



jBPM meets ESB

Prozesse, Agilität, heterogene Systeme – das sind Themen, die uns Informatiker heute umtreiben. Als Lösungsansätze kursieren Konzepte wie BPM und SOA. Als konkrete Produkte werden Business Process Engines und der Enterprise Service Bus (ESB) gehandelt. Dabei können viele Aufgaben prinzipiell von beiden Tools erledigt werden. Wofür soll man sich entscheiden? Oder wie kann man beides sinnvoll kombinieren? Am Beispiel JBoss jBPM, Mule ESB und JBoss ESB soll dies in der Praxis betrachtet werden.

von Markus Demolsky und Bernd Rücker

Sieht man von Use-Case-getriebener Software wie einer Textverarbeitung ab, so bilden die meisten Softwaresysteme in Unternehmen Prozesse ab. Dies kann von der automatischen Archivierung einer eingehenden E-Mail über die punktuelle Anbindung externer Partner über EDIFACT oder Web Services bis hin zu komplexen, langlaufenden Geschäftsprozessen gehen wie etwa einem Auftrags- oder Produktionsprozess. Die Abarbeitung folgt dabei immer einer Reihe von defi-

nierten Aktivitäten und macht dadurch aus einem gegebenen Event (Input) ein Ergebnis (Output). Die Prozesse können durch unterschiedlichste Events gestartet werden, z.B. durch eine eingehende Mail oder JMS-Message, eine neue Datei oder auch einen aufgerufenen (Web-)Service.

Die Implementierung der auszuführenden Aktivitäten und deren Aufruf in der richtigen Reihenfolge können auf unterschiedlichste Weise erfolgen. Klassisch würde der Prozess am einfachsten

hart codiert werden, eventuell in Java. Dies hat jedoch einige Nachteile: Man ist sehr unflexibel, da Änderungen am Prozess Änderungen im Java-Code nach sich ziehen. Änderungen an den Schnittstellen, z.B. wenn die Daten plötzlich als Web-Service-Aufruf übergeben werden sollen, anstatt per E-Mail-Robot, führen ebenfalls zu nicht unerheblichem Änderungsaufwand. Diese Vorgehensweise passt also nicht so richtig zur ständig geforderten Agilität der IT. Eine Lösungsidee zur höheren Agilität stellt das

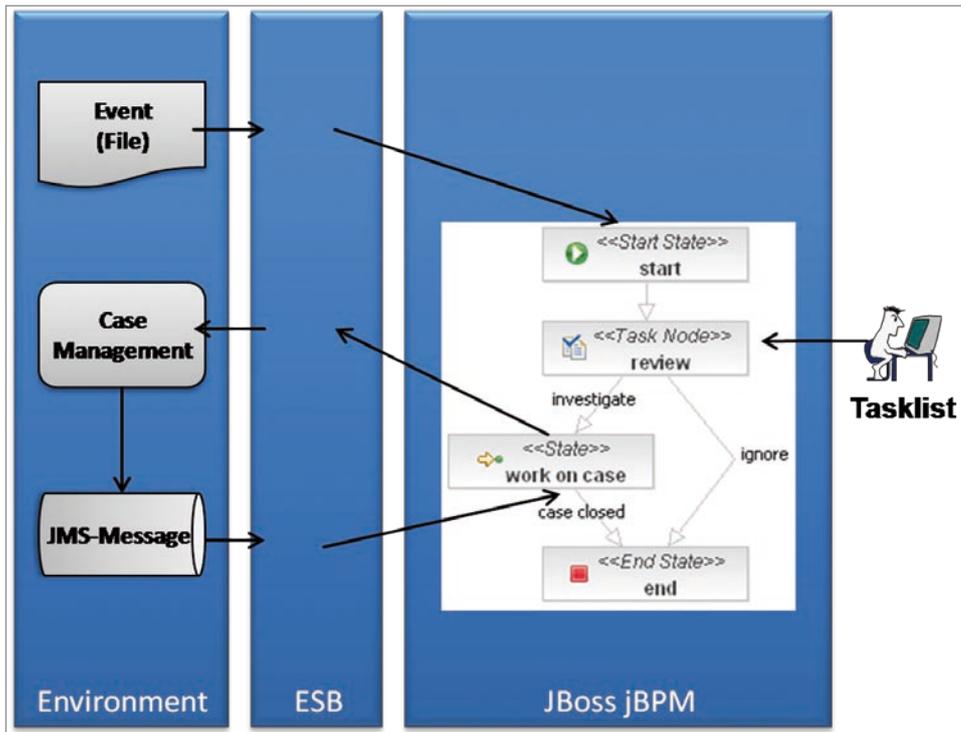


Abb. 1: Beispielanwendung

SOA-Paradigma bereit: Nur die eigentliche Geschäftsfunktionalität wird programmiert und als so genannter Service bereitgestellt. Der Prozess wird nicht programmiert, sondern konfiguriert. Im Prozessverlauf werden dann automatisiert verschiedene Services aufgerufen (man spricht dabei von Orchestrierung). Services können dabei über unterschiedliche Protokolle und Technologien angesprochen werden, etwa über Web Service, REST oder JMS.

