Open-Source-BPEL-Orchester

NEUE SERIE

WS-BPEL ist ein Standard zur Beschreibung von ausführbaren Geschäftsprozessen. Während zeitgleich zum Erscheinen des Standards schon erste kommerzielle Software für WS-BPEL verfügbar war, ist es erst seit Kurzem möglich, den kompletten Lebenszyklus eines BPEL-Geschäftsprozesses (von der Modellierung über die Ausführung bis hin zum Test) mittels Open-Source-Software abzubilden. In dieser dreiteiligen Artikelserie werden wir zeigen, wie man diese Aufgabe mithilfe der drei Softwarepakete Eclipse BPEL Designer, Apache ODE und BPELUnit bewerkstelligen kann.

von Simon Moser, Tammo van Lessen und Daniel Lübke

Mit Screencast und Quellcode auf CD

In der Welt der serviceorientierten Architekturen (SOA) werden Funktionalitäten in interoperable Services zerlegt. Solche Softwareservices können dann kontextunabhängig von verschiedensten anderen Services genutzt werden. Ein Softwareservice zeichnet sich durch eine so genannte „Always-on"-Semantik aus, d. h. er ist zu jeder Zeit erreichbar und verrichtet immer seine Arbeit. Eine Menge solcher Services kann „orchestriert" werden, d. h. verschiedene dieser modularen Funktionalitäten werden in eine Reihenfolge gebracht und bilden

damit wiederum einen höherwertigen, neuen Service. Damit ist auch das Kernparadigma von ausführbaren Geschäftsprozessen beschrieben – das so genannte Two Level Programming. Gemeint ist dabei die Dekomposition in wiederverwendbare Geschäftsfunktionen (Services), die dann wiederum in einer abstrakten Hochsprache zu einer komplexen Geschäftsanwendung komponiert werden.

WS-BPEL [1] ist eine solche Hochsprache und bietet genau diese Funktionalität: Die Verkettung verschiedenster existierender Services, potenziell unter Zuhilfenahme zusätzlicher Logik, in einen neuen Service. Dieses Paradigma mag dem einen oder anderen vertraut erscheinen, und das nicht von ungefähr: Wurden in der Vergangenheit bei Workflow-Management-Systemen (WfMS) Businessfunktionen mithilfe von Workflows integriert [2], so werden heute Services mithilfe von Service-Orchestrierungs-Sprachen in ein höherwertiges Artefakt überführt. Hierbei muss zwischen traditionellen Services, z. B. verteilte CORBA-Services, und Web Services unterschieden werden – der Unterschied ist, dass bei traditionellen Services immer eine starke Kopplung zwischen dem aufrufenden und dem gerufenen Service bestand (Proxy/Stub-Paradigma), wohingegen bei Web Service nur eine lose Kopplung über Nachrichtenaustausch besteht.

Die Aufgabe von WS-BPEL ist es also, über einen einfachen Satz von Befehlen die Orchestrierung von Web Services zu ermöglichen: Einfach gesagt, erlaubt BPEL die Definition von Geschäftsprozessen auf Basis vorhandener Dienste. Dabei wird jeder BPEL-Prozess wiederum als Web Service bereitgestellt. Dieses rekursive Modell erlaubt eine Kaskadierung von Geschäftsprozessen und dadurch die Dekomposition auf verschiedenen Abstraktionsebenen. Mit den entsprechenden Werkzeugen ist eine grafische Modellierung der Geschäftsprozesse möglich; auf diese Weise wird es für Unternehmen leichter, die von den Businessanalysten erfassten fachlichen Prozesse in ausführbare, technische Prozesse zu überführen, da sowohl die fachlichen als auch die technischen Experten ähnliche „Sprachen“ sprechen. Verglichen mit anderen imperativen Programmiersprachen liegen die Schwerpunkte von BPEL demnach auf der direkten Unterstützung von Unternehmensanwendungen. Ein wichtiger Aspekt hierbei ist die grafische Modellierung der Geschäftsprozesse und damit der Servicekomposition. Das Open-Source-Werkzeug Eclipse BPEL Designer [3] bietet hier eine grafische Entwicklungsumgebung an.

> Kernparadigma ausführbarer Geschäftsprozesse ist das Two Level Programming.

