Wenn mal jemand im BPEL-Orchester schief spielt …

Im ersten Teil dieser Artikelserie wurde eine BPEL-Orchestrierung für einen fiktiven Führererscheinprozess entwickelt. Im zweiten Teil wurden für diesen Prozess drei Testfälle entworfen. In diesem Teil werden wir zunächst, dem „Test-First"-Paradigma folgend, einen vierten Testfall definieren, mit dem ein funktionales Problem identifiziert wird. Daraufhin korrigieren wir das Prozessmodell und stellen damit sicher, dass wir mit den vorhandenen Testfällen ein sicheres Regression-Fallnetz für unseren Prozess haben. Abschließend widmen wir uns noch ein wenig dem Management und der Überwachung von Prozessinstanzen.

von Tammo van Lessen, Daniel Lübke und Simon Moser

JAX Speaker

Im letzten Teil der Artikelserie haben wir unseren BPEL-Prozess sowohl einzeln und isoliert als auch vollständig mit allen verwendeten Services getestet. Allerdings gibt es eine fachliche Anforderung, die bisher nicht betrachtet wurde: Ein Prüfling darf maximal dreimal zur Führerscheinprüfung antreten. In diesem Teil werden wir nun diese Funktion nachimplementieren und dabei „Test First" im BPEL-Umfeld anwenden.

„Test First" ist ein zyklisches Vorgehen, in dem Schritte wiederholt werden, bis die Software fertig implementiert ist:

1. Es wird ein Testfall für eine Funktion implementiert
2. Der Test wird ausgeführt, um sicherzustellen, dass er fehlschlägt; damit wird gezeigt, dass der Testfall korrekt ist
3. Die Software (bei BPEL das Prozessmodell) wird so weit implementiert, dass sie den Testfall (und nicht mehr) erfüllt
4. Eventuell wird ein Code Refactoring durchgeführt, um den Code aufzuräumen und wartbarer zu machen

Quellcode auf CD

Insbesondere in vielen agilen Projekten hat sich das Test-driven Development (TDD) als eine vorteilhafte Technik für die Softwareentwicklung etabliert. Der unter Extreme Programming [1] als „Test First" bekannt gewordene Ansatz erreichte viel Aufmerksamkeit. Hierbei wird, wie der Name schon andeutet, die Entwicklung durch Testfälle gesteuert, die vor der eigentlichen Implementierung geschrieben werden. Eine Funktionalität gilt als implementiert, wenn die dazugehörigen Tests von der Software bestanden werden.

Was mit „klassischen" Programmiersprachen wie Java funktioniert, ist auch im „Programming in the Large" möglich. Wir werden dies mit dem dreimaligen Durchfallen durch die Führerscheinprüfung demonstrieren. Aus diesem Grund implementieren wir nun mittels BPELUnit, dem „Test-First"-Paradigma folgend, einen vierten Testfall *UnsuccessfulExamAtThirdTry*, bei dem ein Kandidat dreimal durch die Prüfung fällt. Der fachlichen Anforderung folgend, ist der Test dann erfüllt, wenn die Fahrschule darüber unterrichtet wird, dass der Prüfling die Fahrprüfung für diese Prüfungsperiode endgültig nicht bestanden hat und er sich erst nach einer gesetzlichen Sperrfrist erneut anmelden darf. Dazu duplizieren wir den letzten Testfall *SucessfulExamAtThirdTry*. Dies ist im BPELUnit-Editor leicht möglich. Nach Selektion des Testfalls kann dies über den Button Duplicate erreicht werden. Den kopierten Testfall nennen wir nun *UnsuccessfulExamAtThirdTry*. Nun soll der Prüfling dreimal durch die Führerscheinprüfung fallen. Daher ändern wir die dritte Benachrichtigung über das Prüfungsergebnis: Den Inhalt des Elements *<passed>* setzen wir, wie in Abbildung 1 gezeigt, auf *false*. Zusätzlich entfernen wir alle Aktivitäten in den *partner tracks* für den SMS Service und den Print Service, denn wir erwarten nicht mehr, dass ein