Enterprise Service Bus (ESB)

Für diesen Zweck kann ein ESB zum Einsatz kommen. Er kann eingehende Events und Nachrichten über verschiedenste Technologien und Protokolle entgegennehmen und für eintreffende Events entsprechende Aktivitäten aufrufen. Dies funktioniert meist, indem eine interne Nachricht erstellt wird, die dann entsprechend von Service zu Service geroutet wird. Die Routing-Logik kann dabei beliebig komplex werden, angefangen bei statischem Routing (immer nach Eintreffen der E-Mail wird diese an den Archivierungsservice geschickt) bis zu Content-based Routing (CBR), wo die Nachricht je nach Inhalt an andere Services geroutet wird (ein Produkti-

onsauftrag könnte je nach Ware an den Web Service des PPS im richtigen Werk geschickt werden). Letzteres kann dabei sogar über eine Rule Engine passieren, sodass der Komplexität wirklich keine Grenzen gesetzt sind. Der ESB bringt weiterhin viel Unterstützung für Transformation von Daten mit, sodass er eine gute Grundlage für eine SOA darstellen kann.

Somit können Prozesse mithilfe des ESB durch Zusammenstecken von Services umgesetzt werden, da die Reihenfolge der Aktivitäten durch diverse Routing-Regeln definiert ist. Der Prozess ist also rein konfigurativ entstanden und die geforderte Agilität dadurch erfüllbar. Unsere Anforderungen sind also mit dem ESB prima umgesetzt, oder?

Die offene Frage impliziert es natürlich: Es gibt eine Anforderung, die durch den ESB nicht adressiert wird, die Visualisierung des Geschäftsprozesses. Möchte man mit dem Fachbereich einen Prozess diskutieren, so hilft es genauso wenig, wenn dieser in ESB-Konfiguration statt in Java-Code gegossen ist. Meist wird der Prozess daher im Rahmen der Anforderungen zwar grafisch gezeichnet, dann aber nicht weiterverwendet, geschweige denn aktuell gehalten. Nun

könnte man dank Model-driven Architecture (MDA) ein UML-Aktivitätsdiagramm zeichnen und daraus die ESB-Konfiguration generieren, dann hätte man Diagramm und Code immer in sync. In der Tat wäre dies sicherlich gar keine so schlechte Idee, allerdings hat sie den Nachteil, dass man viel Arbeit in den Generator investieren muss.

Business Process Engine

Eine andere Möglichkeit, den Prozess als Prozess zu modellieren, kommt aus der Ecke des Business Process Modeling (BPM): die Prozessmaschine (Business Process Engine). Sie erlaubt es, Prozessmodelle direkt zu interpretieren und auszuführen. Dies hat den eindeutigen Vorteil, dass der Prozess visualisiert werden kann und an einem Stück vorliegt, also nicht über diverse Routing-Regeln im ESB verteilt ist. Daneben kann die Prozessmaschine zusätzliche Funktionalitäten bereitstellen. Zum Beispiel kann sie Prozesskennzahlen wie die Durchlaufzeit messen, Informationen über aktuell laufende Prozesse bereithalten und auch To-Do-Listen für Benutzer verwalten (das so genannte Human Task Management), falls menschliche Interaktion gefragt ist, wenn doch nicht alles automatisiert werden kann.

Um noch einmal zur MDA-Idee zurückzukommen: Manche Prozessmaschinen sind intern wirklich so ähnlich aufgebaut. Sie arbeiten mit Queues und dem „Prozessfluss“ (routet Nachrichten durch diese Queues). Der Vorteil ist aber immer noch, dass dies nicht selbst implementiert werden muss. Um Missverständnissen vorzubeugen: Es gibt auch viele Process Engines, die intern anders und ohne Queues arbeiten, nämlich meist mit der direkten Umsetzung eines Zustandsautomaten.