BPEL ist eine hybride Prozessmodellierungssprache – d. h. neben bekannten Blockelementen wie *if*, *while* und *forEach* erlaubt es innerhalb von *flow*-Aktivitäten auch rein graphbasierte Modelle. Einzige Einschränkung hierbei ist, dass auf die Definitionen von Zyklen zugunsten einer wohl definierten Ausführungssemantik verzichtet

Artikelserie

Teil 1: Einführung in WS-BPEL, Modellierung und Ausführung von Geschäftsprozessen

Teil 2: Unit Testing von Prozessen

Teil 3: Fehlerszenarien und Regression Testing

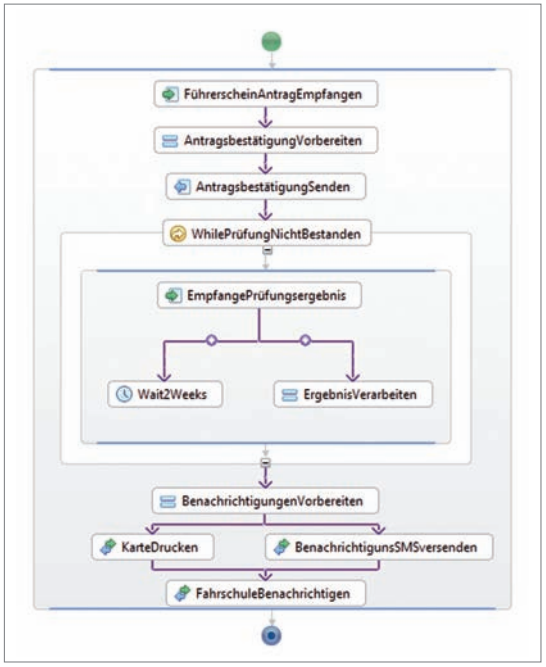

Abb. 1: Der Prozess

wurde. Ein weiteres wichtiges Merkmal von Geschäftsprozessen ist deren potenzielle Langlebigkeit. Eine Prozessinstanz kann durchaus ein Jahr oder länger ausgeführt werden. Das stellt besondere Anforderungen an die Ausführungsumgebung sowie die Sprache selbst. Traditionelle Transaktionen, die ACID-Eigenschaften garantieren, sind in solchen Fällen kaum mehr geeignet und werden durch kompensationsbasierte Transaktionen ersetzt. BPEL stellt dafür die geeigneten sprachlichen Mittel zur Verfügung. Des Weiteren müssen BPEL-Engines, z. B. die Apache ODE BPEL Engine [4], ein so genanntes Phoenix-Verhalten sicherstellen. Stürzt das System während der Ausführung ab, wird die Ausführung der Prozesse nach dem Neustart an der gleichen Stelle fortgesetzt. Weitere bemerkenswerte Eigenschaften der Sprache BPEL sind echte Parallelität (in der BPEL: *flow*-Aktivität), Fault und Compensation Handling für fehlgeschlagene Serviceaufrufe und die Möglichkeit, mittels Sprachmitteln Nachrichten einzelnen Prozessinstanzen zuzuordnen (Correlation). Des Weiteren bieten BPEL Ausführungsumgebungen – dadurch dass sich BPEL-Prozessinstanzen häufig in einem Wartezustand befinden – gute Möglichkeiten zur Verteilbarkeit und damit eine gut skalierende Infrastruktur.

Zunächst soll nun illustriert werden, wie ein Beispielprozess modelliert, ausgeführt und getestet wird. Dafür dient ein Prüfungsprozess einer fiktiven Führerscheinstelle. Wie in Abbildung 1 zu sehen ist, beginnt der Prozess mit der Anmeldung des Prüflings zur praktischen Führerscheinprüfung. Dazu werden einige persönliche Daten wie Name, Vorname, Geburtsdatum sowie die Personalausweisnummer von der Fahrschule an die Führerscheinstelle gesendet. Diese erzeugt eine neue Instanz des Prüfungsprozesses und sendet der Fahrschule eine Bestätigungsnachricht zurück. Danach wartet der Prozess auf das Prüfungsergebnis des Prüflings, übermittelt durch den Fahrprüfer. Wurde die Prüfung erfolgreich abgelegt, gibt der Prozess den Druck des Führerscheins in Auftrag und versendet eine Benachrichtigung per SMS an den Prüfling und benachrichtigt im Anschluss die Fahrschule. War die Prüfung dagegen nicht erfolgreich, müssen zwei Wochen vergehen, bevor der Prüfling einen neuen Versuch starten darf.

