

# WOHIN MIT DER LOGIK? REGELN ALS RETTUNG

Es ist heute (SQL sei Dank!) völlig normal, Datenbankabfragen deklarativ zu beschreiben. Kaum jemand kommt auf die Idee, der Datenbank algorithmisch beibringen zu wollen, wie die gesuchten Daten von der Platte zu lesen sind. Warum beharren so viele IT-Systeme aber darauf, ihre komplexe Geschäftslogik algorithmisch zu formulieren, statt elegante deklarative Regelsysteme dafür zu verwenden? Dieser Artikel beschreibt, wie Sie Fachlogik mit Hilfe von Regelmасhinen aus Ihrem Quellcode auslagern und damit eine Menge Flexibilität gewinnen können, jedoch nicht ohne ein gewisses Risiko – aber lesen Sie selbst.

In der Softwarebranche diskutieren wir hauptsächlich technische Themen, die für Unternehmen – ehrlich gesagt – kaum einen echten Mehrwert bringen. Welche Bank oder Versicherung gewinnt neue Kunden durch ein schickes Persistenz-Frameworks oder ausgeklügelte Transaktionsstrategien?

Letztlich profitieren Unternehmen von ihren wertschöpfenden Prozessen; IT und andere Technik dienen dabei als wichtige Grundlage. Nur in ganz wenigen Fällen steht die Informationstechnik im Vordergrund. Insbesondere nehmen die Trend- und Hype-Themen der Informatik auf die Wertschöpfung nur mittelbaren Einfluss.

Viel höher bewerte ich fachliche Zusammenhänge, die Sie auch unter dem Oberbegriff der *Fachlogik* kennen<sup>1)</sup>.

Wir IT-Leute sollten meiner Meinung nach intensiver über die effiziente und effektive Behandlung dieser Fachlogik nachdenken. Dabei helfen uns die – ach so interessanten – IT-Trends wie Spring, Linq, Groovy, Ruby-on-Rails, ESBs und die diversen Applikationsserver leider überhaupt nicht weiter.

## Was ist Fachlichkeit?

„Fachlichkeit“ beschreibt eine Domäne mit ihren statischen und dynamischen Aspekten (im IT-Slang: Strukturen von Klassen sowie Abläufe zwischen Instanzen). Die Abläufe wiederum gehorchen gewissen Regeln: Wenn in einem vorgegebenen Kontext bestimmte Bedingungen erfüllt sind, dann handle auf diese Art und Weise.

<sup>1)</sup> Fachlogik, Geschäftslogik, Business-Logik, Domänen-Logik – das sind samt und sonders Synonyme. Nennen Sie das Kind, wie Sie mögen, Hauptsache es geht um Anwendungsdomänen.

Statt einer förmlichen Definition möchte ich einige Beispiele für solche Regeln geben:

- **Versicherung:** Schadensfälle mit einem geschätzten Gesamtschaden von mehr als 10.000 Euro werden nur in der Zentrale bearbeitet.
- **Versicherung:** Gehört ein an einem Schadensfall direkt Beteiligter zur oberen Managementebene des Konzerns, so darf der Schaden ausschließlich von Sachbearbeitern mit der Vertraulichkeitsstufe V0 bearbeitet werden.
- **Bank:** Wenn die Überweisungssumme größer ist als der Kontostand zuzüglich dem Dispo-Limit, dann stelle eine Rückfrage beim Auftraggeber.
- **Finanzamt:** Wenn der Steuerpflichtige seinen Zahlungspflichten mehr als zweimal erst nach Erinnerung nachkommt, wird ihm 24 Monate kein Aufschub mehr gewährt.

In vielen Branchen gelten Regeln dieser oder ähnlicher Art. Viele Aktionen oder Abläufe hängen von bestimmten Bedingungen oder Zuständen ab – und genau diese legen die unternehmens- oder domänenspezifische Fachlichkeit fest. Eine Entität „Kunde“ allein mit ihren Attributen macht eben noch keine Fachlichkeit aus – erst die Kombination aus Daten und (durch Regeln gesteuertem) Verhalten macht die „Musik“.