Artikelserie

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

Teil 2: Unit Testing von Prozessen

Teil 3: Fehlerszenarien und Regression Testing

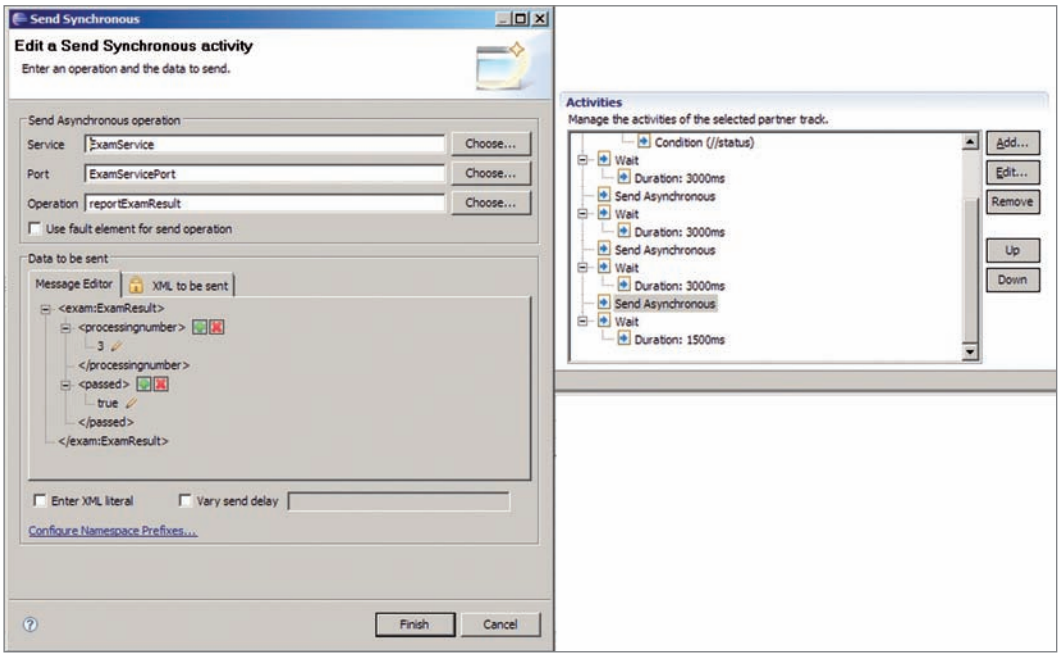

Abb. 1: Parameter des modifizierten Test-Cases „UnsuccessfulExamAtThirdTry"

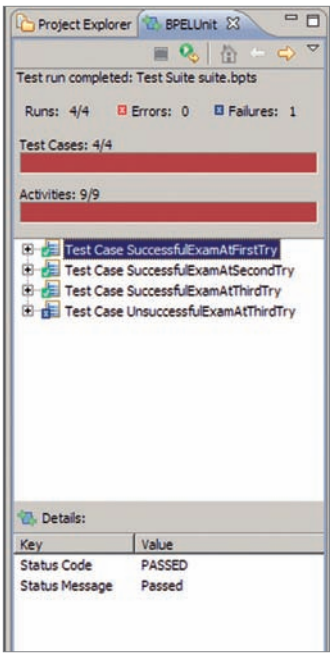

Abb. 2: BPELUnit ohne angepasstes Prozessmodell

Führerschein ausgestellt wird. Auch die Fahrschule erhält nun eine andere, unerfreulichere Nachricht: „Ihr Fahrschüler hat die Prüfung leider nicht bestanden." An dieser Stelle wird deutlich, dass man in Test-driven Development nicht nur testet, sondern ebenfalls genau spezifiziert. Wie die Meldung aussehen soll, die die Fahrschule bekommt, legen wir bereits hier ganz konkret fest. Später muss sie genauso im Prozess implementiert sein, damit der Testfall erfolgreich ist und die Funktionalität als vollständig implementiert gilt.

Wir starten nun die Testsuite und hoffen, dass sich der Balken rot färbt. Wir haben das dreimalige Nichtbestehen bisher noch nicht implementiert, und somit wäre ein grüner Balken ein Zeichen dafür, dass der Testfall noch nicht stimmt. Erst wenn der Balken rot ist, fahren wir fort und implementieren die zusätzliche Logik. Abbildung 2 zeigt, dass das BPEL-Unit-Ergebnis sich mit unseren Erwartungen deckt, der Test fehlschlägt und der Testfall an sich somit also in Ordnung ist.