ESB vs. Prozessmaschine

Okay, wir kennen nun zwei Konzepte, die unsere Agilität in Bezug auf Prozesse erhöhen sollen, den ESB und die Prozessmaschine. Da stellt sich natürlich die Frage, was ich wann benutze und ob es vielleicht auch Sinn macht, beides zu kombinieren.

Prinzipiell macht es Sinn, über den Einsatz einer Prozessmaschine nachzu-

denken, sobald die Prozesse eine gewisse Komplexität erreichen oder Aufgabenverwaltung eine Rolle spielt. Auf der anderen Seite sollte der Einsatz eines ESB erwogen werden, wenn viele heterogene Technologien und Protokolle anzubinden sind. Kombiniert man beide Technologien, was sehr gut möglich ist, bleibt die Frage, welche Logik in den ESB wandert und welche im Prozess in der dafür vorgesehenen Maschine verbleibt. Bei dieser Entscheidung ist in der Praxis Fingerspitzengefühl und Erfahrung gefragt. Leider sind hier noch keine allgemeingültigen Vorgehensweisen verfügbar. Auf jeden Fall kann die Kombination verhindern, dass Prozesslogik im ESB versteckt wird und Integrationslogik den Geschäftsprozess aufbläht.

Beispielanwendung

Die Kombination soll in diesem Artikel am Beispiel der Open Source Process Engine JBoss jBPM sowie zwei verschiedenen, ebenfalls quelloffenen ESB-Implementierungen gezeigt werden: Mule ESB und JBoss ESB. Diese Kombinationen werden hier nur beispielhaft herausgegriffen, da es sich um bekannte und verbreitete Open-Source-Projekte handelt. Letztendlich ist die Integration auch anderer Produkte im Bereich BPM und ESB meist genauso möglich, die Sprache WS BPEL (Business Process Execution Language) ist ja sogar speziell für die Orchestrierung von Web Services in einer SOA ausgelegt und passt damit natürlich sehr gut in die Umgebung eines ESB.

Im Beispiel wird ein vorhandenes Case-Management-System mit verschiedensten Daten gefüllt. Die Startevents können aus unterschiedlichen Quellen stammen, in diesem Beispiel bedeutet ein Event, dass eine neue Datei in einem Verzeichnis eintrifft. Dieses Event wird durch den ESB aufgenommen und ein einfacher jBPM-Prozess wird gestartet (Abb. 1). Der Prozess enthält eine menschliche Interaktion, es soll nämlich entschieden werden, ob der Event relevant ist oder nicht. Wenn ja, wird er über den ESB per Web Service an das Case-Management-System übergeben. Die Rückmeldung über das Schließen des Falls erfolgt als JMS-Message, die über den ESB an jBPM weitergegeben wird.

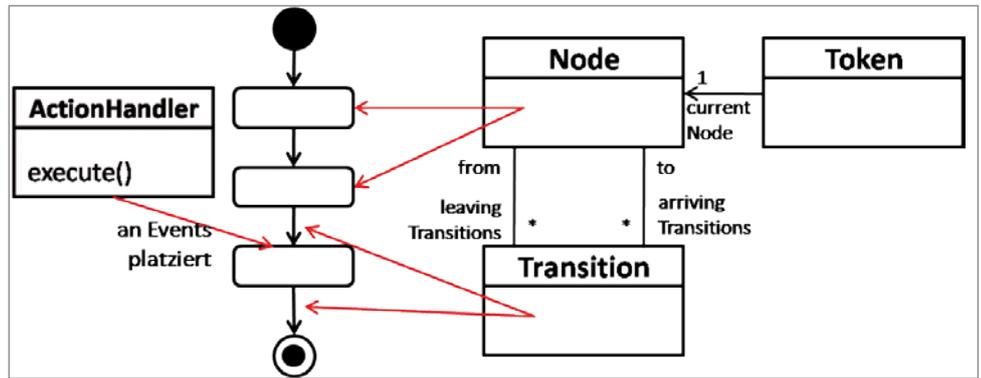


Abb. 2: Interner Aufbau von jBPM

In diesem Fall wird die Process Engine einerseits für das Human Task Management verwendet, andererseits könnte jetzt der Prozess einfach verändert oder Kennzahlen wie die durchschnittliche Bearbeitungsdauer eines Falls oder der prozentuale Anteil der ignorierten Events abgefragt werden.

