. Gemäß dem Singleton-Pattern[20] existiert jeweils nur eine Instanz der Manager-Klasse in jeder Java-Laufzeitumgebung (also im Regelfall einmal pro hybris-Server). Die statische Methode *getInstance()* in der Klasse *RulesManager* erzeugt (beim ersten Aufruf) diese Instanz und gibt sie zurück. Beim ersten Aufruf (bzw. wenn die entsprechenden Objekte noch nicht erzeugt wurden) wird die Rule Engine initialisiert.

```
private void initializeRuleEngine()
{
    log.info("Initializing RuleEngine ...");
    try
    {
        Class.forName( "org.drools.jsr94.rules.RuleServiceProviderImpl" );
        serviceProvider =
            RuleServiceProviderManager.getRuleServiceProvider(
                "http://drools.org/" );
        ruleAdministrator = serviceProvider.getRuleAdministrator( );
    }
    catch (Exception e)
    {
        log.error("Error initializing RuleEngine", e);
    }
}
```

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
}
```

Listing 2.18: Initialisierung der Rule Engine

Ähnlich wie bei JDBC-Treibern wird beim Laden der Klasse *org.drools.jsr94.rules.RuleServiceProviderImpl* der *RuleServiceProviderManager*, die zentrale Klasse zum Verwalten der *RuleServiceProvider*, automatisch registriert. Die Instanzen des verwendeten *RuleServiceProvider* und *RuleAdministrator* werden als Klassenvariable in der Instanz der *RulesManager*-Klasse gespeichert.

Sobald die Rule Engine selbst derart initialisiert wurde, können *Rule Execution Sets* ausgeführt werden. Dabei werden die *Rule Execution Sets* nur beim ersten Mal geladen und das *Rete*-Netz erzeugt. Bei weiteren Ausführungen des selben *Rule Execution Sets* liegt das *Rete*-Netz bereits im Speicher der Rule Engine vor. Der *RulesManager* verwaltet dabei, welche *Rule Execution Sets* bereits geladen wurden. Dazu existiert eine Wrapper Klasse für die von hybris erzeugte *RuleSet*-Klasse namens *CachedRuleSet*, die für das Laden des *RuleSets* selbst zuständig ist. Darüber hinaus ist diese Wrapper-Klasse auch für die Verarbeitung von Änderungen an *RuleSets* zur Laufzeit der hybris Plattform verantwortlich.

Die bereits geladenen *Rule Execution Sets* werden in Form der *CachedRuleSet*-Instanzen in einer *Map*⁵⁵ gespeichert. Sobald ein Regel-Set ausgeführt werden soll, wird zunächst geprüft, ob eine entsprechende *CachedRuleSet*-Instanz bereits in der *Map* vorliegt. Ist dies der Fall, so kann das Regel-Set ausgeführt werden.

Ist eine solche *CachedRuleSet*-Instanz allerdings noch nicht in der *Map* vorhanden, so wurde das *Rule Execution Set* noch nicht geladen und ausgeführt. Das heißt, das *RuleSet* muss zunächst serialisiert und in den Speicher der Rule Engine geladen werden. Dazu verfügt die Klasse *CachedRuleSet* über die Methode *loadRuleset*. Mit Hilfe des *RuleAdministrator* wird dort aus dem serialisierten *RuleSet*-Objekt ein *Rule Execution Set* erzeugt und bei der Rule Engine registriert. Danach kann das *Rule Execution Set* ausgeführt werden.

```
DSL dsl = ruleset.getDsl();  
Map<String, String> parameters = new HashMap<String, String>();  
if (dsl!=null)  
{  
    parameters.put( "source", "drl" );
```

⁵⁵Java-Klasse `java.util.HashMap`

2 Hauptteil

```
parameters.put("dsl", dsl.getMapping());
}

Reader rulesetReader = ruleset.getReader();
RuleExecutionSet res =
    ruleAdministrator.getLocalRuleExecutionSetProvider( null
        ).createRuleExecutionSet( rulesetReader, parameters );
rulesetReader.close( );

ruleAdministrator.registerRuleExecutionSet( ruleset.getName(), res, null);
```

Listing 2.19: Laden eines RuleSet

Ausführung eines Regel-Sets Zur Ausführung von Regeln bietet die *JSR 94* API zwei verschiedene Möglichkeiten:

stateless session Die Regeln werden einmalig, mit einem initialen Satz von Parameter-Objekten, ausgeführt. Es können während der Ausführung keine neuen Objekte dynamisch hinzugefügt werden. Sobald die Auswertung der Regeln mit den initialen Objekten keine weiteren Aktivierungen mehr erzeugt (d.h. alle Regeln sind überprüft und ausgeführt sofern qualifiziert) ist die Auswertung beendet. Als Ergebnis der Auswertung werden alle Objekte (bei Bedarf durch einen *ObjectFilter* gefiltert), die sich am Ende der Auswertung im *Working Memory* befinden haben, zurückgegeben. Diese Ausführungsart entspricht prinzipiell nicht dem eigentlichen „Wesen“ des *Rete*-Algorithmus, der seine Fähigkeiten und Leistung in erster Linie bei langfristigen Auswertungen mit dynamisch sich verändernden Working Memories ausspielen kann. Dennoch ist diese Möglichkeit bei einigen Anwendungsfällen durchaus nützlich und wird im Rahmen dieser Arbeit zum Beispiel zur Anzeige der Promotion-Teaser verwendet.

statefull session Den Gegensatz dazu stellen *statefull sessions* dar. Dies sind langlebige Auswertungssitzungen der Rule Engine, die nicht bereits nach einer Auswertung beendet werden. Stattdessen läuft die Sitzung so lange, bis sie explizit geschlossen wird. Zur Laufzeit können dynamisch Objekte zum *Working Memory* hinzugefügt, daraus entfernt oder auch verändert werden, was sich dann auf die Auswertung der Regeln auswirkt. Da die Auswertung im Endeffekt keine wirklichen Ergebnisse im Sinne eines Rückgabewertes der Auswertungsmethode liefert, ist

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

es jederzeit möglich sich einen „Auszug“ aus dem *Working Memory* der aktuellen Sitzung zu holen. Analog zu den *stateless sessions* kann auch hier ein *ObjectFilter* verwendet werden. Diese Art der Auswertung entspricht exakt dem Verwendungszweck des *Rete*-Algorithmus, so dass hier dessen Leistungsfähigkeit besonders zum Tragen kommt. In der vorliegenden Arbeit werden *statefull sessions* zur Berechnung von Promotions bzw. deren Auswirkungen auf den Warenkorb bzw. die Bestellung verwendet.

Die verwendete *JSR 94*-kompatible Implementierung der *JBoss Rules* unterstützt nativ lediglich *statefull sessions*, da der *Rete*-Algorithmus nur für diesen Verwendungszweck entwickelt wurde und Business Rules allgemein nicht für einmalige Auswertungen entwickelt werden. *Stateless sessions* können dennoch verwendet werden, da sie im Prinzip nichts anderes darstellen als *statefull sessions*, die nach der ersten Auswertung abgebrochen werden.

Ausführung von Stateless Sessions Zur Ausführung von Regeln als *stateless sessions* existiert im *RulesManager* die *executeRules*-Methode. Als Parameter benötigt sie lediglich den Namen des *RuleSet*, welches ausgeführt wird, eine Liste von Objekten auf denen die Regeln ausgewertet werden und einen (optionalen) *ObjectFilter*, um das Ergebnis in Form einer Objektliste einzuschränken.

```
RuleRuntime ruleRuntime = serviceProvider.getRuleRuntime();
Map parameters = new HashMap();
parameters.put("log", Logger.getLogger(ruleset));
objects.add(JaloSession.getCurrentSession());
StatelessRuleSession statelessRuleSession = (StatelessRuleSession)
    ruleRuntime.createRuleSession( ruleset, parameters,
    RuleRuntime.STATELESS_SESSION_TYPE );
return (filter==null) ? statelessRuleSession.executeRules(objects) :
    statelessRuleSession.executeRules(objects, filter);
```

Listing 2.20: Methode *executeRules* im *RulesManager* (Auszug)

Zunächst wird vom *ServiceProvider* die Laufzeitumgebung *RuleRuntime* für die Business Rules angefordert. Danach wird eine *Map* mit optionalen Parametern erzeugt und der *Logger* hinzugefügt. Das ist immer notwendig, da, wie bereits bei der Serialisierung der *RuleSets* erwähnt, in jedem *RuleSet* ein globaler *Logger* angegeben wird. Sobald der *Logger* dort angegeben ist, muss er auch als Parameter übergeben werden. Des Weiteren wird der Objektliste, auf der die Regeln ausgewertet wer-

2 Hauptteil

den sollen (objects) immer die aktuelle *JaloSession* als allgemeiner Zugangspunkt zum hybris System angehängt.

In der im ersten (hier präsentierten) Schritt erhaltenen Laufzeitumgebung für Business Rules wird dann eine Auswertungssitzung erzeugt. Als Parameter dient der Name des *RuleSet*, welches bereits vorher bei der Business Rule Engine registriert und kompiliert worden sein muss, die Parameter-Map und der Typ der angeforderten Sitzung als Konstante. In diesem Fall wird hier eine *stateless session* erzeugt. Wurde das jeweilige *RuleSet* nicht bereits vorher ordnungsgemäß geladen, so kann die Sitzung nicht erzeugt werden und eine Exception wird geworfen. Sobald die Sitzung erfolgreich erzeugt wurde, können die Regeln der Sitzung ausgeführt werden. Je nachdem, ob der optionale *ObjectFilter* angegeben wurde oder nicht, wird das Ergebnis der Auswertung durch diesen Filter gefiltert oder nicht.

Also Resultat der Auswertung wird an die aufrufende Klasse (beispielsweise in der *storefoundation-Frontend-Extension*) eine Liste von Objekten zurückgegeben, gegebenenfalls gefiltert durch den *ObjectFilter*. Einer der Vorteile der einfachen Ausführung von Regeln als *stateless session* ist, dass die aufrufende Klasse bzw. Anwendung überhaupt keine Abhängigkeiten zu den verwendeten Objekten bzw. der *JSR 94 API* hat.

Ausführung von Statefull Sessions Wesentlich komplexer ist die Ausführung bzw. Erzeugung einer stateful

```
session = (StatefulRuleSession)
    JaloSession.getCurrentSession().getAttribute(RULES_SESSION_ATTRIBUTE_PREFIX
        + ruleset);
if (session == null && createIfNecessary)
{
    session = (StatefulRuleSession)
        serviceProvider.getRuleRuntime().createRuleSession(ruleset,
            parameters, RuleRuntime.STATEFUL_SESSION_TYPE);
    JaloSession.getCurrentSession().setAttribute(RULES_SESSION_ATTRIBUTE_PREFIX
        + ruleset, session);
    session.addObjects(objects);
}
```

Listing 2.21: Methode `getStatefulSession` im `RulesManager` (Auszug)

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Um die Sitzung über mehrere Interaktionen und Aufrufe hinweg im Speicher zu behalten und mit dem jeweiligen User verknüpfen zu können, wird die Sitzung unter einem fest definierten Attributnamen in der *JaloSession* (die grundsätzlich nur eine Wrapper-Klasse für die jeweilige *HttpSession* darstellt) gespeichert. So kann zu einem späteren Zeitpunkt die Sitzung mit ihrem aktuellen Zustand weiter verwendet werden. Sobald die Sitzung einmal erzeugt wurde, wird bei jedem folgenden Aufruf die bereits existierende Sitzung aus der *JaloSession* zurückgegeben und nicht neu erzeugt. Da die Auswertung mit einer *statefull session* nicht eine einmalige Ausführung darstellt, ist es hierbei auch möglich, dynamisch Objekte zum *Working Memory* hinzuzufügen, zu entfernen oder zu verändern. Darüber hinaus ist es, wie bereits erwähnt, jederzeit möglich, einen Auszug der Objekte im *Working Memory* zu erhalten, gegebenenfalls gefiltert durch einen *ObjectFilter*. Die Arbeitsweise der *statefull sessions* passt wesentlich besser zur Auswertung durch den *Rete*-Algorithmus, der bei sich dynamisch verändernden Objektmengen seine Leistungsfähigkeit ausspielen kann.

Nachteilig an der Auswertung durch *statefull sessions* ist die höhere Komplexität der Anwendung. Beim Aufruf der *getStatefulSession*-Methode wird lediglich ein Objekt, welches das *JSR 94* Interface *StatefulRuleSession* implementiert, zurückgegeben. Um die weitere Ausführung (also Veränderungen am *Working Memory* oder Extrahieren von Objekten daraus) sowie die Verwaltung der Sitzung muss sich die aufrufende Klasse bzw. Anwendung kümmern. Daraus entstehen auch im aufrufenden Quellcode Abhängigkeiten zur *JSR 94* API, nicht jedoch zur konkret verwendeten Implementierung. Darüber hinaus erzeugen *statefull sessions* einen erhöhten Speicherbedarf. Wenn man davon ausgeht, dass bei einem großen Onlineshop (wie beispielsweise für Virgin Megastores) mit einer großen Anzahl von gleichzeitig aktiven User-Sessions (beispielsweise 1000) jeweils eine zustandsbehaftete Auswertungssitzung pro User-Session angelegt und im Speicher gehalten wird, so kann hier je nach Größe der jeweiligen Working Memories sehr viel Speicher verbraucht werden.

Um eine zustandsbehaftete Auswertungssitzung zu löschen - weil sie nicht mehr benötigt wird oder weil man eine neue Sitzung erzeugen möchte - gibt es die *releaseStatefulSession*-Methode im *Rules-Manager*. Diese holt die angegebene Sitzung aus der aktuellen *JaloSession*, schließt sie und löscht die Referenz in der *JaloSession*. So wird die Sitzung vollständig entfernt. Beim nächsten Aufruf von *getStatefulSession* wird dann, sofern gewünscht, eine neue Sitzung erzeugt.

```
StatefulRuleSession session = (StatefulRuleSession)
    JaloSession.getCurrentSession().getAttribute(RULES_SESSION_ATTRIBUTE_PREFIX
    + ruleset);
```

2 Hauptteil

```
if (session != null)
{
    try
    {
        session.release();
    }
    catch (Exception e)
    {
        log.error("Error releasing Statefull Session " +ruleset);
    }
    JaloSession.getCurrentSession().setAttribute(RULES_SESSION_ATTRIBUTE_PREFIX
        + ruleset, null);
}
```

Listing 2.22: “releaseStatefulSession-Methode im RulesManager“

Ändern der Rules zur Laufzeit

Ein Problem beim Einsatz einer Business Rule Engine innerhalb der hybris Platform ist der getrennte Speicher, den die Rule Engines verwenden. Wie bereits erwähnt werden die Regel-Sets beim Laden durch die Rule Engine kompiliert und dann in einem virtuellen Container gespeichert. Dadurch sind die kompilierten Regel-Sets nicht mehr mit den in hybris gespeicherten Objekten „verbunden“. Wird an einem Regel-Set innerhalb der hybris Platform zur Laufzeit etwas geändert (z.B. werden durch den Sachbearbeiter neue Regeln hinzugefügt), so werden zwar innerhalb der hybris Platform die Caches invalidiert, so dass die Änderung gleich umgesetzt wird. Die im Speicher der Rule Engine befindlichen kompilierten Regel-Sets werden aber über diese Änderung nicht informiert. Beim Ausführen des Regel-Sets ist nur bekannt, dass das Regel-Set bereits geladen ist und deswegen nicht noch serialisiert und geladen werden muss. Sobald ein Regel-Set also verändert wird, kommt diese Änderung nicht zum Tragen, weil die im Speicher der Rule Engine gehaltenen Regel-Sets nicht aktualisiert werden. Erst wenn das System neu gestartet wird und die Regel-Sets neu geladen und kompiliert werden, werden die Änderungen aktiv. Ein Neustart des Servers bei jeder Änderung an einem Regel-Set ist selbstverständlich keine Option.

Aktualisierung durch Überschreiben von Methoden Für dieses Szenario (ein Objekt wird verändert, daraufhin soll eine gesonderte Aktion durchgeführt werden) gibt es eine etablierte Vorgehensweise:

In den von hybris erzeugten Klassen (z.B. *GeneratedRuleSet*), die die Typen abbilden, gibt es für jedes Attribut *setter*-Methoden, die für die Speicherung eines neuen Attributwertes verwendet werden. Diese Methoden werden auch aus der Administrationskonsole *hmc* aus aufgerufen, sobald ein Attribut dort geändert wird.

```
public void setDsl(DSL value)
{
    setDsl( getSession().getSessionContext(), value );
}
```

Listing 2.23: setDsl-Methode in der Klasse GeneratedRuleSet

Um dem Ändern des Attributes eine selbstdefinierte Aktion folgen zu lassen, ist es möglich, die angegebene Methode in der davon erbbenden Klasse (z.B. *RuleSet*) zu überschreiben.

```
@Override
public void setDsl(DSL value)
{
    super.setDsl(value);
    // hier kann beliebiger Code eingefügt werden
}
```

Listing 2.24: überschreibende setDsl-Methode in der Klasse RuleSet

Auf diese Weise kann nach dem Ändern eines Attributes z.B. veranlasst werden, dass das *RuleSet* von der Business Rule Engine neu geladen wird und dann in der „aktualisierten Fassung“ verwendet wird. Wenn jedoch nicht direkt Attribute des Typen *RuleSet*, sondern Attribute von dem *RuleSet* lediglich „angehängten“ Objekten, verändert werden, so wird im *RuleSet*-Objekt keine Methode aufgerufen. Dies ist zum Beispiel bei der Veränderung einzelner Regeln der Fall.

Daher muss auch in allen indirekt beteiligten Typen (also *Rule* und *DSL*) eine spezielle Behandlung der Attributänderungen durchgeführt werden. Hier muss dann zunächst ermittelt werden, welche *RuleSets* von den Änderungen in dem jeweiligen Objekt betroffen sind. Das sind bei Änderungen in einer Regel alle *RuleSets*, in denen die betreffende Regel verwendet wird.

2 Hauptteil

```
for (RuleSet ruleset : (Set<RuleSet>)
    TypeManager.getInstance().getComposedType(RuleSet.class).getAllInstances())
{
    if (ruleset.getRules() != null && ruleset.getRules().contains(this)) {
        //hier kann beliebiger Code eingefügt werden
    }
}
```

Listing 2.25: Codefragment zum Erhalten relevanter Rulesets

Dieses Vorgehen des Überschreibens von Methoden entspricht einem Trigger und ist aus technischer Sicht einfach, elegant und performant. Ursprünglich wurde auch bei der im Rahmen der vorliegenden Arbeit entwickelten *rules*-Extension dieses Vorgehen angewandt.

Leider ist diese Lösung nicht clusterfähig. Sobald die hybris Plattform auf mehreren Servern verteilt läuft, läuft auch auf jedem dieser Rechner eine separate Instanz der Business Rule Engine mit eigenem *Working* und *Production Memory*. Des Weiteren läuft auch auf jedem dieser Rechner die Administrationskonsole. Sobald auf einem dieser Server in der Administrationskonsole ein *RuleSet* (oder ein angehängtes Objekt) geändert wird, wird nur auf diesem einen Server die *setter*-Methode des Attributs aufgerufen. Auf diesem Server wird dann die Rule Engine auch wie vorgesehen aktualisiert und die Regel-Sets sind entsprechend auf dem aktuellen Stand. Die hybris Plattform kümmert sich auch darum, dass auf allen anderen Servern des Clusters die geänderten Objekte aus den Caches entfernt werden und gegebenenfalls neu aus der Datenbank, die ja die aktualisierten Fassungen enthält, geladen werden. Jedoch werden auf allen anderen Servern die *setter*-Methoden nicht aufgerufen, was bedeutet, dass die Rule Engine nicht von den Änderungen „unterrichtet“ wird und weiterhin mit den veralteten Regeln läuft, sofern diese bereits vor dem Änderungszeitpunkt geladen waren. Das bedeutet nicht nur, dass Regel-Sets teilweise nicht mehr zur Laufzeit aktualisiert werden, sondern auch, dass auf verschiedenen Servern unterschiedliche Versionen der Regel-Sets aktiv sein können, was erhebliche Probleme, die nur schwierig rekonstruierbar und nachvollziehbar sein dürften, verursachen könnte.

Aktualisierung durch Abfangen der Cache-Invalidierung Wie bereits erwähnt erfolgt die Cache-Invalidierung in geclusterten Serverumgebungen durch UDP-Multicasts. Dabei werden UDP-

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Pakete an alle im Cluster enthaltenen Server verschickt, in denen mitgeteilt wird, welches Objekt sich geändert hat. Die jeweiligen Server, die diese Pakete enthalten, entfernen die Objekte dann (sofern vorhanden) aus dem Cache, so dass sie beim nächsten Zugriff auf das Objekt aus der bereits aktualisierten Datenbank angefragt werden müssen. Eine Möglichkeit wäre, an der Stelle wo die Cache-Invalidierung stattfindet zusätzlich die Rule Engine über die Änderung zu informieren, sofern es sich bei dem geänderten Objekt um ein Objekt des Typs *Rule*, *RuleSet* oder *DSL* handelt. Leider ist dieser Teil des Quellcodes der hybris Plattform nicht freigegeben bzw. ausreichend dokumentiert und kann daher auch nicht problemlos verändert/erweitert werden.

Aktualisierung durch Zeitstempel Die Variante, die schließlich im Rahmen dieser Arbeit angewendet wird, verwendet Zeitstempel. Jeder von der Rule Engine verwendete Typ verfügt über das geerbte Attribut *modificationtime*. Sobald ein Attribut im jeweiligen Typen verändert wird, so wird das Attribut *modificationtime* auf die aktuelle Systemzeit gesetzt.

Sobald ein *RuleSet* zum ersten mal geladen wird, wird die aktuelle *modificationtime* des *RuleSet* in der jeweiligen Wrapper Klasse *CachedRuleSet* gespeichert. Bei jeder Ausführung des entsprechenden *Rule Execution Set* muss dann geprüft werden, ob die gespeicherte *modificationtime* gleich dem aktuellen *modificationtime*-Attributwert ist. Wenn dies der Fall ist, so wurde seit dem Laden des *RuleSet* kein Attribut verändert und das *Rule Execution Set* im Speicher der Rule Engine ist noch auf dem aktuellen Stand. Wenn die gespeicherte *modificationtime* vor dem aktuellen Attributwert liegt, so wurde seit dem Laden des *Rule Execution Set* ein Attributwert verändert, und das *Rule Execution Set* muss neu geladen werden. Auf diese Weise werden aber nur Änderungen an Attributwerten des *RuleSet*-Objekts selbst erfasst.

Es gibt zwei Möglichkeiten, um auch Änderungen an den angehängten Objekten vom Typ *DSL* und *Rule* zu erfassen.

Überprüfen aller Objekte Bei der Ermittlung der *modificationtime* von Objekten des Typs *RuleSet* werden auch die angehängten Objekte überprüft, und der letzte Änderungszeitpunkt aller Objekte⁵⁶ verwendet. Dadurch wird immer die letzte Veränderung an jedem verwendeten Objekt berücksichtigt und überprüft. Ist die letzte Änderung nach der im *CachedRuleSet* gespeicherten *modificationtime* erfolgt, so muss das gesamte *RuleSet* (unabhängig davon welches Objekt wirklich

⁵⁶ $\max(\text{modificationtime}(\text{ruleset}), \text{modificationtime}(\text{angehängte Rules}), \text{modificationtime}(\text{DSL wenn vorhanden}))$

2 Hauptteil

geändert wurde) neu geladen werden.

Diese Möglichkeit hat den Vorteil, dass nur für die Abfrage des Attributwertes eigener Code geschrieben werden muss. Dabei kann die Abfrage des maximalen Wertes z.B. als reiner Java-Code geschrieben werden.

```
public Date getLastModificationtime()
{
    Date result = this.getModificationTime();

    for (Rule rule : (Collection<Rule>) getRules())
    {
        if (result.before(rule.getModificationTime())) result =
            rule.getModificationTime();
    }

    if (this.getDsl() != null &&
        this.getDsl().getModificationTime().after(result)) result =
        this.getDsl().getModificationTime();

    return result;
}
```

Listing 2.26: Abfrage der modificationtime mittels API

Hierbei wird von jedem relevanten Objekt die *modificationtime* ausgelesen und der größte (also letzte) Zeitstempel zurückgeliefert. Von der Leistungsfähigkeit her betrachtet ist diese Methode nachteilig, weil pro Ausführung eines Regel-Sets sehr viele *modificationtime*-Attribute ausgelesen werden müssen (bei einem Regel-Set mit 20 Regeln und *DSL* genau 22). Selbst wenn man davon ausgeht, dass die Hälfte der angefragten Attributwerte im Cache gehalten sind (die Größe des Caches ist natürlich begrenzt), müssen immer noch bis zu 10 Attributanfragen direkt auf der Datenbank ausgeführt werden, was - obwohl die Anfragen selbst relativ simpel sind - sehr „teuer“ ist.⁵⁷

Alternativ kann die gesamte Abfrage auch nicht auf den Java-Objekten, sondern mittels der Flexible-Search direkt auf der Datenbank ausgeführt werden.

