Prozessorientierte Oberflächen für JBoss jBPM mit JSF oder Java Swing

Neue Kleider für JBoss jBPM

VON BERND RÜCKER

Das Thema Business Process Management (BPM) ist momentan in aller Munde. Daher bleibt es nicht aus, als Entwickler in die Verlegenheit zu kommen, eine prozessorientierte Anwendung schreiben zu müssen. Im Open-Source-Bereich macht zurzeit vor allem JBoss Wirbel, da dort nach und nach ein kompletter SOA-Stack aufgebaut wird. Der vorliegende Artikel zeigt daher auf, wie eine prozessorientierte Oberfläche basierend auf JBoss jBPM entwickelt werden kann, und welche Besonderheiten BPM-Oberflächen ausmachen. Dabei soll auch die oft vernachlässigte Entwicklung einer Rich-Client-Oberfläche mit Java Swing betrachtet werden.

JBoss jBPM [1] ist eine unter LGPL zur Verfügung stehende, sehr flexible Business Process Engine, die sowohl in Java EE, als auch Java SE Umgebungen zum Einsatz kommen kann. Dank Einsatz von POJO's mit Hibernate kann sie mit jeder Datenbank (oder auch ohne Persistenz) arbeiten. Prozesse werden als XML wahlweise im proprietären jPDL-Format oder auch im standardisierten BPEL beschrieben (im Folgenden beschränke ich mich auf die meist benutzte jPDL-Version). Ein grafischer Editor für jPDL steht als Eclipse-Plug-in zur Verfügung. In Prozessen kann an definierten Stellen Java-Code ausgelöst werden, um Verbindung mit der Außenwelt herzustellen. Einführungen zu JBoss jBPM finden sich beispielsweise in [2], [3] oder [4]. Ein grundlegendes Verständnis sowie Kenntnis der Begrifflichkeiten wird im Folgenden vorausgesetzt.

Oberflächen für (j)BPM-Anwendungen

Entwirft man ein System mit einer Business Process Engine, stellt sich eine wichtige Frage: Verändert sich meine Oberfläche? Und wenn ja, wie?

Die Antwort ist natürlich kein großes Geheimnis. BPM hat nicht nur wesentlichen Einfluss auf die Architektur der Anwendung, auch das GUI verändert sich oder sollte es zumindest tun. So ist es ja sogar die Vision der prozessorientierten Anwendung, dass der Prozess in der Anwendung verankert ist und der Benutzer proaktiv auf anstehende Aufgaben hingewiesen wird. Denn bisher mussten die Mitarbeiter irgendwie wissen (evtl. durch Prozessdokumentationen), zu welchen Zeiten etwas zu erledigen ist. Es ergeben sich drei verschiedene Arten von Features in prozessorientierten Oberflächen:

- *Prozessunabhängige Use-Cases*: Es wird immer Features einer Anwendung geben, die nicht Teil eines Prozesses sind.
- *Durch den Prozess ausgelöste Aufgaben*: Der Prozessablauf erfordert manuelle Interaktion, ein Beispiel könnte die manuelle Freigabe eines Auftrags sein.
- Externe Ereignisse: Oft wartet ein Prozess auf das Eintreffen eines externen Ereignis, wie beispielsweise das Eintreffen einer Zahlung. Nach Auftreten des Ereignisses muss der Mitarbeiter den betreffenden Prozess gezielt weiter bearbeiten.

Die prozessunabhängigen Features sind hier nicht weiter spannend, es stellt sich jedoch die Frage, man mit den beiden anderen Anforderungen umgeht. Typischerweise führen Überlegungen zu folgenden Komponenten:

- *Aufgabenliste (Tasklist, Postkorb, Inbox)*: Die Liste enthält alle für einen Benutzer aktuell durch die Engine ausgelöste Aufgaben. Der Mitarbeiter muss lediglich diese Liste abarbeiten, ähnlich beispielsweise dem E-Mail-Eingang.
- *Suche*: Für externe Ereignisse wird meist eine Suche benötigt, damit die betroffenen Prozesse anhand vorhandener Informationen gefunden werden können. Erfasst man beispielsweise anhand eines Kontoauszugs eingegangene Zahlungen, so muss mit der Auftragsnummer im Betreff der richtige Prozess gefunden werden können.
- *Batch-Bearbeitung*: Oft sollen einige externe Ereignisse aus Ergonomiegründen am Stück abgearbeitet werden (z.B. das Eintreffen von Kundenretouren, bei denen es reicht, beim Empfang der

Abb. 1: Ein einfacher Auftragsprozess als Beispiel

Pakete die darauf angegeben Referenznummern zu erfassen).