Apache ODE

Beim Schreiben des Artikels ist uns ein Bug in der verwendeten ODE-Version aufgefallen. Das Problem ist inzwischen bekannt und bestätigt, in dem nächsten offiziellen Release (1.3.4) wird das Problem behoben sein. Damit Sie trotzdem unseren Anwendungsfall nachvollziehen können, haben wir eine gepatchte Vorabversion erstellt und stellen diese zusammen mit dem Quellcode auf der Heft-CD sowie zum Download zur Verfügung (1.3.4-SNAPSHOT).

Auch BPELUnit wurde inzwischen aktualisiert. Bitte stellen Sie über den Update Manager von Eclipse sicher, dass Sie die neuste Version (1.3.0) installiert haben.

Die redigierte Partitur – Modifikationen des Prozessmodells

Das Testergebnis aus dem vorigen Abschnitt zeigt allerdings auch, dass das Prozessmodell erweitert werden muss, um die fachliche Anforderung mit umzusetzen. Dazu kapseln wir die Hauptschleife des Prozesses (*WhilePrüfungNichtBestanden*) in einen *scope*. Das ist deshalb nötig, weil sich dieser Teil des Prozesses nach der dritten fehlgeschlagenen Prüfung mit einem Fault beenden soll, dieser Fault kann dann in einem *fault handler* gefangen und behandelt werden. Da *fault handler* nur an *scopes* (und den Prozess selbst) angeheftet werden können, müssen wir die Schleife in einen *scope* kapseln. Zwei weitere *assign*-Aktivitäten sind nötig, um die Schleifendurchläufe zu zählen: Zunächst eine *assign*-Aktivität, die einen Zähler initialisiert, bevor die Schleife zum ersten Mal betreten wird. Der Zähler ist als lokale Integer-Variable *Versuchscounter* direkt auf dem *scope* definiert, also auch nur innerhalb von diesem gültig. Da ein *scope* nur genau eine Kindaktivität haben darf, nun aber eigentlich zwei Aktivitäten enthalten wären (das initialisierende *assign* und die *while*-Schleife), fügt der Eclipse-BPEL-Designer automatisch eine *sequence*-Aktivität ein, um diese beiden Kindaktivitäten zu kapseln und die syntaktische Korrektheit des Prozesses zu gewährleisten. Darüber hinaus benötigt man ein weiteres *assign*, das, nachdem das

