JAVA Mag inkl. CD

Zweimal Swing: Gestatten SAF und FEST »84

S&S

Deutschland €8,50 Österreich €9,80 Schweiz sFr 16,80 2.2010

Java magazin

Java • Architekturen • SOA • Agile

www.javamagazin.de

CD-INHALT

Swing App Framework
Session von der JAX 2009 in voller Länge

DIE HIGHLIGHTS

Citrus Framework

WEITERE INHALTE

FEST Testing Framework

SAF

Apache Jackrabbit

jCouchDB

Alle CD-Infos ab Seite 3

NoSQL
Alternative Speichersysteme – Schöne neue Welt? »14

Jahres-Archiv 2009
Jetzt exklusiv für Abonnenten online »2

jax 2010
Alle Infos im Heft

SINN UND UNSINN VON CLOUD COMPUTING

»56

Lose Kopplung und BPMN

Passt das? »67

Security-Modularisierung

Modulares Recycling »74

Pragmatic SOA

Beschränken auf das Wesentliche »91

4 194586 708507 02

Datenträger enthält Info- und Lehrprogramme gemäß §14 JuSchG

Vom Ausprobieren zum Testen

Im ersten Teil der Open-Source-Serie haben wir den BPEL-Designer und Apache ODE als BPEL Engine kennen gelernt. Doch was kommt, wenn im Designer alles gut anzusehen ist, es aber in der Ausführung hakt? Wie jede andere Software muss auch der BPEL-Prozess getestet werden. Dies verleiht nicht nur dem Kunden, sondern auch Ihnen als „Komponisten" das gute Gefühl, dass die Services immer ein stimmiges Konzert geben werden. Dazu werden wir in diesem Teil das Open-Source-Werkzeug BPELUnit vorstellen und erste praktische Erfahrungen damit sammeln.

von Daniel Lübke, Simon Moser und Tammo van Lessen

Im ersten Teil haben wir den Prozess der Führerscheinprüfung als BPEL-Orchestrierung mit Web Services umgesetzt. Zum Ende konnten wir mit dem Web Service Explorer von Eclipse den Prozess ausführen. Wenn Sie sich selbst an dem Beispiel versucht haben, werden Sie bemerkt haben, dass man viele kleine Fehler im XML und XPath machen kann. Zur Erinnerung: Ein Kandidat kann sich zu einer Führerscheinprüfung anmelden. Der Prüfer kann im Folgenden das Prüfungsergebnis melden und bei bestandener Prüfung wird der Prüfling informiert und sein Führerschein wird ausgedruckt. Der Prozess ist noch einmal in Abbildung 1 dargestellt.

Stellen Sie sich vor, dass nun bundesweit Führerscheine mit Ihrer Implementierung ausgestellt werden sollen. Könnten Sie gut schlafen, wenn Sie mal kurz mit dem Web Service Explorer den Prozess „getestet" hätten, oder müssten Sie mit unliebsamen Anrufen rechnen, weil die Software nicht funktioniert?

Für BPEL-Prozesse gilt wie für andere Software auch, dass man sie vor der Auslieferung testen sollte. Im Gegensatz zum einfachen Ausführen mit dem Web Service Explorer bedeutet Testen, dass versucht wird, möglichst alle Fälle in dem Prozess einmal auszuführen und zu überprüfen, ob der Prozess sich korrekt verhält und die richtigen Ergebnisse liefert.

„Nur die Streicher bitte" – die Testfälle

Um unseren BPEL-Prozess zu testen, müssen zunächst einmal Testfälle definiert werden. Ein Testfall beschreibt, in welcher Konfiguration die Software mit welchen Daten getestet wird, und welches Verhalten und welche Rückgaben sie liefern soll. Die Konfiguration beschreibt in unserem Fall, auf welchem Server der Prozess ausgeführt wird, also z. B. Apache ODE 1.3.2 in einem Apache Tomcat 6.0. Am besten spiegelt die Testumgebung dabei natürlich die Umgebung wieder, in der die Software später verwendet werden soll. Die Definition der Eingabe- und der erwarteten Ausgabedaten erscheint dagegen wieder offensichtlich. Diese werden dann in Testfällen dokumentiert.