```
SELECT max(modifiedTS) FROM
```

⁵⁷also viel Rechenleistung benötigt

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
{
  SELECT ruleset.modifiedTS
  FROM ruleset AS ruleset
  WHERE name=?name
  UNION
  SELECT rule.modifiedTS
  FROM rule AS rule, ruleset AS ruleset
  WHERE ruleset.rules LIKE rule.pk
  AND ruleset.name=?name
  UNION
  SELECT dsl.modifiedTS
  FROM dsl AS dsl, ruleset AS ruleset
  WHERE ruleset.dsl=dsl.pk
  AND ruleset.name=?name
}
```

Listing 2.27: Flexible-Search Anfrage für letzten Zeitstempel

Im Gegensatz zur Verwendung der jeweiligen Methoden wird hier der Cache der hybris Plattform nicht verwendet. Es wird auf Datenbank-Ebene eine (relativ komplexe) Anfrage durchgeführt. Diese Suchanfrage benötigt als Parameter lediglich den Namen des *RuleSet*, welches überprüft werden soll. Sofern man davon ausgeht, dass Regel-Sets wesentlich häufiger ausgeführt als geändert werden, so sind diese beiden Methoden, Änderungen an den relevanten Objekten zu erfassen, nicht optimal. Zwar erfüllen beide ihren Zweck, es erfolgen jedoch bei beiden Möglichkeiten bei jeder Ausführung des Regel-Sets eine komplexe bzw. viele einfache Anfragen auf die Datenbank.

Aktualisieren bei Veränderung Um dieses Problem zu beseitigen, wird ein anderes Verfahren angewendet. Ähnlich wie die zuvor vorgestellte, nicht clusterfähige Methode des Überschreibens der *setter*-Methoden der relevanten Typen werden auch hierbei die *setter*-Methoden überschrieben. Anders als bei der bereits vorgestellten Vorgehensweise wird aber beim Ändern eines Attributwerts nicht das *RuleSet* neu geladen, sondern die *modificationtime* der zugehörigen *RuleSet*-Objekte aktualisiert. Das bedeutet, dass sobald in einer Regel oder *DSL* ein Attributwert verändert wird auch die *modificationtime* der *RuleSets*, die die jeweilige Regel oder *DSL* verwenden, aktualisiert wird. Bei der Ausführung eines *Rule Execution Set* muss dann nur noch die *modificationtime* des *RuleSet*-

2 Hauptteil

Objekts mit der im *CachedRuleSet* gespeicherten *modificationtime* verglichen werden. Es wird also pro Ausführung des *Rule Execution Set* nur ein Attributwert abgefragt, der sich zu einer hohen Wahrscheinlichkeit auch im Cache befindet. Selbst wenn der Attributwert nicht im Cache vorliegt, ist nur eine einfache Datenbankabfrage nötig.

Im Gegensatz zu den zuvor präsentierten Möglichkeiten müssen also wesentlich weniger bzw. einfachere Anfragen durchgeführt werden. Nachteilig an dieser Methode ist allerdings, dass die Code-Komplexität höher ist, da auch hierbei nicht nur der Vergleich der *modificationtime*, sondern auch die *setter*-Methoden in den Typen *Rule* und *DSL* überschrieben werden müssen. Dennoch stellt diese Methode eine performante und clusterfähige Lösung dar und kommt auch in der vorliegenden Arbeit zur Anwendung.

```
public void setLhs(String lhs)
{
    super.setLhs(lhs);
    touchReferencingRulesets();
}

...

private void touchReferencingRulesets() {
    for (RuleSet ruleset : (Set<RuleSet>)
        TypeManager.getInstance().getComposedType(RuleSet.class).getAllInstances())
    {
        if (ruleset.getRules() != null &&
            ruleset.getRules().contains(this))
        {
            ruleset.setModificationTime(new Date());
        }
    }
}
```

Listing 2.28: Verwendete Lösung zur Aktualisierung der Timestamps

2.3.3 Abbildung von Promotions

Im vorigen Kapitel wurde erläutert, wie die Business Rule Engine in die hybris Plattform eingebunden wird. Damit ist die Grundlage geschaffen Business Rules abhängig vom konkreten Anwendungsfall zu verwenden. In der vorliegenden Arbeit soll demonstriert werden, wie diese Business Rules zur Abbildung von Promotions, als Alternative zu der bei Virgin Megastores verwendeten Implementierung, verwendet werden können. Dabei werden dieselben Anforderungen vorgegeben wie für die projektspezifische Implementierung. Für jeden konkreten Anwendungsfall besteht die Verwendung der Promotions aus zwei Bestandteilen:

- Die Business Rules selbst. Es müssen Regel-Sets mit entsprechenden Regeln angelegt werden, die die Business Logik bzw. die Geschäftsregeln abbilden. Dies geschieht zur Laufzeit und soll (im Idealfall) von Sachbearbeitern mit lediglich grundlegenden Informatik-Kenntnissen durchgeführt werden.
- Die Einbindung in die Anwendung. Innerhalb der Anwendung, beispielsweise im User-Frontend *storefoundation*, muss die Business Rule Engine aufgerufen bzw. verwendet werden. Das geschieht durch Einbindung von Funktionsaufrufen etc. in den Quellcode, kann daher nur zur Kompilierzeit der Anwendung erfolgen und muss von einem Programmierer durchgeführt werden. Auch die Darstellung bzw. Verarbeitung eventueller Ergebnisse muss im Programmcode selbst durchgeführt werden.

Teaser

Zu den aus dem Virgin Megastores Projekt bekannten Anforderungen gehört, vorhandene Promotions auch entsprechend zu bewerben. Das geschieht vor allem auf Produktebene für produktabhängige Promotions (z.B. „Buy 1 get 1 free“). Andererseits ist es auch denkbar, dass Promotions auf Kategorieseiten oder im Warenkorb beworben werden. Zum Bewerben der jeweiligen Promotion sollen sowohl ein Text als auch ein Bild (im hybris Typsystem entsprechend ein Objekt vom Typ *media*) angezeigt werden. Dabei soll die Art, wie der Text bzw. das Bild dargestellt wird, nicht von Business Rules gesteuert, sondern fest in der Anwendung definiert werden. Die Business Rules sollen lediglich entscheiden, ob und welcher Text bzw. Bild dargestellt werden.

Zur Auswertung der Teaser-Regeln wird eine *stateless session*, also eine nicht zustandsbehaftete Aus-

2 Hauptteil

wertungssitzung, verwendet, obwohl dies eigentlich nicht der Natur von Business Rules (bzw. dem verwendeten *Rete*-Algorithmus) entspricht. Dennoch eignet sich eine zustandslose Sitzung für diesen Anwendungsfall. Die Anzeige von Promotion-Bannern oder Text ist ein kurzfristiger Prozess, bei dem die Objektmenge, auf der die Auswertung stattfindet, dynamisch nicht stark verändert wird. Darüber hinaus müssen die Ergebnisse der Auswertung bzw. das *Working Memory* der Rule Engine nicht länger, als für die aktuelle Transaktion, vorliegen. Am Schluss der Auswertung liegt einfach ein Media bzw. Text zur Anzeige vor, oder nicht.

Integration in das Frontend Wie bereits erwähnt verwendet die Frontend-Extension *storefoundation*, die den eigentlichen Shop für die User darstellt, das komponentenbasierte Web-Framework JavaServerFaces (JSF). Mit Hilfe dieses Frameworks ist es möglich, die Darstellungsebene, definiert durch HTML-Code, von der Daten- und Business-Logik-Schicht gemäß dem Model-View-Controller-Konzept zu trennen. Insbesondere soll vermieden werden, dass Java-Quellcode direkt in die HTML-Seiten (wie bspw. bei herkömmlichen JSP-Seiten üblich) integriert wird. Dazu werden so genannte Backing-Beans verwendet, die mit den HTML-Seiten verknüpft werden und den eigentlichen Quellcode enthalten.

Integration im Quellcode Für die Integration der Business Rules bzw. der Promotion-Funktionalitäten und deren Darstellung wird ein eigenes Backing Bean namens PromotionJSFBean bereitgestellt. Diese wird im *request scope* abgelegt, was bedeutet, dass sie keinen request-übergreifenden Zustand besitzt. Dieser wird für die Darstellung von Teasern auch nicht benötigt, da wie bereits erläutert die Auswertung von Teasern für Promotions keine langfristig gespeicherten Zustände benötigt.

```
<!-- Rule-Promotion Bean -->
<managed-bean>
  <managed-bean-name>promotionBean</managed-bean-name>
  <managed-bean-class>ystorefoundationpackage.faces.model.beans.promotion.PromotionJSFBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Listing 2.29: Konfiguration der PromotionBean in der Datei faces-config.xml

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Nachdem die Backing Bean derart in das Frontend eingebunden wurde, steht die Bean ab sofort auf allen html-Seiten zur Verfügung. Das bedeutet, dass ab diesem Zeitpunkt mittels der Expression Language von JSF (mit Hilfe der JSF-Erweiterung facelets) von den html-Seiten auf die Methoden der Promotion-Bean zugegriffen werden kann.

Beispiel:

Der EL-Ausdruck: `#promotionBean.promotionTeaserText`

ruft die Methode `getPromotionTeaserText()` in der Promotion-Bean auf.

Nachdem die Bean im *request scope* definiert ist, muss beim ersten Aufruf der Bean diese zunächst instantiiert werden und im jeweiligen *FacesContext* gespeichert werden. Dies wird von JSF automatisch und für den Programmierer transparent übernommen.

Da zum Bewerben der Promotion gemäß den Anforderungen sowohl ein Objekt des Typs *Media*, als auch ein einfacher Text angezeigt werden sollen, werden diese beiden Möglichkeiten für das Frontend realisiert. Dazu werden neue Methoden in das Promotion-Bean aufgenommen:

```
private Object getPromotionTeaser(Class c)
{
    List objects = new ArrayList();
    objects.add(ProductJSFBean.getInstance().getProduct());
    objects.add(CategoryJSFBean.getInstance().getCategory());
    objects.add(CartJSFBean.getInstance().getCart());

    List result = RulesManager.getInstance().executeRules(RULESET_TEASER,
        objects, new RulesObjectFilter(c));
    return (result != null && result.size() > 0) ? result.get(0) : null;
}

public Object getPromotionTeaserText()
{
    return getPromotionTeaser(String.class);
}

public Object getPromotionTeaserMedia()
{
    return getPromotionTeaser(Media.class);
}
```

2 Hauptteil

}

Listing 2.30: Methoden für das Anteasern durch Text und Media

Die Methoden *getPromotionTeaserText* und *getPromotionTeaserMedia* rufen beide dieselbe Funktion *getPromotionTeaser* auf. Je nachdem welche der beiden Funktionen aufgerufen wird, wird aber eine andere Klasse an die *getPromotionTeaser*-Methode übergeben. Die übergebene Klasse wird im *RulesObjectFilter* verwendet, so dass nur Objekte der jeweiligen Klasse zurückgegeben werden. Die Auswertung selbst findet im *RulesManager* durch die *executeRules*-Methode, die für die Auswertung in zustandslosen Sitzungen zuständig ist, statt. Gibt die Auswertung des Teaser-Regel-Sets mindestens ein Objekt der entsprechenden Klasse zurück, so wird das erste Objekt als Ergebnis ausgegeben, ansonsten wird *null* zurückgegeben.

Der Auswertung wird eine Reihe von Objekten übergeben. Dabei handelt es sich um das aktuell betrachtete Produkt, die aktuelle Kategorie und den jeweiligen Warenkorb des Benutzers. Produkt oder Warenkorb werden aus den jeweiligen Backing Beans ausgelesen und können, sollte dort im Moment kein Produkt bzw. Kategorie gesetzt sein, auch *null* sein.

Integration in die Seite Auf Ebene der html-Seiten muss lediglich die Teaser-Funktionalität aus dem Promotion-Bean aufgerufen werden. Um Promotion-Teaser auf einer Produktseite anzuzeigen, muss auf der entsprechenden html-Seite der Code aufgerufen werden. Die html-Seite für die Produktdetails heißt *productdetail.xhtml*. Um den erforderlichen Code elegant und wiederverwendbar einzubinden, wird für die Anzeige der Promotion-Teaser ein eigenes Tag entwickelt. So kann lediglich durch Einfügen eines einzelnen Tags die Promotion-Teaser-Funktionalität auf einer beliebigen Seite angezeigt werden.

Um das Tag zu definieren muss zunächst der Code für das Tag geschrieben werden. JSF in Verbindung mit Facelets erlaubt es, ein Tag als separate.xhtml-Datei zu definieren, die dann ähnlich wie *server-side-include*s in andere Seiten eingebunden werden können. Es können auch beliebige Attribute als Parameter verwendet werden, was für das Promotion-Teaser-Tag nicht notwendig ist.

```
<div id="promotionTeaser"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:c="http://java.sun.com/jstl/core"
    xmlns:hy="http://hybris.com/jsf">
```

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
Rules DATA
<br/>
<c:set var="promoMedia"
      value="#{promotionBean.promotionTeaserMedia}"/>
<c:set var="promoText" value="#{promotionBean.promotionTeaserText}"/>
<c:choose>
  <c:when test="#{!empty promoMedia or !empty promoText}">
    #{promoText}
    <hy:graphicImage value="#{promoMedia.URL}" rendered="#{!empty
      promoMedia}"/>
  </c:when>
  <c:otherwise>
    No rules promotion teaser found!
  </c:otherwise>
</c:choose>
</div>
```

Listing 2.31: promotionTeaserTag.xhtml

Das Tag definiert im Großen und Ganzen ein *div*-Tag, in dem die entsprechenden Namespaces angegeben werden. Danach wird (für die Übersichtlichkeit während der Tests) statisch der Text „Rules DATA“ ausgegeben, um zu kennzeichnen, dass im Folgenden die von den Business Rules ermittelten Teaser angezeigt werden. Danach erfolgt die Hauptarbeit des Tags, nämlich die Auswertung des Teaser-Regel-Sets. Dazu werden mittels des JSTL-Core-Tags `set` im Promotion-Bean die Methoden `getPromotionTeaserMedia` und `getPromotionTeaserText` aufgerufen und in den Variablen `promoMedia` bzw. `promoText` gespeichert. Die eigentliche Auswertung der Business Rules findet dann im Promotion-Bean statt. Nachdem beim `set`-Tag kein *scope*⁵⁸ explizit angegeben ist, wird standardmäßig *page scope* verwendet. Das bedeutet, dass die Variablen nur auf der jeweiligen Seite gültig sind, was vollkommen ausreichend ist und nicht unnötig Speicher verbraucht. Danach ist die Auswertung der Teaser abgeschlossen und die Arbeit der Business Rules beendet. Die restliche Arbeit des definierten Promotion-Teaser-Tags besteht darin, die Ergebnisse anzuzeigen. Dabei soll, falls weder ein Teaser-

⁵⁸dt. Gültigkeitsbereich

2 Hauptteil

Text noch ein Teaser-Media gefunden wurde der Text „No rules promotion teaser found!“ ausgegeben werden. Falls ein Teaser (egal ob Text oder *Media*) gefunden wurde, so soll der Text (bei leerem Text wird nichts angezeigt) und das Bild angezeigt werden. Um das Bild vom Typ *Media* anzuzeigen wird ein in der *hybris storefront*-Extension definiertes Tag namens *graphicImage* verwendet, was speziell für die Darstellung von *hybris Medias* entwickelt wurde.

Nachdem das Tag selbst definiert wurde, muss es lediglich noch in eine Taglib eingebunden werden. Dazu könnte eine eigene Taglib entwickelt werden, die lediglich Tags für die Promotion- bzw Business Rule-Funktionalität enthält. Da jedoch für die Anwendung der Teaser lediglich ein Tag verwendet wird, wird dieses in die in der *storefoundation*-Extension definierte *hybris*-Taglib aufgenommen.

```
<!-- Show the promotion teaser -->
<tag>
  <tag-name>promotionTeaser</tag-name>
  <source>../../tags/promotionTeaserTag.xhtml</source>
</tag>
```

Listing 2.32: “Ausschnitt aus *jsf-hybris.taglib.xml*“

Dazu muss lediglich in der xml-Konfigurationsdatei *jsf-hybris.taglib.xml* das Tag aufgenommen werden. Es muss lediglich ein Name für das neu aufzunehmende Tag angegeben werden und eine Datei, in der das Tag definiert wurde. Im vorliegenden Fall wird der Tag-Name als „promotionTeaser“ definiert. Als Datei wird die zuvor vorgestellte *promotionTeaserTag.xhtml* angegeben.

Nun kann das neu entwickelte Promotion-Teaser-Tag auf jeder *xhtml*-Seite, die Zugriff auf die *hybris*-Taglib hat, verwendet werden.

```
<hy:promotionTeaser/>
```

Listing 2.33: “Einbindung des Promotion-Teaser-Tags in die Seite

Für die vorliegende Arbeit wird das Tag lediglich auf der Produktdetailsseite verwendet. Für den produktiven Einsatz ist es aber ohne Anpassung am Tag selbst ebenfalls auf beliebigen anderen Seiten einsetzbar. In diesem Fall müssen lediglich neue Rules definiert werden, die dann entsprechende Promotions bewerben. Dabei ist zu bedenken, dass für die vorliegende Arbeit lediglich das aktuelle Produkt, die aktuelle Kategorie und der aktuelle Warenkorb zur Auswertung an die Rule Engine übergeben werden. Werden für andere Anwendungsfälle auch andere Objekte benötigt (beispielsweise der aktuelle Benutzer, so müssen auch diese in das *Working Memory* der Rule Engine aufgenommen

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

werden. Dazu kann der Quellcode der PromotionJSFBean angepasst werden, so dass, analog zu den drei bekannten Objekten, noch weitere übergeben werden. Dabei ist allerdings wie bereits erwähnt eine Anpassung des Quellcodes und der Neustart der hybris Plattform notwendig. Alternativ kann man auch über eigene Production-Rules neue Objekte dem *Working Memory* hinzufügen.

Modellierung der Rules Die Rules für die Teaser-Funktionalität werden zu einem *RuleSet* zusammengefasst. Dieser *RuleSet* hat den Code „promotionteaser“ und wird auch unter diesem Code aus der PromotionJSFBean aufgerufen. Da die Beschreibung nur für den Nutzer bzw. die Mitarbeiter, die die Promotions pflegen, interessant ist, ist der Inhalt der Beschreibung beliebig, sollte jedoch über den Verwendungszweck des *RuleSet*-Objekt Aufschluss geben.

Im Header-Attribut des *RuleSets* ist lediglich die Angabe eines packages Pflicht. Alle weiteren Angaben wie beispielsweise die Definition von *Functions* etc. ist optional bzw. nur bei Verwendung von *Functions* notwendig. Das Package kann prinzipiell beliebig sein, muss jedoch systemweit eindeutig sein. Für die vorliegende Arbeit werden als packages immer *com.drools* gefolgt von dem jeweiligen Verwendungszweck verwendet, für das Teaser-*RuleSet* demnach *com.drools.teaser*.

Für das Teaser-*RuleSet* wird auch eine *Domain Specific Language* verwendet. Dabei gibt es generell zwei Alternativen:

- Es wird ein *DSL*-Objekt für alle Promotion-*RuleSets* verwendet. Diese Alternative hat den Vorteil, dass alle Ausdrücke nur in einem einzelnen Objekt gepflegt werden und keine Redundanzen entstehen. Nachteilig ist, dass das Mapping des einzelnen *DSL*-Objekts sehr umfangreich und unübersichtlich werden kann.
- Es wird ein eigenes *DSL*-Objekt für jedes *RuleSet* separat angelegt. Dadurch kann es zu Redundanzen kommen, wenn in zwei verschiedenen *DSL*-Objekten dieselben Mappings eingetragen werden. Dies kann insbesondere bei einigen „Standard“-Mappings der Fall sein wie „oder=or“ etc. Andererseits wird die Übersichtlichkeit gesteigert, da für jedes *RuleSet* nur die für das jeweilige *RuleSet* selber relevanten Mappings aufgenommen und gepflegt werden müssen.

Für die vorliegende Arbeit wurde entschieden die zweite Variante mit separaten *DSL*-Objekten für jedes *RuleSet* zu verwenden. Die Redundanzen halten sich in Grenzen, da die jeweiligen *RuleSets* der Promotion-Funktionalität relativ strikt getrennt sind und weil nur wenige „Standard“-Mappings verwendet werden. Dafür wird der Umfang der *DSL* auf mehrere Objekte verteilt, was die

2 Hauptteil

Übersichtlichkeit bei der Arbeit mit *DSL*-Objekten enorm steigert. Für das Teaser-*RuleSet* wird ein *DSL*-Objekt mit dem Namen „teaserDSL“ angelegt.

Nun können beliebige Rules für unterschiedliche zu bewerbende Promotions angelegt werden. Wichtig ist hierbei, dass die Teaser vollständig unabhängig von den eigentlichen Promotions sind. Es handelt sich hierbei um getrennte *RuleSets*, die keine Objekte teilen oder in irgendeiner Weise verknüpft sind. Das bedeutet, es ist Aufgabe des Sachbearbeiters, dafür zu sorgen, dass die tatsächlichen Promotions auch in Einklang mit dem Bewerben derselben stehen. Wird z.B. die Promotion selbst gelöscht, so muss auch dafür gesorgt werden, dass aus dem Teaser-*RuleSet* die entsprechenden Regeln entfernt bzw. modifiziert werden, da sonst die Promotion weiterhin beworben wird, obwohl sie nicht mehr existiert.

Buy 2 Get 1 Free Promotion-Teaser Beispiel Zum Bewerben einer „Buy 2 Get 1 Free“-Promotion für ein bestimmtes Produkt wird eine Regel wie in Tabelle 2.3 dargestellt angelegt.

Attribut	Wert
Name	Buy2Get1FreePromotionTeaserRule
Beschreibung	Bewirbt die Buy 2 Get 1 Free Promotion für das Produkt mit dem Code „HW1100-0025“
Imports	de.hybris.jakarta.jalo.product.Product de.hybris.jakarta.jalo.media.MediaManager
opt Attribute	-
LHS	Product(code == 'HW1100-0025')
RHS	assert("Buy 2 Get 1 Free - Beim Kauf von zwei Exemplaren bekommen Sie das dritte geschenkt"); assert(MediaManager.getInstance().getMediasByCode("b2g1fpromopic1").iterator().next());

Tabelle 2.3: Buy 2 Get 1 Free Promotion Teaser

Die Werte der Attribute Name und Beschreibung muss nicht weiter erläutert werden.

Beim *Imports*-Attribut müssen alle Java-Klassen angegeben werden, die innerhalb der Regeln ver-

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

wendet werden. Für die vorliegende Regel sind das die Klassen *Product* und *MediaManager*. Für die vorliegende *Rule* werden keine optionalen Attribute angegeben.

Das eigentlich Interessante an jeder Regel sind jedoch der Bedingungsteil (*LHS*) und der Ausführungsteil (*RHS*). In der vorliegenden Regel besteht der Bedingungsteil lediglich aus einer einzigen Bedingung. Diese Bedingung besagt, dass im *Working Memory* ein Objekt vom Typ *Product* enthalten sein muss. Als zusätzliche Anforderung an dieses Objekt wird gestellt, dass das *code*-Attribut des Objekts den Wert „HW1100-0025“ besitzt. Um den Attributwert zu ermitteln wird gemäß der Java-Bean-Konvention eine entsprechende *getter*-Methode (*getCode()*) aufgerufen. Im daraus kompilierten *Rete*-Netz wird ein *ObjectTypeNode* erzeugt, der nur *Product*-Objekte akzeptiert. Diesem Node wird ein weiterer Alpha-Knoten angehängt, der den Attributwert überprüft (*field constraint*). Sobald ein *Product*-Objekt mit passendem *code*-Attributwert das *Rete*-Netzwerk durchläuft wird die Regel aktiviert.

