

# Business Process Management (BPM) mit einem Enterprise Service Bus (ESB)

von

Robert Switzer

## 1 Einleitung

Anfang der 90er Jahre generierten die folgenden Schlagwörter viel Interesse unter IT-Verantwortlichen:

1. Workflow Management (WfM)
2. Enterprise Application Integration (EAI)
3. Business Process Management (BPM).

Heute haben sich die ersten beiden mehr oder minder unter dem dritten konsolidiert. Denn es hat sich die Erkenntnis durchgesetzt, daß der Begriff des Geschäftsprozesses (Business Process) im Mittelpunkt des Interesses stehen sollte.

Die vielen Anwendungen, die in einem Unternehmen eingesetzt werden, müssen nur dann integriert werden, wenn sie die einzelnen Schritte in einem Geschäftsprozeß bilden. Heute werden diese Einzelschritte als *Dienste* (engl. service) bereitgestellt und ihre Integration als Teil einer dienstorientierten Architektur (Service Oriented Architecture (SOA)) angesehen.

*Beispiel.* Ein typischer Geschäftsprozeß könnte folgendermaßen aussehen:

1. ein Kunde reicht eine Bestellung ein.
2. die Bestellung muß datentechnisch erfaßt werden.
3. die Einzelposten der Bestellung müssen im Lager zusammengetragen werden. In modernen Lagern wird dieser Schritt von Lagerrobotern erledigt.
4. die Bestellung muß versandfertig gemacht werden. Dieser Schritt muß höchstwahrscheinlich von einem menschlichen Mitarbeiter erledigt werden.
5. eine Rechnung muß erstellt und dem Kunden zugestellt werden.
6. der Kunde wird per Email benachrichtigt, daß seine Bestellung bereits unterwegs zu ihm ist.

Bis auf den Schritt 4. werden alle Schritte von Softwareanwendungen (Diensten) erledigt.

Im Vortrag über regelgesteuerte Prozesse haben wir ein Beispiel dafür gesehen, wie ein solcher Prozeß koordiniert werden könnte. In den meisten Fällen jedoch müssen die Dienste untereinander Daten austauschen; d. h. es wird eine Art Datenbus benötigt, der diesen Austausch ermöglicht. Das ist der Punkt wo der zweite Begriff unseres Titels ins Spiel kommt: der Enterprise Service Bus (ESB).

Ich finde, ich kann den ESB am besten erklären, indem ich aus [jdj] zitiere (mit Übersetzung):

Die dienstorientierte Architektur (SOA) ist weniger eine neue Technologie als vielmehr eine neue Geisteshaltung. Technologie, um Anwendungslogik in der mittleren Schicht (middle tier) zu implementieren und diese als Dienst verfügbar zu machen, gibt es schon seit Jahren. Zwar ist diese Technologie jetzt besser standardisiert und erfreut sich einer größeren Verbreitung. Was jedoch wirklich neu ist, ist die immer mehr verbreitete Praxis, SOA mit einem unternehmensweiten Blickwinkel umzusetzen. Dabei sucht das Unternehmen einen holistischen Ansatz zur Identifizierung der gebotenen Dienste.

Der Siegeszug des Enterprise Service Bus (ESB) stammt von dieser neuen unternehmensweiten Sichtweise. ESB und SOA ergänzen sich. Eines der Grundprinzipien von SOA ist, daß Dienste lose gekoppelt sein sollten.

Message Oriented Middleware (MOM) kann benutzt werden, um an sich unverbundene Systeme zu verbinden. Ein ESB wird geschaffen, indem standardbasierte MOM-Provider konsequent eingesetzt werden, um unternehmensweit sonst entkoppelte Dienste miteinander zu vernetzen. So eingesetzt entsteht das Echtzeitunternehmen mit einer ESB-getriebenen SOA als "Antrieb".

Die genaue Natur des ESB ist noch umstritten. Einige sehen ihn als Produkt, andere dagegen eher als architektonisches Entwurfsmuster (design pattern). Aber es herrscht allgemeine Einigkeit darüber, daß man den ESB als eine Art "Nervensystem" des Unternehmens ansehen kann. Er ist wie ein Automat (state machine), der seine Impulse in der Gestalt von Geschäftsereignissen (business events) erhält, die dadurch entstehen, daß Agenten (actors) Dienste in Anspruch nehmen, oder auch dadurch daß Ausnahmestände auftreten.

Diese Ereignisse werden vom ESB transportiert und werden auf der Basis von Geschäftsregeln ausgewertet und entsprechend umgesetzt. Die Regeln verbinden die einzelnen Dienste, um den Geschäftsprozeß zu bilden. Indem die Regeln zusätzliche Dienste aufrufen, werden weitere Ereignisse generiert und der ganze Zyklus beginnt wieder von vorne.