So können in einem Testfall „Erfolgreiche Führerscheinprüfung beim 1. Versuch" die Daten definiert werden, mit denen sich der Kandidat anmeldet, und die Daten, mit denen der Fahrprüfer die erfolgreiche Prüfung bekanntgibt. Zudem muss der Testfall definieren, welche Daten der Prozess im Gegenzug senden soll, d. h. die SMS muss gesendet und der Führerschein ausgedruckt werden etc. Dieser Testfall ist ein möglicher, aber ist er ausreichend? Sicherlich fehlt noch

Artikelserie

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

Teil 2: Unit Testing von Prozessen

Teil 3: Fehlerszenarien und Regression Testing

Mit Screencast auf CD

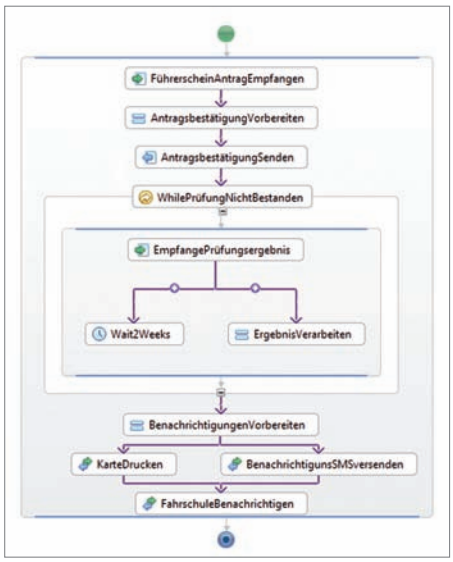


Abb. 1: Der Prozess

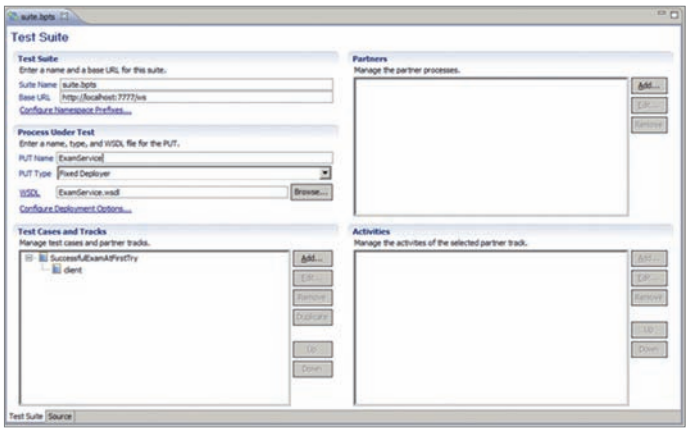


Abb. 2: Der Testsuiteeditor von BPELUnit

ein Testfall für einen nicht bestandenen Führerschein. Ziel beim Testen muss es offensichtlich sein, alle möglichen Szenarien durchzuspielen. Es gibt verschiedene Methoden aus dem Bereich des Softwaretests, mit denen man möglichst gute Testfälle ermitteln kann. Da eine vollständige Beschreibung dieser Methoden den Rahmen dieses Artikels sprengen würde, sei z. B. auf das Buch „Abenteuer Softwarequalität" verwiesen [1].

So könnten wir unseren Prozess mit zwei Testfällen „Bestehen beim ersten Versuch" und „Bestehen beim zweiten Versuch" testen. Diese beiden Testfälle führen zusammen alle Pfade unserer Implementierung der Führerscheinprüfung aus. Für die Testfälle müssen nun Testeingaben und -ausgaben definiert werden.

Dabei stellt sich die Frage, wie der Prozess am besten getestet werden kann. Eine Möglichkeit wäre es, den Web Service Explorer zu nehmen und alle Testfälle nacheinander auszuprobieren. Das ist zwar möglich, aber auch recht zeitaufwändig. Schöner wäre es, wenn wir den Test automatisieren könnten, damit unser Prozess schnell und bei Änderungen immer wieder problemlos von vorne getestet werden kann.

BPELUnit als Dirigent einer Registerprobe

Für die Testautomatisierung gibt es viele Werkzeuge. In diesem Artikel werden wir dazu das Open-Source-Tool BPELUnit [2] benutzen, ein Tool zum (Unit-)Testen von BPEL-Prozessen, das es erlaubt, die Nachrichten, die mit dem Prozess ausgetauscht werden, zu definieren und immer wieder durchzuspielen. Dabei überprüft BPELUnit auch die Sollwerte für empfangene Nachrichten und kann somit verifizieren, ob der Prozess sich immer noch konform zu den Tests verhält. BPELUnit ist also wie ein Dirigent, der in den Proben immer wieder überprüft, ob alles richtig klingt.

BPELUnit unterstützt verschiedene BPEL-Server und kann in verschiedene Entwicklungstools wie Ant und Eclipse integriert werden. So wie der Eclipse-BPEL-Designer kann BPELUnit einfach über eine Updatesite installiert werden. Die Updatesite finden Sie unter http://update.bpelunit.net. Nach erfolgreicher Installation ist BPELUnit auf ähnliche Art und Weise wie JUnit in Eclipse integriert. Es gibt ein Fenster und ebenfalls den berühmten rot-grünen Balken. Zudem gibt es Editoren, um Testfälle zu erstellen.

