MANDARAX
-
Ein OpenSource Ansatz zur Verwaltung und Verarbeitung von Geschäftsregeln

Jens Dietrich
Polytechnic of Namibia

Jochen Hiller
Bauer & Partner AG

Abstract. MANDARAX ist ein OpenSource Projekt zur Verwaltung und Verarbeitung von Geschäftsregeln. MANDARAX ist eine Java basierende Implementierung einer Prädikatenlogik 1-ter Stufe (PL1). Durch die Java basierende Architektur und das flexible Integrationskonzept lässt sich MANDARAX einfach und flexibel in bestehende Java und J2EE Anwendungen integrieren.

1 Einführung

Die letzten Jahre Softwareentwicklung waren vom Siegeszug objektorientierter Technologien geprägt. Dabei sind einige vielversprechende Ansätze aus den späten achtziger Jahren weitgehend in Vergessenheit geraten, obwohl diese sehr vielversprechend waren. Insbesondere meinen wir hiermit die Programmiersprache Prolog und der damit verbundene Ansatz, Programmlogik zu deklarieren anstatt sie zu programmieren. Wir wollen hier einen Ansatz vorstellen, der zeigen soll, wie sich aus Prolog stammende Ideen in Java einbetten lassen, um Anwendungen zu schaffen, in denen Geschäftsregeln nicht hart verdrahtet programmiert sind, sondern in einem separaten Modul als Daten abgelegt (und gepflegt) werden. Der damit verbundene Vorteil ist offensichtlich: wenn sich die Geschäftsregeln eines Unternehmens ändern, können diese Regelmodule bearbeitet werden, ohne erneut in den langwierigen und fehleranfälligen Software-Entwicklungsprozeß eintreten zu müssen.

MANDARAX ist ein OpenSource-Softwareprojekt, das zur Zielsetzung hat, eine Java Klassenbibliothek zu diesem Zweck zur Verfügung zu stellen. Das Projekt selbst besteht seit Ende 1999 und hat mittlerweile schon eine gewisse Reife erreicht, der MANDARAX Code liegt gegenwärtig in der 4ten version (1.3) vor.

MANDARAX ist *kein* Prolog Interpreter, der in Java implementiert ist. Es werden lediglich intern Prolog-Algorithmen verwendet, was für den Benutzer allerdings transparent bleibt. Es gibt insbesondere keine Prolog-artige Scriptsprache, um Geschäftsregeln zu repräsentieren, sondern lediglich Schnittstellen zu entsprechenden Objekten.

2 Die Sprache der Logik

Das theoretische Rüstzeug von MANDARAX ist die Prädikatenlogik 1-ter Stufe (PL1). Mit dieser lassen sich große Teile unserer natürlichen Sprache formalisieren. So etwa Geschäftsregeln, die wie folgt aussehen könnten:

```
(1)  If the value of all transactions of a customer is more
     than 100 $ in one year then he qualifies for a discount of
     10%.
```

Dies könnte eine typische Geschäftsregel in einem Customer-Relationship-Mangement System (CRM) darstellen, und illustriert gleichzeitig die Bedeutung des Ansatzes: in einem wettbewerbsorientiertem Umfeld müssen Unternehmen diese Regeln sehr schnell den Wettbewerbsbedingungen gemäß anpassen, um ihre Kunden zu halten. Ist es erforderlich, zur Veränderung einer solchen Regel Programmcode zu ändern, so bedeutet dies Verzögerung und mithin einen Wettbewerbsnachteil für die betreffende Organisation.

Zurück zur Logik. Den gesamten Ausdruck `(1)` bezeichnet man gemeinhin als *Regel*. Der Teil zwischen „`If`“ und „`then`“ wird als *Voraussetzung* oder *Prämisse* bezeichnet, im allgemeinen handelt es sich dabei um mehrere solcher Prämissen, die mit „und“ verknüpft sind. Der letze Teil ist die *Konklusion* der Regel. Prämissen und Konklusionen selbst haben eine interne Struktur: sie bestehen aus *Prädikaten* und *Termen*. Prädikate sind in obigen Beispiel etwa „`is more than`“ und „`qualifies for`“, Terme sind „`a customer`“ und „`a discount of 10%`“. Die Satzanalyse hier ist nicht unähnlich dem Vorgehen bei der Klassenfindung bei der Objektmodellierung - hier würden die Terme letztendlich zu den Klassen führen.