- *Bearbeitungsmasken*: Für jeden Zustand, in dem ein Prozess sich befinden kann, müssen Masken konfiguriert werden, mit denen die Bearbeitung des Prozesses in diesem Zustand ermöglicht wird. Diese Masken können dann entweder aus der Aufgabenliste oder der Suche geöffnet werden.
- *Admin*: Typischerweise ist auch eine Administrations-Sicht auf die Prozesse hilfreich und erforderlich.

Listing 1

Konfiguration eines TaskController im jPDL

```
<task-node name="Auftrag versenden">
 <task name="Auftrag versenden">
  <controller class="...NavigationTaskController">
   <viewName>shipOrder</viewName>
  </controller>
 </task>
</task-node>
```

Auch wenn dies einfach klingt, birgt es in der Praxis erfahrungsgemäß zwei Probleme. Erstens wird am Anfang meist weder von den Fachanwendern noch von den Entwicklern prozessorientiert gedacht. Und zweitens müssen die genannten Komponenten dann auch entwickelt werden.

Betrachtet man nun JBoss jBPM im Speziellen verhält es sich momentan so, dass die Unterstützung bei der Umsetzung der Oberflächen durchwachsen ist. Von Seiten JBoss wird die mitgelieferte Webkonsole (eine JSF-Anwendung) stetig weiterentwickelt und auch JBoss Seam [5] verheiratet sehr erfolgreich jBPM mit JSF-Oberflächen. Keinerlei Unterstützung gibt es dagegen für Rich-Clients, weswegen diesem Thema hier deutlich mehr Platz eingeräumt wird. Der üblichen Frage, warum heute denn nicht alles als Webanwendung entwickelt wird, möchte ich hier nur mit der knappen Antwort begegnen, dass es genug neue und erfolgreiche Projekte gibt, die auch (oder gerade) heute Java-Swing-Anwendungen entwickeln.

Beispielanwendung

Zur Visualisierung der vorgestellten Ideen gibt es eine kleine Beispielanwendung, die den in Abbildung 1 gezeigten Auftragsprozess umsetzt. Dabei gibt es einen Zustand, in dem die Anwendung auf das externe Ereignis der Zahlung wartet, in jBPM als „State" umgesetzt. Den Auftrag zu versenden, ist dagegen eine durch die Engine auszulösende Aufgabe. Dies wird in jBPM als „Task Node" modelliert. Neben dem Prozess wird die Businesslogik mit einer EJB-3-Entity für einen Auftrag sowie einer zugehörigen Session Bean simuliert.

Wir benötigen für die Oberfläche 3 Masken: Eine für den Start eines Auftrages, eine zum Prüfen der Zahlung sowie eine für den Versand. Daneben werden eine Suche nach einem Auftrag bei eingegangener Zahlung sowie die Aufgabenliste benötigt. Die komplette Beispielanwendung ist online verfügbar [6] und kann aus Platzgründen nicht in allen Details betrachtet werden.

Problem Masken-Mapping

In jeder Anwendung ist eines der ersten Probleme, wo und wie konfiguriert wird, in welchem Prozesszustand welche Bildschirmmaske angezeigt bzw. zur Bearbeitung verwendet wird. Dazu macht es meist Sinn, zuerst einen logischen Namen der Maske einzuführen, welcher von der späteren Implementierung unabhängig ist. Dieser Name kann dabei im einfachsten Fall dem Namen der jBPM-Node selbst entsprechen, aber natürlich auch konfiguriert werden. So könnte auch eine Namenskonvention für Prozesszustände, wie beispielsweise „Warten auf Zahlung/checkPayment", verwendet werden. Allerdings ist dies etwas kritisch, da dann Informationen über die Oberfläche die Geschäftsprozessdiagramme „verunreinigen".