## Zuviel Logik im Code

Meine These zur Fachlogik: (Zu) viel Fachlogik dieser Art steckt verborgen und fest zementiert in Form verschachtelter if-then-else-Konstrukte tief in schwer änderbarem Quellcode. Nur wenige fachliche Abläufe und Regeln können wir in Anwen-



Gernot Starke

([www.gernotstarke.de](http://www.gernotstarke.de)) unterstützt als unabhängiger Berater Unternehmen bei anspruchsvollen IT-Projekten, insbesondere in den Bereichen Softwarearchitektur, iterative Entwicklungsprozesse und Technologiemanagement.

dungssystemen so einfach lesen und verstehen wie die oben genannten Beispiele. Nur mit großen Schwierigkeiten können wir in bestehenden Systemen neue Fachlogik implementieren oder diese an aktuelle Gegebenheiten anpassen. Das genau müssen wir IT-ler jedoch leisten – denken Sie an die *Wertschöpfung*, die ich eingangs erwähnt habe.

Warum sind Änderungen denn so schwierig? Wir alle haben aus diversen Entwurfsmethoden gelernt, fachliche und technische Dinge voneinander zu trennen. In den verbreiteten Schichtenmodellen von Softwarearchitekturen finden wir häufig eine Businessschicht, die fachlich motivierte Domänenklassen enthält, was grundsätzlich eine sehr gute Idee ist. Sie verschafft Überblick und führt zu grundsätzlich sauberen Strukturen. Leider schleichen sich

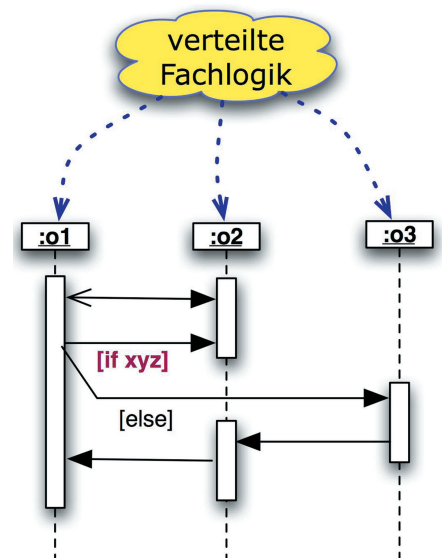


Abb. 1: Fachobjekte mit verteilter Ablauflogik

jedoch in die vermeintlich fachlichen Klassen und Objekte immer wieder technische oder Infrastrukturaspekte ein, die Änderungen erschweren.

Unabhängig von der jeweiligen Implementierungstechnologie stehen *fachliche Objekte* in solchen Architekturen dabei in gegenseitigen Abhängigkeitsbeziehungen. Fachliche Methoden referenzieren jeweils andere fachliche Objekte, um fachliche Abläufe zu zementieren – Verzeihung, zu implementieren. Noch einmal: Fachliche Abläufe, ein Bestandteil der Fachlogik, entstehen aus dem aktiven Zusammenspiel einzelner Fachobjekte.

Beim Entwurf solcher Zusammenarbeit müssen fachliche Abhängigkeiten vorab feststehen. Zwischen den beteiligten Klassen entstehen statische Abhängigkeiten. Änderungen an der Fachlogik bedürfen hierbei der Anpassung von Quellcode, immer verbunden mit einem erneuten *Build-* und *Deploy-Zyklus* der gesamten Applikation.

Zu diesen Abhängigkeiten kommt das Navigationsproblem: Ein Fachobjekt muss möglicherweise vor der eigentlichen fachlichen Tätigkeit durch komplexe Objektgraphen zu anderen Objekten navigieren. So entstehen aus vermeintlich einfachen fachlichen Operationen häufig umfangreiche (und gar nicht mehr einfache) Codefragmente. Die Navigation zwischen Objekten hat nichts mit Fachlichkeit zu tun – sie basiert auf der Architektur und dem Design des jeweiligen Objekt- und Klassengeflechtes. Damit bewerte ich sie als reine Infrastruktur- oder Technikaufgabe, die die Wartbarkeit und Verständlichkeit von Quellcode negativ beeinflusst.

### Regeln zur Rettung?

Als Ergänzung zur oben geschilderten Art der Implementierung von Fachlogik empfehle ich Ihnen Regelsysteme (*Rule-based Systems*). Diese verwenden Geschäftsregeln als eigenständige Einheiten (*First Order Citizens*) mit einer Reihe bestechender Vorteile. Ich möchte aber bereits jetzt ganz offen zugeben: Den Vorteilen von Regelsystemen steht auch eine

<sup>2)</sup> Genau wie „SQL-Queries“ zu erheblich mehr Ordnung im Code führen können, bei Missbrauch jedoch zur Ursache unschöner Debug-Sitzungen mutieren können. Noch deutlicher ist das Beispiel der (ebenfalls deklarativen) regulären Ausdrücke (Regex). Allen Unkenrufen zum Trotz können selbst reguläre Ausdrücke verständlich formuliert werden – und dann auch ordnend und flexibilisierend wirken. Aber zugegeben, reguläre Ausdrücke können auch schwere Gehirnrämpfe verursachen, vgl. [War02].