Gleichgültig, ob ein ESB ein Product ist oder nicht, ist es unerläßlich, daß die Dienste so entworfen werden, daß sie leicht in den ESB "eingesteckt" werden können, und daß sichergestellt ist, daß jeder Dienst ein Ereignis zum ESB veröffentlicht (publish), wenn er beansprucht wird. Dies ist ein typisches Beispiel von einem crosscutting concern. Es liegt also auf der Hand, hierfür AOP

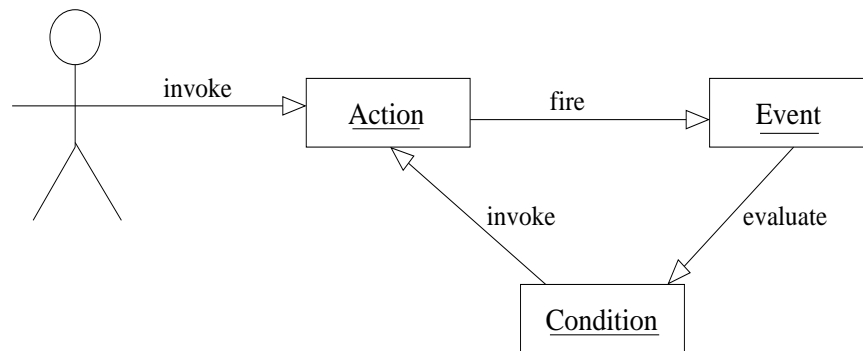


Abbildung 1: Kollaborationsdiagramm des Automaten

einzusetzen. So kann Legacycode in unserem Sinn erweitert und neuer Code kann sauberer gestaltet werden.

Eine sehr einfache Illustration der Funktionsweise unseres Automaten sehen wir in Abbildung 1.

In [jdj] beschreibt Gilbert ein BPM-ESB-Framework, das folgende Technologien einsetzt:

- den Java Message Service (JMS).
- AOP.
- EJB 3.0 in der vorläufigen Version von JBoss.
- den rule engine von drools.

Es ist dieses Framework, das ich in diesem Vortrag vorstellen möchte. Die Grundlagen dafür besitzen wir ja inzwischen.

*Bemerkungen.*

1. Mittlerweile hat Gilbert dieses Framework weiter ausgebaut und unter dem Namen Taylor als OpenSource-Software allgemein verfügbar gemacht.
2. Außer Taylor gibt es ein umfangreiches und recht anspruchsvolles BPM-Framework jBPM von Tom Baeyens, das die JBoss-Gruppe unter seine Fittiche genommen hat. Gilbert möchte Taylor als leichtgewichtige Alternative zu jBPM unter die Leute bringen.

## 2 Eine Beispielanwendung

Um den Gebrauch des Framework aus [jdj] (bzw. Taylor) zu illustrieren, implementieren wir den Prozeß, der ganz am Anfang dieses Vortrags beschrieben wurde.

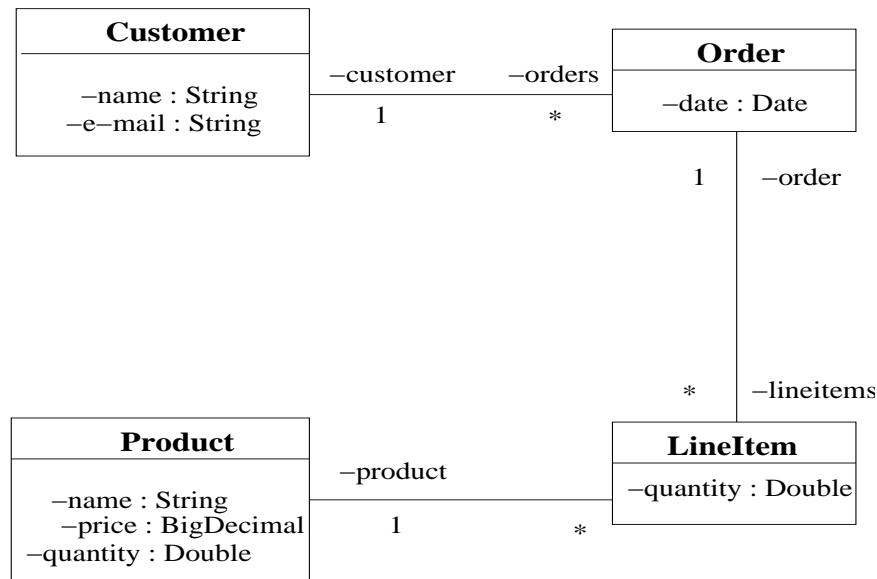


Abbildung 2: Klassendiagramm der Entities

Die Datenobjekte werden als Entity Beans implementiert. Unter EJB 3.0 heißt das, daß wir sie als POJOs implementieren können, wie wir jetzt wissen. Abbildung 2 zeigt das Klassendiagramm unserer Entities.

Außerdem brauchen wir drei Dienste, die wir als zustandslose Session Beans implementieren. Die Abbildung 3 zeigt das Klassendiagramm unserer drei Dienste.

### 3 Die Architektur des Framework

Die bisher beschriebenen Klassen sind natürlich Teil unserer expliziten Anwendung – gewissermaßen das Modell unseres Anwendungsbereichs. Wir wenden uns nun der Struktur des BPM-ESB-Framework zu.