Installation des Open-Source-Orchesters

Das Open-Source-Orchester, mit dem wir in dieser Artikelserie arbeiten, besteht aus drei Teilen: dem Eclipse BPEL Designer [3], der BPEL-Engine Apache ODE [4] und dem Unit-Test-Framework BPELUnit [5]. Die Installation des Modellierungswerkzeugs gestaltet sich mit Eclipse 3.4 einfach: Das Eclipse-BPEL-Team stellt unter [6] eine Update-Site bereit. Darüber lässt sich der Designer mithilfe des Updatemanagers installieren. Das in den folgenden Teilen der Artikelserie besprochene Testframework BPELUnit lässt sich ebenfalls über die Update-Site unter [7] installieren.

Dem Modellierungswerkzeug gegenüber steht die Middleware, die die BPEL-Prozesse ausführt. Hier verwenden wir Apache ODE, das derzeit in zwei Ausführungen für die Zielplattformen JBI sowie für beliebige JEE-Webcontainer vorliegt. Für diesen Artikel haben wir uns für die Axis2-Variante, also für das Deployment in einen Webcontainer wie Tomcat entschieden. Die Installation ist einfach: Die neuste stabile Version kann unter http://ode.apache.org/getting-ode.html heruntergeladen werden. Das in der *.zip*-Datei enthaltene WAR wird nun am einfachsten in das *webapps*-Verzeichnis eines Tomcat 5.x kopiert und dadurch deployt. Die Standardkonfiguration von Apache ODE verwendet eine eingebettete Derby-Datenbank. Das ist für einfache Testzwecke zwar ausreichend, führt aber unter höherer Last immer wieder zu Problemen und Dead-Locks. Daher wird empfohlen, im Produktiveinsatz ODE mit einer externen Datenbank zu betreiben.

Der BPEL-Designer wird mit einer Runtime-Integration für Apache ODE ausgeliefert und ermöglicht das Deployment von Prozessen direkt aus Eclipse heraus. Dazu muss Eclipse zunächst einmal den Installationspfad von ODE kennen. Im Preference-Dialog von Eclipse wird dazu ein neues „Server Runtime Environment" für Apache ODE angelegt und die Pfade für die Home-Verzeichnisse von Tomcat sowie ODE eingetragen. Nun ist Eclipse in der Lage, Prozesse auf ODE zu deployen, zu entfernen sowie den Server selbst zu starten und zu stoppen.

Das Schreiben der Partitur – Die Umsetzung in BPEL

Den fiktiven Prüfungsprozess (Abb. 1) möchten wir nun in BPEL umsetzen. Zunächst beginnen wir mit der Modellierung des Kontrollflusses des Prozesses, später werden wir uns um die Datentypen und technische Details wie Serviceendpunkte und Korrelation von Nachrichten kümmern.

Geschäftsprozesse werden typischerweise als Graphen modelliert, bei denen die Knoten die Aktivitäten und die Kanten den Kontrollfluss visualisieren. Daher fügen wir als Wurzelaktivität eine *flow*-Aktivität in unser Prozessmodell ein, die uns eben diese graphbasierte Modellierung ermöglicht. Innerhalb dieser strukturierten Aktivität, die als Container für andere Aktivitäten fungiert, platzieren wir nun unsere Geschäftslogik und bringen sie mithilfe von so genannten Links in die korrekte Ausführungsreihenfolge. Der Prozess beginnt mit einer *receive*-Ak-

tivität, die die Anmeldung zur Prüfung entgegennimmt. Diese Aktivität lässt die BPEL-Engine auf den Aufruf einer Web-Service-Operation warten und kopiert die Daten in eine Variable. Damit die BPEL-Engine weiß, dass über diese Aktivität eine neue Instanz des Prozesses erzeugt werden soll, muss das *create instance*-Attribut auf *yes* gesetzt werden.

wird, haben wir die Wartephase bewusst einfach gestaltet. Hier müsste eigentlich eine Blacklist bei den Fahrprüfern verwaltet werden, sodass ein gesperrter Prüfling innerhalb dieser zwei Wochen die Prüfung gar nicht erst antreten kann. Nach Verlassen der Schleife werden zwei Aktionen parallel gestartet: Mittels einer *invoke*-Aktivität wird ein Service der

BPEL-Ausführungsumgebungen bieten eine gut skalierende Infrastruktur.