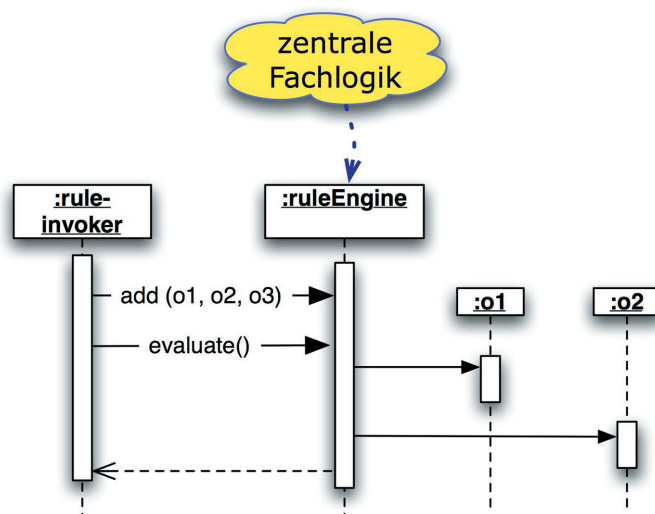


Abb. 2: Zentrale Fachlogik

Menge Nachteile entgegen. Für Ihre eigenen Entwürfe stelle ich Ihnen erst einmal die Konzepte vor – letztlich entscheiden Sie selbst.

Regelsysteme haben einen fundierten theoretischen Hintergrund. Es gibt so genannte *Forward-* und *Backward-Chaining* Regelsysteme und für beide Varianten auch eine ganze Menge von

Implementierungen. Lassen Sie uns diese Details vorläufig ignorieren, da sie für meine Argumentation keine entscheidende Rolle spielen.

Meine These zu Regelsystemen lautet: Regelsysteme können Ihnen zu mehr Ordnung und Flexibilität in Ihren IT-Systemen verhelfen, sie müssen es aber nicht<sup>2)</sup>.

*Forward-Chaining*-Regelsysteme enthalten eine Menge von „Wenn-Dann“-Regeln. Im „Wenn“-Teil (auch genannt *Left-Hand-Side, LHS*) steht eine Menge von Bedingungen. Der „Dann“-Teil der Regeln (genannt *Right-Hand-Side, RHS*) beschreibt, was zu tun ist, falls die LHS erfüllt ist.

Die Regelmaschine arbeitet auf einem so genannten *Working-Memory*, der die so genannten Fakten und den aktuellen Bearbeitungszustand der Regeln enthält. Bei der Auswertung von Regeln geht die Regelmaschine in der Abfolge *Match, Select* und *Act* vor:

- Die *Match*-Phase versucht, die Bedingungssteile sämtlicher Regeln mit den Fakten aus dem *Working-Memory* zu erfüllen. Die Regelmaschine testet dabei sämtliche möglichen Kombinationen von Fakten. Eine erfüllbare Regel mitsamt den dafür notwendigen Fakten wird in die Konfliktmenge (*Conflict-Set*) aufgenommen.
- Die *Select*-Phase ermittelt für sämtliche erfüllbaren Regeln eine Ausführungsreihenfolge, den *Schedule*. Bei den meisten Regelmaschinen erhöht beispielsweise die Komplexität der LHS ihre Priorität, teilweise spielen sogar die Zeitstempel der beteiligten Fakten eine Rolle.
- Die *Act*-Phase führt die Regeln in der vorher bestimmten Reihenfolge aus. Nun kann es spannend werden: Die RHS einer Regel kann dazu führen, dass die Regelmaschine wieder mit einer neuen *Match*-Phase starten muss: RHS können nämlich die Fakten im *Working-Memory* ändern.

Diesen Zyklus implementieren moderne Regelmaschinen durch hochgradig optimierte Algorithmen (RETE, LEAPS), deren Details ich Ihnen hier erspare.