JBoss jBPM

JBoss jBPM ist eine leichtgewichtige Prozessmaschine, die aktuell in Version 3.2.2 vorliegt. Sie ist rein POJO-basiert implementiert, Persistenz wird durch Hibernate umgesetzt. Dies ermöglicht den Einsatz ohne Persistenz oder aber mit allen von Hibernate unterstützten Datenbanken, mit oder ohne Application-Server. Spricht man von jBPM, meint man heute meist die proprietäre Prozesssprache jPDL, so auch in diesem Artikel. jBPM stellt auch eine BPEL-sprechende Engine bereit, die in der Praxis aber eine eher untergeordnete Rolle spielt.

Die Hauptaufgabe der Engine ist die Abbildung eines Zustandsautomaten mit Wartezuständen, in denen die Engine den Zustand persistiert. Dabei gibt es Nodes und Transitionen, wobei letztere zwei Nodes miteinander verbinden (Abb. 2). Dieses Grundprinzip ermöglicht es, Prozesse als freie Grafen zu beschreiben, ist dafür aber auch bereits ausreichend. Dabei gibt es unterschiedliche Arten von Nodes, die das Prozessverhalten beeinflussen können. Die Prozessausführung wird durch ein Token-Objekt gesteuert, das durch den Prozessgraphen „wandert“, es entspricht also einer Prozessinstanz. Während des Prozessdurchlaufs kann die Engine Aktionen anstoßen, beispielsweise das

Versenden einer E-Mail oder der Aufruf eines Web Service. Dies ist durch so genannte „ActionHandler“ umgesetzt, simple Java-Klassen, die ein Interface implementieren müssen, damit sie von der Engine angestoßen werden können. Diese ActionHandler können an definierten Stellen im Prozess angehängt werden, wie etwa beim Eintreffen in einen Zustand oder beim Durchlaufen einer Transition.

Integration ESB in jBPM

Das Grundprinzip der Integration eines ESB mit jBPM ist sehr einfach: Es gibt vorgefertigte ActionHandler, mit denen aus dem Prozess heraus Aktionen im ESB angestoßen werden. Sollten Prozesse auf Input vom ESB warten, so können in die Prozesse Wartezustände („Wait States“) eingebaut werden und der ESB muss sich bei jBPM melden, wenn die Prozessausführung weiterlaufen soll. Technisch bietet jBPM hier einerseits ein direktes API oder aber auch die Möglichkeit, JMS-Messages zu schicken. In beide Richtungen können natürlich auch Daten ausgetauscht werden. Diese einfache Schnittstelle ermöglicht bereits die komplette Integration. Prinzipiell funktioniert die Integration übrigens auch bei anderen Prozessmaschinen nach demselben Muster, auch wenn die konkrete

Listing 1: BPM-Konfiguration in Mule (Ausschnitt)

```

<!-- JBPM Process Engine -->
<spring:bean id="jbpm" class="org.mule.transport.bpm.jbpm.Jbpm"
destroy-method="destroy">
<spring:property name="jbpmConfiguration" ref="jbpmConfiguration"/>
</spring:bean>
<bpm:connector name="jbpmConnector" bpms-ref="jbpm"/>

```

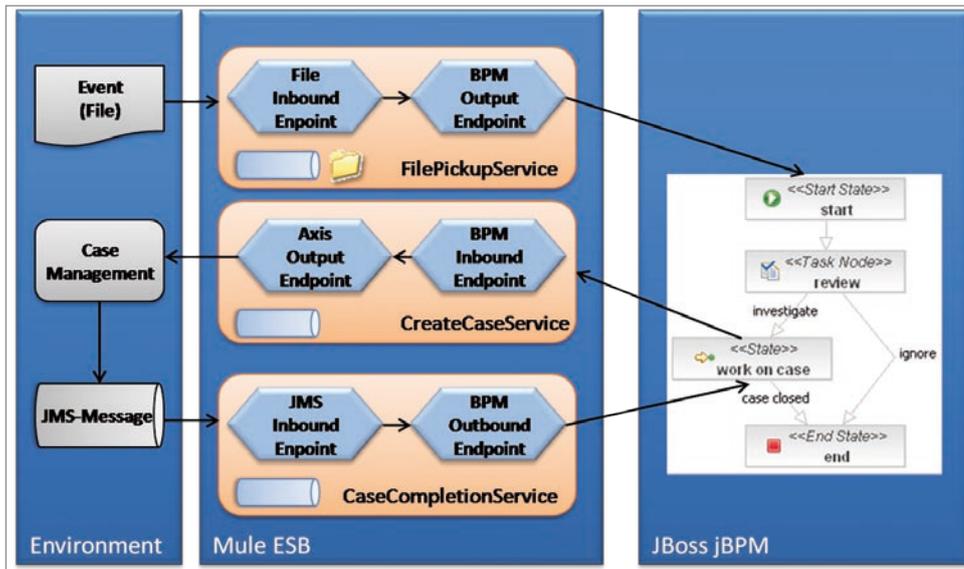


Abb. 3: Das Beispiel mit dem Mule ESB

Implementierung immer ein bisschen anders aussieht.