Die nächste Aktivität ist eine *assign*-Aktivität, die basierend auf den Eingangsdaten die Bestätigungsnachricht (als eine neue Variable) erzeugt. Danach wird diese Nachricht zurück an die Fahrschule geschickt. Da dieser Teil des Geschäftsprozesses garantiert schnell interpretiert wird und nicht von anderen Parteien abhängt, können wir beide Interaktionsaktivitäten an eine synchrone Operation binden. Das machen wir, indem wir die Antwortnachricht anstatt mit einer *invoke*-Aktivität mit einer *reply*-Aktivität an die gleiche Operation und den gleichen Partner senden. Später lässt sich das dann auf einen HTTP-Request/Response-Zyklus abbilden. Im nächsten Schritt warten wir auf das Prüfungsergebnis des Fahrprüfers. Da in BPEL keine Zyklen im grafischen Kontrollfluss erlaubt sind, müssen wir uns mit einer *while*-Aktivität behelfen, die mithilfe einer erneuten *receive*-Aktivität in einen blockierenden Wartemodus versetzt wird. Es wird also erneut auf den Aufruf einer Web-Service-Operation gewartet, in diesem Fall allerdings eine asynchrone *one-way*-Web-Service-Operation. Sobald die Nachricht des Fahrprüfers eintrifft, wird die Blockierung von der BPEL-Engine aufgehoben und die eingehende Nachricht mithilfe einer *transition condition* geprüft (in Abb. 1 als der kleine Kreis auf dem Link zu sehen). Wenn die Fahrprüfung erfolgreich war, wird der nächste Schritt (2 Wochen warten) übersprungen und die Schleife beendet, andernfalls geht sie erneut in den Warten-Zustand. Damit der Beispielprozess nicht zu komplex

Führerscheinstelle aufgerufen, der die Fahrerlaubnis druckt, und parallel dazu wird ein SMS-Service aufgerufen, um den Prüfling über die erfolgreiche Prüfung zu informieren. Als letzter Schritt wird, ebenfalls mittels einer *invoke*-Aktivität, die Fahrschule benachrichtigt.

Das Stimmen der Instrumente – die technische Verfeinerung

Nachdem auf diese Art und Weise der Kontrollfluss modelliert wurde, muss nun der Datenfluss festgelegt werden. Dieser beinhaltet die Daten, die in den Nachrichten von und zu dem Prozess enthalten sind, und die Daten, die der Prozess intern verwendet. Die Nachrichten, die zwischen dem Prozess und anderen Systemen ausgetauscht werden, werden in WSDL-Dokumenten spezifiziert und Operationen zugeordnet. Über diese Operationen kann ein Aufrufer mit einem Service, also auch mit unserem Prozess, interagieren. Für unser Beispiel müssen verschiedene Services definiert werden:

- ein Service, der den Prozess startet, d. h. die Prüflingsdaten entgegennimmt und den Start des Prüfungsprozesses bestätigt
- ein Service, mit dem ein Prüfer das Prüfungsergebnis melden kann
- ein Service, mit dem SMS verschickt werden können
- ein Service, mit dem die Führerscheine gedruckt werden können
- ein Service, mit dem eine Fahrschule über das Prüfungsergebnis benachrichtigt werden kann