Wichtig zu bemerken ist, das wir es mit verschiedenen Sorten von Termen zu tun haben:

„`a discount of 10%`“ ist ein konkreter, *konstanter Term*, der in einem OO Programm gemeinhin durch eine konkrete Objektinstanz repräsentiert würde

„`a customer`“ ist ein *variabler Term*, der durch konkrete Instanzen ersetzbar ist. Das macht Sinn, da ansonsten für jeden existierenden Kunden eine separate Regel verfaßt werden müsste

„`the value of all transactions of a customer`“ ist ein Term der durch Anwendung einer Funktion (hier: „`the value of ...`“) auf weitere Terme (hier: der variable Term „`a customer`“ und der konstante Term „`in one year`“) entsteht. Solche Terme wollen wir als *komplexe Terme* bezeichnen.

Diese Art der Klassifizierung von Termen ist in vielen Programmier- und Abfragesprache zu finden. So kann man beispielsweise in einer SQL-Anfrage (und

damit eigentlich im relationalen Modell) diese Arten von Termen ebenfalls finden. Dazu betrachte man folgende SQL Abfrage:

```
(2)   SELECT UPPER (Name) FROM ANGESTELLTEN WHERE
      Vorname="Hans";
```

Hier sind „`Name`“ und „`Vorname`“ variable Terme, „`Hans`“ ist ein konstanter Term, und „`UPPER (Name)`“ ist ein komplexer Term.

3 Umsetzung in das Java Objektmodell

Insbesondere die Terme habe natürliche Entsprechungen im Java Objektmodell. Konstante Terme werden als konkrete Objekte umgesetzt, während variable Terme als (typisierte und benannte) Parametertypen verwendet werden. Komplexe Terme können durch Anwendung von Java-Methoden auf andere Terme erzeugt werden.

Terme im Sinne von MANDARAX benutzen diese Interpretation, sind aber nicht darauf beschränkt. Der Vorteil dieses Ansatzes ist, daß ein bestehendes Objektmodell, welches ohne Rücksicht auf MANDARAX erstellt wurde, das Rohmaterial zur Erstellung von Regeln (und letztendlich von ganzen Wissensbasen / Knowledge Bases) liefert.

Das folgende Beispiel soll verdeutlichen, wie die obengenannte Regel formuliert werden kann. Vorausgesetzt sei ein Objektmodell, welches die Klassen „`Discount`“ und „`Customer`“ enthält, wobei Discount die Properties „`value`“ und „`isRelative`“ hat (ein relativer Rabatt wird in %, ein absoluter in $ angegeben), und die Klasse „`Customer`“ eine Methode „`int getTurnover (int month)`“ implementiert, welche die Summe aller Transaktionen über den entsprechenden Zeitraum liefert. Der Code ist etwas vereinfacht, insbesondere wird die Fehlerbehandlung ausgelassen.

(3)

```
import org.mandarax.kernel.*;
import org.mandarax.kernel.meta.JFunction;
import org.mandarax.math.IntArithmetic;
import org.mandarax.util.LogicFactorySupport;
// ...

// Instanzen werde mit einer Factory erzeugt,
// zusaetzlich wird eine weitere Utility-Klasse benuzt,
// die das generieren von Objekten weiter vereinfact
LogicFactory factory = LogicFactory.getDefaultFactory ();
LogicFactorySupport factorySupport =
  new LogicFactorySupport (factory);

// Erzeuge Term fuer "a customer"
Term aCustomer =
  factory.createVariableTerm ("aCustomer", Customer.class);

// Erzeuge Term  fuer "one year" (= 12 Monate)
Term oneYear = factory.createConstantTerm (new Integer (12));

// Erzeuge Term fuer "the value of all transactions
// of a customer in one year". new Integer (12) wird
// automatisch in einen constanten Term umbewandelt
Class[] parameterTypes = {Integer.TYPE};
Method aMethod = Customer.class.getMethod (
                   "getTurnover()", parameterTypes);
Function turnover = new JFunction (aMethod);
Term valueOfCustomer = factorySupport.cplx (turnover, aCustomer,
                         new Integer (12));

// Erzeuge Premisse. Das Standardpredikate > ist vordefiniert
Fact premisse = factorySupport.fact (IntArithmetic.GREATER_THAN,
                  valueOfCustomer, new Integer (100));
```