### Kasten 1: Zur Funktionsweise von Regelmaschinen

Vorteile	Nachteile
Flexibilität	Flexibilität
Auch für Fachleute verständliche Beschreibung von Geschäftsregeln	Konsequenzen von Regeln und Regeländerungen aufgrund der Ausführungssemantik der Regelmaschine schwer verständlich
Leichtgewichtigerer Fachobjekte	Erschwertes Testen und Debuggen
Fachexperten können bei Gestaltung der Regeln mitarbeiten. (für Java: durch JSR 94 standardisierte API für Regelsysteme)	Ungewohnte Syntax von Regeln
	Kaum Standards – viele Systeme verwenden proprietäre Sprachen; daher: Methoden- und Technologieauswahl notwendig.
	Management und Administration von Regelsystemen erfordert angepasste Entwicklungs- und Betriebsprozesse.

Tabelle 1: Vor- und Nachteile beim Einsatz von Regelmaschinen

### Funktionsweise von Regelsystemen

Bevor wir die Vor- und Nachteile von Regelsystemen diskutieren, möchte ich

kurz auf das grundlegende Konzept eingehen. Insbesondere stelle ich in diesem Zusammenhang die *konventionelle* Variante der objektorientierten Implemen-

tierung von Fachlogik einer Variante mit Regelsystem gegenüber.

Betrachten Sie zunächst einmal **Abbildung 1**. Das Sequenzdiagramm zeigt drei kooperierende Objekte. Die Fachlogik ist auf diese drei Objekte (genauer: ihre Klassen) verteilt. Beachten Sie insbesondere die Abfrage `if xyz` im Sequenzdiagramm: Sollte sich die Bedingung ändern, folgt daraus zwangsläufig eine Änderung von Quellcode sowie ein erneutes *Deployment* der beteiligten Klassen. Sie könnten solche Abfragen konfigurierbar gestalten – mit dem Effekt, dass die Konfiguration gemeinsam mit dem Programmcode für die Abläufe verantwortlich ist. Das ist auch nicht viel besser als eine *If-then-else*-Logik im Quellcode.

Nun zu einer Alternative mit Geschäftsregeln (*siehe Abb. 2*). Ein Aufrufer (der *ruleInvoker*) übergibt einige Fachobjekte an eine Regelmaschine. Zufällig sind es dieselben Objekte, die Sie schon aus **Abbildung 1** kennen.

Die Regelmaschine führt ihrerseits Operationen auf den Fachobjekten aus, die jetzt untereinander nicht mehr kommunizieren müssen. Die Logik – genauer der objektübergreifende Teil davon – steckt in der Regelmaschine. Um ganz exakt zu sein, steckt die Fachlogik in Regeln, die eine Regelmaschine erst zur Laufzeit einliest.

Die Regelmaschine ruft als Ergebnis der Bearbeitung auf den beteiligten Fachobjekten beliebige Methoden auf, völlig analog zur konventionellen Variante. In dieser regelbasierten Variante weisen die Fachobjekte deutlich weniger Abhängigkeiten voneinander auf – was eine losere Kopplung auf Architektur- und Codeebene bewirkt.

Obendrein lassen sich Regeln häufig in (fast) natürlicher Sprache formulieren:

```
When (Person.Age < 18) Then Person.setCreditAllowed( false );
```

Solche „Sätze“ können auch Personen ohne

Hier einige Kriterien für oder gegen den Einsatz von Regelsystemen (angeregt durch [Rud07]):