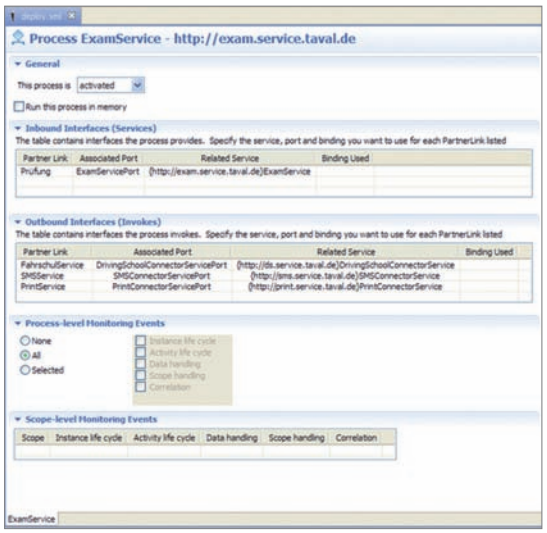

Abb. 2: Der Deployment Descriptor

Einige dieser Services werden vom Prozess angeboten (die ersten beiden), während die anderen von anderen Serviceanbietern (SMS-Dienst, Druckservice und der Fahrschule) angeboten werden müssen. Damit unser Beispiel aber in sich geschlossen nachvollzogen werden kann, verwenden wir JAX-WS-basierte Attrappen, die auch auf der Heft-CD zu finden sind.

In WSDL werden Services zunächst rein logisch in so genannten *port types* definiert. Über *bindings* und *services* werden dann erst konkrete Implementierungen und deren Verfügbarkeit im Netzwerk spezifiziert. Die Daten, die nun an den Service geschickt werden, werden üblicherweise mittels XML Schema definiert. Das WSDL mit den genauen Definitionen der Services kann von der Heft-CD in das Projekt kopiert werden. Eine WSDL mit den Definitionen für Services und den entsprechenden Nachrichten kann problemlos für die Serviceentwicklung z. B. mit Java verwendet werden. Der BPEL-Prozess muss nun allerdings wissen, ob er einen Service implementiert oder einen benutzt. Um das zu beschreiben, werden so genannte *partner link types* definiert. Ein *partner link type* definiert die Interaktion zwischen zwei logischen Services (*port types*), das bedeutet beispielsweise, dass ein Fahrschulservice von einem beliebigen Client aufgerufen werden kann. Im BPEL-Prozess werden dann *partner links* basierend auf den *partner link types* definiert. So kann ein BPEL-Prozess theoretisch mit beliebig vielen Fahrschulen interagieren.

In unserem Prozess müssen nun für jeden verwendeten Service ein *partner link type* und ein *partner link* angelegt werden. Das unterstützt der BPEL-Editor, indem über die Tray in dem *partner link*-Bereich ein neuer *partner link* und im selben Zug bei Bedarf ein *partner link type* angelegt werden kann. Nun können bei allen *receive*-, *invoke*- und *reply*-Aktivitäten die passenden *partner links* angegeben werden.

Das Gleiche muss nun auch mit den Variablen geschehen. BPEL kennt drei Typen von Variablen: Nachrichtenvariablen, die empfangene oder zu sendende Nachrichten enthalten, und Variablen, die entweder XSD-Typdefinitionen- oder -Elementdeklarationen entsprechen. Daher müssen nun für jede Nachricht, die empfangen oder gesendet wird, eine Variable des passenden Typs angelegt werden. In den Eigenschaften der jeweiligen Aktivitäten kann diese Variable dann als *input variable* oder *output variable* festgelegt werden, d. h. die BPEL-Engine wird die empfangenen Nachrichten in dieser Variable speichern oder die Nachricht, die in dieser Variable enthalten ist, senden.

Nachdem die Interaktionsaktivitäten mit Variablen verknüpft wurden, können wir uns dem Datenfluss widmen. In den *assign*-Aktivitäten können Variablen Werte zugewiesen werden. Dazu kann sowohl Quelle als auch Ziel als XPath-Ausdruck angegeben werden. Zu beachten ist hierbei, dass Variablen initialisiert worden sein müssen, bevor mittels XPath auf sie zugegriffen werden kann. Beim Empfang einer Nachricht geschieht das automatisch, in alle anderen Variablen muss zunächst mittels *assign* die XML-Struktur kopiert werden, damit die XPath-Ausdrücke aufgelöst werden können.