Die kompletten Skripts befinden sich in dem `org.mandarax.examples.crm` Package, was der MANDARAX Distribution beiliegt und vom MANDARAX Server heruntergeladen werden kann. Es sei angemerkt, daß es für Demos und Testzwecke zwar angemessen ist, Regeln so zu erzeugen. Im produktiven Einsatz jedoch sollten Regeln über ein User-Interface erzeugt werden und dann im Filesystem oder in einer Datenbank gespeichert werden. Für das Speichern ist insofern Unterstützung eingebaut, als dass alle Klassen, welche zum Aufbau von Regeln verwendet werden, serialisierbar sind. Zudem wird mittlerweile eine Schnittstelle zum RuleML XML Format angeboten. RuleML ist ein vom DFKI koordiniertes Projekt (http://www.dfki.uni-kl.de/ruleml/) welches das Ziel verfolgt, ein XML Format für die Beschreibung von Regelmengen zu definieren.

4 Arbeiten mit Regeln

Regeln werden von speziellen Container-Objekten, sogenannten Wissensbasen (KnowledgeBase), verwaltet. Diese besitzen eine recht einfache Schnittstelle mit `add()` und `get()` Methoden. Darüber hinaus verfügen sie intern über eine Art Indizierung (Keys genannt), welche im wesentlichen der Performanzverbesserung beim Zugriff auf Regeln dient. Ist eine solche Wissensbasis verfügbar, können Anfrage auf diese KnowledgeBase abgesetzt werden. Dazu dient eine sogenannte Inferenzmaschine (inference engine). Diese Inferenzmaschine implementiert die eingangs erwähnten Prolog-Algorithmen zur logischen Ableitung und benutzt darüber hinaus die Semantik des (Java-)Objektmodells um die Termstruktur ständig zu vereinfachen. Für entsprechend vorbelastete Leser sei erwähnt, das Teilalgorithmen wie Unifikation, Selektionsstrategie und Loop-Checking als Policies (in der Referenzimplementierung einer Inferenzmaschine) implementiert sind und so einfach modifiziert werden können.

Anfragen (queries) ähneln formal den einzelnen Prämissen und Konklusionen. Charakteristisch für Anfragen ist, daß sie mindestens einen variablen Term enthalten. Dieser wird versucht von der Inferenzmaschine durch einen konkreten Term zu ersetzen.

So würde etwa in unserem CRM Beispiel eine Anfrage für den Rabatt eines konkreten Kunden wie folgt aussehen:

```
(4)

// Implementierung mit Geschaeftsregelframework
public Discount getDiscount (Customer customer) {
  // Erzeuge Anfrage
  Class[] parameters = {Customer.class, Discount.class};
  Predicate getsDiscount =
    new SimplePredicate ("get discount", parameters);
  Fact query = factorySupport.fact (
                getsDiscount, customer, "a discount");

  // Anfrage stellen
  InferenceEngine ie = new ResolutionInferenceEngine ();
  KnowledgeBase kb = ...;  // hier sind die Regeln gespeichert
  Result result = ie.query (query, kb);
  Replacement replaced = result.getReplacements ()[0];
  Discount discount4Customer =
    (Discount) replaced.replacement.getObject ();
  return discount4Customer;
}
```

Das Resultat-Objekt liefert neben der Ersetzung von Variablen noch eine Instanz der Klasse `Derivation`, welche den Ableitungsbaum, der zu den Ersetzung geführt hat, repräsentiert. Aus dieser Baumstruktur läst sich dann etwa sehr leicht ein Swing-

Treemodel aufbauen, um dem Benutzer eine Übersicht über die benutzten Regeln zu geben. Darin zeigt sich ein grundlegender Vorteil gegenüber Ansätzen wie neuronalen Netzen: ein regelbasiertes System kann seine Entscheidungen erklären, und der Benutzer kann diese Erklärungen benutzen, um das System selbst zu justieren. Diese Transparenz ist insbesondere für unternehmenskritische Anwendungen von großer Bedeutung.