Eine weitere Möglichkeit wäre zum Beispiel, einen eigenen TaskController zu schreiben. Dieser kann zur Laufzeit beim Anlegen eines Tasks den logischen Namen der Oberfläche als Task-Variable in jBPM anlegen. Dadurch erfolgt die Konfiguration der Oberfläche zwar immer noch in der Prozessdefinition, was eben den Vorteil der einfacheren Pflege mit sich bringt, versteckt sich aber gut in den Tiefen des XML der Prozessdefinition (Listing 1) und ist auch nicht im Diagramm zu sehen. Ein weiterer Vorteil an der Lösung ist, dass der TaskController anhand von verfügbaren Prozessvariablen weitere Unterscheidungen treffen könnte.

Der logische Name muss nun noch in die wirkliche Implementierung übersetzt werden. Dies funktioniert bei verschiedenen Oberflächen-Techniken unterschiedlich:

Listing 2

Erweitern eines jBPM-Commands

```
public class MyGetCompleteProcessInstanceCommand
                    extends GetProcessInstanceCommand {

 public MyGetCompleteProcessInstanceCommand
                              (long processInstanceId) {
  super(processInstanceId);
 }
 /**
  * execute method is called by CommandService
  * The JbpmContext is initialized by the environment,
  * for example in the EJB 3.0 Session Bean by DI
  */
 @Override
 public Object execute(JbpmContext jbpmContext)
                                        throws Exception {
  ProcessInstance result = (ProcessInstance) super.
                                    execute(jbpmContext);
  // now retrieve standard graph also for super process
  if (result!=null && result.getSuperProcessToken()!=null)
                                                      {
   super.retrieveProcessInstance(
    result.getSuperProcessToken().getProcessInstance());
  }
  return result;
 }
}
```

- JSF: Typischerweise wird der logische Name dem JSF-Outcome entsprechen und daher von der *faces-config* in den korrekten View übersetzt.
- Rich-Client: Es wird ein eigenes Mapping benötigt.

Möglich ist natürlich auch, das Mapping zu sparen, und statt dem logischen Namen direkt die Implementierung (Klassenname oder endgültige URL) im Prozess zu konfigurieren. Dies führt zwar zu weniger „seperation of concerns", hat aber auf der anderen Seite den Vorteil, dass die GUI-Konfiguration direkt mit dem Prozess versioniert wird. Dies ist zwar auch in anderen Lösungen möglich, aber aufwändiger.

Implementierung mit JSF

Die Implementierung einer JSF-Anwendung als Oberfläche für jBPM soll hier nicht eingehender betrachtet werden. Ein guter Startpunkt bei einem solchen Vorhaben ist die bei jBPM mitgelieferte Webconsole, die inzwischen komplett auf JSF basiert und zumindest Inspiration für die eigene Anwendung liefern kann. Ebenfalls sehr interessant ist das (zum Zeitpunkt des Artikels sehr frische) Projekt jbpm4jsf [7], welches eine Facelet-Bibliothek für jBPM-Oberflächen implementiert. jbpm4jsf soll zukünftig auch in der Webconsole zum Einsatz kommen. Des Weiteren kann JBoss Seam [5] als Ideengeber sehr hilfreich sein. Dieses macht allerdings umfangreiche Eingriffe in den JSF-Lebenszyklus, was eine Einarbeitung in die interne Funktionsweise nicht gerade vereinfacht. Auf der anderen Seite kann man dadurch auch die eine oder andere gute Idee für JSF-Anwendungen, auch über jBPM hinaus, mitnehmen.

Grundsätzlich benötigt die Webanwendung neben Standardkomponenten wie der Taskliste oder einem Administrationsbereich ein durchgängiges Konzept für das Öffnen von Tasks. Dabei sind zwei unterschiedliche Ansätze denkbar:

- Beim Aufbau eines Links (z.B. in der Taskliste) wird der für den Task konfigurierte Outcome ermittelt und in den *commandLink* auf der JSF-Seite eingebaut.
- Es wird ein globaler Outcome verwendet, um Tasks (oder States) anzuzeigen. Mit einem geeigneten PhaseListener wird dieser Outcome „abgefangen" und nach Untersuchung des anzuzeigenden Tasks auf die gewünschte Seite umgeleitet.