Mule

Mule ist ein leichtgewichtiges Integrations-Framework, das seit April 2008

Listing 2: Prozess starten

```
<bpm:endpoint name="TestProcess" process="testProcess"/>

<service name="FilePickupService">
  <inbound>
    <file:inbound-endpoint path="/test-data/in"
      transformer-refs="FileToString XmlToObject"/>
    </file:inbound-endpoint>
  </inbound>
  <outbound>
    <outbound-endpoint ref="TestProcess"/>
  </outbound>
</service>
```

Listing 3: Event Trigger, der einen Prozess weiterleitet

```
<service name="CaseCompletionService">
  <inbound>
    <jms:inbound-endpoint topic="services"
      connector-ref="activeMqJmsConnector"
      transformer-refs="JMSMessageToObject"/>
  </inbound>
  <outbound>
    <outbound-pass-through-router>
      <!--Referenz auf den globalen BPM Endpoint in Listing 2-->
      <outbound-endpoint ref="TestProcess"/>
    </outbound-pass-through-router>
  </outbound>
</service>
```

in der Version 2 zur Verfügung steht. Eine Einführung in die Grundarchitektur und die Konzepte von Mule können unter [1] und die Neuerungen und Abgrenzungen zur Vorgängerversion unter [2] nachgelesen werden. Zusatzerweiterungen wie Transformer oder andere hilfreiche Features fassen sich bei Mule in Modulen zusammen. Auf Muleforge [3] haben sich in den letzten Jahren eine Menge an Zusatztransporten und Modulen für Mule entwickelt, so auch der BPM Transport. Aus der Sicht von Mule wird eine Prozessmaschine als externes System gesehen und über diesen Transport angesprochen. Wie jeder Transport in Mule, hat auch der BPM Transport den gleichen Aufbau und erleichtert für geübte Mule-User den Umgang mit BPM-Systemen. Er ist für die Integration von unterschiedlichen BPM-Systemen ausgelegt, die ein API zur Verfügung stellen.

Betrachten wir zunächst den grundsätzlichen Aufbau des BPM Transport. Das BPMS-Interface stellt das zu integrierende BPM-System dar. Der Message-Service im BPMS-Interface wird für die Erstellung von Mule-Nachrichten verwendet und ermöglicht dem BPM-System Nachrichten an den Mule ESB zu senden. Im Falle der jBPM gibt es eine entsprechende Implementierung des BPMS-Interface. Das Starten, Beenden und Weiterleiten wird entsprechend über das von der jBPM zur Verfügung

gestellte API implementiert. Listing 1 veranschaulicht die Konfiguration eines BPM-Konnektors. Der Konnektor erhält als Referenz das zu integrierende BPM-System, in unserem Fall jBPM.

Ein Prozess kann auf unterschiedlichste Art und Weise gestartet werden. Dies kann in Mule sehr elegant mit der Angabe unterschiedlicher *Inbound Endpoints* (Einstiegspunkte für ein Service) gelöst werden. Ein Service kann in Mule mehrere Einstiegswege haben, und das über unterschiedliche Transporte hinaus. Listing 2 veranschaulicht, wie der in Abbildung 2 dargestellte Prozess über das *FilePickupService* gestartet werden kann. Das File wird eingelesen und es werden dabei auch zwei Transformationen vorgenommen.

Der BPM Endpoint ist mit dem fix definierten Prozess *testProcess* verbunden. Es besteht jedoch auch die Möglichkeit, einen globalen Endpoint für die BPM Engine zu definieren, dabei muss der Prozess dann als Message Property mitgeliefert werden. Wird dieser Endpoint als *OutboundEndpoint* angegeben, so können die Grundaktionen, wie oben angeführt, auf den Prozess angewendet werden. In Listing 2 wird eine neue Instanz des Prozesses angelegt. Soll eine Prozessinstanz weitergeleitet, terminiert oder aktualisiert werden, so benötigt Mule die ID der Prozessinstanz. Diese wird üblicherweise als Message Property in der Nachricht mitgeliefert. Die Aktion ist nur im Falle der Terminierung und des Updates anzugeben, da Mule, falls eine Instanz bereits existiert, diese weiterleitet bzw. neu anlegt.