Im vorigen Abschnitt haben wir unsere Dienste (Services) kennengelernt. Wir müssen natürlich dafür sorgen, daß unsere Dienste Ereignisse zum ESB veröffentlichen. Das ist, wie ich bereits sagte, ein crosscutting concern, den wir am besten mit AOP erledigen. Dafür setzen wir einen Interzeptor ein, wie wir sie aus meinem AOP-Vortrag kennen. Hier ist die Klasse unseres Interzeptors:

```

package esb.framework.event.interceptor;

import java.security.Principal;
import java.util.logging.*;
import javax.ejb.*;

import esb.framework.event.entity.Event;
  
```

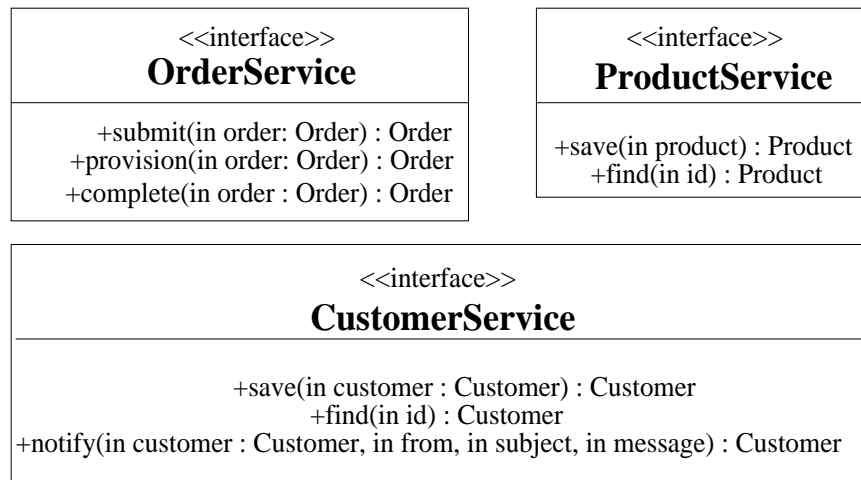


Abbildung 3: Klassendiagramm der Dienste

```

/**
 * All invocations that reach one of our service beans
 * are intercepted by this interceptor; corresponding
 * Event objects are built and dispatched.
 *
 */
public class EventInterceptor
{
    static private Logger logger =
        Logger.getLogger(EventInterceptor.class.getName());

    /**
     * This method is called before the invocation being
     * intercepted is actually executed.
     * @param ctx transports all relevant information about
     * the method call being intercepted.
     * @return - value returned by intercepted call.
     * @throws Exception any exception thrown by intercepted call.
     */
    @AroundInvoke
    public Object invoke(InvocationContext ctx)
        throws Exception
    {
        Event event = null;
        try
        {
            logger.info(ctx.toString());

```

```

        event = new Event();
        event.setName(ctx.getMethod().getName());
        event.setSource(ctx.getBean().getClass().getName());
        event.setArguments(ctx.getParameters());
        event.setUserId(getUserId(ctx));

        Object output = ctx.proceed();
        event.setOutput(output);
        return output;
    }
    catch (Exception e)
    {
        logger.throwing(this.toString(), e.getMessage(), e);
        event.setName(event.getName() + "Fault");
        event.setException(e);
        throw e;
    }
    finally
    {
        PublisherUtil.publish(event);
    }
}

protected String getUserId(InvocationContext ctx)
{
    Principal principal = ctx.getEJBContext().getCallerPrincipal();
    return (principal == null ? null : principal.getName());
}
}

```

*Kommentar:* Unsere Methode `invoke` ist mit der Annotation `@AroundInvoke` versehen, die wir aus dem AOP-Vortrag nicht kennen. Diese Annotation ist in Wirklichkeit die spezielle Art, wie EJB 3.0 AOP (und speziell Interzeptoren) in EJB einführt. Am Ende des AOP-Vortrags habe ich moniert, daß es noch keine AOP-Spezifikation gibt. Das wird bei EJB 3.0 anders sein; da sind die Interzeptoren Bestandteil der neuen EJB-Spezifikation – wenngleich ich den Eindruck habe, daß dieser Teil der Spezifikation noch nicht seine endgültige Gestalt gefunden hat. Es scheint dort noch keine Möglichkeit zu geben zu spezifizieren, dieser Interzeptor fängt nur bestimmte Methoden auf und andere nicht.