Beide Ansätze haben Vor- und Nachteile, sodass man in der Praxis durchaus eine Mischung erleben wird. Andere Webframeworks werden prinzipiell sehr ähnlich angebunden, auch wenn dann mehr Handarbeit ansteht, da sich JSF bei jBPM durchgesetzt hat.

Rich-Clients und jBPM

Rich-Clients stehen neben dem Masken-Mapping noch vor weiteren Problemen, denn normalerweise finden sich diese Oberflächen nur in mehrschichtigen Architekturen mit „echter" Remote-Kommunikation. Daher muss man sich mit dieser Kommunikation auch auseinandersetzen. Typischerweise helfen in diesem Fall, zumindest in Java-EE-Umgebungen, Stateless Session Beans als Fassaden aus, sodass sich der Java-EE-Container um Remoting, Transaktionssteuerung sowie benötigte Ressourcen (z.B. per Dependency Injection) kümmern kann. Ab jBPM-Version 3.2 wird dies durch einen eingeführten CommandService gut unterstützt. Wichtige Funktionalitäten der Prozess-Engine sind als Command-Objekte verfügbar und können über Implementierungen des CommandService ausgeführt werden. In jBPM ist auf Grund der Java-1.4-Kompatibilität lediglich die Implementierung als EJB 2.1 Session Bean enthalten. Eine EJB 3.0 Session Bean ist allerdings sehr einfach, für den Quellcode sei auf das später im Artikel eingeführte tk4jbpm [8] verwiesen.

Durch die Remote-Kommunikation entsteht aber noch ein weiteres großes Problem: Es müssen Daten wie der Prozesszustand oder Prozessvariablen über die Leitung transportiert werden. Dies ist oft die Stunde der nicht sehr beliebten Data Transfer Objects (DTO). Da jBPM intern mit reinen POJOs arbeitet, kommt man aber auch wunderbar ohne DTOs aus. Das einzige noch verbleibende Problem ist, dass Rich-Clients nicht wie Weboberflächen bei Bedarf Daten nachladen können. Benötigt man beispielsweise den Namen des aktuellen Zustands des Prozesses, so muss sichergestellt sein, dass dieser auch bereits an den Client übertragen wurde. In Webanwendungen kann man dagegen das von Hibernate angebotene Lazy Loading verwenden, sofern das Rendering in einer aktiven Transaktion stattfindet (was allerdings nicht schwierig zu realisieren ist).

Die Lösung sieht bei JBPM so aus, dass der auf dem Client benötigte Objektgraph auf dem Server bereits komplett geladen wird, was durch Traversieren realisiert werden kann. Weitere benötigte Teile des Graphs können nachgeladen werden.

Masken-Mapping im Überblick

Maskennamen können sein:
- Name des Tasks oder der Node
- Konfigurierter Name
  - über TaskController
  - per Namenskonvention

Dabei wird normalerweise ein logischer Maskenname verwendet (beispielsweise der Outcome in JSF), wobei auch der Name der echten View verwendet werden kann.
Der logische Name wird durch eine eigene Konfigurationsdatei in die echte View übersetzt, dies übernimmt bei JSF meist die *faces-config.xml*, bei Rich-Client muss ein eigenes Mapping definiert werden.

Lösung bei Alfresco

Das Enterprise Content Management System Alfresco [9] setzt intern stark auf JBoss jBPM. Dort ist das Problem des Masken-Mappings durch eine eigene Konfiguration gelöst, die auf den Namen der Node Bezug nimmt:

```
<config evaluator="node-type" condition="
                wf:submitAdhocTask" replace="true">
 <property-sheet>
  ...
 </property-sheet>
</config>
```