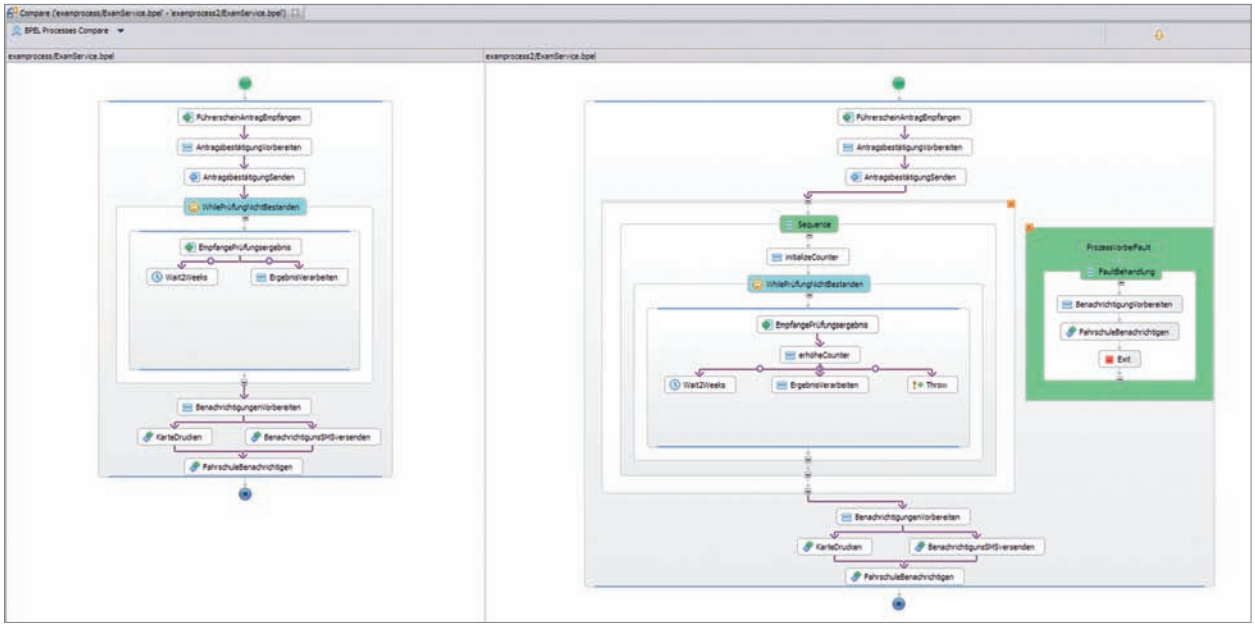

Abb. 3: Der Führerscheinprozess vorher/nachher

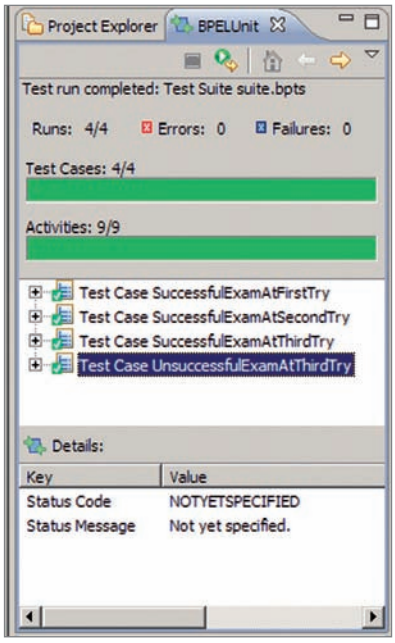

Abb. 4: Erfolgreicher Test mit dem angepassten Prozessmodell

Prüfungsergebnis bekannt ist, innerhalb der Schleife den Zähler erhöht. Dadurch, dass dieses zweite *assign* nach dem Empfangen des Prüfungsergebnisses aber vor der weiteren Prozesslogik eingefügt werden muss, muss die weitere Prozesslogik an dieses *assign* angehängt werden, d. h. die Links zu *wait2Weeks* bzw. *ErgebnisVerarbeiten* wurden umgehängt. Die beiden bestehenden Links müssen ihre *transition condition* anpassen, und zusätzlich benötigt man noch einen dritten Link, der prüft, ob noch ein weiterer Versuch erlaubt ist (Zähler < 3) oder ob dies der letzte Versuch war. Tabelle 1 stellt die bestehende und die neue Logik der *transition conditions* gegenüber.

Der neu eingefügte dritte Link erzeugt dann mittels einer *throw*-Aktivität einen Fehler. Abbildung 3 stellt den ursprünglichen und den modifizierten Prozess mithilfe der Betaversion eines grafischen *BPEL-Process-Compare*-Werkzeugs als Vorher/nachher-Bild gegenüber.

Falls mithilfe der *throw*-Aktivität ein Fehler erzeugt wird, wird er von dem *fault handler* gefangen und verarbeitet, um sicherzustellen, dass der Prozess korrekt beendet wird. Dazu wird, analog zum normalen Beenden des Prozesses, eine zu sendende Nachricht vorbereitet (in diesem Fall mit dem unerfreulichen Text „Ihr Fahrschüler hat die Prüfung leider nicht bestanden." – siehe Abschnitt 1). Anschließend wird mittels der *invoke*-Aktivität die Fahrschule benachrichtigt und der Prozess schlussendlich mit einer *exit*-Aktivität komplett terminiert. Nachdem die zusätzliche Funktionalität des endgültigen Nichtbestehens implementiert ist, können wir unsere Testsuite noch einmal ausführen. Haben wir keine Fehler gemacht, so sollten nun alle Tests grün sein. Der neue Testfall sollte also von rot nach grün gewechselt sein und uns so zeigen, dass die neue Funktion wirklich richtig implementiert ist. Das ist der Test-driven-Development-Anteil der Testsuite.

Auf der anderen Seite sollten alle anderen Tests weiterhin grün geblieben sein. Durch das schnelle und unkomplizierte Ausführen der Testsuite haben wir so automatisch Regressionstests, die uns anzeigen können, wenn bisherige Funktionen aus Versehen kaputtgegangen sind. So könnten z. B. *transition conditions* auf Links nicht konsistent angepasst worden sein oder der neue Kontrollfluss könnte bestehende Funktionen unerreichbar ma-

Link von Aktivität 'EmpfangePrüfungsergebnis'/ 'initializeCounter' zu Aktivität ...	Alte transitionCondition-Logik	Neue transitionCondition-Logik
Wait2Weeks	[...]passed='false'	[...]passed='false'and $Versuchscounter < 3
ErgebnisVerarbeiten	[...]passed='true'	[...]passed='true' and $Versuchscounter <= 3
Throw	[Link noch nicht vorhanden]	([...]passed='false' and $Versuchscounter >= 3) or ([...]passed='true' and $Versuchscounter > 3)

Tabelle 1: Navigationslogik vorher/nachher

Abb. 5: Webkonsole von Apache ODE

chen. Dies ist bei unserem Prozess glücklicherweise nicht der Fall. Wie in Abbildung 4 dargestellt, zeigen die Tests nun an, dass sowohl die neue als auch die alte Funktionalität für den Einsatz bereit sind.