1. Müssen Sie innerhalb Ihrer Anwendungen *Entscheidungen* auf Basis vieler bedingter Verzweigungen (z. B. verschachtelte *if-then-else*-Konstrukte) treffen? Dann passt eine Regelmaschine sicherlich gut zu Ihrem Problem. Können Sie Ihre Entscheidungen jedoch durch Datenbankabfragen oder rein algorithmische Berechnungen treffen, verwenden Sie besser keine Regelmaschine.
2. Wenn sich Ihre Geschäftsregeln nur sehr selten ändern, können Sie statt Regeln ebenso konventionellen Programmcode schreiben und sich das Risiko und die Aufwände von Regelmaschinen ersparen.
3. Versuchen Sie einmal, einige Ihrer Geschäftsregeln in „Wenn-Dann“-Form aufzuschreiben. Lassen Sie sich ruhig von einem Nicht-IT-ler dabei unterstützen. Wenn Ihnen nach einer halben Stunde immer noch keine fachlich sinnvollen Regeln eingefallen sind, spricht das gegen eine Regelmaschine. Aber: Lassen Sie sich vorher von Beispielregeln einer Regelmaschine (*siehe Kasten 1*) inspirieren – manchmal erfordert das Umdenken von klassischen Programmiersprachen zu Regelsprachen einige Zeit.
4. Wenn Ihre Entscheidungsprozesse einfach sind, d. h. wenn Sie immer nur ein oder zwei *if*-Anweisungen zur endgültigen Entscheidung benötigen, dann ist eine Regelmaschine wahrscheinlich nicht angemessen.
5. Falls es bei Ihnen auf Millisekunden ankommt, weil Sie höchste Anforderungen an die Performance stellen, dann vergessen Sie Regelmaschinen (obwohl die in den letzten Jahren wirklich flott geworden sind). Gleiches gilt für Echtzeitanforderungen. Zumindest sind mir keine Regelmaschinen bekannt, die Laufzeiten im Echtzeit-Sinn *garantieren* können.
6. Haben Sie Druck im Projekt? Stehen Sie kurz vor Fertigstellung? Neue Technologien stellen ein zusätzliches Risiko dar.
7. Bedenken Sie, dass die Einarbeitung in die Methode und Technik sowie die Erstellung der Regeln hohen Aufwand benötigen. Falls Sie die Rendite für eine Regelmaschine in weniger als 12 Monaten erreichen müssen, wird es riskant.
8. Wenn Sie eine System- oder Softwarearchitektur entwerfen, bei der die Fachlogik klar von der Technik getrennt sein soll, können Regelsysteme Sie dabei unterstützen.
9. Sind Sie erfahren im Umgang mit Neuerungen? Suchen Sie stets nach Verbesserungsmöglichkeiten und Ergänzungen Ihres Werkzeugkastens? Stellen Sie vorhandene Prozesse und Technologien in Frage? Dann sollten Sie sich auf einen Flirt mit einer Regelmaschine einlassen – spätere Heirat nicht ausgeschlossen.

Kasten 2: Kriterien zum Einsatz von Regelsystemen

- **JBoss-Rules**(vormals **Drools**): Open-Source-Projekt, gute Performance, aktive Entwicklergemeinde, recht gute Eclipse-basierte Entwicklungsumgebung inklusive Editor- und Debugging-Unterstützung, leicht verständliche und anpassbare Regelsyntax; meine persönliche Empfehlung zum Einstieg ([labs.jboss.com/portal/jbossrules/](http://labs.jboss.com/portal/jbossrules/)).
- **JESS**: der „Klassiker“ unter den Java-basierten Regelmaschinen, kommerziell, aber recht günstig, kostenfreie Variante erhältlich, gute Performance, guter Support (vom Regel-Guru Dr. Friedmann-Hill), konfigurierbare Auswertungsstrategie; Regelsyntax erinnert an Lisp ([www.jessrules.com/jess/index.shtml](http://www.jessrules.com/jess/index.shtml)).
- **ILOG Rules, BLAZE-Advisor**: Java-basierte kommerzielle Regelmaschinen der Luxusklasse.
- **InRULE**: .NET-basierte Inferenzmaschine mit (laut Hersteller) „pragmatischer“ und anwenderorientierter Regelsyntax ([www.inrule.com/default.aspx](http://www.inrule.com/default.aspx)).

Darüber hinaus gibt es noch die „klassischen“ Regelsprachen wie Prolog und OPS5, zu denen es unterschiedliche Implementierungen gibt. Prolog wird – viele mögen es kaum glauben – heutzutage im kommerziellen Kontext mit großem Erfolg eingesetzt.

### Kasten 3: Einige Tools zum Thema (ohne Anspruch auf Vollständigkeit, lediglich als erste Übersicht)

umfangreiche Informatikausbildung verstehen und schreiben. Falls Sie sich jetzt fragen, ob so etwas in der Praxis wirklich funktioniert, habe ich hier einige Beispiele für (erfolgreiche) Anwendungen von Regelsystemen parat (was soviel bedeutet wie „ja, es funktioniert an vielen Stellen“):

- *content-based Routing* von Nachrichten in serviceorientierten Architekturen
- Tarif- oder Preisberechnung in Versicherungen
- Steuerung von Call-Center-Abläufen
- Steuerung der Datensatzverarbeitung von Batch-Prozessen

Etwas mehr zur Funktionsweise von Regelmaschinen finden Sie in [Kasten 1](#).

### Flexibilität gewinnen – mit Risiken

Einen Vorteil erlebe ich mit solchen Systemen in der Praxis häufig: Während der Implementierungsphase kennen wir nur wenige einfache Geschäftsregeln. Während der Lebenszeit des Systems kommen immer komplexere Regeln oder Sonderfälle dazu, die wir zum großen Teil durch neue oder verfeinerte Regeln abbilden können. Dafür müssen wir keinen Quellcode ändern, sondern lediglich Regeln. Die werden nicht kompiliert, sind (meist) relativ einfach für Fachleute zu verstehen und auch zur Laufzeit noch änderbar.