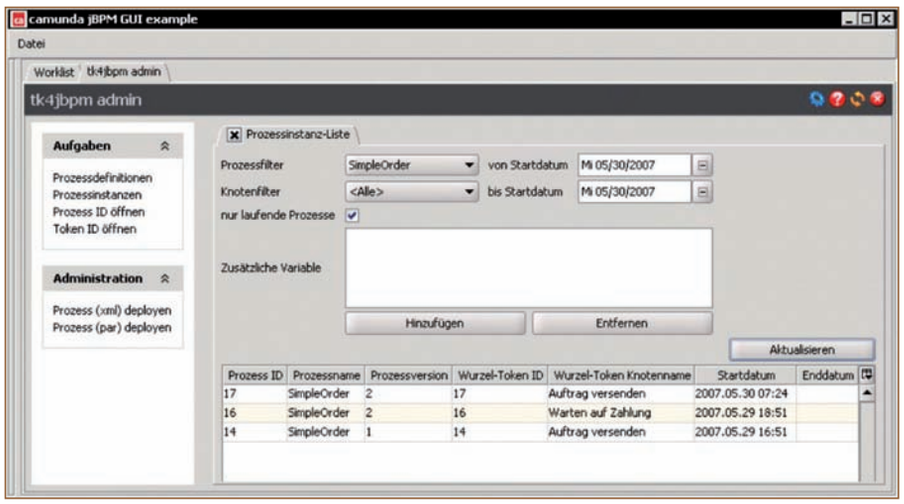

Abb. 2: Prozessliste im Admin-Client des tk4jbpm

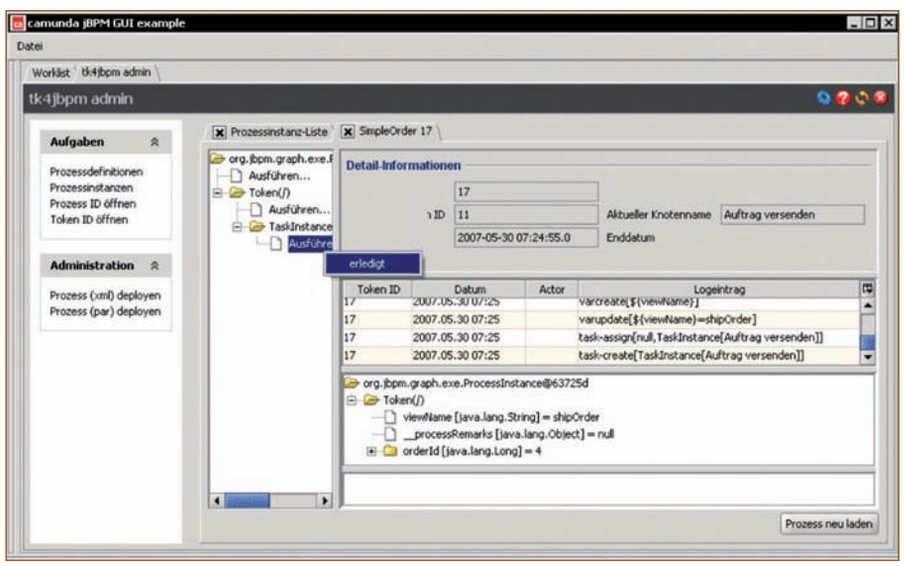

Abb. 3: Prozessdetailansicht im Admin-Client des tk4jbpm

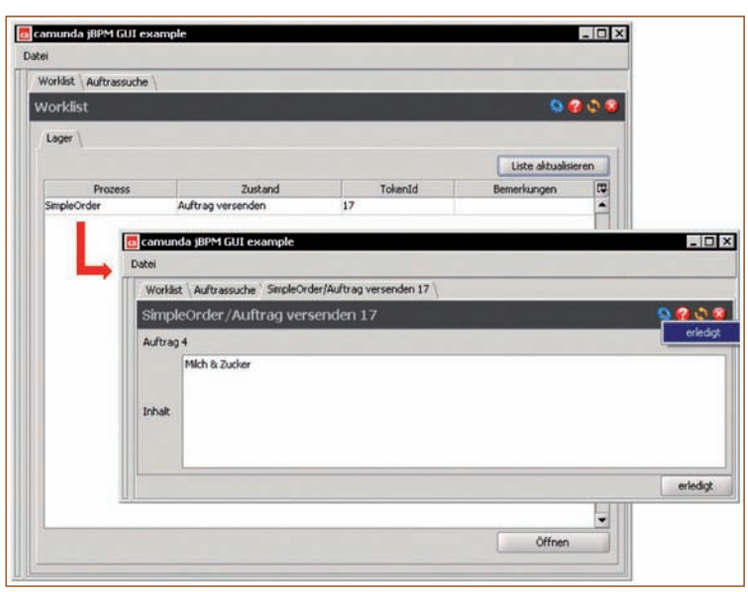

Abb. 4: Taskliste und Taskpanel

Die Lösung sieht bei jBPM so aus, dass der auf dem Client benötigte Objektgraph auf dem Server bereits komplett geladen wird, was durch einfaches Traversieren realisiert werden kann. Nun benötigen unterschiedliche Clients aber oft unterschiedliche Objektgraphen. Da im Standard aus Performanzgründen nicht zu tiefe Graphen geladen werden, empfiehlt es sich bei Bedarf Subklassen der Commands zu schreiben, die nach dem Aufruf der eigentlichen Logik noch benötigte Teile des Graphs nachladen. Ein Beispiel hierfür zeigt Listing 2, dort ist auch der Aufbau eines Command zu sehen.

Implementierung mit Java Swing

Anders als bei JSF-Anwendungen bietet jBPM bei der Umsetzung einer Java Swing Oberfläche keine Unterstützung. Aus diesem Grund wurde das, auch unter LGPL zur Verfügung stehende, Toolkit für jbpm (tk4jbpm [8]) vom Autor ins Leben gerufen. Es kann als Grundlage (oder Vorlage, da der Quellcode zur Verfügung steht) der eigenen Oberfläche dienen und wartet mit folgenden Features auf:

- *Admin-Client* (Abb. 2 und 3): Mit dem Admin-Client können neue Prozessversionen aufgespielt, aber auch laufende Instanzen überwacht und beeinflusst werden. Er ist generisch aufgebaut, sodass er in beliebigen Anwendungen zum Einsatz kommen kann, auch als Ergän-

Scruffy