In unserem Prozessmodell haben wir drei Bedingungen platziert, die den Kontrollfluss auf Basis der Prozessvariablen beeinflussen. Zum einen ist das die Schleifenbedingung, die die Schleife beendet, wenn ein positives Prüfungsergebnis vorliegt. Diese Bedingung prüft per XPath, ob die Variable *$bestanden* auf *true* gesetzt ist. Zum anderen sind das die zwei *transition conditions*, die nach dem Empfang des Prüfungsergebnisses evaluiert werden müssen, um zu testen, ob die Fahrprüfung erfolgreich war. In diesem Fall evaluiert der XPath-Ausdruck *$PrüfungsergebnisRequest.report-ExamResultParameters/passed!=false()* zu *true* und aktiviert damit die *assign*-Aktivität, die die Variable für die Schleifenbedingung entsprechend setzt. Die Negation des XPath-Ausdrucks findet in der anderen *transition condition* Anwendung und aktiviert bei Nichtbestehen die *wait*-Aktivität.

Da ein Prozess häufig mit mehreren unabhängigen Partnern interagiert (in unserem Beispiel mit dem Kartendruck-, dem SMS-, dem Fahrschul- und dem Fahrprüferservice), muss sichergestellt werden, dass eingehende Nachrichten der richtigen Prozessinstanz zugeordnet werden. Das dazugehörige Konzept nennt man *correlation*. Dafür müssen Partner und Prozessinstanz ein gemeinsames Identifikationsmerkmal definieren. Dieses Merkmal setzt sich normalerweise aus Businessdaten des Prozesses zusammen, in unserem Fall einer eindeutigen Nummer für jeden Prüfling. Dazu definiert man im BPEL-Designer ein so genanntes *correlation set* und fügt diesem eine *property* hinzu – wir nennen sie in unserem Beispiel *pid* (für *processing ID*). Für diese *property* müssen nun *property aliases* definiert werden, sodass für jede eingehende Nachricht ein Feld dieses Identifizierungsmerkmal trägt. In den XML-Schemata der beiden eingehenden Nachrichten haben wir ein Element *processingnumber* definiert, d. h. wir müssen für beide Nachrichten einen *property alias* definieren, der auf dieses Element zeigt. Die Aktivitäten, die Nachrichten von außen empfangen, müssen dann dieses *correlation set* verwenden, die erste dieser Aktivitäten muss das *correlation set* initialisieren.

Das Betreten des Konzertsaals – das Deployment

Der nächste und letzte Schritt vor der Ausführung unseres Beispielprozesses ist das so genannte Deployment. BPEL-Prozesse machen zunächst keine Annahmen über die zu verwendende

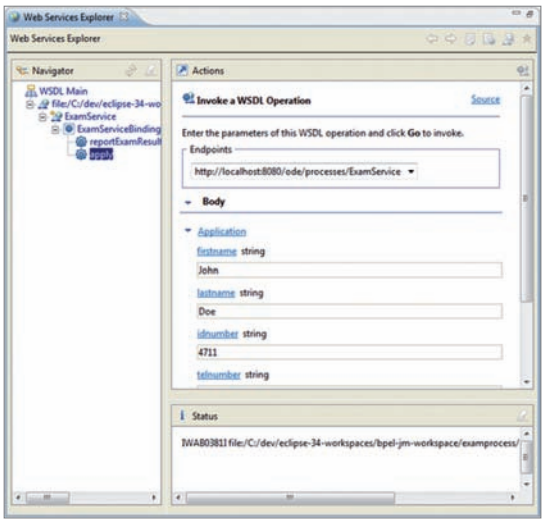

Abb. 3: Web Service Explorer

Infrastruktur. Um einen Prozess auf eine BPEL-Engine zu deployen, müssen also noch plattformspezifische Eigenschaften definiert werden. Diese Aufgabe übernimmt der Deployment-Deskriptor, im Fall von Apache ODE, die *deploy.xml*. In dieser Datei werden zum einen die physischen Endpunkte an die im Prozessmodell verwendeten *partner links* gebunden, zum anderen kann die Engine weiter parametrisiert werden, z. B. welche Events während der Ausführung erzeugt werden sollen und ob die Prozessausführung für langlaufende Prozesse persistent sein soll.