Das Wort `publish`, das hier mehrfach vorgekommen ist, legt nahe, daß wir unsere Ereignisse mit einem Message Service (wie etwa JMS) propagieren. Das ist in der Tat richtig, wie wir in unserer Hilfsklasse `PublisherUtil` sehen:

```
package esb.framework.event.interceptor;
```

```

import javax.jms.*;
import javax.naming.*;

import esb.framework.event.entity.Event;

public class PublisherUtil
{
    static private ConnectionFactory connectionFactory = null;
    static final private String topicName = "topic/esb.EventTopic";
    static private Destination topic = null;

    synchronized private static ConnectionFactory getConnectionFactory()
    {
        if (connectionFactory == null)
        {
            try
            {
                InitialContext ctx = new InitialContext();
                connectionFactory = (ConnectionFactory) ctx
                    .lookup("XAConnectionFactory");
            } catch (NamingException e)
            {
                throw new RuntimeException(e);
            }
        }
        return connectionFactory;
    }

    static public void setConnectionFactory(ConnectionFactory cf)
    {
        PublisherUtil.connectionFactory = cf;
    }

    synchronized public static Destination getTopic()
    {
        if (topic == null)
        {
            try
            {
                InitialContext ctx = new InitialContext();
                topic = (Destination) ctx.lookup(topicName);
            } catch (NamingException e)
            {
                throw new RuntimeException(e);
            }
        }
    }
}

```

```

    }
    return topic;
}

public static void setTopic(Destination topic)
{
    PublisherUtil.topic = topic;
}

/*
 * Publish the event to a well-know topic.
 */
static public void publish(Event event) throws Exception
{
    boolean fault = false;
    Connection connection = null;
    Session session = null;
    MessageProducer publisher = null;
    try
    {
        //on a fault we want to publish the event on its own transaction
        //this allows the original transaction to rollback
        fault = event.getName().endsWith("Fault");

        ConnectionFactory cf = getConnectionFactory();
        connection = cf.createConnection();
        connection.start();
        session = connection.createSession(fault, Session.AUTO_ACKNOWLEDGE);

        Destination topic = getTopic();
        publisher = session.createProducer(topic);

        ObjectMessage msg = session.createObjectMessage();
        msg.setObject(event);
        publisher.send(msg);
        if (fault)
        {
            session.commit();
        }
    } finally
    {
        if (publisher != null)
        {
            publisher.close();
        }
    }
}

```

```

        if (session != null)
        {
            session.close();
        }
        if (connection != null)
        {
            connection.close();
        }
    }
}
}

```

*Kommentar:*

1. Wer meinen Vortrag über JMS verinnerlicht hat, dürfte fast alles in dieser Klasse vertraut finden.
2. Hier gehen wir davon aus, daß der JMS-Administrator einen Topic erzeugt und unter dem Name `topic/esb.EventTopic` in den JNDI-Baum gehängt hat.
3. Wie man hier sieht, werden Ereignisse durch POJOs vom Typ `Event` repräsentiert. Ich werde die Klasse `Event` nicht zeigen, da man aus dem Code der oben gezeigten Klassen, schließen kann, welche Attribute diese Klasse haben muß.
4. Die Nachrichten die hier veröffentlicht werden, sind vom Typ `ObjectMessage` und transportieren ein Objekt vom Typ `Event`.
5. Wir wissen noch nicht, wer der Empfänger (Subscriber) unserer Ereignisse sein wird. Den werden wir im nächsten Abschnitt kennenlernen.

## 4 Der rule engine betritt die Bühne

Wir haben bisher keine Spur eines rule engine gesehen. Der tritt in unserem Empfänger (Subscriber) in seiner Hauptrolle auf.

In unserem EJB3-Vortrag haben wir gelernt, daß es außer den Session und den Entity Beans auch noch die Message-Driven Beans gibt, die nicht klassische RPC-Middleware sondern eher Message Oriented Middleware (MOM) implementieren.

Die Klasse eines Message Driven Bean (MDB) erweitert ein `Listener`-Interface; welches Interface das ist, hängt von der eingesetzten Nachrichtentransporttechnologie ab. Mit JMS ist dieses Interface `MessageListener` – ein Interface, das nur eine einzige Methode besitzt:

```
void onMessage(Message m);
```

Es scheint, daß unser Empfänger fast zwangsläufig ein MDB sein sollte. Wir akzeptieren diese Erkenntnis und setzen folgende Klasse als Empfänger ein:

```

package esb.framework.workflow.listener;

import java.util.*;
import java.util.logging.*;
import javax.annotation.Resource;
import javax.ejb.*;
import javax.jms.*;
import javax.rules.*;
import esb.framework.event.entity.Event;
import esb.framework.util.JMSUtil;

/**
 * A message driven bean to support asynchronous handling of events.
 */
@MessageDriven(activateConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "destination",
        propertyValue = "topic/esb.EventTopic"),
    @ActivationConfigProperty(propertyName = "transaction-type",
        propertyValue = "Container"),
    @ActivationConfigProperty(propertyName = "acknowledge-mode",
        propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(propertyName = "subscription-durability",
        propertyValue = "Durable") })
public class RulesMDB implements MessageListener
{
    static private Logger logger =
        Logger.getLogger(RulesMDB.class.getName());

    @Resource
    private MessageDrivenContext context;

    @Resource(name = "esb.RuleRuntime")
    private RuleRuntime rt;

    public RulesMDB()
    {
        super();
    }

    /**
     * The MessageListener callback.
     * It does the following:
     * 1) extracts the event from the message.