Damit besitzen Regeln den gleichen Stellenwert wie Java-, C#- oder COBOL-

Code. Sie müssen getestet werden und beeinflussen den Ablauf der Programme. Aus rein fachlicher Sicht aber gewinnen wir damit eine Menge Flexibilität. Regeln können Sie ohne die Umstände starrer Release-Zyklen ändern. Jetzt jubeln die Perl-, Bash- und Groovy-Fraktionen – und die Vorsichtigen unter den Lesern bekommen es (spätestens) jetzt mit der Angst zu tun, denn diese Flexibilität hat ihren Preis in Form von einem höheren Risiko. Wir sind erwachsene Menschen und wissen um die mit Vorteilen verbundenen Nachteile<sup>3)</sup>. So verhält es sich auch mit Geschäftsregeln und regelbasierten Systemen: Die einfache Änderbarkeit birgt das gravierende Risiko, dass Änderungen *einfach so* geschehen: Mit dem Texteditor im Produktivsystem flugs einige Regeln geändert – und schon ist es um die Wertschöpfung geschehen.

Zusätzlich droht noch eine weitere Nebenwirkung: Die Reihenfolge innerhalb der Regelbearbeitung wird durch die Regelmaschine und die zur Verfügung stehenden Ausgangsdaten bestimmt – ganz anders als bei konventionellen Sprachen. Eine deklarative Regelsyntax liest sich völlig anders als sequenzieller Java- oder C#-Programmcode. Selbst hartgesottene Entwickler finden es oft schwierig, zwischen sequenziellem und deklarativem Paradigma umzudenken und in der Praxis hängen beide – sprich: Fachlogik und IT – doch meistens irgendwie zusammen.

### Fazit

Licht und Schatten liegen auch bei regelbasierten Systemen eng zusammen. Nicht jedes algorithmisch schwere Problem eignet sich zur Behandlung mit deklarativen Geschäftsregeln (vgl. [Tabelle 1](#)). Regelsysteme folgen einem ganz anderen Para-

digma als konventionelle Programmiersprachen – das steigert die Komplexität (und erhöht gleichzeitig den Reiz wie häufig bei so genannten neuen Technologien).

Meiner Erfahrung nach lassen sich Regelmaschinen in vielerlei IT-Projekten jedoch produktiv einsetzen – zumindest gehören sie in den Werkzeugkasten von Softwarearchitekten. Falls in Ihrer Fachdomäne Abläufe oder Zusammenhänge häufig in Form von „Wenn-Dann“-Sätzen formuliert sind, ist dies ein Indikator für den Einsatz von Regelsystemen. Weitere Kriterien für (und gegen) den Einsatz von Regelmaschinen enthält [Kasten 2](#).

Weitere Kriterien für (und gegen) den Einsatz von Regelmaschinen enthält [Kasten 2](#) und einige technische Alternativen finden Sie in [Kasten 3](#). ■

### Links

**[Bus06]** Business Rules Group, Business Rule Manifesto, 2006, siehe: [www.businessrulesgroup.org/brmanifesto.htm](http://www.businessrulesgroup.org/brmanifesto.htm)

**[Dro06]** Drools Project, FIT for Rules – Keep your business rules in shape (Test-Framework für Regelsysteme), 2006, siehe: [fit-for-rules.sourceforge.net/](http://fit-for-rules.sourceforge.net/)

**[Rud07]** G. Rudolph, Some Guidelines For Deciding Whether To Use A Rules Engine, 2007, siehe: [herzberg.ca.sandia.gov/jess/guidelines.shtml](http://herzberg.ca.sandia.gov/jess/guidelines.shtml)

**[Rul07]** Rule Works, User Guide Introduction, 2007 (beschreibt methodische Grundlagen, etwa Recognize-Act-Cyle und Conflict-Resolution), siehe: [www.ruleworks.co.uk/uguide/rwug1.html](http://www.ruleworks.co.uk/uguide/rwug1.html)

**[War02]** P. Warren, Regulärer Ausdruck zur Validierung von Mailadressen gemäß RFC 822, 13.4.2002, siehe: [www.ex-parrot.com/~pdw/Mail-RFC822-Address.html](http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html)

<sup>3)</sup> Auch bekannt als TANSTAAFL: *There Aint No Such Thing As A Free Lunch*.