Durch die Aktivierung der Regel wird dann der Ausführungsteil der Regel ausgeführt. In der oben dargestellten Regel besteht der Ausführungsteil aus zwei *assert*-Befehlen. Durch das *assert*-Schlüsselwort werden neue Objekte dem *Working Memory* des aktuell ausgeführten Regel-Sets hinzugefügt. Die derart zum *Working Memory* hinzugefügten Objekte werden dann automatisch durch das *Rete*-Netz weiter verarbeitet. Obwohl das bei der betrachteten Regel im *Teaser-RuleSet* keine Rolle spielt, können auf diese Weise natürlich auch neue Objekte, die von anderen Regeln verarbeitet werden, dem *Working Memory* hinzugefügt werden.

Der Ausführungsteil fügt dem *Working Memory* zwei Objekte, gewissermaßen das Ergebnis der Auswertung der Regel, hinzu:

- Erstens wird ein Objekt vom Typ *String* dem *Working Memory* hinzugefügt. Dieser String ist der Text, der als Werbebotschaft für die beworbene Promotion angezeigt wird.
- Zweitens wird ein *Media*-Objekt dem *Working Memory* hinzugefügt. Dieses *Media* stellt ein Bild dar, was als Teaser für die Promotion angezeigt werden kann. Wichtig ist hierbei, dass das *Media* vor Ausführung der Regel im *hybris* System via der Administrationskonsole hoch geladen werden und mit dem Code „b2g1fpromopic1“ gespeichert werden muss. Andernfalls würde das Fehlen des angefragten *Medias* zu einer Exception bei der Auswertung der Regel führen. Obwohl in der vorliegenden Arbeit immer von einem Bild als *Media* die Rede ist, wäre es auch zulässig statt einem Bild bspw. eine Flash-Animation in dem *Media*-Objekt zu speichern, ohne dass im Frontend oder bei den Rules eine Veränderung nötig wäre.

2 Hauptteil

Diese beiden dem *Working Memory* hinzugefügten Objekte werden durch das *Rete*-Netz ausgewertet, was aber keine weiteren Auswirkungen hat. Stattdessen werden alle Objekte des *Working Memory*, idealerweise gefiltert durch den angegebenen *ObjectFilter*, als Ergebnis der für die Teaser-Funktionalität verwendeten *executeRules*-Methode zurückgegeben. Wird (wie von der *PromotionJSFBean*) das *RuleSet* mit einem *Media-ObjectFilter* bzw. *String-ObjectFilter* ausgeführt, so wird jeweils das *Media*-Objekt oder der String zurückgegeben. Soll im Frontend sowohl der Text als auch das *Media* zum Bewerben der Promotion angezeigt werden, so muss (zumindest bei der gegenwärtigen Implementierung) das Regel-Set zweimal, jeweils mit anderem *ObjectFilter*, ausgeführt werden. Um mehrere Objekte verschiedener Klassen als Ergebnis erhalten zu können, müsste lediglich die vorhandene Implementierung des *RulesObjectFilter* modifiziert werden, so dass dieser nicht mehr nur für Objekte einer einzelnen Klasse, sondern mehrerer verschiedener Klassen „durchlässig“ würde.

Mit dieser einen Regel, die das Anteausern einer Promotion für ein ganz bestimmtes Produkt regelt, ist das „promotionteaser“ *RuleSet* bereits lauffähig. Sobald man auf die Produktdetailseite des entsprechenden Produktes geht, werden dort das unter dem Code „b2g1fpromopic1“ gespeicherte *Media* sowie der Teaser-Text angezeigt.

Erstellung einer DSL Da die in Tabelle 2.3 angeführte Regel in ihrer dargestellten Form für Sachbearbeiter ohne Informatik-Ausbildung zu komplex und wenig aussagekräftig ist, müssen entsprechende Mappings in das „teaserDSL“-Objekt eingefügt werden. Idealerweise sollte eine *DSL* so konzipiert werden, dass sie so viele „technische“ Details wie möglich verbirgt. Das heißt, die Regeln sollen unter Einsatz einer *DSL* nicht mehr wie technischer Programmcode, sondern wie natürliche Sprache aussehen. So wird die Verständlichkeit gefördert und die Erweiterbarkeit durch Sachbearbeiter ermöglicht. Gleichzeitig soll eine *DSL* jedoch so einschränkend wie möglich sein, das heißt, dass auch durch den Einsatz einer *DSL* der Funktionsumfang nicht zu stark gemindert werden sollte.

Um das Mapping für den Bedingungs- und Ausführungsteil zu erstellen, wird zeilenweise vorgegangen. In jeder Zeile müssen die Teile identifiziert werden, die variabel sind. Im vorliegenden Beispiel sind das genau die Teile, die mit ” bzw. „ umrahmt sind. Dies muss aber nicht immer der Fall sein. Sobald die Variablen ermittelt sind, wird ein Konstrukt in natürlicher Sprache gebildet, mit dem die Code-Zeile ausgedrückt werden kann. Die Variablen werden dort an der passenden Stelle eingefügt. Nachdem der von *JBoss Rules* verwendete Parser nicht absolut zuverlässig arbeitet, wird geraten,

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

auch in den natürlichsprachlichen Teilen die Variablen durch Anführungszeichen zu begrenzen. Sobald bei dem *RuleSet*-Objekt eine *DSL* angegeben wird, erwartet *JBoss Rules*, dass jede Zeile der Regel durch die *DSL* übersetzt werden kann. Das heißt, dass entweder für jede Zeile ein entsprechendes *DSL*-Mapping definiert werden muss, oder die entsprechende Zeile durch Voranstellen des Symbols `>` escaped werden muss.

Für die oben angegebene Regel wird ein, in der Tabelle 2.4 präsentiertes, Mapping innerhalb der *DSL* definiert.

DRL	DSL
<code>Product(code == '{code}')</code>	Produkt mit dem Code '{code}'
<code>assert("{text}");</code>	zeige Text '{text}'
<code>assert(MediaManager.getInstance().getMediasByCode("{code}").</code>	zeige Bild '{code}'
<code>iterator().next());</code>	

Tabelle 2.4: DSL-Mapping für Teaser #1

In dem oben angegebenen Mapping sind alle Variablen durch geschweifte Klammern gekennzeichnet. Alles was beim Übersetzen vom *DSL* in die DRL in der eingelesenen Zeichenkette an Stelle der Variable steht, wird der Wert der Variablen. Dieser Wert wird dann an die entsprechende Stelle in der übersetzten DRL gesetzt. Mit der Verwendung dieses Mappings kann die Regel wie in Tabelle 2.5 angegeben werden.

Attribut	Wert
LHS	Produkt mit dem Code 'HW1100-0025'
RHS	zeige Text 'Buy 2 Get 1 Free - Beim Kauf von zwei Exemplaren bekommen Sie das dritte geschenkt'
	zeige Bild 'b2g1fpromopic1'

Tabelle 2.5: Buy 2 Get 1 Free Promotion Teaser mit DSL

2 Hauptteil

Erweiterung Nachdem die Promotion in dieser Version lauffähig ist, können beispielsweise auch weitere Produkt hinzugefügt werden, so dass derselbe Teaser bzw. dieselbe Werbebotschaft bei mehreren Produkten oder auch auf Kategorienseiten angezeigt wird. Um die vorhandene Regel derartig zu erweitern, sind nur geringfügige Änderungen notwendig.

Um weitere Produkte aufzunehmen, reicht es generell, weitere Produkt-Zeilen in den Bedingungsteil aufzunehmen. Dabei dürfen die Produkt-Zeilen, die im kompilierten *Rete*-Netz α -Knoten darstellen, jedoch nicht direkt aufeinander folgen. In der Drools Rule Language (DRL) ist die Standard-Verknüpfung zweier Bedingungen immer die \wedge -Verknüpfung (UND)[7]. Nachdem (zumindest bei dem gegenwärtigen Design des Beispiel-Shops) nur ein Produkt detailliert dargestellt wird, könnte die Regel mit zwei unterschiedlichen Produktbedingungen niemals erfüllt werden. Darum müssen die einzelnen Bedingungen mit \vee (ODER) verknüpft werden.

In DRL müssen dazu die zu verknüpfenden Zeilen mit einem „or“ verbunden werden.

```
Produkt mit dem Code 'HW1100-0025'  
>or  
Produkt mit dem Code 'HW1200'
```

Listing 2.34: Bedingungen mit ODER verknüpft

Nachdem in der Regel eine *DSL* verwendet wird, ist es notwendig, das „or“-Schlüsselwort zu escapen. Um diesen Makel zu vermeiden, ist es sinnvoll, die *DSL* entsprechend zu erweitern. Dazu muss lediglich das Mapping „or=oder“ aufgenommen werden, so dass die Regel wieder komplett in natürlicher Sprache formuliert werden kann.

```
Produkt mit dem Code 'HW1100-0025'  
oder  
Produkt mit dem Code 'HW1200'
```

Listing 2.35: Bedingungen mit ODER verknüpft und angepasster DSL

Spend over x€ and get y Promotion-Teaser Beispiel Ein weiterer Promotion-Teaser, der in der vorliegenden Arbeit exemplarisch dargestellt werden soll, ist ein Teaser für die Promotion „Spend over x€ and get y“. Bei dieser Art von Promotion geht es darum, dass dem Kunden ab einem Bestellwert von x€ eine Vergünstigung in irgendeiner Art eingeräumt wird. Dabei ist es für die Promotion-Teaser, also zum Bewerben der Promotion selbst, unerheblich welcher Art diese

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Vergünstigung ist. Wichtig bei den Teasern ist lediglich, dass ein Media bzw. ein Text angezeigt wird. Deswegen ist bei allen Teaser-Regeln der Ausführungsteil schematisch identisch.

Ein sinnvoller Teaser für die Beispiel-Promotion „Spend over 50 € and get free shipping“ (Bei Einkäufen über 50 € ist der Versand umsonst) könnte beispielsweise darin bestehen, dass ab einem Bestellwert von 20 € die Nachricht erscheint „Kaufen Sie noch für weitere xyz Euro ein, und der Versand ist umsonst“. Dabei sollte immer der Betrag angezeigt werden, der noch zu dem angegebenen Schwellwert fehlt. Ist der Schwellwert überschritten, sollte der Teaser nicht mehr angezeigt werden. Die Regel, die den vorgeschlagenen Teaser abbildet, kann wie in Tabelle 2.6 angegeben werden.

Attribut	Wert
Name	Spend50GetFreeShippingPromotionTeaserRule
Beschreibung	Bewirkt die Spend 50 € and get free shipping Promotion ab 20 €
Imports	de.hybris.jakarta.jalo.product.Product de.hybris.jakarta.jalo.media.MediaManager
opt Attribute	-
LHS	cart : Cart (total > 20.00, total < 50.00)
RHS	assert("Kaufen Sie noch für weitere " + 50.00 - cart.getTotal() + "Euro ein "...); assert(MediaManager.getInstance().getMediasByCode(βspend50promopic1").iterator().next());

Tabelle 2.6: Spend 50 € and get free shipping Promotion Teaser

Im Bedingungsteil der Regel wird der (von der PromotionJSFBean übergebene) Warenkorb betrachtet. Wenn sein Gesamtwert (total) größer als 20 € und kleiner als 50 € ist, so ist die Bedingung für diese Regel erfüllt. Eine Besonderheit an dieser Bedingung ist, dass zusätzlich zu der Auswertung der Bedingung selbst noch eine Variable erzeugt wird. An diese Variable wird das Objekt, das in der jeweiligen Bedingung überprüft wird, gebunden. Diese Variable kann dann später sowohl im Bedingungs- als auch im Ausführungsteil verwendet werden. Im vorliegenden Beispiel wird die Variable im Ausführungsteil verwendet, um dynamisch den Fehlbetrag zum Schwellwert der Promotion zu ermitteln.

2 Hauptteil

Erweiterung der DSL Um die oben genannte Regel in natürlicher Sprache beschreiben zu können, muss die Teaser-*DSL* erweitert werden (siehe Tabelle 2.7). Dabei müssen für die vorliegende Regel nur zwei neue Mappings aufgenommen werden, weil für die zweite Zeile des Ausführungsteils bereits ein passendes Mapping existiert.

DRL	DSL
<code>cart : Cart (total > {value1}, total < {value2})</code>	Warenkorbwert zwischen {value1} und {value2}
<code>assert(("{text1}" + {value}-cart.getTotal() + "{text2}");</code>	zeige Fehlbetragtext '{text1}' '{value}' '{text2}'

Tabelle 2.7: DSL-Mapping für Teaser #2

Mit der erweiterten *DSL* kann die Regel dann wie in Tabelle 2.8 geschrieben werden.

Attribut	Wert
LHS	Warenkorbwert zwischen 20.00 und 50.00
RHS	zeige Fehlbetragtext 'Kaufen Sie noch für weitere' '50.00' ' Euro ein... ' zeige Bild 'spend50promopic1'

Tabelle 2.8: Spend 50 € and get free shipping Promotion Teaser mit DSL

Promotions

Im vorigen Kapitel wurde erläutert, wie Teaser für Promotions mit Hilfe der *rules*-Extension und *stateless session* dargestellt werden können. Nun soll darauf eingegangen werden, wie die Promotions selbst mit Hilfe dieser Extension verarbeitet werden können. Im Gegensatz zu den Promotion-Teasern, wo die Ausführungsteile der Regeln immer nach demselben Muster aufgebaut waren, sind bei den Promotions selber auch die Ausführungsteile der Regeln stark verschieden und spielen eine große Rolle.

Dabei spielen drei Aspekte eine Rolle:

Erfüllung der Voraussetzungen Zunächst muss bei der Auswertung einer Promotion geprüft

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

werden, ob die Bedingungen für diese Promotion erfüllt sind. Diese Auswertung ist generell ähnlich zur Auswertung der Promotion-Teaser, jedoch wesentlich komplexer, worauf später noch ausführlich eingegangen wird.

Anwendung der Promotions Die Anwendung der Promotions unterscheidet sich stark von der der Teaser. Während alle Teaser-Regeln im Endeffekt immer einen Text und ein *Media* zum *Working Memory* hinzufügen, müssen bei der Anwendung der Promotions unterschiedlichste Effekte (Rabatt gewähren, Versandart ändern, Produkte in den Warenkorb aufnehmen etc) möglich sein.

UNDO Es muss die Möglichkeit bestehen, bereits gewährte Promotions wieder rückgängig zu machen. Dies kann beispielsweise der Fall sein, wenn bestimmte Produkte aus dem Warenkorb entfernt werden, so dass bestimmten Promotions die Grundlage entzogen wird.

Erfüllung der Voraussetzungen Wie bereits erwähnt verlaufen die Tests, ob die Voraussetzungen der Promotions erfüllt sind, im Wesentlichen ähnlich zu den bereits zuvor erläuterten Teasern. In den Bedingungssteilen der Promotion-Rules werden einfach die Voraussetzungen für jede einzelne Promotion festgelegt und durch die Rule Engine überprüft. Dennoch ist die Auswertung der Voraussetzungen der Promotions gegenüber den Teaser wesentlich komplexer.

Verbrauchte Produkte Die Herausforderung beim Auswerten der Bedingungen der Promotions besteht in der Wechselwirkung der Promotions untereinander. Vor allem bei produktabhängigen Promotions (wie „Buy 2 get 1 free“ etc) sind solche Wechselwirkungen zu beobachten. Diese Promotions „verbrauchen“ die Produkte, die die Promotion qualifiziert haben, bei der Anwendung. Das bedeutet, dass die Produkte nach der Anwendung nicht mehr für andere Promotions verfügbar sein sollen. Dadurch wird es notwendig, dieses „Verbrauchen“ von Produkten nachvollziehbar zu machen. Eine auf den ersten Blick einfache Lösung wäre die verbrauchten Produkte bei der Berechnung im Warenkorb zu löschen, so dass sie für andere Promotions nicht mehr zur Verfügung stehen. Diese Lösung würde zwar einerseits teilweise die Anforderung erfüllen, jedoch auch Probleme mit sich bringen. Zum einen müssten nach der Berechnung der Promotions die entfernten Produkte dem Warenkorb wieder hinzugefügt werden, weil sie sonst tatsächlich gelöscht wären und der Kunde die Produkte nicht bestellen könnte. Andererseits würden durch das Entfernen der Produkte aus dem Warenkorb andere Eigen-

2 Hauptteil

schaften des Warenkorbs verändert, z.B. der Gesamtwert. Bei produktabhängigen Promotions sollen die Produkte zwar nicht mehr für andere produktabhängige Promotions zur Verfügung stehen, für andere Promotions (z.B. „Keine Versandkosten ab 50 € Bestellwert“) aber sehr wohl. Durch das Entfernen der qualifizierenden Produkte aus dem Warenkorb würde der Gesamtwert der Bestellung evtl. zeitweilig unter den gesetzten Schwellwert fallen, und die entsprechende Promotion nicht angewendet. Nach der Berechnung aller Promotions würden die Produkte wieder dem Warenkorb hinzugefügt, und der Bestellwert wieder über den Schwellwert steigen, ohne die entsprechende Promotions auszulösen.

Aus diesem Grund muss ein neuer Mechanismus zur Verwaltung von „verbrauchten“ Produkten eingeführt werden. Dabei handelt es sich um eine Sicht (view) auf den Warenkorb, die nur Produkte enthält, die noch nicht verbraucht wurden bzw. die verbrauchten Produkte ausblendet. Die Promotion-Rules arbeiten dann lediglich auf dieser Sicht und nicht direkt auf dem Warenkorb. Zu Beginn der Auswertung der Regeln entspricht die Sicht genau dem Inhalt des Warenkorbs, da zu diesem Zeitpunkt noch keine Promotion angewendet und aus diesem Grund auch noch kein Produkt verbraucht wurde. Sobald durch eine Promotion bestimmte Produkte verbraucht werden, müssen diese aus der Sicht entfernt werden und stehen dadurch anderen Promotions nicht mehr zur Verfügung.

Wie bereits erwähnt, existiert ein eigenes package bzw. Verzeichnis in der *rules*-Extension, wo alle speziell für die Promotion-Funktionalität entwickelten Klassen gespeichert werden. Um die Sicht der unverbrauchten Produkte im Warenkorb zu erzeugen, müssen dort neue Klassen erzeugt werden.

Zum einen ist das die Klasse *ProductCartView*. Diese Klasse stellt die eigentliche Sicht auf den Warenkorb dar. Die Klasse stellt eine Tabelle der Produkte mit der jeweiligen Menge der unverbrauchten Produkte im Warenkorb dar.

```
private static class ProductCartView extends HashMap<Product, Long>
{
    public ProductCartView(AbstractOrder order,
        List<RulesPromotionResult> results) {...}
    public boolean hasProducts(Product p, long quantity) {...}
    public void removeProduct(Product product, long quantity) {...}
}
```

Listing 2.36: Outline der Klasse ProductCartView

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Die Klasse erbt von der Klasse *HashMap* und bringt dadurch schon die Basisfunktionalität zum Speichern der Produkte (als Key) und der unverbrauchten Anzahl (als Value) mit. Es muss lediglich ein Konstruktor und die Methoden zur Abfrage bzw. zum „Verbrauchen“ von Produkten hinzugefügt werden. Dem Konstruktor wird der Warenkorb und eine Liste von Promotion-Ergebnissen übergeben. Wie diese Ergebnisse aussehen ist an dieser Stelle nicht relevant und wird in einem späteren Kapitel erläutert. Wichtig ist im Moment nur, dass aus den Promotion-Ergebnissen die „verbrauchten“ Produkte ermittelt werden können. Zusammen mit dem Warenkorb können so die „unverbrauchten“ Produkte ermittelt werden. Durch die *hasProducts*-Methode kann auf einfache Weise ermittelt werden, ob ein bestimmtes Produkt in ausreichender Stückzahl vorhanden ist. Durch die *removeProduct*-Methode kann ein bestimmtes Produkt mit einer bestimmten Anzahl „verbraucht“ werden und wird aus der Sicht entfernt.

Verwaltet wird die Sicht aus einer statischen Klasse, die bei der Auswertung der Promotions direkt aus dem Bedingungsteil der Regeln heraus aufgerufen werden kann. Die Klasse *PromotionChecker* bietet unter anderem die Methode *hasUnboundProducts*, der ein Produktcode (als String) und eine Anzahl übergeben wird.

```
public static boolean hasUnboundProducts(String product, long quantity)
{
    try
    {
        Product p = (Product)
            ProductManager.getInstance().getProductsByCode(product).iterator().next();
        List<RulesPromotionResult> objects =
            getPromotionResultsFromSession(RULESET_PROMOTION);
        if (objects == null) objects = new
            ArrayList<RulesPromotionResult>();
        ProductCartView view = new
            ProductCartView(JaloSession.getCurrentSession().getCart(),
                objects);
        return view.hasProducts(p, quantity);
    }
    catch (Exception e)
    {
        log.error("Error in hasUnboundProducts", e);
    }
}
```

2 Hauptteil

```
        return false;
    }
}
```

Listing 2.37: statische Methode `hasUnboundProducts` in der Klasse `PromotionChecker`

Innerhalb der Methode wird zunächst anhand des übergebenen Codes das Produkt-Objekt aus der Datenbank ermittelt. Danach werden aus dem gerade auszuwertenden *Promotion-Rule Execution Set* die bereits vorhandenen Promotion-Ergebnisse ausgelesen. Mit diesen Ergebnissen und dem aktuellen Warenkorb des Users kann dann die Sicht der unverbrauchten Produkte in Form eines *ProductCart-View*-Objekts erzeugt werden. Bei dieser Sicht kann dann angefragt werden, ob von dem angefragten Produkt die entsprechende Stückzahl vorhanden ist.

Einmalige Anwendung Eine weitere Herausforderung, die bei den Promotion-Teasern nicht auftritt, ist die einmalige bzw. mehrfache Anwendung von Promotions. Es ist möglich, dass mehr als eine Regel dieselbe Promotion auslöst oder verschiedene Promotions denselben Rabatt (der nur einmal gewährt werden soll) bieten. Genauso kann es sein, dass eine Regel mehrfach aktiviert wird, aber nur einmal ausgelöst werden soll. Deshalb kann es notwendig werden zu überprüfen, ob eine bestimmte Promotion bereits aktiviert wurde, bzw. ob schon ein bestimmtes Promotion-Ergebnis vorliegt.