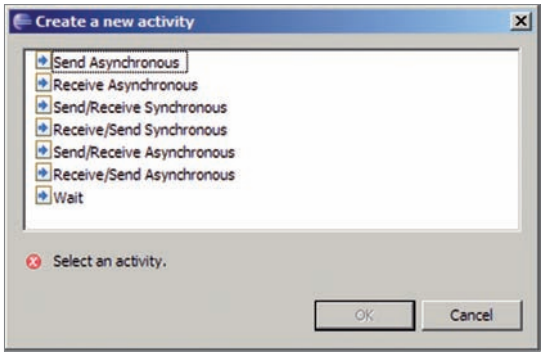


Abb. 3: Verfügbare Testaktivitäten

Wir erstellen nun den ersten Testfall. Dazu erstellen wir zunächst über File | New | Other | BPEL Unit | BPELUnit Test Suite eine Testsuite. Eine Testsuite ist ein Container für beliebig viele Testfälle, die denselben BPEL-Prozess testen. Wir konfigurieren die Testsuite nun wie in Abbildung 2 gezeigt. Hierbei ist aktuell besonders die WSDL-Datei zu beachten, über die der Service und der verwendete Server angesprochen werden können. BPELUnit bietet über die Deployer-Einstellungen die Möglichkeit, den Prozess automatisch auf verschiedene Server zu deployen oder aber über den „Fixed Deployer" einen bereits deployten Prozess zu testen. Letzteres ist insbesondere praktisch, wenn ein BPEL-Server verwendet wird, der von BPELUnit noch nicht unterstützt wird oder wenn – wie in unserem Fall – das Deployment nicht von BPELUnit, sondern von Eclipse verwaltet werden soll. Im unteren Bereich des Editors finden sich die Bereiche zur Definition der Testfälle. Um nun den ersten Testfall zu erstellen, wählen wir Add im Bereich Test Cases and Tracks und erstellen einen Testfall *SuccessfulExamAtFirstTry*. BPELUnit erstellt daraufhin den Testfall und einen Partner Track *client*. Partner Tracks entsprechen Abläufen zwischen dem Prozess und den Partner Links des Prozesses. BPELUnit erstellt per Default den *client* Partner Track, der dafür zuständig ist, den BPEL-Prozess aufzurufen, also die Rolle des primären Servicekonsumenten übernimmt. Um nun eine Testinteraktion mit

dem Prozess zu erstellen, markieren wir den *client* Partner Track und wählen danach ADD im Bereich ACTIVITIES.

Wir werden nun gefragt, welches Interaktionsmuster wir mit dem Prozess erstellen wollen (Abb. 3). Unser Prozess wird synchron aufgerufen und liefert eine Antwort, d. h. wir wählen SEND/RECEIVE SYNCHRONOUS. Im darauf folgenden Assistenten können wir nun die Nachrichten beschreiben, die mit dem Prozess ausgetauscht werden sollen. Wir wollen dabei folgende Nachricht an die Operation *apply* schicken:

```
<exam:Application>
<firstname>Donald</firstname>
<lastname>Duck</lastname>
<idnumber>DDuck123</idnumber>
<telnumber>0049111111111</telnumber>
<birthdate>1900-01-01</birthdate>
</exam:Application>
```

In der darauffolgenden Seite können wir Bedingungen für die Antwortnachricht definieren. Wir möchten, dass der Prozess im Element *processingnumber* unsere Vorgangsnummer *DDuck123* zurück gibt und dass der Antrag erfolgreich erstellt worden ist. Dazu definieren wir folgende Bedingungen:

```
Condition: //processingnumber Value: 'DDuck123'
Condition: //status Value: 'true'
```

Dabei ist zu beachten, dass der Wert selbst ein XPath-Ausdruck ist, d. h. dass z. B. Strings in einfache Anführungszeichen gesetzt werden müssen.

Somit haben wir die erste Nachricht zum Prozess definiert. Als Nächstes muss der Prüfer melden, dass die Prüfung bestanden wurde. Dazu erstellen wir wieder eine neue Aktivität im Partner Track *client*. Diesmal erwarten wir aber keine Antwort, d. h. der Typ ist *Send Asynchronous*. Wir senden einfach folgende Nachricht:

```
<exam:ExamResult>
  <processingnumber>DDuck123</processingnumber>
  <passed>true</passed>
</exam:ExamResult>
```

Nun haben wir einen ersten Testfall erstellt und können das erste Mal BPELUnit starten. Dazu wählen wir im Kontextmenü der Datei RUN AS... | BPELUNIT TEST SUITE. Wenn der BPEL-Prozess auf einem Server läuft, sollte der Test erfolgreich ausgeführt werden. Dabei erscheint ein Fenster, das analog zum JUnit-Fenster aufgebaut ist (Abb. 4).