5 Integration von Fakten

Regeln operieren auf einer Faktenbasis. Wenn etwa Regel `(1)` ausgewertet wird, wird versucht entweder Regeln auszuwerten, welche die Prämissen von `(1)` als Konklusion haben, oder aber entsprechende Fakten zu finden. Fakten können in MANDARAX explizit erzeugt und zu einer Wissensbasis hinzugefügt werden. Das macht aber in den meisten Fällen wenig Sinn: die Fakten sollten zur Zeit der Anfrage generisch aus dem Objektmodell oder aus dem Inhalt einer Datenbank erzeugt werden. An dieser Stelle setzt das Integrationskonzept von MANDARAX an: Wissenbasen verwalten sogenannte Klauselmengen (clause sets), welche eine Menge von Klauseln beschreiben. Klauseln sind dabei entweder Regeln oder Fakten. Beschreiben bedeutet hier, das diese Menge nicht fix definiert ist, sondern nur ein Rezept zur Erzeugung einer solchen vorgegeben ist („deskriptive" Menge). Das ähnelt der Vorgehensweise in der Mathematik, wo etwa die Menge der Quadratzahlen nicht durch Angabe aller Elemente (was auf Grund der Unendlichkeit dieser Menge auch unmöglich wäre) angegeben wird, sondern durch die Angabe einer definierenden Eigenschaft. Charakteristisch für Klauselmengen ist ein Iterator. Solch ein Iterator wendet die Definition der Klauselmenge an und erzeugt die entsprechenden Klauseln (in der Regel sind dies Fakten). Ein Beispiel sollen das verdeutlichen. Angenommen, es sollen Fakten gefunden werden für Kunden `c`, für welche der Fakt „`the value of all transactions of a customer in one year is more than 100$`" wahr ist.

Fakten werden dynamisch aus einer Datenbankabfrage erzeugt. Dazu enthält die Wissenbasis eine Klauselmenge, die beim Aufruf der Methode[1] `clauses ()` eine SQL Anfrage[2] via JDBC auf eine Tabelle stellt, welche Kundentransaktionen enthält. Bei jedem Aufruf von `nextClause ()`[3] wird die nächste Zeile im Resultset der Anfrage gesucht, so daß der Umsatz im letzten Jahr grösser als 100$ war. Wenn ein solcher Datensatz gefunden ist (etwa für den Kunden „`Meier`"), wird ein entsprechender Fakt „`the value of all transactions of a customer` **`Meier`** `in one year is more than 100$`" erzeugt und von der Methode zurückgegeben.

---

[1] Diese Methode ist das Equivalent zu `iterator ()` in `java.util.Collection`.
[2] Wir gehen hier nicht weiter auf die Struktur der Tabellen ein, möglicherweise wuerde es sich hier um eine `group by` Abfrage handeln.
[3] Diese Methode entspricht `next ()` in `java.util.Iterator`.

Dieser Ansatz erweist sich als ausserordentlich flexibel. MANDARAX enthält zudem eine Menge von Utility-Klassen, die den Aufbau einer solchen Faktenbasis sehr erleichtern. Die Klasse `AutoFacts` beispielsweise benötigt nur ein Prädikat und die Informationen über die Mengen der Instanzen der beteiligten Klassen, um eine Faktenbasis aufzubauen. Im CRM Beispiel sieht die Definition der ensprechenden Klauselmenge wie folgt aus:

```
(6)

// DB ist hier eine Klasse die eine Datenbank simuliert
AutoFacts af = new AutoFacts () {
  public Collection getExtension (Class clazz) {
    if (clazz==Customer.class)
      return DB.getCustomers ().values ();
    if (clazz==Category.class)
      return DB.getCategories ().values ();
    if (clazz==KindOfPayment.class)
      return DB.getKindsOfPayment ().values ();
    return super.getExtension(clazz);
  }
};

// benutze vordefiniertes Standardpraedikat
// aus dem org.mandarax.math package
Predicate[] predicates = {DoubleArithmetic.GREATER_THAN};
af.setPredicates (predicates);
```

Wenn dann über diese AutoFacts-Instanz iteriert wird, werden alle Kombinationen der beteiligten Objekte gebildet, und auf diese Kombinationen werden rekursiv die dem Prädikat (und eventuellen Funktionen) entsprechenden Java-Methoden angewendet. Wenn das Resultat das Java-Objekt `Boolean.TRUE` ist, wird der entsprechende Fakt gebildet.

6 Randbedingungen und Anwendungsszenarien

Um MANDARAX oder verwandte Ansätze erfolgreich einsetzen zu können, sollten eine Reihe von Randbedingungen berücksichtigt werden. Wir wollen im folgenden auf diese Aspekte eingehen.

1. Persistenz der Wissensbasen. Wissenbasen/ Geschäftsregeln müssen persistent gespeichert werden. Dazu sind in der Regel relationale Datenbanken nicht besonders gut geeignet, da das entsprechende relationale Modell recht komplex ist und unter Umständen eine Vielzahl von Joins notwendig ist, um einzelne Regeln auszulesen. XML Files und objektorientierte Datenbanken sind hier besser geeignet. Auch Serialisierung

ist möglich, ist allerdings als Persistenzmechanismus in Produktionssystemen aufgrund der mangelhaften Skalierbarkeit und Versionierungsproblemen eher ungeeignet.