Für die Nachrichtenübermittlung an die BPM Engine ist der *ProcessMessageDispatcher* zuständig. Der Dispatcher empfängt das Message-Event und wertet die auszuführende Aktion aus. Der Dispatcher erstellt eine Map von Prozessvariablen, die im Anschluss in der Engine als Variablen zu der Prozessinstanz persistiert werden. Die Map beinhaltet alle Message Properties, die im Zuge des Requests an die Message angehängt wurden. Abhängig davon, über welchen Endpoint und Transport ein Prozess gestartet wird, werden unterschiedliche Parameter mitgeliefert. Wird ein Prozess beispielsweise über

E-Mail gestartet, so werden alle vom POP3 Transport spezifischen Parameter als Variablen abgelegt, der File Transport liefert wieder andere Parameter. Es hängt also vom zugrunde liegenden Transport ab, welche Parameter an die Nachricht angehängt werden. Der BPM Transport definiert ebenfalls ein Set von Variablen. Zu den wichtigsten zählen *incoming* (Message Payload) oder *incomingSource*. Die *incomingSource*-Variable kann sehr hilfreich sein, wenn ein Prozess unterschiedliche Startevents hat und spätere Entscheidungen vom Startevent abhängen. Die Variablen werden immer wieder neu gesetzt, wenn die Prozessmaschine ein Event von Mule erhält. Im Prozessverlauf kommt es häufig vor, dass auf ein externes Ereignis gewartet wird, bis der Prozess weitergeleitet werden kann. Listing 3 demonstriert, wie ein Prozess aufgrund einer eintretenden JMS-Nachricht weitergeleitet werden kann, die dann auch die ID des Prozesses enthalten muss.

Um Events aus einem Prozess an den ESB zu senden, kommen die jBPM-ActionHandler zum Einsatz, Tabelle 1 listet die von Mule zur Verfügung gestellten Handler auf. Am Wichtigsten für dieses Beispiel ist das *SendMuleEvent* bzw. *SendMuleEventAndContinue*, da mit diesen ActionHandlern definierte Endpoints in Mule angesprochen werden. Listing 4 zeigt den entsprechenden Ausschnitt aus der Prozessdefinition. Dem ESB-Service können beliebige Prozessvariablen über den Payload mitgeliefert werden. Der Endpoint definiert den konkreten Service, der angesprochen werden soll. Schließlich setzen wir den Aufrufasynchron ab. Im Falle eines synchronen Aufrufs würde der Prozess auf eine Antwort warten und das Ergebnis in die Variable *incoming* abgelegt werden.

Zuletzt betrachten wir noch, wie Mule eingehende Events von der jBPM entgegennimmt und verarbeitet. In dem Beispiel haben wir einen generellen Empfangsservice für Events aus der BPM durch den *InboundEndpoint* definiert. Aufgrund des gegebenen Endpoints, leitet Mule die Anfrage an den konkreten Service-Endpoint weiter, in unserem Fall an den *CreateCaseService*,

der wiederum einen Web Service Call über den Axis Transport absetzt.

JBoss ESB

Das gleiche Beispiel haben wir auch mit dem JBoss ESB umgesetzt, der Überblick ist in Abbildung 4 gezeigt. Der JBoss ESB arbeitet etwas anders als Mule: Es existieren ebenfalls Queues, die vom ESB abgehört werden. Liegt eine Nachricht vor, wird diese Nachricht an einen Service weitergegeben, der die Action Pipeline reicht. Actions setzen dann die eigentliche Funktionalität um, wobei viele vorgefertigte Actions bereits existieren. Eigene Anforderungen können mit eigenen Actions einfach in Java implementiert werden. Eine detaillierte Einführung in den JBoss ESB ist in [4] zu finden.

Zur Anbindung von jBPM existiert dem entsprechend eine vorgefertigte Action, die im ESB direkt verwendet werden kann. Diese muss lediglich konfiguriert werden, also welcher Prozess z.B. zu starten ist. Listing 6 zeigt die Action zum Starten eines Prozesses und Listing 7 die zum späteren Weitertriggern. Dabei können auch Daten aus der Message des ESB an den Prozess übergeben werden. Diese Actions müssen dann nur noch entsprechend in der Action Pipeline der Services des ESB platziert werden.

Soll eine laufende Prozessinstanz angesprochen werden, so muss der ESB die Instanz identifizieren können, dies passiert in jBPM durch die Token ID. Nach dem Starten des Prozesses befindet sich daher durch Zauberhand die Variable *jbpmTokenId* nebst anderer Informationen aus jBPM in den Properties der ESB-Message. Um den Prozess später wieder zu identifizieren, wird im Beispiel diese ID an das Case-Management-System übergeben. Es wären hier jedoch auch zahlreiche andere Korrelationsmechanismen vorstellbar.