```

```

    * 2) applies the ruleset named "OrderProcess" to this event.
    */
public void onMessage(Message message)
{
    try
    {
        Event event = getEvent(message);
        logger.info(JMSUtil.toString(message));
        onEvent(event);
    }
    catch (RuntimeException e)
    {
        logger.throwing(this.toString(), e.getMessage(), e);
        throw e;
    }
    catch (Throwable t)
    {
        logger.throwing(this.toString(), t.getMessage(), t);
        context.setRollbackOnly();
        throw new RuntimeException(t);
    }
}

/**
 * Extract the event object from the message.
 * @param message the message transporting the event.
 * @return - the extracted event.
 * @throws JMSEException
 */
protected Event getEvent(Message message) throws JMSEException
{
    Event event = null;
    event = (Event) ((ObjectMessage) message).getObject();
    event.setId(message.getJMSMessageID());
    return event;
}

/**
 * Apply ruleset named "OrderProcess" to input consisting of one event.
 * @param event to take as input.
 * @throws Exception
 */
protected void onEvent(Event event) throws Exception
{
    List<Event> input = new ArrayList<Event>();

```

```

        input.add(event);
        execute("OrderProcess", input);
    }

    /**
     * Run rule runtime on a given rule set with given input.
     * @param ruleset the name of the ruleset to apply.
     * @param input the input to the ruleset.
     * @return - the output generated by the ruleset.
     * @throws Exception
     */
    protected List execute(String ruleset, List input) throws Exception
    {
        StatelessRuleSession srs = (StatelessRuleSession)
            rt.createRuleSession(ruleset,
                                null,
                                RuleRuntime.STATELESS_SESSION_TYPE);
        return srs.executeRules(input);
    }
}

```

*Kommentar:*

1. Ein MDB wird mit der Annotation `@MessageDriven` gekennzeichnet. Diese Annotation hat ein Element `activateConfig`, in dem diverse Festlegungen gemacht werden können, die früher in einem deployment descriptor stehen mußten. Dazu gehören Angaben über den Nachrichtenendpunkt sowie über die Art, wie Transaktionen gehandhabt werden sollen.
2. Unser MDB bekommt den Zugriff auf den rule engine mittels Dependency Injection, wie wir sie im Vortrag über EJB 3.0 kennengelernt haben.
3. Die Methode `onMessage` extrahiert das Objekt vom Typ `Event` aus der Nachricht und delegiert die Behandlung dieses Ereignisses an eine `protected` Methode `onEvent`. Diese delegiert wiederum an eine Methode `execute`, wo die rule execution set aktiviert wird.
4. Die Methode `onMessage` differenziert in den `catch`-Klauseln zwischen "unchecked" Exceptions (Typ `RuntimeException`) einerseits und allen anderen Exceptions andererseits (zu diesen können auch anwendungsdefinierte Exceptions gehören). Erstere werden nur im Logfile vermerkt und weitergeworfen, während die anderen zusätzlich dazu führen müssen, daß die aktuelle Transaktion zurückgedreht wird (`rollback`) und außerdem müssen diese in einer `RuntimeException` verpackt werden, bevor sie weitergeworfen werden.

Daß wir hier die aktuelle Transaktion nicht unmittelbar mit `rollback` beenden können, liegt daran, daß wir nicht der Initiator der Transaktion waren; das war

tatsächlich der JMS-Provider. Darum können wir allenfalls dafür sorgen, daß die aktuelle Transaktion nur mit `rollback` beendet wird. Wie man das in einem Enterprise Bean macht, sehen wir hier.

5. Der Name der einzusetzenden rule set ist hier hardwired (`OrderProcess`). In einem Framework zum allgemeinen Gebrauch, wie es Taylor ist, kann man natürlich nicht so vorgehen. Dort hat die Methode `execute` kein Argument `ruleset`; stattdessen iteriert sie über alle registrierten rule execution sets.

## 5 Unsere Regeln

Als nächstes wollen wir die Regeln ansehen, die unseren Prozeß steuern. Hier sind sie:

```
<?xml version="1.0"?>

<rule-set name="OrderProcess"
  xmlns="http://drools.org/rules"
  xmlns:java="http://drools.org/semantics/java"
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xs:schemaLocation="http://drools.org/rules rules.xsd
                    http://drools.org/semantics/java java.xsd">

  <java:import>java.lang.Object</java:import>
  <java:import>java.lang.String</java:import>

  <java:import>esb.framework.event.entity.Event</java:import>
  <java:import>esb.feamework.workflow.entity.Task</java:import>
  <java:import>esb.framework.workflow.service.task.*</java:import>

  <java:import>esb.model.entity.*</java:import>
  <java:import>esb.model.service.order.*</java:import>
  <java:import>esb.model.service.customer.*</java:import>
  <java:import>esb.model.service.product.*</java:import>

  <java:functions>
    import esb.model.service.customer.*;
    import esb.model.service.order.*;
    import esb.model.service.product.*;
    import esb.framework.workflow.service.task.*;
    import esb.framework.workflow.service.task.*;
    import esb.framework.workflow.listener.RulesUtil;
    import java.util.logging.*;

    public void log(String rule, String source, String name)
    {
```

```

        Logger.getLogger("esb.framework.rules").info(
            "ASSERTED:" + rule + ":" + source + ":" + name);
    }

    public CustomerService getCustomerService() {
        return (CustomerService)
            RulesUtil.lookup(CustomerServiceLocal.class);
    }

    public OrderService getOrderService() {
        return (OrderService)
            RulesUtil.lookup(OrderServiceLocal.class);
    }

    public TaskService getTaskService() {
        return (TaskService)
            RulesUtil.lookup(TaskServiceLocal.class);
    }

    public ProductService getProductService() {
        return (ProductService)
            RulesUtil.lookup(ProductServiceLocal.class);
    }
}
</java:functions>