Damit wären nun die Modellierung, Unit-Test, Remodellierung, Regressionstest und Ausführung des Prozessmodells abgeschlossen. Der fiktive Führerscheinprozess könnte nun bei den Führerscheinstellen in Produktion gehen, sodass alle Führerscheinprüfungen in Zukunft über dieses System abgewickelt werden könnten und jede Anmeldung eines Fahrschülers eine neue Prozessinstanz im System erzeugt. Obwohl mit der Modellierung und dem Automatisieren von Geschäftsprozessen schon viel gewonnen ist, übergibt man auf diese Weise auch die Kontrolle und zum Teil auch Verantwortung an Softwaresysteme, deren Fehlerfreiheit trotz gründlicher Tests nie garantiert werden kann. Deshalb ist es wichtig, dass das Business-Process-Management-System (BPMS) die Benutzer bei der Überwachung der Prozesse unterstützt.

Das Orchester auf Tournee – Monitoring von Prozessen

Für den reibungslosen Ablauf eines Produktionssystems stellen sich häufig dieselben Fragen: Wie viele Prozessinstanzen wurden ordnungsgemäß ausgeführt? Gibt es welche, die stehen geblieben sind, z. B. weil ein externer Dienst nicht erreichbar war? Wie kann ich solche Instanzen wieder in Gang bringen? Das sind Fragen, die bei der Überwachung von technischen Prozessen relevant sind. Apache ODE unterstützt die Anwender hier in zweierlei Hinsicht: Zum einen stellt es eine einfache Webkonsole (Abb. 5) bereit, mit deren Hilfe man sich einen Überblick über die Prozesse und den Zustand ihrer Instanzen verschaffen kann. Diese Konsole kann in unserem Fall unter http://localhost:8080/ode aufgerufen werden. Zum anderen stellt ODE ein Management-API für Prozesse und deren Instanzen als Web Service bereit. Da ist zum einen das Deployment-API, mit dessen Hilfe neue Prozessmodelle in ODE deployt werden können. Ist bereits ein Prozessmodell gleichen Namens aktiv, so greift die Prozessversionierung von ODE und schickt die ältere Version in den Ruhestand, bevor die neue aktiviert wird. Existierende Prozessinstanzen werden dann noch nach dem alten Modell verarbeitet. Ebenso lassen sich Prozessmodelle oder Versionen davon wieder entfernen oder auflisten. Das Prozess-Management-API unterstützt den Anwender bei der Parametrisierung von Prozessmodellen und erlaubt es, Prozessmodelle manuell zu aktivieren/deaktivieren. Die umfangreichste Schnittstelle ist das Instanzmanagement-API, mit ihm kann man laufende Prozessinstanzen abfragen und steuern. Tabelle 2 zeigt einen Auszug aus dem ODE-Instanz-Management-API – das vollständige API findet man unter [2].

Eine Übersicht der bereitgestellten Dienste (inklusive der durch Prozessmodelle implementierten) lässt sich über

die Axis2-Webkonsole abrufen: http://localhost:8080/ode/services/listServices

Fazit der Artikelserie

In dieser Artikelserie haben wir gezeigt, wie man ausschließlich mit Open-Source-Software ein System zur BPEL-Serviceorchestrierung aufbauen und betreiben kann. Damit haben Sie einen Überblick über alle wichtigen Features von BPEL, dem BPEL Designer und erste Einblicke in die Arbeit mit Apache ODE und BPELUnit erlangt. Mit dieser Version des Fahrprüfungsprozesses, den man nun produktiv einsetzen könnte, beenden wir unsere Serie. Wir hoffen, dass wir Ihnen BPEL näher bringen und die ersten Berührungsängste nehmen konnten. Nun ist es an Ihnen, als Dirigent aufzutreten und Services zu orchestrieren! Sollten dabei Fragen aufkommen, helfen Ihnen sicherlich die vielen Leute auf den ODE-, BPEL-Designer- und BPELUnit-Mailinglisten. Darüber hinaus gibt es unter [3] auch im Internet noch weitere How-tos für BPEL und Apache ODE. ■