Das Anstoßen eines ESB-Service aus dem Prozess heraus funktioniert wie in Mule über einen speziellen ActionHandler (Listing 8). Auch hier können natürlich Prozessvariablen an den ESB übergeben werden. Dieser grobe Überblick soll an dieser Stelle reichen,

Listing 4: Aufruf von Services aus dem Prozess

```
<state name="Work on case">
  <event type="node-enter">
    <action class="org.mule.transport.bpm.jbpm.actions.
                                     SendMuleEvent">
      <payloadSource>serviceRequest</payloadSource>
      <endpoint>CreateCaseService</endpoint>
      <synchronous>false</synchronous>
    </action>
  </event>
</state>
```

Listing 5: Events des BPM in Mule verarbeiten

```
<service name="fromBPM">
  <inbound>
    <!--Referenz auf den globalen BPM Endpoint in Listing 2 -->
    <inbound-endpoint ref="TestProcess"/>
  </inbound>
  <outbound>
    <endpoint-selector-router selectorExpression="header:endpoint">
      <outbound-endpoint ref="CreateCaseService"/>
    </endpoint-selector-router>
  </outbound>
</service>
```

```
<service name="chargeServiceCall">
  <inbound>
    <inbound-endpoint ref="CreateCaseService"/>
  </inbound>
  <outbound>
    <outbound-pass-through-router>
      <axis:outbound-endpoint .../>
    </outbound-pass-through-router>
  </outbound>
</service>
```

Listing 6: JBoss ESB Action zum Starten eines Prozesses

```
<action name="startProcessInstance" class="org.jboss.soa.esb.
                                     services.jbpm.actions.BpmProcessor">
  <property name="command" value="StartProcessInstanceCommand" />
  <property name="process-definition-name" value="EsbShowcase"/>
  <property name="object-paths">
    <object-path bpm="caseContent" esb="content" />
  </property>
</action>
```

Listing 7: JBoss ESB Action zum Triggern eines Prozesses

```
<action class="org.jboss.soa.esb.services.jbpm.actions.BpmProcessor">
  <property name="command" value="SignalCommand" />
</action>
```

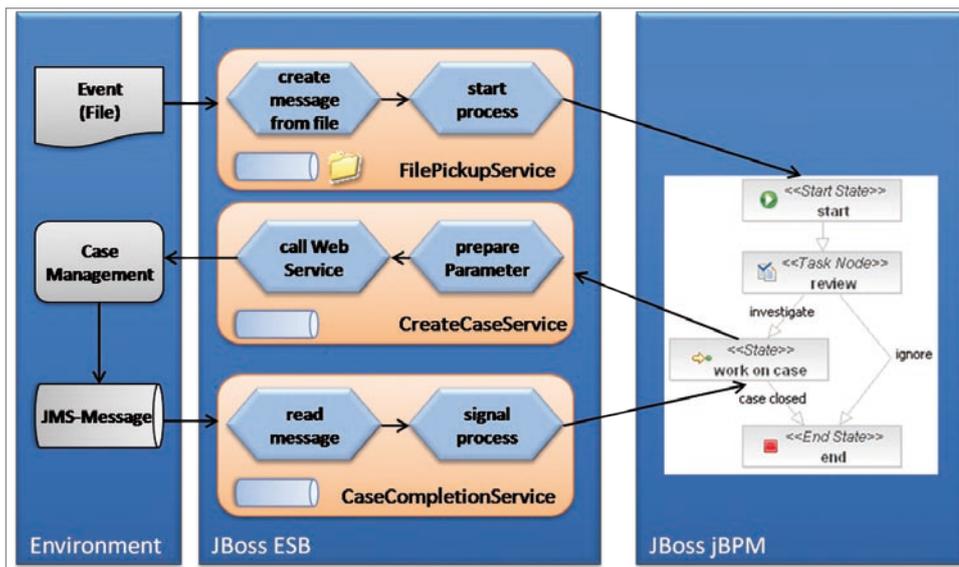


Abb. 4: Das Beispiel mit dem JBoss ESB

um die Integration mit dem JBoss ESB grundsätzlich vorzustellen. Der interessierte Leser findet den Quellcode zur Beispielanwendung (inklusive der vollständigen ESB-Konfiguration) sowie eine ausführlichere Beschreibung unter [5].