2. Benutzerschnittstellen. Die (graphischen) Benutzerschnittstellen müssen den (nicht IT-) Endanwender in die Lage versetzen, die Regeln zu verstehen und intuitiv zu bearbeiten. Das zu erreichen erweist sich unter Umständen als sehr schwierig, und war eines der Hauptprobleme etwa von Expertensystemen in den 80er Jahren. Allerdings sind hier (verbunden mit OO-Technologien wie Swing) große Fortschritte erzielt wurden, der Leser sollte sich davon selbst ein Bild machen (Verweise finden sich am Ende des Artikels).
3. Größe der Wissensbasis (= Anzahl der Regeln). Die Größe der Wissensbasis sollte nach oben beschränkt werden. Einige dutzend, möglicherweise auch einige hundert Regeln machen Sinn, große Wissensbasen mit tausenden von Regeln lassen sich vom Benutzer oft sehr schwer verwalten, da die Interaktionen und Abhängigkeiten der Regeln kaum noch nachvollziehbar sind. Tendenziell stoßen dann die Systeme auch an Performancegrenzen. Kommerzielle Systeme, die einen ähnlichen Ansatz wie MANDARAX verfolgen (wie ILOG JRules) skalieren hier (laut Aussagen des Herstellers) sehr gut, ihre Integration erfordert aber entsprechend mehr Aufwand. Obiges Beispiel ist ein recht gutes Anwendungsbeispiel für Geschäftsregelmodule, es ist realistisch anzunehmen, daß solcher Rabattrechner irgendwo zwischen 10 und 50 Regeln haben würde.
4. Komplexität der Regeln. Wenn die Komplexität der Regeln gering ist, kann es Sinn machen, einige Regeln zwar fix zu programmieren, diese aber mit Parametern zu versehen, die dann zur Laufzeit eingestellt werden können. Die Aufwände, was etwa die Persistenz und die Benutzerschnittstellen betrifft, sind dann oft geringer.

Insgesamt ist MANDARAX und verwandte Ansätze eher dazu geeignet, punktuell als gut gekapselte Module (hier sog. Rule-Engine) eingesetzt zu werden. Große Teile der Geschäftslogik sind nach wie vor relativ statisch, und dort sind fest implementierte oder parametrisierbare Regeln durchaus akzeptabel. Neben Lösungen, die dem oben diskutiertem Rabattrechner-Beispiel ähneln, dürfte insbesondere bei Systemen, die der Personalisierung von Diensten im Internet/Internet entsprechen, größeres Potential für regelbasierte Systeme liegen. Der MS-Outlook Email Client etwa erlaubt es dem Benutzer über einen Regel-Wizard Regeln zur automatisierten Verwaltung von eingehenden Emails zu erstellen. Ähnliche Anwendungen machen auch serverseitig Sinn: so könnten etwa Benutzer von Finanzportalen Regeln konfigurieren, nach welchen bestimmte Depot-Verwaltungsfunktionen automatisiert werden. Eine solche Regel könnte etwa wie folgt aussehen:

```
(7)  Wenn eine Aktie x um mehr als 5 % verliert und der Anteil
     dieser Aktie im Portfolio größer als 2% ist, schicke
     Warnung per SMS.
```

Die Regeln würden hier gewissermaßen das Gehirn eines Programmes bieten, welches serverseitig im Auftrage des Benutzers das Depot überwacht. Solche Ansätze entsprechen dem Konzept des *Softwareagenten*.

7 Verweise

Die Adresse der Homepage des MANDARAX-Projektes ist http://www.mandarax.org. Das MANDARAX-Projekt verwendet JUnit (http://ww.junit.org) für die Implementierung von Regressionstests. Darüber hinaus wird log4j als Logging Komponente verwendet. Log4j ist inzwischen Teil des Apache Jakarta-Projektes und kann von folgender Seite geladen werden: http://jakarta.apache.org/log4j

Mittlerweile existieren eine Reihe von Erweiterungen zum MANDARAX-Projekt. Dabei handelt es sich insbesondere um Module zur Persistenz (via XML Files) und Präsentation (Swing - Komponenten, JSP-Tag Libaries) von Regeln. Verweise zu ensprechenden Online-Beispielen finden sich auf der MANDARAX-Homepage. Siehe hierzu http://www.jbdietrich.de