Scruffy ist ein eigenes Framework für Java-Swing Oberflächen, das im Rahmen der camunda-commons Open Source zur Verfügung steht. Ähnlich wie bei aktuellen Browsern wird durch Tabs eine Multifenster-Umgebung realisiert, wobei Scruffy daneben Superklassen für Standardaufgaben und Zusatzdienste wie beispielsweise das Sperren von Fenstern, das modale Überblenden von einzelnen Tabs oder den Aufbau der Menüs bereitstellt. Ein weiteres Feature ist die Konfiguration von Verknüpfungen, die es ermöglichen, dass ein Klick auf die Prozess-ID in der Tabelle des Suchergebnisses direkt die konfigurierte Oberflächenkomponente öffnet. Dies unterstützt ideal das Öffnen eines Prozesses zu einem bestimmten Auftrag.

zung zu einer Weboberfläche. Der Admin-Client des tk4jbpm ist dabei „schlüsselfertig“, im eigenen Projekt muss er nur mit der Referenz auf jBPM versorgt und eventuell in die eigene Swing-Oberfläche eingebaut werden.

- *Taskliste*: Eine Tabelle mit den Aufgaben des Benutzers, wobei zusätzliche Spalten eingeblendet werden können, die bestimmte Prozessvariablen anzeigen.
- *Panel-Mapping-Funktionalität*: Das Toolkit übernimmt das notwendige Mapping von logischen Namen (beim tk4jbpm werden die Namen der Nodes verwendet) auf Swing-Panels. Einzige Voraussetzung ist, Oberflächen-Komponenten als Subklassen des Toolkits zu erstellen. Das tk4jbpm befreit dann den Entwickler von der gesamten Logik des Einlesens der Konfiguration bis zur Instanziierung und Initialisierung der Panels.
- *Prozessmodell*: Es wird ein Prozessmodell bereitgestellt, das beim Zugriff auf jBPM von der exakten Umgebung abstrahiert. So kann die gleiche Logik sowohl innerhalb des Prozesses selbst als auch auf der Oberfläche verwendet werden. Auch ermöglicht es, dass die gleichen Oberflächenkomponenten Tasks oder States in verschiedenen Zuständen darstellen können.