Dafür existiert in der statischen Klasse *PromotionChecker* die Methode *isApplied*.

```
public static boolean isApplied(String ruleset, String rule)
{
    try
    {
        List<RulesPromotionResult> objects =
            getPromotionResultsFromSession(ruleset);
        if (objects == null) return false;
        for (RulesPromotionResult result : objects)
        {
            if (result.getRule().equals(rule)) return true;
        }
    }
    catch (Exception e)
    {
```

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
        log.error("Error in isApplied", e);
    }
    return false;
}
```

Listing 2.38: statische Methode `isApplied` in der Klasse `PromotionChecker`

Der Methode wird der Name des *RuleSet* und der Name der Regel bzw. Promotion übergeben, von der geprüft werden soll, ob sie bereits angewendet wurde. Ähnlich wie bei der Erzeugung der Sicht der unverbrauchten Produkte wird zunächst eine Liste der bereits vorliegenden Promotion-Ergebnisse aus dem *Working Memory* abgerufen. Danach wird geprüft, ob der Name der Regel bzw. Promotion, der beim Methodenaufruf übergeben wurde, bereits in einem der vorliegenden Ergebnisse auftaucht. Wenn dem so ist, gilt die Regel bzw. Promotion als bereits angewendet und als Ergebnis wird *true* zurückgeliefert. Wenn keine Entsprechung bei den vorliegenden Ergebnissen gefunden wird, so ist die Regel bzw. Promotion noch nicht angewendet worden und es wird *false* zurückgeliefert.

Die *isApplied*-Methode kann wie die *hasUnboundProducts*-Methode direkt aus dem Bedingungsteil der Regel aufgerufen werden. Dazu muss einfach eine *eval*-Bedingung nach folgendem Muster hinzugefügt werden:

```
eval(PromotionChecker.isApplied(„Name des RuleSet“, „Name der Promotion“);
```

Anwendung Im Falle des positiven Ausgangs der Prüfung, ob die Bedingungen einer Regel erfüllt sind, wird diese angewendet. Das bedeutet, dass der Ausführungsteil der Regel ausgeführt wird. Dadurch werden die Effekte der Promotion wirksam. Beispielsweise werden Rabatte direkt aus den Ausführungsteilen von Regeln auf den Warenkorb angewendet oder zusätzliche Produkte hinzugefügt etc. Das steht im Gegensatz zu den Promotion-Teasern, wo lediglich neue Objekte dem *Working Memory* hinzugefügt werden und von der „externen“ Anwendung ausgelesen und verwendet. Statt dessen wird bei den Promotions selbst die „externe“ Anwendung durch die Rules beeinflusst, bzw. manipuliert.

Diese Anwendung der Promotions geschieht durch einen zweistufigen Prozess. Zunächst wird für jede Promotion eine eigene Regel definiert. Diese Regel enthält im Bedingungsteil die Anforderungen, die zur Anwendung der Promotion notwendig sind. Im Ausführungsteil wird die jeweilige Auswirkung (also beispielsweise ein Rabatt oder ein zusätzliches Produkt) als Objekt dem *Working Memory* hinzugefügt. Nachdem diese Objekte dem *Working Memory* hinzugefügt wurden, gibt es spezielle Regeln,

2 Hauptteil

die diese Objekte dann auf den Warenkorb etc. anwenden. So wird redundanter Code innerhalb der Regeln vermieden. Ohne diesen zweistufigen Prozess müsste der Code, um bspw. einen Rabatt auf den Warenkorb anzuwenden, in jeder Regel gleich vorhanden sein. Mit Hilfe des zweistufigen Prozesses kann daher die Komplexität der einzelnen Regeln verringert werden.

Die anzuwendenden Objekte werden jedoch nicht direkt dem *Working Memory* hinzugefügt, sondern im Rahmen eines Promotion-Ergebnis-Objekts. Diese Objekte der Klasse *RulesPromotionResult* enthalten hauptsächlich einen so genannten Effekt (*PromotionEffect*), der das Objekt, welches angewendet werden soll, enthält. Für jeden Effekt bzw. jeden potenziell darin gespeicherten Objekttypen gibt es eine *Rule*, die diesen speziellen Effekt dann anwendet. Im einfachsten Fall ist dieses Effekt-Objekt bspw. ein Objekt vom Typ *Discount* (also Rabatt). Es gibt dann eine spezielle Regel, die im Bedingungsteil *RulesPromotionResult*-Objekte erwarten, die als Effekt ein Objekt vom Typ *Discount* besitzen. Sobald ein solches Objekt gefunden wurde, feuert die Regel und wendet das jeweilige Objekt tatsächlich an.

Für diese Anwendung der Promotions existiert ein eigenes *Rule Execution Set* namens „Promotion“. Hier sind alle Regeln für Promotions und deren Anwendung enthalten. Da aus Sicht der hybriden Plattform die Anwendung der Promotions ein atomarer Vorgang ist, könnte hier ebenfalls, wie bei den Promotion-Teasern, lediglich eine einfache *stateless session*, also zustandslose Auswertungssitzung, verwendet werden. Diese Sitzung würde aufgerufen werden („jetzt Promotions berechnen“) und nach Beendigung der Ausführung wären die Promotions fertig angewendet. Um Promotions allerdings rückgängig machen zu können, ist es notwendig, die Auswertung in der zustandsbehafteten Sitzung (*statefull session*) auszuführen.

UNDO Bei jeder Warenkorb-Interaktion ist es notwendig zu prüfen, ob neue Promotions anwendbar sind. Darüber hinaus ist es aber auch notwendig zu prüfen, ob bereits gewährte und angewendete Promotions noch immer zutreffend sind, oder ob ihnen mittlerweile die Grundlage entzogen wurde. Dies kann insbesondere beim Entfernen von Produkten aus dem Warenkorb der Fall sein, wenn dadurch bestimmte Schwellwerte für den Warenkorbwert oder minimale Stückzahlen für bestimmte Produkte (bspw. bei einer „buy 2 get 1 free“ Promotion etc.) unterschritten werden. Aber auch das Hinzufügen von neuen Produkten kann theoretisch Ursache sein, wenn bspw. durch eine größere Stückzahl andere Promotions greifen als die ursprünglich angewendeten etc. Darüber hinaus kann es noch weitere Ursachen für potenzielle Veränderungen an den angewendeten Promotions geben, wie z.B. die Änderung

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

der Lieferadresse oder der Lieferart.

Aus diesen Gründen kann die Notwendigkeit bestehen, dass bereits gewährte Promotions rückgängig gemacht werden müssen (UNDO). Sobald eine Promotion angewendet wurde, bspw. in Form eines Rabatts oder eines zusätzlichen, kostenlosen Produktes für den Warenkorb, gibt es jedoch von Seiten der hybris Platform keine Möglichkeit mehr, zu entscheiden, ob dieser Rabatt oder dieses Produkt durch die Promotion-Regeln oder eine andere Aktion auf den Warenkorb angewendet bzw. hinzugefügt wurde. Es besteht schließlich auch die Möglichkeit, dass z.B. die Anwendung eines Gutscheins durch die Voucher-Extension oder andere Effekte Rabatte o.ä. zum Warenkorb hinzugefügt wurden. Es gibt im hybris Datenmodell keine Möglichkeit zu definieren, durch welche Aktion bzw. durch welchen Umstand ein Rabatt oder Produkt zum Warenkorb hinzugefügt wurde. Bei Rabatten mag es noch möglich sein, über den Code des Rabatts zu ermitteln, durch welche Aktion er erzeugt wurde (was jedoch keine saubere Lösung darstellt und zudem höchst fehleranfällig ist). Bei Produkten hingegen ist es unmöglich zu sagen, ob sie durch absichtliche Userinteraktion („dieses Produkt in den Warenkorb legen“) oder durch Manipulation von Seiten der Promotion-Rules in den Warenkorb gelangt sind.

Möglichkeiten zum UNDO Es besteht also die Notwendigkeit, die Auswirkungen der Promotion-Regeln auf das hybris System gesondert zu speichern, so dass diese Aktionen gezielt rückgängig gemacht werden können. Dazu dienen die bereits vorgestellten *RulesPromotionResult*-Objekte. Die dort enthaltenen *PromotionEffect*-Objekte speichern die angewendeten Objekte. Ist dort bspw. ein Rabatt-Objekt gespeichert, so kann beim UNDO der Promotions genau dieser Rabatt aus dem Warenkorb entfernt werden. Zusätzlich enthalten die *PromotionEffect*-Objekte ein spezielles UNDO-Objekt. Mit diesem Objekt ist es möglich, einen evtl. vorher vorhandenen Zustand wieder herzustellen.

Beim Neuberechnen nach Warenkorb-Interaktion von Promotions gibt es grundsätzlich zwei Alternativen:

Globales UNDO Bei der Neuberechnung werden zunächst alle angewendeten Promotions gelöscht, egal ob sie noch gültig sind oder nicht. Danach werden alle Promotions ausgehend von dem geänderten, „promotion-freien“ Warenkorb neu berechnet.

Selektives UNDO Es werden lediglich die Promotions rückgängig gemacht, die von den Änderungen des Warenkorbs betroffen sind. Danach wird berechnet, ob durch das UNDO der betroffenen Promotions andere Promotions qualifiziert werden können.

2 Hauptteil

Auf den ersten Blick erscheint das selektive UNDO die bessere, weil vor Allem performantere, Option. Dadurch, dass lediglich die letzte Änderung des Warenkorbs berücksichtigt und neu berechnet werden muss, besteht die Möglichkeit, dass häufig keinerlei Änderungen der Promotions notwendig sind. Dadurch, dass lediglich ein Teil der Promotions neu berechnet werden muss, kann viel Rechenzeit eingespart werden, was vor allem bei großen Systemen mit vielen parallelen Sitzungen durchaus ins Gewicht fallen kann.

Für die vorliegende Arbeit wird dennoch das globale UNDO verwendet. Begründet wird diese Entscheidung damit, dass es schlicht zu aufwändig bzw. teilweise unmöglich sein dürfte, ein selektives UNDO effizient durchzuführen. Zunächst müsste die der Berechnung vorhergehende Interaktionsart ermittelt werden. Allein diese Ermittlung gestaltet sich als schwierig, da für die Berechnung der Promotions generell alle Änderungen des Warenkorbs relevant sein können (abhängig von den konkret verwendeten Regeln). Nicht nur das Hinzufügen oder Entfernen von Produkten, sondern auch das Ändern der Versand- und Zahlungsart, das Einlösen eines Gutscheins, der Login des Users (durch das Einloggen können andere Preise verwendet werden, bspw. für Händler) und andere Faktoren müssen berücksichtigt werden. Sofern man die Ursache ermittelt hat, müsste man die davon betroffenen Promotions ermitteln, was sich wiederum als sehr aufwändig erweisen kann. Danach müsste eine „was wäre wenn“-Kalkulation durchgeführt werden, um zu bestimmen welche Promotions durch das UNDO anderer Promotions wirksam werden könnten. Das bezieht den Umstand mit ein, dass durch das Wegfallen einer Promotion eine andere Promotion zwar noch nicht qualifiziert wird, aber durch das weitere Löschen einer dritten (eigentlich nicht betroffenen) Promotion diese (evtl. bessere oder höher priorisierte) Promotion qualifiziert werden könnte. Insgesamt erweist sich der selektive UNDO als höchst komplex und fehleranfällig, so dass es die bessere Alternative ist, die Promotions bei jeder Warenkorbänderung komplett zu widerrufen und neu zu berechnen.

Speichern der Promotions Um ein UNDO überhaupt durchführen zu können, ist es wie bereits erwähnt notwendig, die angewendeten Promotions zumindest bis zum Abschluss der Transaktion (also in der Regel bis zur Bestellung bzw. bis zum Abbruch der Session) zu speichern. Dazu wird für die Auswertung der Promotions eine zustandsbehaftete Sitzung (*statefull session*) verwendet. Die Objekte im *Working Memory* dieser Session, in erster Linie also die erzeugten *RulesPromotionResult*-Objekte, bleiben während der gesamten Sitzungsdauer, also länger als die eigentliche Auswertung bzw. Anwendung der Promotions, gespeichert. Erst sobald die Auswertungssitzung geschlossen wird,

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

gehen diese Objekte, und damit die Verknüpfung der Promotion-Rules zu den auf den Warenkorb angewendeten Objekten, verloren. Das bedeutet, dass erst, sobald es nicht mehr notwendig ist ein UNDO durchzuführen, diese Auswertungssitzung geschlossen werden darf.

UNDO-Rule Execution Set Um dieses UNDO durchzuführen, existiert ein eigenes *Rule Execution Set* namens „PromotionUndo“. Dieses wird ähnlich dem *Teaser-Rule Execution Set* als *stateless session* ausgeführt, da das UNDO aller Promotions aus Sicht der hybriden Anwendung und des Users ein atomarer Vorgang ist. In diesem *Rule Execution Set* gibt es für jede Art von Objekt, die von den Promotions auf den Warenkorb angewendet werden (also bspw. *Discount*- oder *Product*-Objekte) eine entsprechende UNDO-Regel. Diese Objekte werden, in Form von aus der laufenden Promotion-Auswertungssitzung ausgelesenen *RulesPromotionResult*, bzw. den darin enthaltenen *PromotionEffect*-Objekten, beim Start der Ausführung dem *UNDO-Rule Execution Set* übergeben. Dort werden die Effekte der Promotions rückgängig gemacht.

Integration der Promotions in das Frontend Die Integration der Promotions selber (inklusive UNDO-Funktionalität) in die *storefoundation*-Extension gestaltet sich als wesentlich komplexer als die Integration der Teaser-Funktionen. Einstiegspunkt ist hierbei kein direkter Aufruf in einer xhtml-Seite, sondern lediglich eine modifizierte Methode in einem vorhandenen JSF-Bean. Das Bean *CartJSFBean* stellt im Frontend Methoden zur Arbeit mit dem Warenkorb bereit. Unter anderem enthält dieses Bean die Methode *recalculateCart*, die immer aufgerufen wird, sobald im Frontend eine Aktion ausgeführt wird, die den Zustand des Warenkorbs in irgendeiner Weise beeinflussen könnte. Innerhalb der Methode wird zunächst der Warenkorb (ohne Berücksichtigung von Promotions) neu berechnet und aktualisiert. Danach wird die *invalidateModel*-Methode aufgerufen, die alle zwischengespeicherten Ergebnisse (wie z.B. den Warenkorbwert) löscht. Erst danach wird die Berechnung der Promotions über Business Rules ausgeführt.

Dazu wird das bereits für die Teaser definierte *PromotionJSFBean* verwendet. Dort wird die Methode *calculatePromotions* aufgerufen, die wiederum lediglich die *recalculatePromotions*-Methode aus der bekannten statischen Klasse *PromotionChecker* aufruft. Dort geschieht dann die eigentliche Arbeit.

Die Berechnung der Promotions erfolgt immer in zwei Schritten:

- UNDO aller vorheriger Promotions

2 Hauptteil

- Berechnung der Promotions

UNDO-Schritt Zunächst wird vom *RulesManager* die zustandsbehaftete Auswertungssitzung (*statefull session*) der Promotions angefordert.

```
StatefulRuleSession session =
    RulesManager.getInstance().getStatefulSession(RULESET_PROMOTION,
        false, null, null);
if (session != null)
{
    undoChangesByRules(session);
    assureCartCalculated();
}
```

Listing 2.39: Aufruf des UNDO-Schrittes

Die *statefull session* wird vom *RulesManager* so angefordert, dass sie, falls sie noch nicht existiert, auch nicht erzeugt werden soll. Das heißt, es wird nur ein *StatefulRuleSession*-Objekt zurückgeliefert, wenn bereits eine Sitzung existiert. Falls die Sitzung noch nicht existiert, muss auch kein UNDO der Promotions durchgeführt werden, weil noch keine Auswertung stattgefunden hat. In der Praxis ist dies genau dann der Fall, sobald die erste Warenkorb-Interaktion einer User-Session durchgeführt wird. Falls die Sitzung existiert (also nicht *null* ist), wird ein UNDO durchgeführt.

```
public static void undoChangesByRules(StatefulRuleSession session)
{
    //get the RulesPromotionResult objects from the current promotion
    session
    if (session == null) return;
    List<RulesPromotionResult> results = (List<RulesPromotionResult>)
        session.getObjects(new
            RulesObjectFilter(RulesPromotionResult.class));
    if (results == null || results.size()==0) return;

    //add the PromotionEffect objects from the the results and put them
    into a separate list
```

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
List objects = new ArrayList();
objects.add(JaloSession.getCurrentSession().getCart());
for (RulesPromotionResult result : results)
{
    objects.add(result.getEffect());
}

//call the UNDO ruleset with the list of effects
RulesManager.getInstance().executeRules(RULESET_UNDO, objects, null);

//release the promotion session
RulesManager.getInstance().releaseStatefulSession(RULESET_PROMOTION);
}
```

Listing 2.40: Durchführung des UNDO

Zunächst wird die Liste der *RulesPromotionResult*-Objekte aus der aktuellen Promotion-Auswertungssitzung ausgelesen. Dazu wird der bereits vorgestellte *RulesObjectFilter* verwendet. Falls die Sitzung *null* ist (was eigentlich bereits vor dem Aufruf der Methode überprüft werden sollte) bzw. die Liste der *RulesPromotionResult*-Objekte leer ist (falls zwar bereits eine Auswertung stattgefunden hat, aber keine Promotions angewendet wurden) ist der UNDO-Schritt an dieser Stelle bereits beendet.

Liegen jedoch bereits Ergebnisse der Auswertung vor, so wurden bereits Promotions angewendet, die rückgängig gemacht werden müssen. Dazu wird zunächst eine Liste der Effekt-Objekte aus den Ergebnissen zusammengestellt. Danach wird das UNDO-*Rule Execution Set* aufgerufen und diese Effekt-Objekte mit übergeben. Bei der Ausführung des *Rule Execution Set* werden dann alle Effekte (bzw. alle Effekte, für die es eine UNDO-Regel gibt) rückgängig gemacht. Nach diesem Schritt sind alle Effekte der Promotions auf den Warenkorb entfernt worden.

Abschließend wird die Auswertungssitzung geschlossen. Dies ist notwendig, da diese Auswertungssitzung nicht mehr „synchron“ mit dem Warenkorb ist. Unter anderem sind dort noch die vorherigen Promotion-Ergebnisse gespeichert, die zwischenzeitlich aber rückgängig gemacht wurden. Alternativ könnten auch die entsprechenden *PromotionResult*-Objekte aus dem *Working Memory* der Promotion-Auswertungssitzung entfernt werden. Bei der Verwendung der *JSR 94* API können die Objekte jedoch

2 Hauptteil

nicht direkt, sondern nur via eines *Handles*⁵⁹ aus dem *Working Memory* entfernt werden. Ein *Handle* wird zum eindeutigen Identifizieren von Objekten verwendet. Da die Promotion-Ergebnisse jedoch nicht von der hybris Anwendung an das *Working Memory* übergeben werden, sondern innerhalb der Ausführung der Regeln erzeugt werden, sind diese *Handles* nicht vorhanden. Deshalb können von außerhalb der Regeln die Ergebnis-Objekte nicht mehr aus dem *Working Memory* entfernt werden.

Mit speziellen Regeln wäre es möglich, beispielsweise durch das Hinzufügen eines speziellen Objekts, welches das Entfernen auslöst, auch von innerhalb des Regel-Sets die Objekte zu entfernen. Diese Möglichkeit bietet den Vorteil, dass die Sitzung nicht bei jeder Berechnung neu aufgebaut werden muss. Allerdings müssten zwei neue (sehr einfache) Regeln entwickelt werden. Aus Gründen der Übersichtlichkeit wird bei der Implementierung der vorliegenden Arbeit darauf verzichtet. Darüber hinaus müsste ein spezielles UNDO-Objekt neu definiert werden.

Also letzter Schritt im UNDO-Prozess wird die *assureCartCalculated*-Methode aufgerufen, die sicherstellt, dass der Warenkorb neu berechnet wird.

Berechnungs-Schritt Der eigentliche Berechnungs-Schritt der Promotions erfolgt nach dem UNDO. Nach dem UNDO ist die Promotion-Auswertungssitzung nicht (mehr) vorhanden. Entweder erfolgt die erste Auswertung der Promotions, so wurde noch keine Sitzung erzeugt und auch kein UNDO durchgeführt, oder es gab bereits eine vorherige Auswertungssitzung, deren Ergebnisse im UNDO rückgängig gemacht wurden und die Sitzung danach geschlossen.

```
Map parameters = new HashMap();
parameters.put("log", Logger.getLogger(RULESET_PROMOTION));
session =
    RulesManager.getInstance().getStatefulSession(RULESET_PROMOTION,
        true, parameters, new ArrayList());
try
{
    session.addObject(JaloSession.getCurrentSession());
    session.addObject(JaloSession.getCurrentSession().getCart());
    session.executeRules();
}
catch (Exception e)
```

⁵⁹Java-Interface `javax.rule.Handle` - wird beim Hinzufügen des Objekts zum *Working Memory* als Methodenergebnis zurückgeliefert

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

```
{
    log.error("Error calculating Promotions", e);
    RulesManager.getInstance().releaseStatefulSession(RULESET_PROMOTION);
}
assureCartCalculated();
```

Listing 2.41: Durchführung der Promotion-Berechnung

Es wird dem entsprechend eine neue Auswertungssitzung angefordert. Dieser Sitzung werden das *JaloSession*-Objekt als zentraler Zugangspunkt zum hybris System und der aktuelle Warenkorb des Nutzers übergeben. Danach werden die Promotion-Regeln ausgeführt und die Promotions angewendet. Tritt bei der Auswertung ein Fehler auf, so wird versucht die Auswertungssitzung wieder zu schließen. Als letzter Schritt der Promotion-Berechnung wird der Warenkorb neu berechnet, so dass alle evtl. vorgenommenen Änderungen (Rabatte hinzugefügt etc.) angezeigt werden können.

Beispiele Im Folgenden sollen exemplarisch einige interessante Regeln bzw. Promotions vorgestellt werden.

Ruleset Promotion Im *RuleSet* Promotion sind alle Regeln zusammengefasst, die für die Anwendung der Promotions notwendig sind. Dies umfasst sowohl die Regeln für die einzelnen Promotions selbst, also auch die Regeln, die die Ergebnisse der Auswertung dann anwenden.

Promotion: OrderThresholdPromotion - Alle Bestellungen über 500 € erhalten 10% Rabatt!

Um eine einfache Promotion, die ab einem bestimmten Schwellwert einen prozentualen Rabatt gewährt, zu anzulegen, wird eine Regel wie in Tabelle 2.9 definiert.

Die Regel prüft, ob der Warenkorbwert (total) 500 € übersteigt. Dazu wird ein *ObjectTypeNode* mit einem *field constraint* verwendet. Wenn dies der Fall ist, wird ausgewertet, ob sich bereits ein *RulesPromotionResult*-Objekt im *Working Memory* des *Rule Execution Set* „promotion“ befindet, welches der Regel *OrderThresholdPromotionRule* zugeordnet ist. Dazu wird das DRL-Schlüsselwort *eval* verwendet. Dieses Schlüsselwort erlaubt es, beliebigen Code, der zu einem boolean ausgewertet wird, zu prüfen. Im Gegensatz zu normalen *ObjectTypeNodes* mit *field constraints* (wie hier in der ersten Bedingungszeile) werden diese *eval*-Bedingungen dynamisch ausgewertet. Das bedeutet, sie

2 Hauptteil

Attribut	Wert
Name	OrderThresholdPromotionRule
Beschreibung	alle Bestellungen über 500€ erhalten 10% Rabatt
Imports	de.hybris.jakarta.jalo.order.Cart de.hybris.jakarta.util.DiscountValue; com.arithnea.rules.promotions.PromotionChecker; com.arithnea.rules.promotions.RulesPromotionResult;
opt Attribute	-
LHS	cart : Cart(total > 500.00) eval(!PromotionChecker.isApplied(„promotion“, „OrderThresholdPromotionRule“))
RHS	RulesPromotionResult result = new RulesPromotionResult(„OrderThresholdPromotionRule“, null, new DiscountValue(„above 500“, 10, false, cart.getCurrency().getIsoCode())); assert(result);

Tabelle 2.9: OrderThreshold Promotion

können verwendet werden, wenn sich der Wert der Auswertung dynamisch ändert. Bei normalen *ObjectTypeNodes* mit *field constraints* wird die Bedingung nur einmal, sobald das Objekt in das *Rete*-Netz aufgenommen wird, ausgewertet. Solange nicht mittels des *modify*-Operators (bzw. durch die Verwendung von *PropertyChangeListener*) explizit bekannt gemacht wird, dass sich das Objekt verändert hat, werden die Felder nicht mehr ausgewertet. Die *eval*-Anweisungen werden jedoch (sobald alle *ObjectTypeNodes* und *field constraints* erfüllt sind) bei jeder Ausführung des *Rule Execution Set* erneut überprüft. Deshalb sollten *eval*-Anweisungen immer am Schluss des Bedingungssteils stehen. *Eval*-Anweisungen sollten aus Performancegründen nur für dynamische Abfragen verwendet werden. Die *eval*-Anweisung ruft im vorliegenden Fall die *isApplied*-Methode der statischen Klasse *PromotionChecker* auf, die bereits zuvor erläutert wurde. Dort wird geprüft, ob bereits ein *RulesPromotionResult*-Objekt für die aktuelle Regel im *Working Memory* des *Rule Execution Set* „promotion“ vorliegt. Wenn dies der Fall ist, so wurde die Regel bereits angewendet und soll nicht mehr aktiviert werden.

Sind die beiden Bedingungen der Regel erfüllt, so wird ein neues *RulesPromotionResult*-Objekt

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

erzeugt. Diesem Objekt wird als Effekt-Objekt ein Objekt vom Typ *DiscountValue* übergeben. Dieser *DiscountValue* stellt den Rabatt dar, der gewährt werden soll. Im vorliegenden Fall erhält der Rabattwert den Code „above 500“, der später im Warenkorb angezeigt wird. Als eigentlichen Wert erhält der Rabattwert die Zahl 10, die einen relativen Wert (*false* beim Parameter *isAbsolute*) darstellt. Als Währung wird die Währung des Warenkorbs angegeben. Das neu erzeugte *RulesPromotionResult*-Objekt wird danach dem *Working Memory* hinzugefügt (*assert*-Schlüsselwort).

Durch Ausführung dieser Regel alleine wird der Warenkorb noch nicht manipuliert. Wie bereits erläutert wird lediglich ein Auswertungs-Ergebnis-Objekt erzeugt und in das *Rete*-Netz aufgenommen. Die Anwendung des Ergebnisses (in diesem Fall des Rabatts) wird durch eine separate Regel erledigt, die später vorgestellt wird.