Der BPEL-Designer bringt einen Editor für die ODE *deploy.xml* gleich mit. Wir legen also in unserem Projekt eine *deploy.xml* an und öffnen sie. Wie in Abbildung 2 zu sehen, werden automatisch die Prozesse des Eclipse-Projekts importiert. Per Doppelklick in der *PartnerLink*-Tabelle können nun den *PartnerLinks* die Ports und Endpunkte der verknüpften WSDL-Dateien zugeordnet werden. Nun sind alle zur Ausführung notwendigen Artefakte vorhanden und dem Deployment unseres Beispielprozesses steht nichts mehr im Wege. Über das Kontextmenü des ODE-Server-Eintrags in der *Servers*-View kann unser BPEL-Projekt direkt auf den Server deployt und gestartet werden.

Das Konzert – die Ausführung

Nach dem Deployment wartet unser BPEL-Prozess auf eine eingehende Nachricht für die initialisierende *receive*-Aktivität. Der BPEL-Prozess stellt sein WSDL unter http://localhost:8080/ode/processes/ExamService?wsdl bereit. Dieses Dokument kann verwendet werden, um eine Clientanwendung an den Prozess zu binden. Zum Test verwenden wir jedoch den Eclipse Web Service Explorer, der mithilfe des WSDL-Dokuments geeignete SOAP-Nachrichten erstellen kann. Zunächst müssen allerdings die Serviceattrappen gestartet werden. Hierzu muss das auf der Heft-CD mitgelieferte Shell-Skript *run-services.(bat|sh)* gestartet werden. Im Anschluss starten wir aus dem Kontextmenü (Web Services | Test with Web Services Explorer) auf die *ExamService.wsdl* den Eclipse Web Service Explorer (Abb. 3). Eine Instanz unseres Prozesses wird mit der Operation *apply* erzeugt. Die Prüfungsergebnisse des Fahrlehrers werden über die Operation *reportExamResult* übermittelt. Auf der Konsole der Serviceattrappen werden dann die Ausgaben des SMS-, Print- und Fahrschulservices angezeigt.

Fazit

Einen Geschäftsprozess in WS-BPEL abzubilden, zu implementieren und mittels unseres Open-Source-Orchester zu „spielen" war das Ziel für diesen ersten Teil der Artikelserie. Damit die einzelnen Schritte besser nachvollzogen werden können, haben wir auf der Heft-CD einen Screencast bereitgestellt. In der nächsten Folge werden wir darauf aufbauen und eine Regressionstest-Suite für unser Beispiel erstellen. Diese wird uns helfen, falsche Töne bei der Weiterentwicklung des Prozesses sofort zu erkennen. ■

Dipl.-Inf. Tammo van Lessen arbeitet als unabhängiger Berater im Bereich SOA/BPM und ist Doktorand am Institut für Architektur von Anwendungssystemen der Universität Stuttgart bei Prof. Dr. Frank Leymann. Seine Forschungsschwerpunkte liegen in den Bereichen Conversational Web Service Interactions, BPEL sowie Semantic Web Services und sBPM. Er ist zudem Committer und PMC Member bei Apache ODE.

Dipl.-Ing. Simon Moser ist seit 2003 als Softwareentwickler und Architekt im Bereich Business Integration Tools im Forschungs- und Entwicklungszentrum der IBM in Böblingen angestellt. Er war aktiv an der Entwicklung des WS-BPEL-Standards beteiligt und hat IBM dort als Mitglied des OASIS Technical Committees repräsentiert. Seit 2006 leitet er zudem das BPEL Designer Project bei der Eclipse Software Foundation.

Dr. Daniel Lübke ist Senior Consultant bei der innoQ Schweiz GmbH und arbeitet in Kundenprojekten im Bereich SOA und modellgetriebener Softwareentwicklung. Zuvor hat er am Fachgebiet Software Engineering der Leibniz Universität Hannover im Bereich SOA promoviert. Daniel Lübke ist Maintainer des Open-Source-Frameworks BPELUnit zum Testen von BPEL-Prozessen.

Links & Literatur

[1] OASIS WS-BPEL TC: Web Services Business Process Execution Language Version 2.0, OASIS-Standard, 2007: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[2] Leymann, F.; Roller, D.: „Production Workflow", Prentice Hall, 2000

[3] Eclipse BPEL Designer: http://www.eclipse.org/bpel

[4] Apache ODE: http://ode.apache.org

[5] BPELUnit: http://www.bpelunit.net

[6] http://download.eclipse.org/technology/bpel/update-site/

[5] http://update.bpelunit.net