```

```

<rule name="fault">
    <parameter identifier="event">
        <class>Event</class>
    </parameter>

    <java:condition>
        event.getName().endsWith("Fault")
    </java:condition>

    <java:consequence>
        log("fault", event.getSource(), event.getName());
    </java:consequence>
</rule>

```

```

<rule name="onSubmitOrder">
    <parameter identifier="event">
        <class>Event</class>
    </parameter>

    <java:condition>
        event.getName().equals("submit")
    </java:condition>
    <java:condition>

```

```

        event.getSource().equals(OrderServiceBean.class.getName())
</java:condition>

<java:consequence>
    log("onSubmitOrder", event.getSource(), event.getName());

    Order o = (Order) event.getOutput();
    OrderService service = getOrderService();
    service.provision(o);
</java:consequence>
</rule>

<rule name="onProvisionOrder">
    <parameter identifier="event">
        <class>Event</class>
    </parameter>

    <java:condition>
        event.getName().equals("provision")
    </java:condition>
    <java:condition>
        event.getSource().equals(OrderServiceBean.class.getName())
    </java:condition>

    <java:consequence>
        log("onProvisionOrder", event.getSource(), event.getName());

        Order o = (Order) event.getOutput();
        Task task = new Task(o, "Ship Order: " + o.getId(), "Shipper");
        TaskService service = getTaskService();
        service.initiate(task);
    </java:consequence>
</rule>

<rule name="onProvisionOrder2">
    <parameter identifier="event">
        <class>Event</class>
    </parameter>

    <java:condition>
        event.getName().equals("provision")
    </java:condition>
    <java:condition>
        event.getSource().equals(OrderServiceBean.class.getName())
    </java:condition>

```

```

<java:consequence>
    log("onProvisionOrder2", event.getSource(), event.getName());

    Order o = (Order) event.getOutput();
    Customer c = o.getCustomer();

    String from = "customer.service@acme.com";
    String subject = "Order Accepted";
    String message = "Order #" + o.getId() +
        " has been received and processed.\n\n";
    message += event.toString();
    CustomerService service = getCustomerService();
    service.notify(c, from, subject, message);
</java:consequence>
</rule>

<rule name="onProvisionOrderFault">
    <parameter identifier="event">
        <class>Event</class>
    </parameter>

    <java:condition>
        event.getName().equals("provisionFault")
    </java:condition>
    <java:condition>
        event.getSource().equals(OrderServiceBean.class.getName())
    </java:condition>

    <java:consequence>
        log("onProvisionOrderFault", event.getSource(), event.getName());

        Order o = (Order) event.getArguments()[0];
        Customer c = o.getCustomer();
        Exception e = event.getException();

        String from = "customer.service@acme.com";
        String subject = "Order Errors";
        String message = "Order #" + o.getId() +
            " has the following errors:\n";
        message += e.getMessage();

        CustomerService service = getCustomerService();
        service.notify(c, from, subject, message);
    </java:consequence>
</rule>

```

```

</rule>

<rule name="onShippingComplete">
  <parameter identifier="event">
    <class>Event</class>
  </parameter>

  <java:condition>
    event.getSource().equals(TaskServiceBean.class.getName())
  </java:condition>
  <java:condition>
    event.getName().equals("complete")
  </java:condition>
  <java:condition>
    event.getArguments()[1].equals("shipped")
  </java:condition>

  <java:consequence>
    log("onShippingComplete", event.getSource(), event.getName());

    Task t = (Task) event.getOutput();
    Order o = (Order) t.getTarget();
    Customer c = o.getCustomer();

    String from = "customer.service@acme.com";
    String subject = "Order Shipped";
    String message = "Order #" + o.getId() +
        " has been shipped.\n\n";
    message += event.toString();
    CustomerService service = getCustomerService();
    service.notify(c, from, subject, message);
  </java:consequence>
</rule>

<rule name="onShippingComplete2">
  <parameter identifier="event">
    <class>Event</class>
  </parameter>

  <java:condition>
    event.getSource().equals(TaskServiceBean.class.getName())
  </java:condition>
  <java:condition>
    event.getName().equals("complete")
  </java:condition>