Um die Regel in natürlicher Sprache abbilden zu können, muss die verwendete *Domain Specific Language* einige neue Mappings enthalten (siehe Tabelle 2.10).

DRL	DSL
<code>cart : Cart(total > {value})</code>	Warenkorbwert über {value}
<code>eval(!PromotionChecker.isApplied("promotion", "{rule}"))</code>	Rabatt für die Regel '{rule}' noch nicht gegeben
<code>RulesPromotionResult result = new Ru- lesPromotionResult("{rule}", null, new DiscountValue("{code}", {value}, false, cart.getCurrency().getIsoCode()));</code>	{value} Prozent Rabatt für die Regel '{rule}' unter dem Code '{code}' geben
<code>assert(result);</code>	Ergebnis berechnen

Tabelle 2.10: DSL-Mapping für Promotion #1

Nachdem bisher noch keine Mappings im *DSL* definiert waren, muss für jede Zeile ein neues Mapping definiert werden. Dies erscheint zu Beginn umständlich und es ist nicht ersichtlich, wo der Vorteil der *DSL* liegt, wenn sowieso das Mapping neu definiert werden muss. Mit wachsender Anzahl von Rules und Mappings wird aber deutlich, dass immer mehr Mappings bereits vorliegen und nicht neu definiert werden müssen. Für die weiteren Beispiele wird das Mapping nicht mehr komplett angegeben, sondern nur noch die neuen Elemente, die hinzugefügt werden.

2 Hauptteil

Sobald das Mapping definiert ist, kann die Regel wie in Tabelle 2.11 in natürlicher Sprache angegeben werden.

Attribut	Wert
Name	OrderThresholdPromotion
Beschreibung	alle Bestellungen über 500€ erhalten 10% Rabatt
Imports	de.hybris.jakarta.jalo.order.Cart de.hybris.jakarta.util.DiscountValue; com.arithnea.rules.promotions.PromotionChecker;
opt Attribute	-
LHS	Warenkorbwert über 500.00 Rabatt für die Regel 'OrderThresholdPromotionRule' noch nicht gegeben
RHS	10 Prozent Rabatt für die Regel 'OrderThresholdPromotionRule' unter dem Code 'above 500' geben Ergebnis berechnen

Tabelle 2.11: OrderThreshold Promotion mit DSL

Promotion: MultiBuy - Kaufen Sie 2 Digicams, und sie erhalten 5 € Rabatt! Kaufen Sie 5 Digicams, und sie erhalten 20 € Rabatt!

Diese Promotion ist komplexer. Sie wird als zwei getrennte Regeln abgebildet. Eine Regel stellt den Rabatt für den Bereich von zwei bis vier Digicams, die andere Regel den Bereich von fünf und mehr Digicams dar. Die Promotion für zwei bis vier Digicams kann wie in Tabelle 2.12 angegeben definiert werden.

Der Bedingungsteil der Regel enthält zunächst einen einfachen Test, ob ein Objekt vom Typ *Cart* im *Working Memory* enthalten ist. Diese Bedingung ist an sich nicht notwendig, da es für die Berechnung unerheblich ist ob dieses Warenkorb-Objekt vorhanden ist und aus dem Warenkorb auch keine Felder getestet werden. Der einzige Grund, warum diese Bedingung benötigt wird, ist eine Besonderheit der *Rete*-Implementierung innerhalb von *JBoss Rules*. Wie bereits erwähnt werden die *eval*-Bedingungen nur ausgewertet, wenn alle vorherigen *ObjectTypeNodes* (mit ihren *field*

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Attribut	Wert
Name	DigicamMultibuy2
Beschreibung	beim Kauf von zwei Digicams gibt es 5 € Rabatt
Imports	de.hybris.jakarta.jalo.order.Cart de.hybris.jakarta.util.DiscountValue; com.arithnea.rules.promotions.PromotionChecker; com.arithnea.rules.promotions.RulesPromotionResult;
opt Attribute	-
LHS	cart : Cart() eval(PromotionChecker.hasUnboundProducts("HW1230-0200", 2)); eval(!PromotionChecker.hasUnboundProducts("HW1230-0200", 5));
RHS	RulesPromotionResult result = new RulesPromotionResult("DigicamMultibuy2", null, new DiscountValue("DigicamMultibuy2", 5.00, true, cart.getCurrency().getIsoCode())); result.addConsumedProduct("HW1230-0200", 2); assert(result);

Tabelle 2.12: Multibuy Promotion 2 Produkte / 5 €

constraints) erfüllt wurden. Wenn der Bedingungsteil nur aus *eval*-Tests besteht, so wird die Regel genau einmal (beim ersten Test) überprüft. Falls jedoch beispielsweise vier Digicams im Warenkorb liegen, tritt folgender Effekt auf: Die Regel wird einmal überprüft, es wird erkannt, dass die beiden *eval*-Bedingungen zutreffen und die Regel wird angewendet. Durch das Anwenden wird u.a. der *Working Memory* verändert (genau genommen wird durch die später vorgestellte Anwendungsregel der Warenkorb modifiziert). Durch diese Änderung werden alle betroffenen Regeln neu überprüft. Da das entsprechende *Cart*-Objekt jedoch nicht im Bedingungsteil der Regel auftaucht, wird die Regel nicht erneut überprüft, und die Promotion nicht ein zweites Mal angewendet, obwohl dies möglich wäre.

Es ist denkbar, dass es sich hierbei nicht um einen Fehler handelt, sondern die Implementierung absichtlich so gelöst wurde. Es besteht ansonsten nämlich die Gefahr, dass durch unvorsichtige Definition der *eval*-Bedingungen (genauer: wenn die *eval*-Bedingungen immer wahr sind) eine

2 Hauptteil

Endlosschleife entsteht. Für die vorliegende Regel ist dies jedoch nicht der Fall und die erste Bedingung muss als reiner Trigger, der eine erneute Überprüfung erzwingt, aufgenommen werden. Im Ausführungsteil wird, analog zu der vorher bereits vorgestellten *OrderThresholdPromotion*, zunächst ein *RulesPromotionResult*-Objekt mit einem darin enthaltenen 5 €-Rabatt erzeugt. Zusätzlich wird dem Ergebnis-Objekt noch übergeben, dass durch das vorliegende Ergebnis zwei Produkte mit dem Code „HW1230-0200“ verbraucht wurden, und für weitere Promotions nicht mehr zur Verfügung stehen. Diese verbrauchten Produkte werden bei der Erzeugung der Sicht von unverbrauchten Produkten, die in der Klasse *PromotionChecker* durch die *hasUnboundProducts*-Methode verwendet wird, berücksichtigt. Im letzten Schritt wird das Ergebnis-Objekt zum *Working Memory* hinzugefügt, so dass es von einer anderen Regel auf den Warenkorb angewendet werden kann. Die zweite Regel für mehr als fünf Digicams wird kann, ohne hier nochmals auf die Einzelheiten einzugehen, wie in Tabelle 2.13 angegeben definiert werden.

Attribut	Wert
Name	DigicamMultibuy5
Beschreibung	beim Kauf von fünf Digicams gibt es 20 € Rabatt
Imports	de.hybris.jakarta.jalo.order.Cart de.hybris.jakarta.util.DiscountValue; com.arithnea.rules.promotions.PromotionChecker; com.arithnea.rules.promotions.RulesPromotionResult;
opt Attribute	-
LHS	cart : Cart() eval(PromotionChecker.hasUnboundProducts("HW1230-0200", 5));
RHS	RulesPromotionResult result = new RulesPromotionResult("DigicamMultibuy5", null, new DiscountValue("DigicamMultibuy5", 20.00, true, cart.getCurrency().getIsoCode())); result.addConsumedProduct("HW1230-0200", 5); assert(result);

Tabelle 2.13: Multibuy Promotion 5 Produkte / 20 €

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Auch für diese Regel sollen entsprechende Mappings in die Promotion-*DSL* aufgenommen werden. Hier müssen bereits nicht mehr für alle Zeilen der Regel Mappings definiert werden, da für sie schon passende Mappings im *DSL* enthalten sind. Für die erste (`RulesPromotionResult result ...`) und dritte (`assert(result);`) Zeile des Ausführungsteils ist es nicht notwendig neue Mappings zu definieren. Für die restlichen Zeilen des Bedingungs- und Ausführungsteils werden neue Mappings entsprechend der Tabelle 2.14 definiert.

DRL	DSL
<code>cart : Cart()</code>	Warenkorb beobachten
<code>eval(PromotionChecker.hasUnboundProducts (“{code}”, {quantity}));</code>	es gibt mindestens {quantity} Produkte ‘{code}’ im Warenkorb
<code>eval(!PromotionChecker.hasUnboundProducts (“{code}”, {quantity}));</code>	es gibt weniger als 5 Produkte ‘HW1230-0200’ im Warenkorb
<code>result.addConsumedProduct(“{code}”, {quantity});</code>	dabei {quantity} Produkte ‘{code}’ verbrauchen

Tabelle 2.14: DSL-Mapping für Promotion #2

Mit Hilfe der (erweiterten) *DSL* kann die erste Multibuy-Regel entsprechend der Tabelle 2.15 angegeben werden.

Die zweite Multibuy-Regel (für fünf Digicams) kann analog definiert werden.

Promotion: Buy 2 get 1 free - Kaufen Sie zwei T-Shirts, und sie erhalten ein drittes umsonst!

Die bisher vorgestellten Promotions gewähren dem Kunden Rabatte. Diese können sowohl relativ (Prozentsatz) oder absolut (fester Betrag) sein. Die nun präsentierte Regel soll verdeutlichen, wie eine andere Art von Vergünstigung gewährt werden kann. Die in Tabelle 2.16 angegebene Regel soll eine Buy 2 get 1 free-Promotion abbilden, bei der für zwei gekaufte Produkte ein drittes geschenkt wird.

Die Regel ist generell den zuvor präsentierten Multibuy-Regeln sehr ähnlich. Sie unterscheidet sich von diesen in einem veränderten *Import* (nicht mehr *DiscountValue* sondern *ProductManager* werden importiert) und erzeugt ein anderes *RulesPromotionResult*-Objekt. Diesem *RulesPromotionResult*-

2 Hauptteil

Attribut	Wert
Name	DigicamMultibuy2
Beschreibung	beim Kauf von zwei Digicams gibt es 5 € Rabatt
Imports	de.hybris.jakarta.jalo.order.Cart de.hybris.jakarta.util.DiscountValue; com.arithnea.rules.promotions.PromotionChecker; com.arithnea.rules.promotions.RulesPromotionResult;
opt Attribute	-
LHS	Warenkorb beobachten es gibt mindestens 2 Produkte 'HW1230-0200' im Warenkorb es gibt weniger als 5 Produkte 'HW1230-0200' im Warenkorb
RHS	RulesPromotionResult result = new RulesPromotionResult(„DigicamMultibuy2“, null, new DiscountValue(„DigicamMultibuy2“, 5.00, true, cart.getCurrency().getIsoCode())); dabei 2 Produkte 'HW1230-0200' verbrauchen Ergebnis berechnen

Tabelle 2.15: Multibuy Promotion 2 Produkte / 5 € mit DSL

Objekt wird nicht mehr wie zuvor ein Rabatt (*DiscountValue*), sondern ein Produkt als Effekt-Objekt übergeben. Dieses Produkt wird aufgrund des angegebenen Codes von der Manager-Klasse *Product-Manager* ermittelt. Für die Anwendung des Ergebnis-Objekts (also das Hinzufügen des Produktes zum Warenkorb) ist auch hier eine separate Anwendungsregel verantwortlich.

Auch bei dieser Regel tritt der unschöne Effekt auf, dass eine Mehrfachanwendung - da der Bedingungsteil eigentlich nur aus einer *eval*-Bedingung besteht - nur mit der Verwendung des prinzipiell unnötigen *ObjectTypeNodes* für das *Cart*-Objekt möglich ist.

Für die *Buy2get1freeTshirt*-Promotion muss das *DSL*-Mapping lediglich um eine einzige Zeile erweitert werden (siehe Tabelle 2.17). Hier wird schon klar ersichtlich, dass die Definition von neuen *DSL*-Mappings mit wachsender Anzahl von Regeln nur noch selten nötig ist. Lediglich wenn vollkommen neuartige Regeln definiert werden (wie später noch die Anwendungsregeln), müssen viele neue *DSL*-Mappings aufgenommen werden.

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Attribut	Wert
Name	Buy2get1freeTshirt
Beschreibung	beim Kauf von zwei T-Shirts gibt es das dritte umsonst
Imports	de.hybris.jakarta.jalo.order.Cart de.hybris.jakarta.jalo.product.ProductManager; com.arithnea.rules.promotions.PromotionChecker; com.arithnea.rules.promotions.RulesPromotionResult;
opt Attribute	-
LHS	cart : Cart() eval(PromotionChecker.hasUnboundProducts(„CL1200-aaa010x04-20“, 2));
RHS	RulesPromotionResult result = new RulesPromotionResult(„Buy2get1freeTshirt“, null, ProductManager.getInstance().getProductsByCode(„CL1200-aaa010x04- 20“).iterator().next()); result.addConsumedProduct(„CL1200-aaa010x04-20“, 2); assert(result);

Tabelle 2.16: Buy2get1freeTshirt Promotion

Sobald dieses Mapping in die *DSL* aufgenommen wurde, kann die Regel wie in Tabelle 2.18 angegeben werden.

Function: Beschränken der Promotions auf einen Zeitraum

In den vorgestellten Promotions werden keine Einschränkungen bezüglich des Zeitraums definiert. Da dies in der Praxis jedoch mit Sicherheit erforderlich sein wird, soll diese Anforderung hier gesondert betrachtet werden.

Die Beschränkung auf ein bestimmtes Datum bzw. einen bestimmten Zeitraum wird in den Bedingungsteil der jeweiligen Promotion-Regel aufgenommen. Erstaunlicherweise verfügt die verwendete Implementierung *JBoss Rules* (zumindest in der verwendeten Version) lediglich über rudimentäre Daten-Vergleich-Fähigkeiten. So wird zwar ein < bzw. >-Operator definiert und die Möglichkeit ein neues Datum per String zu definieren „29-Oct-2007“. Es gibt jedoch keine einfache Möglichkeit das

2 Hauptteil

DRL	DSL
<pre>RulesPromotionResult result = new RulesPromotionResult("{promotion}", null, ProductManager.getInstance().getProductsByCode("{code}").iterator().next());</pre>	<pre>kostenloses Produkt '{code}' für die Promotion '{promotion}' erzeugen</pre>

Tabelle 2.17: DSL-Mapping für Promotion #3

aktuelle Datum zu definieren und dann mit den vorgegebenen Daten zu vergleichen.

Aus diesem Grunde wird eine *Function* verwendet. Diese wird im Header des *Promotion-RuleSet*-Objekts gespeichert.

```
function boolean validDate(java.util.Date startDate, java.util.Date
    endDate)
{
    java.util.Date now = new java.util.Date();
    boolean afterStart = (startDate != null) ? startDate.before(now) :
        true;
    boolean beforeEnd = (endDate != null) ? endDate.after(now) : true;
    return (afterStart && beforeEnd);
}
```

Listing 2.42: Function zum komfortablen Datumsvergleich

Die *Function validDate* nimmt als Parameter zwei Daten auf. Dann wird geprüft ob das erste angegebene Datum vor dem aktuellen Datum liegt und ob das zweite angegebene Datum nach dem aktuellen Datum liegt. Es wird also geprüft, ob das aktuelle Datum innerhalb des von den beiden angegebenen Daten aufgespannten Zeitraumes liegt. Wird kein Start- bzw. Enddatum angegeben (*null*), so gilt die jeweilige Teilbedingung automatisch als erfüllt. Auf diese Weise können auch nur einseitig begrenzte Zeiträume überprüft werden.

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Attribut	Wert
Name	Buy2get1freeTshirt
Beschreibung	beim Kauf von zwei T-Shirts gibt es das dritte umsonst
Imports	de.hybris.jakarta.jalo.order.Cart de.hybris.jakarta.jalo.product.ProductManager; com.arithnea.rules.promotions.PromotionChecker; com.arithnea.rules.promotions.RulesPromotionResult;
opt Attribute	-
LHS	Warenkorb beobachten es gibt mindestens 2 Produkte 'CL1200-aaa010x04-20' im Warenkorb
RHS	kostenloses Produkt 'CL1200-aaa010x04-20' für die Promotion 'Buy2Get1FreePromotion' erzeugen dabei 2 Produkte 'CL1200-aaa010x04-20' verbrauchen Ergebnis berechnen

Tabelle 2.18: Buy2get1freeTshirt Promotion mit DSL

Die *Function* wird im Rahmen einer *eval*-Bedingung aufgerufen.

```
Datum zwischen dem '{date1}' und dem '{date2}' =
eval (validDate (new
    java.text.SimpleDateFormat ("dd.MM.yyyy").parse ("{date1}"), new
    java.text.SimpleDateFormat ("dd.MM.yyyy").parse ("{date2}")));
```

Listing 2.43: eval-Bedingung DSL-Mapping für Datumsvergleich

Das Datum wird immer im Format „dd.MM.yyyy“ angegeben, also bspw. „27.03.2008“. Soll eine Promotion lediglich im Dezember 2008 gelten, so kann folgende Bedingung hinzugefügt werden: „Datum zwischen dem '01.12.2008' und dem '31.12.2008'“. Für offene Zeiträume muss ein gesondertes *DSL*-Mapping, welches an den „offenen“ Ende der Zeiträume null-Werte übergibt, definiert werden.

Anwendungsregel: DiscountValues - Die Anwendung von Rabatten

Wie bereits zuvor erläutert sind die Regeln der Promotions (bspw. *OrderThresholdPromotionRule*) lediglich dafür verantwortlich, ein entsprechendes *RulesPromotionResult*-Objekt zu erzeugen und in

2 Hauptteil

den *Working Memory* der aktuellen Auswertungssitzung einzufügen. Diese *RulesPromotionResult*-Objekte enthalten ein Effekt-Objekt, welches die eigentliche Auswirkung der Regel darstellt. Dabei kann es sich beispielsweise um ein Objekt vom Typ *DiscountValue* (für Rabatte) oder *Product* (für kostenlose Produkte) handeln. Für die tatsächliche Anwendung dieser Effekte auf den Warenkorb (oder auf andere Bestandteile der hybris Plattform) werden eigene Regeln definiert. Um diese Regeln kenntlich zu machen und von den Promotion-Regeln besser unterscheiden zu können, werden die Anwendungsregeln immer „do“ zusammen mit dem ObjektTypen, für den sie zuständig sind, genannt, also z.B. „doDiscountValues“ oder „doProducts“.

Da diese Regeln nicht von Sachbearbeitern bearbeitet werden sollen, ist es nicht notwendig für sie Mappings in die *Domain Specific Language* aufzunehmen. Mit den vorhandenen doX-Regeln sollten alle vorhandenen und neu zu definierenden Promotions angewendet werden können. Falls wirklich der Fall eintritt, dass für eine komplett neue Promotion auch eine neue Anwendungsregel definiert werden muss, so muss dies aufgrund der Komplexität der Anwendungsregeln ohnehin von einem Programmierer mit genauer Kenntnis der hybris-API und der Drools Rule Language übernommen werden.

Um *DiscountValue*-Objekte auf den Warenkorb anzuwenden, wird die Regel „doDiscountValues“ definiert (siehe Tabelle 2.19). Alle *RulesPromotionResult*-Objekte, die als Effekt ein *DiscountValue*-Objekt enthalten, werden von dieser Regel verarbeitet.

Im Bedingungsteil der Regel wird zunächst wieder ein *ObjectTypeNode* für Objekte des *Cart*-Typen definiert. Für die vorliegende Regel hat dies aber nicht den Zweck eine erneute Überprüfung der Regel bei Änderung des *Working Memory* zu erzwingen. Statt dessen wird die Variable *cart* belegt, so dass diese im Ausführungsteil verwendet und auf den Warenkorb zugegriffen werden kann. Des Weiteren werden zwei zusätzliche Variablen definiert. In der Variable *result* wird das *RulesPromotionResult*-Objekt und in der Variable *resultEffect* das darin enthaltene *PromotionEffect*-Objekt gespeichert. Als letzter Schritt des Bedingungsteils erfolgt eine dynamische *eval*-Auswertung, die überprüft, ob das vorhandene Ergebnis schon angewendet wurde, und ob es sich bei dem Effekt um einen Rabattwert handelt. Nur wenn das Ergebnis noch nicht angewendet wurde und es sich um einen Rabattwert handelt, wird die Regel angewendet.

Im Ausführungsteil der Regel wird der *DiscountValue* mittels der *addGlobalDiscountValue*-Methode auf den Warenkorb angewendet. Danach wird das Ergebnis als angewendet markiert, so dass es später

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Attribut	Wert
Name	doDiscountValues
Beschreibung	wendet DiscountValues auf den Warenkorb an
Imports	de.hybris.jakarta.jalo.order.Cart com.arithnea.rules.promotions.RulesPromotionResult de.hybris.jakarta.util.DiscountValue
opt Attribute	-
LHS	<pre> cart : Cart() result : RulesPromotionResult (resultEffect : effect) eval(!result.isApplied() && resultEffect.getEffect() instanceof DiscountValue) </pre>
RHS	<pre> cart.addGlobalDiscountValue((DiscountValue)resultEffect.getEffect()); result.setApplied(); cart.calculateTotals(true); retract(result);assert(result); retract(cart);assert(cart); </pre>

Tabelle 2.19: doDiscountValues Regel

nicht ein weiteres Mal angewendet werden kann. Der Warenkorb muss neu berechnet werden, weil sich sein Wert durch die Anwendung des Rabatts geändert haben könnte. Schließlich wird dem *Working Memory* mitgeteilt, dass sich die beiden Objekte *result* und *cart* verändert haben. Normalerweise würde hierfür das Schlüsselwort *modify* verwendet werden. Aufgrund eines Bugs in der verwendeten *JBoss Rules*-Version funktioniert dieses *modify*-Schlüsselwort jedoch nicht immer einwandfrei[8]. Als Workaround wird das Objekt einfach aus dem *Working Memory* entfernt und sofort wieder neu hinzugefügt, was den selben Effekt wie *modify* hat.

Nachdem diese Regel ausgeführt wurde, wurde die Promotion vollständig auf den Warenkorb angewendet.

Anwendungsregel: Products - Die Anwendung von Produkten

Eine weitere Anwendungsregel ist die Regel „doProducts“ zum Hinzufügen von kostenlosen Produkten zum Warenkorb, die in der Tabelle 2.20 definiert wird.

2 Hauptteil

Attribut	Wert
Name	doProducts
Beschreibung	fügt kostenlose Produkte dem Warenkorb hinzu
Imports	de.hybris.jakarta.jalo.order.Cart com.arithnea.rules.promotions.RulesPromotionResult de.hybris.jakarta.jalo.product.Product de.hybris.jakarta.jalo.order.AbstractOrderEntry
opt Attribute	-
LHS	cart : Cart() result : RulesPromotionResult (resultEffect : effect) eval(!result.isApplied() && resultEffect.getEffect() instanceof Product)
RHS	Product p = (Product) resultEffect.getEffect(); AbstractOrderEntry entry = cart.addNewEntry(p, 1, p.getUnit()); entry.setGiveAway(true); resultEffect.setEffect(entry); result.setApplied(); retract(result);assert(result); retract(cart);assert(cart);

Tabelle 2.20: doProducts Regel

Der Bedingungsteil der doProducts-Regel ähnelt dem der *doDiscountValues*-Regel sehr stark. Der einzige Unterschied ist, dass als Effekt-Objekt kein Objekt vom Typ *DiscountValue*, sondern vom Typ *Product* erwartet wird. Diese Regel wird also bei allen noch nicht angewendeten *RulesPromotionResult*-Objekten angewendet, die ein Effekt-Objekt vom Typ *Product* enthalten. Dieses Produkt soll dann kostenlos zum Warenkorb hinzugefügt werden.