QuickFacts zum BPEL-Orchester

Eclipse BPEL Designer
- BPEL-2.0-kompatibles, grafisches Modellierungswerkzeug für BPEL-Prozessmodelle
- XML-Editing über synchronisierte XML View möglich
- Eclipse-integriert, BPEL Validator und grafisches Compare von Prozessen möglich

Apache ODE
- Schlanke BPEL-Maschine mit Support für BPEL4WS 1.1 und WS-BPEL 2.0 [4]
- Modulare und erweiterbare Architektur, integriert sowohl mit Axis2 als auch JBI
- Unterstützt das HTTP WSDL-Binding und erlaubt so den Aufruf von REST-style Web Services.

BPEL Unit
- Einfaches Testframework zum Testen von BPEL-Prozessen auf beliebigen BPEL-Servern
- Ersetzen und transparentes Mocking von verwendeten Services
- Integration in Eclipse und Ant

Suspend	Hält eine Prozessinstanz an. Dies kann hilfreich sein, wenn Wartungsarbeiten an Partnerdiensten geplant sind und die Prozessinstanz solange ruhen soll.
Resume	Setzt eine angehaltene Prozessinstanz fort.
Terminate	Beendet eine Prozessinstanz manuell.
Delete	Löscht eine oder mehrere Prozessinstanzen über einen Filterausdruck. Mit dem Löschen werden auch alle Logging-Informationen, die mit den ausgewählten Prozessinstanzen verbunden sind, entfernt. Diese Operation soll die Datenflut, die durch Log/Audit-Informationen entsteht, beherrschbarer machen.
listInstances queryInstances listAllInstances	Gibt eine Liste aller Prozessinstanzen zurück. Diese Liste kann durch einen Filterausdruck eingeschränkt, auf eine Anzahl begrenzt und sortiert werden. So ist es beispielsweise möglich, sich alle fehlgeschlagenen Instanzen eines bestimmten Prozessmodells anzeigen zu lassen. Aus dem Ergebnis können wir den Zustand und einige Zeitstempel der Instanz ablesen. Für Analysezwecke sehr praktisch ist die Auflistung der Correlation Properties.
getInstanceInfo	Gibt detaillierte Informationen über eine Prozessinstanz zurück. Diese Instanz wird durch eine eindeutige IID (Instance ID) ausgedrückt, wie sie beispielweise in dem Rückgabewert von *listInstances* zu finden ist.
recoverActivity	Kann eine fehlgeschlagene Aktivität „wiederbeleben". In der Web-Services-Welt unterscheidet man *faults* und *failures*. Faults sind erwartete (fachliche) Fehler, während man eine Netzwerkfragmentierung, einen Timeout oder einen nicht korrekt funktionierenden Dienst als Failure klassifiziert. Mit solchen Problemen muss man in der Regel anders umgehen als mit fachlichen Fehlern, die man in der Prozesslogik entsprechend behandelt, denn ein Failure ist in der Regel ein temporäres Problem, wegen dem man einen langlaufenden Prozess nicht unbedingt terminieren möchte. Tritt ein Failure bei dem Aufruf eines Partnerdienstes auf, so setzt ODE diese Aktivität in einen Wartezustand. Mit *recoverActivity* weckt man sie wieder aus diesem Winterschlaf. Dazu kann man die Aktivität a) noch einmal starten (*retry*), b) abbrechen (*cancel*) oder c) einen Fehler erzeugen (*fault*), der von den FaultHandlern in BPEL behandelt werden kann.

Tabelle 2: Auszug aus dem ODE-Instanzmanagement-API

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.

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.

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.

Links & Literatur

[1] Extreme Programming: http://www.extremeprogramming.org/

[2] ODE-Instanzmanagement-API: http://ode.apache.org/bpel-management-api-specification.html bzw. http://ode.apache.org/javadoc/org/apache/ode/bpel/pmapi/InstanceManagement.html

[3] BPEL und BPELUnit Tutorials der Leibniz Universität Hannover, FG Software Engineering: http://www.se.uni-hannover.de/lehre/2008winter/ws2008_soa.php#tut

[4] Apache ODE – WS-BPEL 2.0 Specification Compliance