```

```

<java:condition>
    event.getArguments()[1].equals("shipped")
</java:condition>

<java:consequence>
    log("onShippingComplete2", event.getSource(), event.getName());

    Task t = (Task) event.getOutput();
    Order o = (Order) t.getTarget();

    OrderService service = getOrderService();
    service.complete(o);
</java:consequence>
</rule>

<rule name="debug">
    <parameter identifier="event">
        <class>Event</class>
    </parameter>

    <java:consequence>
        log("debug", event.getSource(), event.getName());
    </java:consequence>
</rule>

</rule-set>

```

*Kommentar:*

1. Jedem, der meinen Vortrag über rule engines verinnerlicht hat, müßten diese Regeln im Großen und Ganzen vertraut erscheinen.
2. Die etwas umständliche Aufgabe, den einzusetzenden Dienst ausfindig zu machen, wird zunächst an eine im Abschnitt `java:functions` definierte Methode delegiert. Diese delegiert ihrerseits weiter an eine Hilfsklasse `RulesUtil`, die die langweilige Arbeit des Suchens im JNDI-Baum erledigt. Auf diese Weise gewinnt unsere rule execution set an Lesbar- und Verständlichkeit.
3. In unserer Regel `onProvisionOrder` ist von einer Task Service die Rede. Im Rahmen dieses Framework ist ein *Task* eine Aufgabe, die von einem menschlichen Mitarbeiter zu erledigen ist. Die Vorstellung ist, daß die beteiligten Mitarbeiter einen elektronischen "Eingangskorb" haben werden, wo sie die Tasks vorfinden, die aktuell anliegen und die sie qualifiziert sind, auszuführen.
4. Hier sehen wir, wie unser ESB mit auftretenden Fehlern fertig wird: auch diese werden als Ereignisse behandelt, deren Namen mit `Fault` endet. So brauchen wir keinen komplizierten Fehlerbehandlungscode, der mit der eigentlichen Anwendung

wenig zu tun hat. Auch die Fehlerbehandlung wird deklarativ geregelt (Wortspiel ist volle Absicht!).

## 6 Der Rule Engine als JBoss-Service

Es ist noch ziemlich unklar geblieben, wie der rule engine unserem ESB verfügbar gemacht wird. In der Klasse `RulesMDB` haben wir zwar gesehen, daß unser MDB den Zugriff auf ein Objekt vom Typ `RuleRuntime` durch Dependency Injection erhält; das heißt letztlich, daß der Container dieses Objekt beschaffen und in unser MDB “injizieren” muß. Unsere Frage reduziert sich letztendlich auf die Frage, wieso der Container (d. h. JBoss) den Zugriff auf den rule engine hat.

Um dies zu erklären muß ich an Stefan’s Vortrag im Januar über JMX und MBeans erinnern. JMX ist die Java Management eXtension, mit der jede Anwendung von außen beobachtet und gesteuert werden kann. JMX ist das Erfolgsgeheimnis von JBoss. Der JBoss-Kern ist eine Art von Rückgrat, an dem beliebig viele Mbeans (Mbean = Managed Bean) angeschlossen werden können. Alle JBoss-Services – ob JNDI, JMS, Sicherheit oder Transaktionen – werden durch entsprechende MBeans implementiert.

Jeder kann eigene maßgeschneiderte JBoss-Services kreieren: er muß nur passende Mbeans programmieren und diese in einem sog. Service-Archive mit Extension `sar` verpacken, in dem sein Service installiert (deployed) werden kann. So machen wir es mit dem rule engine.

Hierzu müssen wir drei Artefakte schreiben:

- ein Interface für unser MBean; wir nennen es `RulesMbean`
- eine Implementationsklasse für unser MBean; wir nennen sie einfach `Rules`.
- einen Deskriptor `jboss-service.xml`, der alles Wesentliche über unser Mbean festlegt.

Unser Interface `RulesMBean` enthält das absolute Minimum, das in einem MBean-Interface stehen muß:

```
package esb.framework.rules.jmx;

/**
 * Interface for MBean for JSR-94 compliant Rules Engines
 */
public interface RulesMBean
{

    /**
     * The location in JNDI where the engine will be bound
     */
    public String getJndiName();
}
```

```

    public void setJndiName(String jndiName);

    /**
     * The start lifecycle operation
     */
    public void start() throws Exception;

    /**
     * The stop lifecycle operation
     */
    public void stop() throws Exception;
}

```

Und hier nun ist die zugehörige Implementationsklasse:

```

package esb.framework.rules.jmx;

import java.io.InputStream;
import java.util.logging.Logger;

import javax.naming.InitialContext;
import javax.rules.*;
import javax.rules.admin.*;

/**
 * An MBean for JSR-94 compliant Rules Engines
 */

public class Rules implements RulesMBean
{
    static private Logger logger =
        Logger.getLogger(Rules.class.getName());

    private boolean started = false;

    private String jndiName = "";

    public Rules() {}

    public String getJndiName() { return jndiName;}

    public void setJndiName(String jndiName)
    {
        String oldName = this.jndiName;