Fazit

Sowohl Business Process Engine als auch Enterprise Service Bus sind heute bereits akzeptierte Technologien. Beide haben dabei ihre Berechtigung, wobei gewisse Funktionalitäten wie intelligentes Routing durchaus überlappen. Zwar gibt es genauso Projekte, die keinen ESB benötigen, wie es Projekte gibt, bei denen eine Prozessmaschine keinen Sinn macht. Oft ist es jedoch

auch eine gute Idee, beide Technologien zu kombinieren. Der Prozess kann so von Transformations- oder Integrationsproblematiken befreit, durch unterschiedlichste Technologien mit Daten versorgt und nebenbei noch von der Implementierung der Services entkoppelt werden. Auf der anderen Seite muss durch Nutzung einer Prozessmaschine Geschäftsprozesslogik nicht länger in ESB-Konfigurationen oder Content-based-Routing-Regeln versteckt werden. Die nicht triviale Frage ist dann allerdings in der Praxis, welche Probleme in welchem Tool zu lösen sind.

Die Integration der Process Engine JBoss jBPM mit einem ESB geht technisch sehr leicht von der Hand. Es wurde gezeigt, wie dies in Mule und dem JBoss ESB bereits vorbereitet ist. Möchte man andere ESB-Produkte anbinden, können diese Implementierungen auch als Vorlage dienen. Schön an dieser Lösung ist, dass die bekannte Umgebung, nämlich Java, eventuell nicht verlassen werden muss, da die in Java implementierten ESBs intern genauso selbstverständlich mit Java hantieren können wie es auch jBPM kann. So können auch ohne Web Services tragfähige Lösungen entstehen.

Ein kleiner Wehrmutstropfen bleibt natürlich auch, dies ist die steigende Komplexität der Konfiguration. In den vorliegenden Beispielen müssen bereits

einige Queues angelegt, ESB-Services konfiguriert und Prozesse modelliert werden. Dies führt zu zahlreichen XML-Dateien, die auch noch zusammenpassen müssen – bei steigender Projektgröße ein nicht zu unterschätzendes Problem. Auch kann man zu einem Punkt gelangen, an dem nicht mehr wirklich klar ist, welche Geschäftslogik wo abgebildet ist. Hier gilt es bei größeren Projekten aufzupassen.

Insgesamt sind wir aber mit dem heute bereits Open Source verfügbaren Stand der Technik schon sehr zufrieden, und wir stehen eher am Anfang. Es bleibt also spannend, bleiben Sie am Ball!



Bernd Rücker ist Geschäftsführer der camunda services GmbH. Er verfügt über mehrjährige Projekterfahrung als Softwarearchitekt, Coach, Berater, Trainer und Entwickler im Umfeld von Unternehmensanwendungen, Java EE, BPM und SOA. Er ist Autor eines EJB3-Buchs, zahlreicher Fachartikel, Sprecher auf Konferenzen sowie Committer im JBoss-jBPM-Projekt.



Markus Demolsky ist Software Engineer und Berater bei der Soreco Group. Seine Schwerpunkte liegen bei Softwarearchitektur, Workflow-Management-Systemen und Open-Source-Technologien im Java-EE-Umfeld. Er ist Committer bei MuleForge und arbeitet aktuell mit Dr. Alexander Schatten an einem Buch zum Thema „Software Engineering – Best Practices“.

Listing 8: jBPM ActionHandler zum Ansprechen des JBoss ESB

```
<state name="work on case">
  <event type="node-enter">
    <action class="org.jboss.soa.esb.services.jbpm.actionhandlers.
      EsbActionHandler">
      <esbCategoryName>EsbShowcase</esbCategoryName>
      <esbServiceName>CreateCaseService</esbServiceName>
      <jbpmToEsbVars>
        <mapping jbpm="caseContent" esb="someVariableWithContent" />
      </jbpmToEsbVars>
    </action>
  </event>
  <transition to="end" name="case closed" />
</state>
```

Links & Literatur

- [1] Markus Demolsky: Open Source ESB Mule, in: Java Magazin 05.2008, S. 80–86
- [2] Markus Demolsky: Mule 2: it-republik.de/jaxenter/artikel/Mule-2---The-Next-Generation-1772.html
- [3] muleforge.org
- [4] Bernd Rücker, Sebastian Lasek: Der JBoss ESB im Einsatz, in: Java Magazin 08.2008, S. 40–45
- [5] JBoss-ESB-Beispielanwendung: www.camunda.com/knowledge/jbpm_meets_esb.html