Im Ausführungsteil der Regel wird ein neues Objekt vom Typ *AbstractOrderEntry* angelegt. Dieses Objekt entspricht einer Bestellzeile im Warenkorb. Diese Zeile enthält genau eine Basiseinheit (also bspw. „Stück“) des angegebenen Products. Die Bestellzeile wird als Geschenk markiert (*setGiveAway(true)*), so dass bei der Berechnung des Warenkorbs der Preis der Zeile auf 0 gesetzt wird. Da ein späteres UNDO nicht das Produkt selbst, sondern die gesamte Bestellzeile entfernen muss,

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

wird das Effekt-Objekt des angewendeten *RulesPromotionResult*-Objekts geändert. Wo zuvor nur das *Product*-Objekt selbst verwendet wurde, wird nun das *AbstractOrderEntry*-Objekt, welches das Produkt enthält, verwendet. Das Ergebnis wird als bereits angewendet markiert und das Ergebnis- und Warenkorb-Objekt werden ähnlich wie bei der *doDiscountValues*-Regel aus dem *Working Memory* entfernt und neu hinzugefügt, um ein *modify* zu simulieren.

Im Gegensatz zur *doDiscountValues*-Regel muss der Warenkorb nach Hinzufügen des kostenlosen Produktes nicht neu berechnet werden, da sich der Warenkorbwert durch das kostenlose Produkt nicht verändert hat.

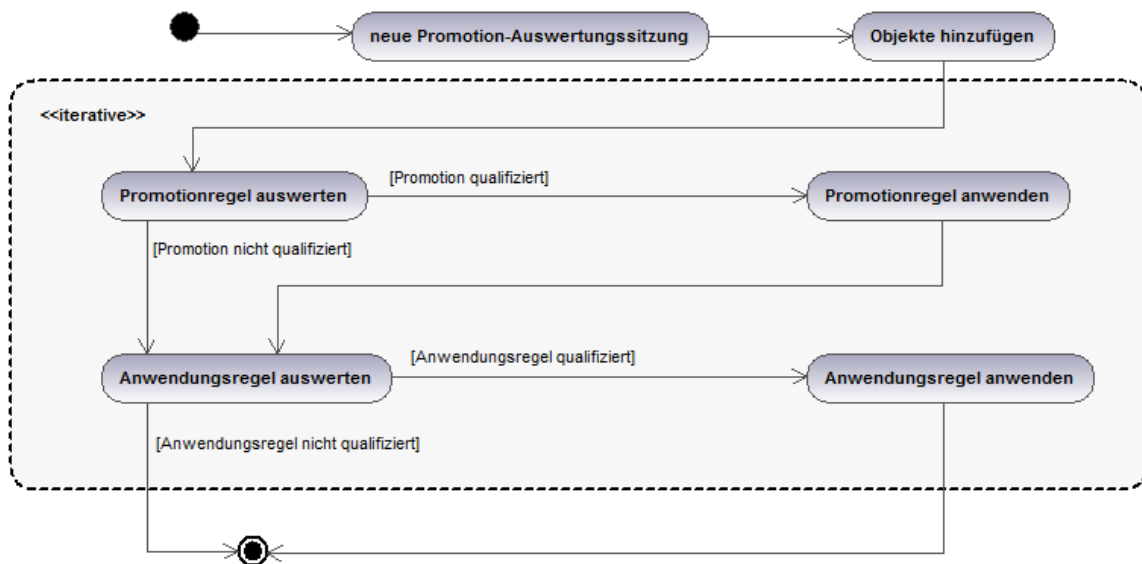


Abbildung 2.15: vereinfachtes Aktivitätsdiagramm Auswertung von Promotions mit Business Rules

Ruleset UNDO Nachdem im vorigen Teil die Regeln zum Anwenden der Promotions definiert wurden, sollen jetzt die Regeln zum UNDO der Promotions dargestellt werden. Wie bereits erwähnt wird das UNDO vor jeder Berechnung der Promotions global durchgeführt. Das heißt, dass alle Effekte der Promotions zunächst entfernt und dann gegebenenfalls neu angewendet werden. Für das UNDO der Promotions werden lediglich die *PromotionEffect*-Objekte benötigt und kein Wissen über die Promotion-Regeln selbst, bzw. wie einzelne Ergebnisse zustande gekommen sind. Diese *RulesPromotionResult*-Objekte werden aus der zustandsbehafteten Promotions-Auswertungssitzung aus-

2 Hauptteil

gelesen und bei der Anwendung des UNDO-*RuleSet* in den *Working Memory* aufgenommen. Generell muss für jede Anwendungsregel (*doDiscountValues*, *doProducts*...) eine entsprechende UNDO-Regel aufgenommen werden. Sobald eine neue Anwendungsregel für eine vollkommen neuartige Promotion definiert wird, ist es wahrscheinlich, dass auch eine neue UNDO-Regel hierfür angelegt werden muss. Eine eigene *Domain Specific Language* für das UNDO-*RuleSet* wird nicht definiert, da die Regeln dieses *RuleSet* (genau wie die Anwendungsregeln) nicht von Sachbearbeitern, sondern nur von Personen mit Programmierkenntnissen gepflegt werden.

UNDO-Regel: Products - UNDO von kostenlosen Produkten

Diese Regel kann als der Gegenspieler der *doProducts*-Anwendungsregel verstanden werden. Analog zu dieser entfernt sie die Bestellzeilen, die innerhalb der *PromotionEffect*-Objekte gespeichert sind (siehe Tabelle 2.21).

Attribut	Wert
Name	undoProductsRule
Beschreibung	entfernt kostenlose Produkte aus dem Warenkorb
Imports	de.hybris.jakarta.jalo.order.Cart com.arithnea.rules.promotions.RulesPromotionResult de.hybris.jakarta.jalo.product.Product de.hybris.jakarta.jalo.order.AbstractOrderEntry
opt Attribute	-
LHS	cart : Cart() result : RulesPromotionResult.PromotionEffect (resultEffect : effect) eval(resultEffect instanceof AbstractOrderEntry)
RHS	cart.removeEntry((AbstractOrderEntry)resultEffect);

Tabelle 2.21: undoProducts Regel

Im Bedingungsteil wird zunächst der Warenkorb als Variable definiert, so dass dann im Ausführungsteil später darauf zugegriffen werden kann. Danach wird aus dem *RulesPromotionResult*-Objekt das *PromotionEffect*-Objekt in der Variable *resultEffect* gespeichert. In der *eval*-Bedingung wird geprüft, ob das Effekt-Objekt vom Typ *AbstractOrderEntry* (also Bestellzeile)

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

ist. Wenn dies der Fall ist, kann die Regel ausgeführt werden.

Im Ausführungsteil wird die Bestellzeile aus dem Warenkorb entfernt. Dadurch wird implizit das kostenlose Produkt aus der Bestellung gelöscht. Die Promotion wurde dadurch vollständig rückgängig gemacht.

UNDO-Regel: DiscountValues - UNDO von Rabatten

Um *DiscountValues* (also Rabatte) aus dem Warenkorb zu entfernen, wird die *undoDiscountValues*-Regel definiert. Sie ist der Gegenspieler der *doDiscountValues*-Anwendungsregel.

Attribut	Wert
Name	undoDiscountValuesRule
Beschreibung	entfernt Rabatte aus dem Warenkorb
Imports	de.hybris.jakarta.jalo.order.Cart com.arithnea.rules.promotions.RulesPromotionResult de.hybris.jakarta.jalo.product.Product de.hybris.jakarta.util.DiscountValue
opt Attribute	-
LHS	cart : Cart() result : RulesPromotionResult.PromotionEffect (resultEffect : effect) eval(resultEffect instanceof DiscountValue)
RHS	cart.removeGlobalDiscountValue((DiscountValue)resultEffect);

Tabelle 2.22: undoDiscountValues Regel

Diese Regel stellt sich ähnlich zur *undoProducts*-Regel dar, und bedarf deshalb keiner weiteren Erläuterung.

2.3.4 Fazit

In der vorliegenden Arbeit wurde demonstriert, dass es möglich ist eine Business Rule Engine mit der *hybris* Anwendung zu verknüpfen und sie für Promotion-Funktionalitäten zu nutzen. Dabei ist

2 Hauptteil

der Spielraum der Business Rules groß. Mit entsprechend modellierten Regeln können viele Anforderungen abgebildet werden. Leider ist es nicht möglich, ohne neu programmierte, speziell auf die Promotions zugeschnittenen, Java-Klassen zu arbeiten. Für die Promotions mussten einige Klassen entwickelt werden.

Im Vergleich zur typbasierten, projektspezifischen Promotion-Implementierung von Virgin Megastores bietet die regelbasierte Lösung wesentlich höhere Flexibilität. Hier können neuartige Promotions definiert werden, ohne dass das Typsystem verändert, Java-Code geschrieben oder das System neu gestartet werden muss. Dennoch ist es natürlich nicht auszuschließen, dass für komplett neue Promotions auch bei der regelbasierten Lösung neue Klassen geschrieben werden müssen.

Vor allem für den Bedingungsteil, also die Filterlogik, ob eine Promotion angewendet werden darf oder nicht, eignen sich die Business Rules besonders. Hier kann die Flexibilität und Erweiterbarkeit der regelbasierten Lösung gut genutzt werden, da es zwar relativ viele verschiedene Vorbedingungen für eine Promotion geben kann, diese aber im Einzelnen nicht besonders komplex sind.

Im Ausführungsteil, also der Anwendung der Promotion, liegt der Fall anders. Hier gibt es nur wenige verschiedene Auswirkungen (zumeist Rabatte), die aber komplexer abzubilden und zu programmieren sind. Deswegen ist auch ein hybrider Ansatz denkbar: Die Bedingungen der Promotions basieren auf Business Rules, die Anwendung der Promotions hingegen wird durch eigene hybris Typen realisiert. Die Bedingungen der Promotions könnten auf diese Weise dynamisch mit allen Vorteilen der Business Rules definiert werden. Die Anwendung der Promotions wäre fest in Java-Quellcode definiert und würde, da sie auf dem hybris-Typsystem aufbaut, die Vorteile z.B. der persistenten Speicherung der Anwendung von Promotions auf eine Bestellung, mitbringen.

Usability Größtes Manko der regelbasierten gegenüber der typbasierten Lösung ist die Benutzerfreundlichkeit.

hmc Die gesamte Verwaltung der hybris Plattform läuft über die Administrationskonsole *hmc*. Deswegen ist es nicht denkbar, für die Pflege der Business Rules andere Werkzeuge (wie z.B. das von *JBoss Rules* bereitgestellte Plugin für die Eclipse IDE) zur Definition der Regeln zu verwenden. Mit der Administrationskonsole ist die Definition von Regeln nicht optimal. Es steht keinerlei Syntax-Highlighting, Syntax-Überprüfung, Code-Vervollständigung, *Import*-Verwaltung etc. zur Verfügung, wodurch die Regeldefinition aufwändig und fehleranfällig wird. Ohne Zugriff auf die Logfiles der

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

Anwendung ist für einen Sachbearbeiter nicht erkenntlich, ob definierte Regeln fehlerfrei kompiliert und angewendet werden, oder nicht.

Analog zur typbasierten Lösung für Virgin Megastores ist es auch bei der regelbasierten Lösung notwendig, Prioritäten zu vergeben, um Konflikte bei der Anwendung der Promotions zu vermeiden. Hierfür scheint es keine andere Möglichkeit zu geben.

DSL Die Verwendung einer *Domain Specific Language* erlaubt es, Regeln in natürlicher Sprache zu definieren. Dies erleichtert in erster Linie das Verständnis von bereits vorhandenen Regeln. Um neue Regeln zu definieren, ist es zumindest notwendig, die in der *DSL* definierten Mappings zu kennen. Mit der *JBoss Rules DSL* ist es nicht möglich eine Art fehlertolerantes Mapping zu definieren. Das bedeutet, dass die Regeln auf jedes Zeichen genau dem Mapping entsprechend angegeben werden müssen.

Um vollkommen neue Regeln (bzw. Bedingungen oder Ausführungszeilen) zu definieren, für die noch keine Mappings in der *DSL* existieren, ist genaue Kenntnis der hybriden API und *Drools Rule Language* erforderlich. Daher ist es zwar durchaus möglich zur Laufzeit vollkommene neue Promotions zu definieren, dies kann aber von einem Sachbearbeiter meist nicht geleistet werden.

imports Ein weiteres Problem für die Benutzbarkeit ist die Notwendigkeit Java-Klassen zu importieren. Ohne Kenntnis der API bzw. der package-Strukturen ist es nicht möglich diese *Imports* zu definieren. Deswegen kann dies auch nur durch einen erfahrenen Programmierer erfolgen.

Darüber hinaus wird durch die Verwendung einer *DSL* die dahinter liegende Technik verschleiert. Sobald eine Bedingung in natürlicher Sprache (z.B. „Datum zwischen dem '01.12.2008' und '31.12.2008'“) eingefügt wird, muss die Klasse *java.util.Date* importiert werden. Dies wird aber nur ersichtlich, wenn man das *DSL*-Mapping kennt.

Um Abhilfe zu schaffen empfiehlt es sich, in den *DSL*-Mappings alle Klassen implizit mit vollständiger package-Angabe zu versehen. Also statt die Klasse *Date* mit dem separat definierten *Import java.util.Date* direkt die Klasse *java.util.Date* angeben. Dadurch muss der Bearbeiter beim Hinzufügen von neuen, bereits in der *DSL* definierten Zeilen, nicht mehr mühsam die *Imports* erweitern.

Ein ähnliches Problem ist die Notwendigkeit, *eval*-Bedingungen immer am Ende des Bedingungssteils

2 Hauptteil

zu platzieren. Durch die Verwendung einer *DSL* ist nicht sofort erkennbar, ob es sich bei der Bedingung um einen herkömmlichen Objekt- bzw. Feldtest (*ObjectTypeNode* bzw. *field constraint*) oder um eine *eval*-Bedingung handelt.

JSR 94 Die Verwendung der *JSR 94* API bietet in der Theorie den Vorteil, dass die Rule Engine problemlos ausgetauscht werden kann. Durch eine einheitliche API muss beim Austauschen der Rule Engine kein Quellcode (bzw. nur sehr wenig Quellcode) verändert werden. In der Praxis bringt der Einsatz der *JSR 94* API nicht nur Vorteile. Einerseits kann durch den Einsatz der generellen API immer nur der kleinste gemeinsame Nenner der individuellen Fähigkeiten der Rule Engines verwendet werden. Hauptnachteil des *JSR 94*-Standards ist jedoch, dass lediglich eine einheitliche API, nicht jedoch eine einheitliche Sprache, um Regeln zu definieren, festgelegt wird. So verfügt jede Business Rule Engine über ihre eigene Regeldefinitionssprache, oft sogar über mehr als eine. Man kann davon ausgehen, dass der Hauptteil des Arbeitsaufwandes bei der Verwendung einer Business Rule Engine, in der Erstellung der Regeln und nicht in der Einbindung der Rule Engine in das Programm liegt. Würde man tatsächlich versuchen die Business Rule Engine auszutauschen, so könnte zwar ein Großteil des Quellcodes, der gemäß der *JSR 94* API gestaltet wurde, beibehalten werden. Der Großteil der Arbeit, die Regeln selbst, wären jedoch verloren bzw. müssten aufwändig portiert werden. Darüber hinaus muss, um strukturiert mit den Regeln arbeiten zu können, das Typsystem von *hybris* auch an die Struktur der jeweiligen Regelsprache angepasst werden.

Aus Sicht des Autors macht die Verwendung der *JSR 94* API noch keinen Sinn. Erst wenn zusätzlich zur *JSR 94* API auch eine einheitliche Regeldefinitionssprache zur Verfügung steht, kann der Austausch von Business Rule Engines untereinander sinnvoll werden. An einer einheitlichen Regeldefinitionssprache wird bereits im Rahmen der RuleML-Initiative gearbeitet. Bisher ist RuleML jedoch nicht ausgereift und wird nur von wenigen Business Rule Engines unterstützt.

Performance Bezüglich der Performance der Business Rule Engines bei der Verwendung für Promotions lässt sich keine eindeutige Aussage treffen. Gegenüber der projektspezifischen und komplett auf *hybris* Typen und Java-Code basierenden Lösung für Virgin Megastores kann folgendes festgestellt werden:

- Die kompilierten *Rete*-Netze und die zustandsbehafteten Auswertungssitzungen werden im Arbeitsspeicher gehalten, wodurch hier erhöhter Bedarf besteht.

2.3 Implementierung von Promotions mit Hilfe einer Business Rule Engine

- Gleichzeitig werden jedoch nicht so viele Objekte aus dem Cache bzw. der Datenbank der hybris Anwendung geladen, so dass der Cache bzw. die Datenbank entlastet wird.

Eigentlich entspricht die Auswertung von Promotions jedoch nicht dem eigentlichen Sinn von Business Rule Engines bzw. des zugrunde liegenden *Rete*-Algorithmus. Die Auswertung einer Promotion ist aus Anwendungssicht ein atomarer Vorgang, an dessen Ende ein Ergebnis, inkl. Anwendung einer Promotion, steht, oder nicht. Der für langfristige Auswertungen, mit sich dynamisch über einen längeren Zeitraum verändernden Objekten, entwickelte *Rete*-Algorithmus kann hierbei seine Leistungsfähigkeit nicht ausspielen.

Persistenz Die Anforderung, dass angewendete Promotions zu jeder Bestellung gespeichert werden und damit nachvollziehbar bleiben, ist teilweise erfüllt. Die auf einen Warenkorb bzw. eine Bestellung angewendeten Rabatte werden persistent gespeichert und können durch den übergebenen Code eindeutig einer Promotion zugeordnet werden. Kostenlose Produkte sind als Geschenk gekennzeichnet. Darüber hinaus wird jedoch nicht gespeichert durch welche Promotion das kostenlose Produkt in die Bestellung aufgenommen wurde.

Um diesen Umstand zu beseitigen müssen eigene hybris-Typen definiert werden, die dann durch die hybris Plattform in der Datenbank gespeichert werden können. Dadurch büßt man allerdings den eigentlich Vorteil der Business Rules ein, ohne zusätzliche hybris-Typen auszukommen.

mögliche Verbesserungen Bisher werden *Functions* direkt im Header der jeweiligen *RuleSet*-Objekte definiert. Wie das Beispiel der Datum-*Function* verdeutlicht, kann es aber durchaus sein, dass dieselbe *Function* von mehreren *RuleSet*-Objekten verwendet werden soll. Um redundanten Code zu vermeiden, wäre es sinnvoll, einen eigenen hybris-Typen für *Functions* anzulegen. So könnten die einzelnen *Function*-Objekte von mehreren *RuleSets* gleichzeitig verwendet werden.

Da die schlechte Benutzerfreundlichkeit die größte Schwachstelle der Business Rules darstellt, wäre es mittelfristig sinnvoll die Fähigkeiten der Regelerstellungs-Werkzeuge (wie das *JBoss Rules* Plugin für die Eclipse-IDE) in die Administrationskonsole zu integrieren. Da es sich bei der Administrationskonsole um eine Webanwendung handelt, wäre der Aufwand hierfür allerdings enorm hoch und im Rahmen der vorliegenden Arbeit nicht auch nur ansatzweise durchführbar.

3 Zusammenfassung

In der vorliegenden Arbeit wurde dargelegt, was für Anforderungen es seitens der Online-Shop-Betreiber bezüglich Promotions gibt, wie diese Anforderungen in einem aktuellen Großprojekt umgesetzt wurden und wie man die annähernd gleiche Funktionalität mit Hilfe von Business Rule Engines modellieren kann.

Bei den Anforderungen steht nicht nur die Funktionalität selbst, also welche Promotions überhaupt angeboten werden können, im Vordergrund. Von enormer Bedeutung sind auch Fähigkeiten, die für den Kunden des Onlineshops nicht wahrnehmbar sind. Dazu gehört, auch aus rechtlichen Gründen, die persistente Speicherung und Rückverfolgbarkeit der angewendeten Promotions zu den Bestellungen. Vor Allem ist jedoch die Pflegbarkeit und Benutzerfreundlichkeit entscheidend.

Die projektspezifische Implementierung im Virgin Megastores Projekt basiert komplett auf dem hybris Typsystem. Jede Promotion wird als Instanz eines hybris Typen angelegt und in der Datenbank gespeichert. Gepflegt werden die Promotions wie jeder andere hybris Typ auch in der Administrationskonsole. Dort werden die Attributwerte der jeweiligen Promotion einfach eingetragen.

Bei der regelbasierten Lösung werden die Regeln selbst auch als Instanzen von hybris Typen gespeichert. Beim ersten Aufruf werden die Regeln jedoch zum *Rete*-Netz kompiliert und danach im flüchtigen Arbeitsspeicher gehalten. Sobald Regeln in der Administrationskonsole geändert werden, muss das jeweilige Regel-Set beim nächsten Aufruf neu kompiliert und geladen werden. Die Regeln selbst werden dabei in der für JBoss Rules spezifischen Regeldefinitionssprache Drools Rule Language definiert. Es ist jedoch möglich mit einem entsprechenden Mapping die Regeln auch in natürlicher Sprache anzugeben.

3.1 Beurteilung

Vom Funktionsumfang her ist es möglich, mit dem regelbasierten Ansatz die gleichen Promotions wie mit der typbasierten Implementierung abzubilden. Darüber hinaus sind die Regeln wesentlich einfacher zu erweitern als die projektspezifische Virgin Megastores Lösung. Für die Abbildung der Promotions selbst ist die regelbasierte durchaus der projektspezifischen Lösung ebenbürtig bzw. sogar überlegen.

Jedoch ist jede noch so „mächtige“ Promotion-Funktionalität unnütz, wenn sie durch die eigenen Mitarbeiter nicht sinnvoll und fehlerfrei bedient werden kann. Hier liegt der große Vorteil der projektspezifischen Implementierung mit eigenen hybris-Typen für jede Promotion. Diese Promotions sind intuitiv durch Sachbearbeiter ohne besondere technische Kenntnisse pflegbar. Vor Allem ist eine Fehlbedienung praktisch ausgeschlossen. Im Gegensatz dazu ist es bei der auf JBoss Rules basierten Lösung für ungeübte und technisch weniger versierte Bearbeiter schwierig, eine auf Anhieb funktionierende Promotion zu definieren. Dies ist in erster Linie dem Umstand geschuldet, dass die Administrationskonsole nicht die Funktionalitäten, wie die von den Herstellern der Rule Engines angebotenen Werkzeuge, bieten kann.

Im Rahmen dieser Arbeit ist eine funktional sehr mächtige regelbasierte Lösung für Promotions entstanden. Die Promotions können wie gewünscht abgebildet werden. Die ebenfalls sehr wichtigen sekundären Anforderungen wie Persistenz, Rückverfolgbarkeit, Call-Center-Fähigkeit und allen voran Usability können bisher allerdings nicht ausreichend abgedeckt werden.

Vor diesem Hintergrund ist die typbasierte Lösung zumindest für Promotions der regelbasierten Lösung vorzuziehen. Sie mag zwar weniger flexibel und umfangreich sein, dafür jedoch wesentlich einfacher zu bedienen. Solange die Art der angebotenen Promotions gleich bleibt und nicht häufig neuartige Promotions hinzugefügt werden müssen, tritt der Nachteil der mangelnden Flexibilität selten in Erscheinung. Nur wenn häufig vollkommen neue Promotions definiert werden, lohnt sich die Verwendung von Business Rules. Die Erfahrung im Virgin Megastores Projekt lehrt jedoch, dass, sofern zu Beginn die benötigten Promotions gründlich definiert werden, Erweiterungen im Nachhinein nur sehr selten notwendig sind.

3.2 Ausblick

Wie bereits erwähnt wird die Business Rule Extension in einem kommerziellen Projekt erfolgreich eingesetzt. Insbesondere an Stellen, wo das Verhalten der Anwendung dynamisch, kurzfristig und zur Laufzeit verändert werden muss, kann der Einsatz von Business Rules vorteilhaft sein. Die im Rahmen der Arbeit erstellte *rules*-Extension ist so implementiert, dass sie auch ohne Änderungen der Extension für andere Zwecke als die der Promotions benutzt werden kann.

Um die Arbeit mit der *rules*-Extension weiter zu erleichtern, wäre es in erster Linie sinnvoll und notwendig, die Integration in die Administrationsoberfläche zu verbessern. Insbesondere eine Überprüfung der Syntax auf Fehler inkl. entsprechenden Fehlermeldungen wäre wünschenswert. Aufgrund der verwendeten Technologie sind diese Verbesserungen jedoch nur mit großem Aufwand zu erreichen.

A Anhang

A.1 Promotion-Extension

A.1.1 Itemdefinition

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
  Hybris Promotions Engine

  Copyright (c) 2007 Neoworks Limited
  All rights reserved.
-->
<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="items.xsd">

  [...]