```

```

        this.jndiName = jndiName;
        recycle(oldName);
    }

    public void start()
    {
        started = true;
        bind();
    }

    public void stop() throws Exception
    {
        started = false;
        unbind(jndiName);
    }

    private void bind()
    {
        try
        {
            logger.info(jndiName);
            InitialContext rootCtx = new InitialContext();
            Object o = getServiceProvider().getRuleRuntime();
            rootCtx.bind(jndiName, o);
        }
        catch (Exception e)
        {
            logger.throwing(this.toString(), e.getMessage(), e);
            throw new RuntimeException(e);
        }
    }

    private void unbind(String jndiName)
    {
        logger.info(jndiName);
        try
        {
            InitialContext rootCtx = new InitialContext();
            rootCtx.unbind(jndiName);
        }
        catch (Exception e)
        {
            logger.throwing(this.toString(), e.getMessage(), e);
        }
    }
}

```

```

private void recycle(String jndiName)
{
    if (started)
    {
        unbind(jndiName);
        bind();
    }
}

private RuleServiceProvider getServiceProvider() throws Exception
{
    logger.info(this.toString());
    String provider =
        "org.drools.jsr94.rules.RuleServiceProviderImpl";
    ClassLoader cl = this.getClass().getClassLoader();
    Class clazz = cl.loadClass(provider);
    RuleServiceProviderManager.registerRuleServiceProvider(
        provider, clazz, cl);
    RuleServiceProvider serviceProvider =
        RuleServiceProviderManager.getRuleServiceProvider(provider);
    RuleAdministrator ruleAdministrator =
        serviceProvider.getRuleAdministrator();

    registerRuleExecutionSet(ruleAdministrator, "/rules.xml");
    return serviceProvider;
}

private void registerRuleExecutionSet(RuleAdministrator ruleAdministrator,
    String url) throws Exception
{
    logger.info(url);
    InputStream inStream =
        this.getClass().getResourceAsStream(url);
    RuleExecutionSetProvider resp =
        ruleAdministrator.getLocalRuleExecutionSetProvider(null);
    RuleExecutionSet res1 = resp.createRuleExecutionSet(
        inStream, null);

    inStream.close();
    String uri = res1.getName();
    logger.info(uri);
    ruleAdministrator.registerRuleExecutionSet(uri, res1, null);
}
}

```

*Kommentar:* Alles Interessante passiert in den privaten Hilfsmethoden – namentlich in

```
getServiceProvider  
registerRuleExecutionSet
```

Aber spätestens seit meinem Vortrag über rule engines müßte der Inhalt dieser Methoden vertraut sein.

Und nun – damit wir auch wirklich alles wissen – wollen wir unseren Deskriptor sehen:

```
<?xml version="1.0" encoding="UTF-8"?>  
  
<service>  
  
  <mbean code="esb.framework.rules.jmx.Rules"  
        name="esb.framework.rules:service=Rules">  
  
    <!--Attribute JndiName, type java.lang.String, The location in  
         JNDI where the engine will be bound-->  
    <attribute name="JndiName">esb.RuleRuntime</attribute>  
  
    <depends>jboss:service=Naming</depends>  
  
  </mbean>  
  
</service>
```

*Kommentar:*

1. Das Wurzelement dieses Deskriptors heißt **service**. Es kann beliebig viele **mbean**-Subelemente enthalten, die alle MBeans beschreiben, die zum Service gehören. In unserem Fall haben wir nur ein solches MBean.
2. Das **mbean** Element muß die zwei Attribute **code** und **name** besitzen. Das erste nennt den qualifizierten Klassennamen der Implementationsklasse des Means. Das **name**-Attribut gibt dem MBean einen Namen, mit dem es beim JMX-Kern registriert werden kann; dieser Name muß im ganzen Server eindeutig und auch ein nach der JMX-Spezifikation gültiger Name sein.
3. Ein **mbean**-Element kann beliebig viele **depends**-Subelemente haben. Diese nennen Services, von denen das aktuelle MBean abhängt.

MBean *M* hängt von Service *S* ab, wenn es keinen Zweck hat, *M* zu instanziiieren, solange *S* noch nicht läuft. In unserem Fall z. B. kann unser MBean nicht funktionieren, wenn das JNDI noch nicht da ist. Darum hängt das MBean vom Service **Naming** (=JNDI) ab.

## 7 Schlußbemerkungen

Die Dienste sind die Grundbausteine von einem Geschäftsprozeß. Sie können so programmiert werden, daß sie sich auf die eigentliche Anwendungslogik konzentrieren. Sie brauchen keinen störenden Code zu enthalten, der dafür sorgt, daß passende Ereignisse propagiert werden. Es können beliebig viele Dienste an den ESB angeschlossen werden, ohne daß sie sich gegenseitig stören. Regeln können ebenfalls hinzugefügt, weggenommen oder modifiziert werden, ohne daß die Dienste deswegen angepaßt werden müssen.

## Literatur

- [jdj] <http://java.sys-con.com/read/84658.htm>. Hier findet man den JDJ-Artikel *AOP-enabled ESB* von John Gilbert aus der Mai-Ausgabe des *Java Developer's Journal*.
  
- [esb] <http://www.bijonline.com/pdf/sep03saini.pdf>. Und hier ein Artikel von Atul Saini, der ESB erklärt.