Die Konfiguration für das kleine Beispiel ist in Listing 3 zu sehen, genauere Details können dem Quellcode der Beispielanwendung entnommen werden. Sind die entsprechenden Panels dann implementiert, beispielhaft sei auf Listing 4 verwiesen, kann die Anwendung relativ einfach zusammengebaut werden. Das Beispiel bedient sich dem Swing-Framework Scruffy (siehe Kasten), was die Aufgabe zusätzlich erleichert. Das tk4jbpm kann jedoch auch ohne bzw. im eigenen Swing-Framework zum Einsatz kommen. Der Code zum Starten der Anwendung ist in Listing 5 zu sehen, wobei vorausgesetzt wird, dass die Beispielanwendung korrekt auf einem erreichbaren JBoss Application Server deployed wurde.

Wie erläutert übernimmt das tk4jbpm die Erstellung der Panels, wobei automatisch Buttons für jede ausgehende Transition hinzugefügt werden, wie in Abbildung 4 zu sehen ist. Des Weiteren kann das tk4jbpm auch komplexere Panels für eine Art

Listing 3

tk4jbpm-Konfiguration

```
<tk4jbpm>
 <process name="SimpleOrder" model="com.camunda.
                    jmgui.swing.SimpleOrderProcessModel">
  <start-state name="start" start-class="com.camunda.jmgui.swing.StartOrderPanel" />
  <state name="Warten auf Zahlung" start-class="com.camunda.jmgui.swing.
                                                          CheckPaymentPanel">
   <transition name="Zahlung OK" confirm="true" />
  </state>
  <state name="Auftrag versenden" start-class="com.
                  camunda.jmgui.swing.ShipmentPanel" />
 </process>
</tk4jbpm>
```

Wizard darstellen. Die Idee dahinter ist, verschiedene Panels für ausgehende Transitionen des Prozesses zu konfigurieren. Nach der Entscheidung des Benutzers, welcher Prozesspfad zu nehmen ist, bekommt er ein für diese Transition aufbereitetes Panel angezeigt.

Somit wird einem durch das Toolkit sehr viel Arbeit abgenommen, was einen Einsatz in vielen jbpm-Swing-Projekten interessant macht. Die einzige Anforderung an die Oberflächenkomponenten ist dabei das Erben von einer Frameworkklasse. Ob bestehende Anwendungen einfach umgebaut werden können, führt meist zu der Frage, wie das Data-Binding umgesetzt ist, da Änderungen an Prozessvariablen in der Oberfläche dem Prozessmodell mitgeteilt werden müssen. In eigenen Projekten haben wir dabei mit dem JGoodies Binding Framework [10] gute Erfahrungen gemacht.

Listing 4

Implementierung eines Swing-Panels

```
public class CheckPaymentPanel extends AbstractProcessDetailBasePanel {
 private SimpleOrderProcessModel model;
        public CheckPaymentPanel(ProcessModel model) {
 super(model);
 this.model = (SimpleOrderProcessModel)model;
 }

 @Override
 public void initialize() {}

 @Override
 public JComponent build() {
 FormLayout layout = new FormLayout("pref, 2dlu,
                80dlu:grow", "pref, 2dlu, pref, 2dlu, pref");
 DefaultFormBuilder builder = new DefaultFormBuilder
(layout, new JPanel());
 builder.setBorder(Borders.createEmptyBorder("4dlu,
                                4dlu, 4dlu, 4dlu"));
 CellConstraints cc = new CellConstraints();
 builder.addLabel("Auftrag", cc.xy(1, 1));
 builder.addLabel(String.valueOf( model.loadOrder().
                                getId() ), cc.xy(3, 1));
 builder.addLabel("Kunde", cc.xy(1, 3));
 builder.addLabel(model.loadOrder().getCustomer
                                Name(), cc.xy(3, 3));
 builder.addLabel("Preis", cc.xy(1, 5));
 builder.addLabel(String.valueOf( model.loadOrder().
                                getPrice() ), cc.xy(3, 5));
 return panel;
 }
}
```