  <itemtypes>

    <itemtype code="PromotionResult" extends="GenericItem"
      jaloclass="de.hybris.jakarta.ext.promotions.jalo.PromotionResult"
      deployment="de.hybris.jakarta.entity.PromotionResult"
      autocreate="true" generate="true">

      <attributes>

        <attribute autocreate="true" qualifier="order"
          type="AbstractOrder">
          <description>The order </description>
          <persistence type="property" />
        </attribute>
        <attribute autocreate="true" qualifier="actions"
          type="PromotionActionCollectionType">
          <description>List of actions </description>
          <persistence type="jalo" />
          <modifiers read="true" write="true" search="false"
            initial="false" optional="true" partof="true" />
        </attribute>
        <attribute autocreate="true" qualifier="consumedEntries"
          type="PromotionOrderEntryConsumedCollectionType">
          <description>List of consumed entries </description>
          <persistence type="jalo" />
          <modifiers read="true" write="true" search="false"
            initial="false" optional="true" partof="true" />
      </attributes>
    </itemtype>
  </itemtypes>
</items>
```

```

    </attribute >
    <attribute autocreate="true" qualifier="promotion"
        type="AbstractPromotion">
        <description>The promotion </description >
        <persistence type="property" />
    </attribute >
    <attribute autocreate="true" qualifier="certainty"
        type="java.lang.Float">
        <defaultvalue >
            new java.lang.Float(0.0F)
        </defaultvalue >
        <description>Certainty </description >
        <persistence type="property" />
    </attribute >
    <attribute autocreate="true" qualifier="custom"
        type="java.lang.String">
        <description >
            Custom data stored on this promotion result by
            the promotion.
        </description >
        <modifiers read="true" write="true" />
        <persistence type="property" />
    </attribute >

</attributes >

[...]

</itemtype >

<!-- *** PROMOTION ACTIONS *** -->

<itemtype code="AbstractPromotionAction" extends="GenericItem"
    jaloclass="de.hybris.jakarta.ext.promotions.jalo.AbstractPromotionAction"
    deployment="de.hybris.jakarta.entity.AbstractPromotionAction"
    autocreate="true" generate="true">
<attributes >

    <attribute qualifier="markedApplied" autocreate="true"
        type="java.lang.Boolean">
        <description >
            Flag to indicate that this promotion is applied.
        </description >
        <defaultvalue >Boolean.FALSE</defaultvalue >
        <persistence type="property" />
        <modifiers read="true" write="true"
            optional="false" />
    </attribute >
    <attribute autocreate="true" qualifier="guid"
        type="java.lang.String">
        <description >
            The unique identifier for this action.
        </description >

```

```

        <modifiers read="true" write="true" />
        <persistence type="property" />
    </attribute>
    <attribute autocreate="true" qualifier="promotionResult"
        type="PromotionResult">
        <description>The promotion result.</description>
        <persistence type="property" />
    </attribute>

</attributes>

[...]

</itemtype>

[...]

<itemtype code="PromotionOrderAddFreeGiftAction"
    extends="AbstractPromotionAction"
    jaloclass="de.hybris.jakarta.ext.promotions.jalo.PromotionOrderAddFreeGiftAct
    autocreate="true" generate="true">
    <attributes>
        <attribute autocreate="true" qualifier="freeProduct"
            type="Product">
            <description>
                The product given away as a gift
            </description>
            <modifiers read="true" write="true" search="true"
                optional="false" />
            <persistence type="property" />
        </attribute>
    </attributes>
</itemtype>

[...]

<!-- *** PROMOTION ORDER ENTRY CONSUMED *** -->

[...]

<!-- *** PROMOTION PRICE ROW *** -->

[...]

<!-- *** PROMOTION QUANTITY PRICES ROW *** -->

[...]

<!-- *** RESTRICTIONS *** -->

<itemtype code="AbstractPromotionRestriction"
    extends="GenericItem"
    jaloclass="de.hybris.jakarta.ext.promotions.jalo.AbstractPromotionRestriction

```

A Anhang

```
deployment="de.hybris.jakarta.entity.AbstractPromotionRestriction"
autocreate="true" generate="true">

<attributes>

  <attribute autocreate="true" qualifier="title"
    type="java.lang.String">
    <persistence type="property" />
    <description>
      Title for this restriction
    </description>
  </attribute>

  <attribute qualifier="description"
    type="java.lang.String">
    <description>
      Description of this restriction
    </description>

    [...]
  </attribute>

  <attribute autocreate="true" qualifier="promotion"
    type="AbstractPromotion">
    <persistence type="property" />
    <description>
      The promotion that this restriction is part of
    </description>
  </attribute>

</attributes>

[...]

</itemtype>

<itemtype code="PromotionProductRestriction"
  extends="AbstractPromotionRestriction"
  jaloclass="de.hybris.jakarta.ext.promotions.jalo.PromotionProductRestriction"
  autocreate="true" generate="true">

  <attributes>
    <attribute autocreate="true" generate="true"
      qualifier="products" type="ProductCollection">
      <description>
        The products that the promotion should not be
        applied to
      </description>
      <persistence type="property" />
      <modifiers read="true" write="true" search="false"
        optional="true" />
    </attribute>
  </attributes>
```

```

</itemtype>

[...]

<!-- *** PROMOTIONS *** -->

<itemtype code="AbstractPromotion" extends="GenericItem"
  jaloclass="de.hybris.jakarta.ext.promotions.jalo.AbstractPromotion"
  deployment="de.hybris.jakarta.entity.AbstractPromotion"
  autocreate="true" generate="true">

  <attributes>

    <attribute autocreate="true" qualifier="code"
      type="java.lang.String">
      <description>
        Identifier for this promotion
      </description>
      <persistence type="property" />
    </attribute>

    <attribute autocreate="true" qualifier="title"
      type="java.lang.String">
      <description>Title for this promotion</description>
      <persistence type="property" />
    </attribute>

    <attribute qualifier="description"
      type="java.lang.String">
      <description>
        Description of this promotion
      </description>

      [...]
    </attribute>

    <attribute autocreate="true" qualifier="startDate"
      type="java.util.Date">
      <description>
        Date on which this promotion becomes available ,
        if not set the promotion will not be available .
      </description>
      <defaultvalue>
        de.hybris.jakarta.ext.promotions.util.Helper.buildDateForYear(2000)
      </defaultvalue>
      <modifiers read="true" write="true" search="true"
        optional="true" />
      <persistence type="property" />
    </attribute>

    <attribute autocreate="true" qualifier="endDate"
      type="java.util.Date">

```

```

<description>
    Date on which this promotion stops being
    available , if not set the promotion will not be
    available .
</description>
<defaultvalue>
    de . hybris . jakarta . ext . promotions . util . Helper . buildDateForYear (2099)
</defaultvalue>
<modifiers read="true" write="true" search="true"
    optional="true" />
<persistence type="property" />
</attribute >

<attribute autocreate="true" qualifier="detailsURL"
    type="java . lang . String">
<description>
    URL to a content page with further details of
    this promotion
</description>
<persistence type="property" />
</attribute >

<attribute qualifier="restrictions" autocreate="true"
    type="PromotionRestrictionsCollection">
<description>
    Collection of restrictions that are applied to
    this promotion
</description>
<persistence type="jalo" />
<modifiers partof="true" />
</attribute >

<attribute qualifier="enabled" autocreate="true"
    type="java . lang . Boolean">
<description>
    Flag to indicate if this promotion is enabled .
</description>
<defaultvalue>Boolean .FALSE</defaultvalue >
<persistence type="property" />
<modifiers read="true" write="true"
    optional="false" />
</attribute >

<attribute autocreate="true" qualifier="priority"
    type="java . lang . Integer">
<defaultvalue>
    new java . lang . Integer (500)
</defaultvalue >
<description>
    Value to indicate relative priority of
    promotions . The higher the value the higher the
    priority .
</description >

```

```

        <persistence type="property" />
    </attribute >

    [...]

</attributes >

[...]

</itemtype >

<itemtype code="ProductPromotion" extends="AbstractPromotion"
    jaloclass="de.hybris.jakarta.ext.promotions.jalo.ProductPromotion"
    autocreate="true" generate="true">

    <attributes >

        <attribute autocreate="true" qualifier="productBanner"
            type="Media">
            <description >
                Media to display on the product page when this
                promotion is available.
            </description >
            <persistence type="property" />
            <modifiers read="true" write="true" search="true"
                optional="true" />
        </attribute >

        <!--
            Priority attribute redeclared to change the default
            priority for ProductPromotion to 1000
            and therefore above (and higher priority than)
            OrderPromotions.
        -->
        <attribute redeclare="true" autocreate="true"
            qualifier="priority" type="java.lang.Integer">
            <defaultvalue >
                new java.lang.Integer(1000)
            </defaultvalue >
            <description >
                Value to indicate relative priority of
                promotions. The higher the value the higher the
                priority.
            </description >
            <persistence type="property" />
        </attribute >

    </attributes >
</itemtype >

<itemtype code="OrderPromotion" extends="AbstractPromotion"
    jaloclass="de.hybris.jakarta.ext.promotions.jalo.OrderPromotion"
    autocreate="true" generate="true">

```

```

    <attributes />

</itemtype>

<!-- *** PRODUCT PROMOTIONS *** -->

[...]

<itemtype code="ProductBOGOFPromotion"
    extends="ProductPromotion"
    jaloclass="de.hybris.jakarta.ext.promotions.jalo.ProductBOGOFPromotion"
    autocreate="true" generate="true">

    <attributes>

        <attribute autocreate="true" qualifier="qualifyingCount"
            type="java.lang.Integer">
            <defaultvalue>
                new java.lang.Integer(2)
            </defaultvalue>
            <description>
                The number of products required in the cart to
                activate the promotion. (For standard BOGOF this
                is 2).
            </description>
            <modifiers read="true" write="true" search="true"
                optional="true" />
            <persistence type="property" />
        </attribute>

        <attribute autocreate="true" qualifier="freeCount"
            type="java.lang.Integer">
            <defaultvalue>
                new java.lang.Integer(1)
            </defaultvalue>
            <description>
                The number of products within the cart to give
                away free. (For standard BOGOF this is 1).
            </description>
            <modifiers read="true" write="true" search="true"
                optional="true" />
            <persistence type="property" />
        </attribute>

        <attribute qualifier="messageFired"
            type="localized:java.lang.String">
            <defaultvalue>
                de.hybris.jakarta.ext.promotions.util.Helper.buildLocalisedStringDefa
                "en", "These items qualify for our buy
                {0,number,integer} get {1,number,integer} free
                offer – You have saved {3}" )
            </defaultvalue>

```

```

    <description>
        The message to show when the promotion has
        fired.
    </description>
    <modifiers read="true" write="true" optional="true" />
    <persistence type="property" />
</attribute>

<attribute qualifier="messageCouldHaveFired"
    type="localized:java.lang.String">
    <defaultvalue>
        de.hybris.jakarta.ext.promotions.util.Helper.buildLocalisedStringDefa
        "en", "Buy {0,choice,1#one more
        item|1&lt;another {0,number,integer} items} to
        qualify for our buy {1,number,integer} get
        {2,number,integer} free offer" )
    </defaultvalue>
    <description>
        The message to show when the promotion could
        have potentially fire.
    </description>
    <modifiers read="true" write="true" optional="true" />
    <persistence type="property" />
</attribute>

</attributes>
</itemtype>

[...]

<!-- *** ORDER PROMOTIONS *** -->

[...]

<itemtype code="OrderThresholdFreeGiftPromotion"
    extends="OrderPromotion"
    jaloclass="de.hybris.jakarta.ext.promotions.jalo.OrderThresholdFreeGiftPromot
    autocreate="true" generate="true">
<attributes>

    <attribute autocreate="true" qualifier="thresholdTotals"
        type="PromotionPriceRowCollectionType">
        <description>
            The cart total value threshold in specific
            currencies.
        </description>
        <persistence type="property" />
        <modifiers read="true" write="true" search="false"
            initial="false" optional="true" partof="true" />
    </attribute>

    <attribute autocreate="true" qualifier="giftProduct"
        type="Product">

```

```

        <description>
            The free gift product to add to the cart.
        </description>
        <modifiers read="true" write="true" search="true"
            optional="true" />
        <persistence type="property" />
    </attribute>

    <attribute qualifier="messageFired"
        type="localized:java.lang.String">
        <defaultvalue>
            de.hybris.jakarta.ext.promotions.util.Helper.buildLocalisedStringDefault(
                "en", "You got a free ProductName for spending
                over {1}" )
        </defaultvalue>
        <description>
            The message to show when the promotion has
            fired.
        </description>
        <modifiers read="true" write="true" optional="true" />
        <persistence type="property" />
    </attribute>

    <attribute qualifier="messageCouldHaveFired"
        type="localized:java.lang.String">
        <defaultvalue>
            de.hybris.jakarta.ext.promotions.util.Helper.buildLocalisedStringDefault(
                "en", "Spend {1} to get a free ProductName –
                Spend another {3} to qualify" )
        </defaultvalue>
        <description>
            The message to show when the promotion could
            have potentially fire.
        </description>
        <modifiers read="true" write="true" optional="true" />
        <persistence type="property" />
    </attribute>

    </attributes>
</itemtype>

[... ]

<!-- *** ABSTRACT ORDER *** -->

<!-- Extend AbstractOrder by adding in the ability to store a
previous delivery mode -->
[... ]

<!-- *** CREATE PROMOTION WIZARDS *** -->
[... ]

</itemtypes>

```

`</items>`

Listing A.1: items.xml der Promotion-Extension

A.1.2 Java Klassen

```
/*
 * Hybris Promotions Engine
 *
 * Copyright (c) 2007 Neoworks Limited
 */

package de.hybris.jakarta.ext.promotions.jalo;

[...]

public class PromotionsManager extends GeneratedPromotionsManager
{
    [...]

    public List<ProductPromotion> getProductPromotions(final
        SessionContext ctx, final Product product, final boolean
        evaluateRestrictions, Date date)
    {
        try
        {
            if (log.isDebugEnabled())
            {
                log.debug("getProductPromotions for [" + product + "]
                    evaluateRestrictions=[" + evaluateRestrictions + "]
                    date=[" + date + "]");
            }

            if (product != null)
            {
                if (date == null)
                {
                    // Default date to now
                    date = new Date();
                }

                // Build query to find all distinct list of all promotions
                // that are related to
                // the source product either via the
                // ProductPromotionRelation or via
                // the CategoryPromotionRelation.
                // Filter the results so that only Promotions with Start and
                // End dates valid for time 'now'
                // order by Priority with the highest value first
                final String query =
                    "SELECT DISTINCT {promo:" + Item.PK + "}, {promo:" +
                    AbstractPromotion.PRIORITY + "}" +
                    "FROM {" +
                    TypeManager.getInstance().getComposedType(ProductPromotion.class)
                    + " as promo}" +
                    "WHERE " +
                    "( " +
```

```

" {promo:" + Item.PK + "} IN ( {{ SELECT
  {prod2promo:" + Link.TARGET + "} " +
" FROM {" +
  PromotionsConstants.Relations.PRODUCTPROMOTIONRELATION
+ " AS prod2promo} " +
" WHERE {prod2promo:" +
  Link.SOURCE + "} = ?product }} ) " +
" OR " +
" {promo:" + Item.PK + "} IN ( {{ SELECT {cat2promo:"
+ Link.TARGET + "} " +
" FROM {" +
  PromotionsConstants.Relations.CATEGORYPROMOTIONRELATION
+ " AS cat2promo " +
" LEFT JOIN " +
  CategoryConstants.Relations.CATEGORYPRODUCTRELATION
+ " as cat2prod ON {cat2promo." + Link.SOURCE +
  "}"={cat2prod:" + Link.SOURCE + "}" } " +
" WHERE {cat2prod:" + Link.TARGET
+ "}" = ?product }} ) " +
") " +
"AND " +
"( " +
" {promo:" + AbstractPromotion.ENABLED + "}=1 AND
  {promo:" + AbstractPromotion.STARTDATE + "} <= ?now
AND ?now <= {promo:" + AbstractPromotion.ENDDATE +
  "}" " +
") " +
"ORDER BY {promo:" + AbstractPromotion.PRIORITY + "}
DESC";

```

```

final Flat3Map args = new Flat3Map();
args.put("product", product);
args.put("now", date);

final Collection<ProductPromotion> allPromotions =
  getSession().getFlexibleSearch().search(ctx, query, args,
  Collections.singletonList(ProductPromotion.class), true,
  false, 0, -1).getResult();

final ArrayList<ProductPromotion> availablePromotions = new
  ArrayList<ProductPromotion>(allPromotions.size());

for (ProductPromotion promotion : allPromotions)
{
  boolean satisfiedRestrictions = true;

  // Run Restrictions checks
  if (evaluateRestrictions)
  {
    final Collection<AbstractPromotionRestriction>
      restrictions = promotion.getRestrictions();
    if (restrictions != null)
    {

```

```

        for (AbstractPromotionRestriction restriction :
            restrictions)
        {
            // Check restriction
            final
                AbstractPromotionRestriction.RestrictionResult
                result = restriction.evaluate(ctx, product,
                    date, null);
            if (result ==
                AbstractPromotionRestriction.RestrictionResult.DENY
                || result ==
                AbstractPromotionRestriction.RestrictionResult.ADJUSTED_PRO
            {
                if (log.isDebugEnabled())
                {
                    log.debug("getProductPromotions for [" +
                        product + "]. Promotion [" + promotion
                        + "] blocked by restriction [" +
                        restriction + "]");
                }

                satisfiedRestrictions = false;
                break;
            }
        }
    }

    if (satisfiedRestrictions)
    {
        availablePromotions.add(promotion);
    }
}

// Dump out list of available promotions
if (log.isDebugEnabled())
{
    for (ProductPromotion promotion : availablePromotions)
    {
        log.debug("getProductPromotions for [" + product + "]
            available promotion [" + promotion + "]");
    }
}

return availablePromotions;
}
}
catch (Exception ex)
{
    log.error("Failed to getProductPromotions", ex);
}
return new ArrayList<ProductPromotion>(0);

```

```

}

[...]

public PromotionOrderResults updatePromotions(final SessionContext
    ctx, final AbstractOrder order, final boolean
    evaluateRestrictions, final AutoApplyMode productPromotionMode,
    final AutoApplyMode orderPromotionMode, Date date)
{
    try
    {
        if (log.isDebugEnabled())
        {
            log.debug("updatePromotions for [" + order + "]
                evaluateRestrictions=[" + evaluateRestrictions + "]
                productPromotionMode=[" + productPromotionMode + "]
                orderPromotionMode=[" + orderPromotionMode + "] date=[" +
                date + "]");
        }

        if (order != null)
        {
            if (!order.isCalculated(ctx))
            {
                log.error("updatePromotions order [" + order + "] not
                    calculated, calculating");
                order.calculate(date);
            }

            if (date == null)
            {
                // Default value is Now
                date = new Date();
            }

            // Record list of promotions to keep applied if
            // AutoApplyMode.KEEP_APPLIED is specified
            final List<String> promotionResultsToKeepApplied = new
                ArrayList<String>();

            // Find the current total value of promotions active
            final List<PromotionResult> currResults =
                this.getPromotionResultsInternal(ctx, order);
            double oldTotalAppliedDiscount = 0;
            if (currResults != null && !currResults.isEmpty())
            {
                for(PromotionResult pr : currResults)
                {
                    if(pr.getFired(ctx))
                    {
                        final boolean prApplied = pr.isApplied(ctx);
                        if (prApplied)
                        {

```

```

        // We want to capture the total applied discount
        // even for promotion results that are now
        // invalid.
        oldTotalAppliedDiscount +=
            pr.getTotalDiscount(ctx);
    }

    if (pr.isValid(ctx))
    {
        if ( ( (productPromotionMode ==
            AutoApplyMode.KEEP_APPLIED &&
            pr.getPromotion(ctx) instanceof
            ProductPromotion) ||
            (orderPromotionMode ==
            AutoApplyMode.KEEP_APPLIED &&
            pr.getPromotion(ctx) instanceof
            OrderPromotion) ) &&
            prApplied )
        {
            final String prKey = pr.getDataUnigueKey(ctx);
            if (prKey != null && prKey.length() > 0)
            {
                if (log.isDebugEnabled())
                {
                    log.debug("updatePromotions found
                        applied PromotionResult [" + pr + "]
                        key [" + prKey + "] that should be
                        reapplied");
                }
                promotionResultsToKeepApplied.add(prKey);
            }
        }
    }
}

// Delete any results stored from a previous run
deleteStoredPromotionResults(ctx, order, true);

// Find all runnable promotions in the system

// Get the list of base products in the cart
final Collection<Product> products =
    getBaseProductsForOrder(ctx, order);

// Find the promotions that can be evaluated
// will find all OrderPromotions and any ProductPromotions
// that are related to the products specified
final List<AbstractPromotion> activePromotions =
    findOrderAndProductPromotionsSortByPriority(ctx,
        getSession(), products, date);

```

```

if (log.isDebugEnabled())
{
    log.debug("updatePromotions found [" +
        activePromotions.size() + "] promotions to run");
}

final List<PromotionResult> results = new
    LinkedList<PromotionResult>();

double newTotalAppliedDiscount = 0.0D;
if (!activePromotions.isEmpty())
{
    final PromotionEvaluationContext promoContext = new
        PromotionEvaluationContext(order,
            evaluateRestrictions, date);

    for (AbstractPromotion promotion : activePromotions)
    {
        if (log.isDebugEnabled())
        {
            log.debug("updatePromotions evaluating promotion [" +
                promotion + "]");
        }
        final List<PromotionResult> promoResults =
            promotion.evaluate(ctx, promoContext);
        if (log.isDebugEnabled())
        {
            log.debug("updatePromotions promotion [" +
                promotion + "] returned [" + promoResults.size()
                + "] results");
        }

        // Work out if we need to apply this promotion
        boolean autoApply = false;
        boolean keepApplied = false;

        if ( (productPromotionMode == AutoApplyMode.APPLY_ALL
            && orderPromotionMode == AutoApplyMode.APPLY_ALL) ||
            (productPromotionMode == AutoApplyMode.APPLY_ALL
            && promotion instanceof ProductPromotion) ||
            (orderPromotionMode == AutoApplyMode.APPLY_ALL &&
            promotion instanceof OrderPromotion) )
        {
            autoApply = true;
        }
        else if ( (productPromotionMode ==
            AutoApplyMode.KEEP_APPLIED && orderPromotionMode ==
            AutoApplyMode.KEEP_APPLIED) ||
            (productPromotionMode ==
            AutoApplyMode.KEEP_APPLIED && promotion
            instanceof ProductPromotion) ||
            (orderPromotionMode ==
            AutoApplyMode.KEEP_APPLIED && promotion

```

```

        instanceof OrderPromotion) )
    {
        keepApplied = true;
    }

    boolean needsCalculateTotals = false;

    if (autoApply || keepApplied)
    {
        // Apply the promotion results if required
        for (PromotionResult pr : promoResults)
        {
            if (pr.getFired(ctx))
            {
                if (autoApply)
                {
                    if (log.isDebugEnabled())
                    {
                        log.debug("updatePromotions auto
                            applying result [" + pr + "] from
                            promotion [" + promotion + "]");
                    }
                    needsCalculateTotals |= pr.apply(ctx);

                    // Add this promotion to the new total
                    newTotalAppliedDiscount +=
                        pr.getTotalDiscount(ctx);
                }
            }
            else if (keepApplied)
            {
                final String prKey =
                    pr.getDataUnigueKey(ctx);
                if (prKey == null || prKey.length() == 0)
                {
                    log.error("updatePromotions promotion
                        result [" + pr + "] from promotion
                        [" + promotion + "] returned NULL or
                        Empty DataUnigueKey");
                }
            }
            else
            {
                // See if the promotion result is in
                // the list of promotions to keep
                // applied
                if
                    (promotionResultsToKeepApplied.remove(prKey))
                {
                    if (log.isDebugEnabled())
                    {
                        log.debug("updatePromotions
                            keeping applied the result ["
                                + pr + "] from promotion [" +

```

```

                promotion + "]");
            }
            needsCalculateTotals |=
                pr.apply(ctx);