Listing 5

Starten der Anwendung (Auszug)

```
public void initApplication() throws Exception {
 initInitialContext();
 setLookAndFeel();
 initTk4jbpm();
 initBusinessLogicFactory();
 configureScruffy();
 buildMenu();
 ScruffyManager.getMainWindowManager().
                                createAndShow();
}

private void initBusinessLogicFactory() throws
                                Exception {
 OrderService orderService = (OrderService) ctx.
                lookup("/jmgui/OrderService/remote");
 Factory.initOrderService(orderService);
}

private void initTk4jbpm() throws Exception {
 CommandService commandService = (CommandService)
ctx.lookup("/jmgui/CommandServiceBean/remote");
 Tk4jbpmConfiguration.initTk4jbpm(commandService,
                                "/tk4jbpm.xml", false);
 Tk4jbpmConfiguration.setUseScruffy(true);
}

private void buildMenu() {
 WindowManager wm = ScruffyManager.getMain
                                WindowManager();
 ScruffyManager.getInstance().addMenu("Datei:
                BpmAdmin", new ShowComponentAction
                (JbpmAdminGuiComponent.class, wm));
 ScruffyManager.getInstance().addMenu("Datei:
                Worklist", new ShowWorklistComponent(wm));
 ScruffyManager.getInstance().addMenu("Datei:
OpenToken", new OpenProcessWithTokenIdAction(wm));
 ScruffyManager.getInstance().
                addMenuSeperator("Datei:---");
 ScruffyManager.getInstance().addMenu("Datei:StartOr-
der", new StartSelectedProcessWithFormAction
                (wm, "SimpleOrder"));
 ScruffyManager.getInstance().addMenu
     ("Datei:AuftragSuche", new ShowSearchComponent
                Action(OrderSearch.class, wm));
 ScruffyManager.getInstance().
                addMenuSeperator("Datei:---");
 ScruffyManager.getInstance().addMenu("Datei:
    ExceptionLog", new OpenExceptionLogAction(wm));
 ScruffyManager.getInstance().addMenu("Datei:End",
                new TerminateScruffyAction());
}
```

Fazit

Oberflächenentwicklung für jBPM-Anwendungen bedarf einiger zusätzlicher Überlegungen sowie prozessorientierten Denkens der Entwickler, was gerne unterschätzt wird. Aus technischer Sicht ist die Unterstützung für Webanwendungen mit JSF bereits heute gut und verbessert sich ständig. Nachholbedarf besteht allerdings im Bereich der Rich-Clients. Dazu hat der Autor das tk4jbpm ins Leben gerufen, das bereits heute in mehreren Java-Swing Anwendungen seinen Dienst verrichtet. Mit diesem ist auch die Swing-Entwicklung kein Hexenwerk mehr. Welche Art von Oberfläche zu bevorzugen ist, ist damit wieder eine rein fachliche Entscheidung. Aus technischer Sicht ist es eigentlich unerheblich.

Bernd Rücker ist Berater und Geschäftsführer bei der camunda GmbH. Er verfügt über mehrjährige Projekterfahrung als Softwarearchitekt und Entwickler im Umfeld von Unternehmensanwendungen in Java EE. Er ist Autor eines EJB3-Buches, einiger Fachartikel sowie Committer im JBoss jBPM Projekt.
Kontakt: bernd.ruecker@camunda.com.

Links & Literatur

[1] JBoss jBPM: labs.jboss.com/jbossjbpm/

[2] Martin Backschat, Bernd Rücker: Enterprise JavaBeans 3.0, Spektrum Akademischer Verlag, 2007

[3] Bernd Rücker: jBPM – Ein Erfahrungsbericht, in JavaSpektrum 6.2005

[4] Adam Bien: JBoss jBPM: Graphentheorie für Geschäftsprozesse, in Java Magazin 5.2006

[5] JBoss Seam: labs.jboss.com/jbossseam

[6] Beispielanwendung: www.camunda.com/toolkit_for_jbpm/jbpm_gui_tutorial.html

[7] jbpm4jsf: wiki.jboss.org/wiki/Wiki.jsp?page=Jbpm4Jsf

[8] tk4jbpm: www.camunda.com/toolkit_for_jbpm/toolkit_for_jbpm.html

[9] tk4jbpm: www.alfresco.com/

[10] JGoodies Binding: binding.dev.java.net