            // Add this promotion to the new
            // total
            newTotalAppliedDiscount +=
                pr.getTotalDiscount(ctx);
        }
    }
}

if (needsCalculateTotals)
{
    order.calculateTotals(true);
}

results.addAll(promoResults);
}
}

// Log all the PromotionResults that could not be reapplied.
if (log.isDebugEnabled())
{
    for(String prKey : promotionResultsToKeepApplied)
    {
        log.debug("updatePromotions PromotionResult not
            reapplied because it did not fire [" + prKey + "]");
    }
}

final double appliedDiscountChange = newTotalAppliedDiscount
    - oldTotalAppliedDiscount;

if (log.isDebugEnabled())
{
    log.debug("updatePromotions for [" + order + "] returned
        [" + results.size() + "] PromotionResults
        appliedDiscountChange=[" + appliedDiscountChange +
            "]");
}

return new PromotionOrderResults(ctx, order,
    Collections.unmodifiableList(results),
    appliedDiscountChange);
}
}
catch (Exception ex)
{

```

A Anhang

```
        log.error("Failed to updatePromotions", ex);
    }
    return null;
}

/**
 * Which changes should the promotions manager apply to an order
 * automatically.
 * <p/>
 * Used in updatePromotions. The mode is specified seperatly for
 * product and order promotions.
 */
public enum AutoApplyMode
{
    /**
     * Do not apply any promotions.
     */
    APPLY_NONE,

    /**
     * Reapply promotions that are currently applied if they can still
     * be applied.
     */
    KEEP_APPLIED,

    /**
     * Apply all promotions that can be applied.
     */
    APPLY_ALL
}

[...]

private static List<AbstractPromotion>
findOrderAndProductPromotionsSortByPriority(final SessionContext
ctx, final JaloSession jaloSession, final Collection<Product>
products, final Date date)
{
    String query = "" +
        "SELECT DISTINCT {promo:" + Item.PK + "}, {promo:" +
        AbstractPromotion.PRIORITY + "}" +
        "FROM {" +
        TypeManager.getInstance().getComposedType(AbstractPromotion.class).getC
        + " as promo}" +
        "WHERE " +
        "( " +
        " {promo:" + AbstractPromotion.ENABLED + "}" = 1 AND
        {promo:" + AbstractPromotion.STARTDATE + "}" <= ?now AND
        ?now <= {promo:" + AbstractPromotion.ENDDATE + "}" " +
        ") " +
        "AND " +
        "( ";
```

```

if (products != null && !products.isEmpty())
{
    // Add query for ProductPromotions
    query +=
        " ( " +
        "     {promo:" + Item.PK + "} IN ( " +
        "         {{ " +
        "             SELECT {p2:" + Item.PK + "} FROM {" +
        "                 TypeManager.getInstance().getComposedType(ProductPromotion.class).ge
        "             + " as p2}" +
        "         }} " +
        "     )" +
        "     AND " +
        "     ( " +
        "         {promo:" + Item.PK + "} IN ( " +
        "             {{ " +
        "                 SELECT {prod2promo:" + Link.TARGET + "} FROM
        "             {" +
        "                 PromotionsConstants.Relations.PRODUCTPROMOTIONRELATION
        "             + " AS prod2promo}" +
        "             WHERE {prod2promo:" + Link.SOURCE + "} IN
        "             (?productPks) " +
        "             }} " +
        "         )" +
        "     OR " +
        "     {promo:" + Item.PK + "} IN ( " +
        "         {{ " +
        "             SELECT {cat2promo:" + Link.TARGET + "} " +
        "             FROM {" +
        "                 CategoryConstants.Relations.CATEGORYPRODUCTRELATION +
        "             AS cat2prod " +
        "             JOIN " +
        "                 PromotionsConstants.Relations.CATEGORYPROMOTIONRELATION
        "             + " AS cat2promo ON {cat2prod:" + Link.SOURCE +
        "             "}= {cat2promo:" + Link.SOURCE + "} } " +
        "             WHERE {cat2prod:" + Link.TARGET + "} IN
        "             (?productPks) " +
        "             }} " +
        "         )" +
        "     )" +
        " );";
}
else
{
    // Skip products query
    query += " (1=0) ";
}

// Add query for OrderPromotions
query +=
    " OR " +
    " ( " +
    "     {promo:" + Item.PK + "} IN ( " +

```

A Anhang

```
        {{ " +
        "      SELECT {p3:" + Item.PK + "} FROM {" +
        "      TypeManager.getInstance().getComposedType(OrderPromotion.class).getCode
        + " as p3} " +
        "      }}" +
        "    )" +
        "  ) ";

// Close AND and add OrderBy
query +=
    ") " +
    "ORDER BY {promo:" + AbstractPromotion.PRIORITY + "} DESC";

// log.debug("findOrderAndProductPromotionsSortByPriority query
    [[{" + query + "}]");

final Flat3Map queryParams = new Flat3Map();
queryParams.put("now", date);
queryParams.put("productPks", products);

return jaloSession.getFlexibleSearch().search(ctx, query,
    queryParams, AbstractPromotion.class).getResult();
}

[...]
}
```

Listing A.2: PromotionsManager.java

A.2 Rules-Extension

A.2.1 Itemdefinition

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!--
```

```
[y] hybris E-Business Platform
```

```
Copyright (c) 2000-2007 hybris Holding AG
```

```
All rights reserved.
```

This software is the confidential and proprietary information of hybris ("Confidential Information"). You shall not disclose such Confidential Information and shall use it only in accordance with the terms of the license agreement you entered into with hybris.

```
—>
```

```
<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="items.xsd">
```

```
[...]
```

```
<itemtypes>
```

```
  <itemtype code="RuleSet"
    extends="GenericItem"
    autocreate="true"
    generate="true">
    <attributes>
      <attribute qualifier="name" autocreate="true"
        type="java.lang.String">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
          optional="false" unique="true"/>
      </attribute>
      <attribute qualifier="description" autocreate="true"
        type="localized:java.lang.String">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
          optional="true"/>
      </attribute>
      <attribute qualifier="header" autocreate="true"
        type="java.lang.String">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
          optional="false"/>
      </attribute>
      <attribute qualifier="dsl" autocreate="true" type="DSL">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
          optional="true"/>
    </attributes>
  </itemtype>
</itemtypes>
```

```

        </attribute >
        <attribute qualifier="rules" autocreate="true"
            type="RulesCollection">
            <persistence type="property"/>
            <modifiers read="true" write="true" search="true"
                optional="true"/>
        </attribute >
    </attributes >
</itemtype >

<itemtype code="Rule"
    extends="GenericItem"
    autocreate="true"
    generate="true"
    deployment="com.arithnea.rules.entity.Rule">
<attributes >
    <attribute qualifier="name" autocreate="true"
        type="java.lang.String">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="false" unique="true"/>
    </attribute >
    <attribute qualifier="description" autocreate="true"
        type="localized:java.lang.String">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="true"/>
    </attribute >
    <attribute qualifier="lhs" autocreate="true"
        type="java.lang.String">
        <persistence type="property"/>
        [...]
        <modifiers read="true" write="true" search="true"
            optional="false"/>
    </attribute >
    <attribute qualifier="rhs" autocreate="true"
        type="java.lang.String">
        <persistence type="property"/>
        [...]
        <modifiers read="true" write="true" search="true"
            optional="false"/>
    </attribute >
    <attribute qualifier="optionalAttributes" autocreate="true"
        type="RulesAttributeMap">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="true"/>
    </attribute >
    <attribute qualifier="imports" autocreate="true"
        type="StringCollection">
        <persistence type="property"/>
        <modifiers read="true" write="true" search="true"
            optional="true"/>

```

```

        </attribute >
    </attributes >
</itemtype >

<itemtype code="DSL"
    extends="GenericItem"
    autocreate="true"
    generate="true">
    <attributes >
        <attribute qualifier="name" autocreate="true"
            type="java.lang.String">
            <persistence type="property"/>
            <modifiers read="true" write="true" search="true"
                optional="false" unique="true"/>
        </attribute >
        <attribute qualifier="description" autocreate="true"
            type="localized:java.lang.String">
            <persistence type="property"/>
            <modifiers read="true" write="true" search="true"
                optional="true"/>
        </attribute >
        <attribute qualifier="mapping" autocreate="true"
            type="java.lang.String">
            <persistence type="property"/>
            <modifiers read="true" write="true" search="true"
                optional="false"/>
        </attribute >
    </attributes >
</itemtype >

</itemtypes >
</items >

```

Listing A.3: items.xml der Rules-Extension

A.2.2 Java Klassen

```
/*
 * [y] hybris E-Business Platform
 *
 * Copyright (c) 2000–2007 hybris Holding AG
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of
 *   hybris
 * ("Confidential Information"). You shall not disclose such Confidential
 * Information and shall use it only in accordance with the terms of the
 * license agreement you entered into with hybris.
 *
 */
package com.arithnea.rules.jalo;

[...]

public class RulesManager extends GeneratedRulesManager // implements
    WebRequestInterceptor
{
    public static final String RULES_SESSION_ATTRIBUTE_PREFIX =
        "RULES_SESSION_";

    // Log4J implementation – edit log4j.properties to log to your own
    // log channel
    private static final Logger log = Logger.getLogger(
        RulesManager.class.getName() );

    private static final String RULE_SERVICE_PROVIDER =
        "http://drools.org/";
    private static final String RULE_CLASS =
        "org.drools.jsr94.rules.RuleServiceProviderImpl";

    private RuleServiceProvider serviceProvider = null;
    private RuleAdministrator ruleAdministrator = null;

    private Map<String, CachedRuleSet> loadedRules =
        Collections.synchronizedMap(new HashMap<String, CachedRuleSet>());

    [...]

    public static RulesManager getInstance()
    {
        JaloSession js = JaloSession.getCurrentSession();
        RulesManager rm = (RulesManager)
            js.getExtensionManager().getExtension(
                RulesConstants.EXTENSIONNAME );

        if (rm.serviceProvider == null || rm.ruleAdministrator == null) {
            rm.initializeRuleEngine();
        }
    }
}
```

```

    }

    return rm;
}
[...]

private void initializeRuleEngine() {
    log.info("Initializing RuleEngine ...");
    try {
        Class.forName(RULE_CLASS);
        serviceProvider =
            RuleServiceProviderManager.getRuleServiceProvider(RULE_SERVICE_PROVIDER);
        ruleAdministrator = serviceProvider.getRuleAdministrator();
    } catch (Exception e) {
        log.error("Error initializing RuleEngine", e);
    }
}

public List executeRules(String ruleset, List objects, ObjectFilter
    filter) {
    log.info("Executing RuleSet: " + ruleset);

    CachedRuleSet cRuleSet = loadedRules.get(ruleset);
    if (cRuleSet == null) cRuleSet =
        CachedRuleSet.getCachedRuleSet(ruleset);
    if (cRuleSet == null) {
        log.error("Ruleset " + ruleset + " was not found!");
    }
    else {
        loadedRules.put(ruleset, cRuleSet);
        try {
            cRuleSet.prepare(false, ruleAdministrator);

            RuleRuntime ruleRuntime = serviceProvider.getRuleRuntime();
            Map parameters = new HashMap();
            parameters.put("log", Logger.getLogger(ruleset));

            objects.add(JaloSession.getCurrentSession());

            StatelessRuleSession statelessRuleSession =
                (StatelessRuleSession)
                ruleRuntime.createRuleSession(ruleset, parameters,
                    RuleRuntime.STATELESS_SESSION_TYPE);

            return (filter == null) ?
                statelessRuleSession.executeRules(objects) :
                statelessRuleSession.executeRules(objects, filter);

        } catch (RuleLoadException rle) {
            log.error("Error executing ruleset " + ruleset + ".");
            log.error(rle.getText(), rle.getRoot());
        } catch (Exception e) {
            log.error("Error executing Ruleset " + ruleset, e);
        }
    }
}

```

A Anhang

```
    }
  }
  return null;
}

public StatefulRuleSession getStatefulSession(String ruleset , boolean
  createIfNecessary , Map parameters , List objects)
{
  StatefulRuleSession session = null;
  CachedRuleSet cRuleSet = loadedRules.get(ruleset);
  if (cRuleSet == null) cRuleSet =
    CachedRuleSet.getCachedRuleSet(ruleset);
  if (cRuleSet == null)
  {
    log.error("Ruleset " +ruleset + " was not found!");
    return null;
  }
  else
  {
    loadedRules.put(ruleset , cRuleSet);
    try {
      cRuleSet.prepare(false , ruleAdministrator);

      session = (StatefulRuleSession)
        JaloSession.getCurrentSession().getAttribute(RULES_SESSION_ATTRIBUTE.PR
          + ruleset);
//      log.info("getStatefulSession called , session is " +session);
      if (session == null && createIfNecessary)
      {
        session = (StatefulRuleSession)
          serviceProvider.getRuleRuntime().createRuleSession(ruleset ,
            parameters , RuleRuntime.STATEFUL_SESSION_TYPE);
        JaloSession.getCurrentSession().setAttribute(RULES_SESSION_ATTRIBUTE
          + ruleset , session);
        session.addObject(objects);
      }
    }
    catch (RuleLoadException e)
    {
      log.error("Error during creation of RulesSession",
        e.getRoot());
    }
    catch (Exception e)
    {
      log.error("Error during creation of RulesSession", e);
    }
//    log.info("getStatefulSession called: " +ruleset+", "
+createIfNecessary +", " +parameters +", " +objects);

    return session;
  }
}
```

```

public void releaseStatefulSession(String ruleset)
{
    StatefulRuleSession session = (StatefulRuleSession)
        JaloSession.getCurrentSession().getAttribute(RULES_SESSION_ATTRIBUTE_PREFIX
            + ruleset);
    if (session != null)
    {
        try
        {
            session.release();
        }
        catch (Exception e)
        {
            log.error("Error releasing Statefull Session " +ruleset);
        }
        JaloSession.getCurrentSession().setAttribute(RULES_SESSION_ATTRIBUTE_PREFIX
            + ruleset, null);
    }
}
}
}

```

Listing A.4: RulesManager.java

```

package com.arithnea.rules.promotions;

[...]

public class PromotionChecker {

    private static final String RULESET_PROMOTION           = "promotion";
    private static final String RULESET_UNDO                = "promotion_undo";
    private static final String RULES_SESSION_CART_HANDLE  =
        "RULES_SESSION_CART_HANDLE";

    private static final Logger log =
        Logger.getLogger(PromotionChecker.class);

    public static boolean isApplied(String ruleset, String rule)
    {
        log.info("isApplied called with " +ruleset +", " +rule);
        try
        {
            List<RulesPromotionResult> objects =
                getPromotionResultsFromSession(ruleset);
            if (objects == null) return false;
            for (RulesPromotionResult result : objects)
            {
                if (result.getRule().equals(rule)) {
                    log.info("true");
                    return true;
                }
            }
        }
    }
}

```

A Anhang

```
    }
  }
  catch (Exception e)
  {
    log.error("Error in isApplied", e);
  }
  log.info("false");
  return false;
}

[...]

public static boolean hasUnboundProducts(String product, long
  quantity)
{
  try
  {
    Product p = (Product)
      ProductManager.getInstance().getProductsByCode(product).iterator().next();

    List<RulesPromotionResult> objects =
      getPromotionResultsFromSession(RULESET_PROMOTION);
    if (objects == null) objects = new
      ArrayList<RulesPromotionResult>();

    ProductCartView view = new
      ProductCartView(JaloSession.getCurrentSession().getCart(),
        objects);
    return view.hasProducts(p, quantity);
  }
  catch (Exception e)
  {
    log.error("Error in hasUnboundProducts", e);
    return false;
  }
}

private static List<RulesPromotionResult>
  getPromotionResultsFromSession(String ruleset) throws
  RemoteException, InvalidRuleSessionException {
  StatefulRuleSession session =
    RulesManager.getInstance().getStatefulSession(ruleset, false,
      null, null);
  if (session == null) return new ArrayList<RulesPromotionResult>();
  List<RulesPromotionResult> results = (List<RulesPromotionResult>)
    session.getObjects(new
      RulesObjectFilter(RulesPromotionResult.class));
  return results;
}
```

```

private static class ProductCartView extends HashMap<Product, Long>
{
    public ProductCartView(AbstractOrder order,
        List<RulesPromotionResult> results)
    {
        super();

        for (AbstractOrderEntry entry :
            (Collection<AbstractOrderEntry>) order.getAllEntries())
        {
            if (!entry.isGiveAway())
            {
                this.put(entry.getProduct(), entry.getQuantity());
            }
        }

        for (RulesPromotionResult result : results)
        {
            for(RulesPromotionResult.ProductEntry entry :
                result.getConsumedProducts())
            {
                this.removeProduct(entry.getProduct(),
                    entry.getQuantity());
            }
        }
    }
    [...]

    public boolean hasProducts(Product p, long quantity)
    {
        return (this.get(p) != null &&
            this.get(p).longValue()>=quantity);
    }

    public void removeProduct(Product product, long quantity)
    {
        if (!this.containsKey(product) || !this.hasProducts(product,
            quantity))
        {
            log.equals("Could not remove Product from view: " +product
                +", " +quantity);
            return;
        }
        this.put(product, this.get(product)-quantity);
    }
}

public static void recalculatePromotions()
{
    //undo all previous changes made by rules and release the session

```

A Anhang

```
    StatefulRuleSession session =
        RulesManager.getInstance().getStatefulSession(RULESET_PROMOTION,
            false, null, null);
    if (session != null)
    {
        undoChangesByRules(session);
        assureCartCalculated();
    }

    // start rules session and evaluate the promotion results
    Map parameters = new HashMap();
    parameters.put("log", Logger.getLogger(RULESET_PROMOTION));

    session =
        RulesManager.getInstance().getStatefulSession(RULESET_PROMOTION,
            true, parameters, new ArrayList());

    try
    {
        session.addObject(JaloSession.getCurrentSession());
        session.addObject(JaloSession.getCurrentSession().getCart());
        session.executeRules();
    }
    catch (Exception e)
    {
        log.error("Error calculating Promotions", e);
        RulesManager.getInstance().releaseStatefulSession(RULESET_PROMOTION);
    }

    assureCartCalculated();
}

private static void assureCartCalculated() {
    Cart cart = JaloSession.getCurrentSession().getCart();
    try
    {
        cart.calculateTotals(true);
    }
    catch (JaloPriceFactoryException e1)
    {
        log.error("Error calculating Cart", e1);
    }
}

public static void undoChangesByRules(StatefulRuleSession session)
{
    try
    {
        if (session == null) return;
        List<RulesPromotionResult> results =
            (List<RulesPromotionResult>) session.getObjects(new
                RulesObjectFilter(RulesPromotionResult.class));
    }
}
```

```

    if (results == null || results.size()==0) return;

    List objects = new ArrayList();
    objects.add(JaloSession.getCurrentSession().getCart());
    for (RulesPromotionResult result : results)
    {
        objects.add(result.getEffect());
    }

    log.info("Executing UNDO");

    RulesManager.getInstance().executeRules(RULESET.UNDO, objects,
        null);
    RulesManager.getInstance().releaseStatefulSession(RULESET.PROMOTION);
}
catch (Exception e)
{
    log.error("Error during Undo", e);
}
}
[...]
}

```

Listing A.5: PromotionChecker.java

A Anhang

Literaturverzeichnis

- [1] ARITHNEA GMBH. <http://www.arithnea.de>.
- [2] FACELETS. <https://facelets.dev.java.net>.
- [3] FRITZ BERGER ONLINESHOP. <http://www.fritz-berger.de>.
- [4] THE HYBRIS DEVELOPMENT NETWORK. <http://dev.hybris.de>.
- [5] J2EE JAVASERVER PAGES TECHNOLOGY. <http://java.sun.com/products/jsp>.
- [6] JAVA PLATFORM, ENTERPRISE EDITION (JAVA EE) JAVASERVER FACES TECHNOLOGY. <http://java.sun.com/javaee/javaserverfaces>.
- [7] JBOSS RULES DOCUMENTATION LIBRARY. <http://labs.jboss.com/jbossrules/docs/index.html>.
- [8] JIRA ISSUE TRACKER JBOSS RULES. <http://jira.jboss.com/jira/browse/JBRULES>.
- [9] APACHE TOMCAT, 2007. <http://tomcat.apache.org>.
- [10] HIBERNATE - RELATIONAL PERSISTENCE FOR JAVA AND .NET, 2007. <http://www.hibernate.org>.
- [11] HSQL DATABASE ENGINE - 100% JAVA DATABASE, 2007. <http://www.hsqldb.org>.
- [12] ILOG JRULES, 2007. <http://www.ilog.com/products/jrules>.
- [13] JESS, THE RULE ENGINE FOR THE JAVATM PLATFORM, 2007. <http://herzberg.ca.sandia.gov/jess>.
- [14] MYSQL - DIE POPULÄRSTE OPEN-SOURCE-DATENBANK DER WELT, 2007. <http://www.mysql.de>.
- [15] Alberto Avritzer, Johannes P. Ros und Elaine J. Weyuker: ESTIMATING THE CPU UTILIZATION OF A RULE-BASED SYSTEM. (29(1)):1–12, 2004.

LITERATURVERZEICHNIS

- [16] Don Batory: THE LEAPS ALGORITHMS, 2007. <ftp://ftp.cs.utexas.edu/pub/predator/tr-94-28.pdf>.
- [17] Hendrik Beck: EINSATZ VON RULE-ENGINES ZUR FLEXIBLEN WISSENSVERARBEITUNG IM BETRIEBLICHEN UMFELD. Diplomarbeit, Fachhochschule Darmstadt, 2005.
- [18] Wolfgang Bibel, Steffen Hölldobler und Torsten Schaub: WISSENSREPRÄSENTATION UND INFERENZ. Vieweg Verlagsgesellschaft, ISBN 978-3528053741, 1998.
- [19] Charles Forgy: RETE: A FAST ALGORITHM FOR THE MANY PATTERN/MANY OBJECT PATTERN MATCH PROBLEM. (2):17–37, 1982.
- [20] Erich Gamma *et al.*: ENTWURFSMUSTER. ELEMENTE WIEDERVERWENDBARER OBJEKTORIENTIERTER SOFTWARE. Addison-Wesley, ISBN 978-3827321992, 2004.
- [21] A.C. Nielsen GmbH: NEUE PROMOTIONS UNTER DER LUPE, 2002. http://www.acnielsen.de/pubs/documents/Promotion_Studie_Highlights2002.pdf.
- [22] Oliver Ihns, Stefan M. Heldt und Ralf Wirdemann: ENTERPRISE JAVABEANS KOMPLETT. GRUNDLAGEN, ÜBERBLICK UND EINSATZ VON EJB 2.1. Oldenbourg, ISBN 978-3486273793, 2003.
- [23] artificialmemory net: LINKSAVE AT ARTIFICIALMEMORY, 2007. <http://www.artificialmemory.net/artificialmemory.aspx?ID=32353VP>.
- [24] Java Specification Requests: JSR 94: JAVATM RULE ENGINE API, 2003. <http://jcp.org/en/jsr/detail?id=94>.
- [25] Ronald G. Ross: PRINCIPLES OF THE BUSINESS RULE APPROACH. Addison-Wesley Professional, ISBN 978-0201788938, 2003.
- [26] The Free Encyclopedia Wikipedia: BUSINESS RULE, 2007. http://en.wikipedia.org/wiki/Business_rules.
- [27] The Free Encyclopedia Wikipedia: CROSS-SELLING, 2007. <http://de.wikipedia.org/wiki/Cross-Selling>.
- [28] The Free Encyclopedia Wikipedia: DEKLARATIVE PROGRAMMIERUNG, 2007. http://de.wikipedia.org/wiki/Deklarative_Programmierung.

LITERATURVERZEICHNIS

- [29] The Free Encyclopedia Wikipedia: GESCHÄFTSREGEL, 2007.
<http://de.wikipedia.org/wiki/Gesch%C3%A4ftsregel>.
- [30] The Free Encyclopedia Wikipedia: PATTERN MATCHING, 2007.
http://de.wikipedia.org/wiki/Pattern_Matching.
- [31] The Free Encyclopedia Wikipedia: PROZEDURALE PROGRAMMIERUNG, 2007.
http://de.wikipedia.org/wiki/Prozedurale_Programmierung.
- [32] The Free Encyclopedia Wikipedia: RETE ALGORITHM, 2007.
http://en.wikipedia.org/wiki/Rete_algorithm.
- [33] The Free Encyclopedia Wikipedia: UPSELLING, 2007. <http://de.wikipedia.org/wiki/Upselling>.
- [34] The Free Encyclopedia Wikipedia: VERKAUFSFÖRDERUNG, 2007.
<http://de.wikipedia.org/wiki/Verkaufsf%C3%B6rderung>.
- [35] Lars Wunderlich: JAVA RULES ENGINES - ENTWICKLUNG VON REGELBASIERTEN SYSTEMEN. entwickler.press, ISBN 3-935042-75-2, 2006.

LITERATURVERZEICHNIS