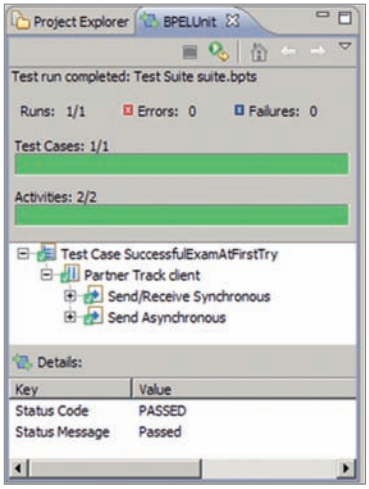
Ist der Prozess und der Testfall korrekt, so erscheint ein grüner Balken, ansonsten ein roter Balken. Der Test Case mit allen Partner Tracks wird im oberen Bereich hierarchisch dargestellt. Hier können weitere Details wie SOAP-Nachrichten ausgewählt werden, die dann im unteren Bereich angezeigt werden. Auf dieselbe Art und Weise kann nun ein weiterer Testfall erstellt werden, bei dem erst beim zweiten Anlauf die Prüfung bestanden wurde.

Dabei fallen nun zwei Probleme auf: Zum einen dauert der Testfall für das Bestehen im zweiten Versuch zwei Wochen für die Ausführung. Daher sollte man für das Testen die Dauer der *wait*-Aktivität auf ein paar Sekunden verkürzen. Zum anderen kann man in der Führerscheinprüfung nicht endgültig durchfallen. Wir könnten zwar einen Testfall erstellen, bei dem ein Kandidat dreimal durch die Prüfung fällt, aber das hat keine Konsequenzen.

Begleitautomatik – Mocking für Web Services

Wie gut ist nun unser Testfall? Der grüne Balken täuscht ein wenig, denn wir haben bisher nicht überprüft, ob eine SMS gesendet oder der Führerschein wirklich ausgedruckt worden ist. Dazu müsste jemand in einer echten Umgebung andauernd sein Handy bereithalten und öfters zum Drucker laufen. Dazu verursachen der SMS-Versand und das Druckmaterial Kosten. Der SMS- und der Druckservice sind scheinbar Dienste, die man nicht bei jedem Test nach jeder kleinen Änderung ausführen möchte. Solche Services gibt es häufig; sei es, weil der Service zu lange in der Ausführung braucht, kostenpflichtig ist oder weil man die Ergebnisse nicht automatisch überprüfen kann. In solchen Fällen sollte man die echten Services durch Dummyimplementierungen ersetzen. Diese nennt man auch Mocks und sind von anderen Programmiersprachen und deren Testtools wie JMock [3] bekannt. Durch den Einsatz von Mocks ist es möglich, den BPEL-Prozess komplett zu isolieren, sodass er keine externen Services mehr benötigt. Dadurch kann man beim Testen einige Probleme umgehen:

Abb. 4: Ausführen des Tests



Partner Track	Bedingung	Erwarteter Wert
FahrschulService	//ds:learnerId	'DDuck123'
SMSService	//sms:recipient	49111111111
PrintService	//prin:firstName	'Donald'
	//prin:lastName	'Duck'
	//prin:birthday	'1900-01-01'

Tabelle 1: Bedingungen für empfangene Nachrichten

Einstellung	Wert	Beschreibung
DeploymentArchive	.	Paketiert den Prozess im aktuellen Verzeichnis, d. h. das ganze Projektverzeichnis
ODEDeployment-ServiceURL	http://localhost:8080/ode/processes/DeploymentService	Pfad zum ODE Deployment Service (typischerweise auf localhost)

Tabelle 2: Deployer-Optionen

- kostenpflichtige Services müssen nicht aufgerufen werden
- langsame Services müssen nicht aufgerufen werden
- noch nicht fertig entwickelte Services behindern nicht die Entwicklung und den Test des BPEL-Prozesses
- Fehlerbedingungen wie SOAP Faults können leicht simuliert werden

Um Services mit BPELUnit zu mocken, muss man lediglich angeben, welche Nachrichten ein Service erhalten hat und was er darauf antworten soll. Der Serviceaufruf des BPEL-Prozesses muss dazu auf den simulierten und durch BPELUnit bereitgestellten Service umgeleitet werden.

Um das auszuprobieren, erstellen wir eine neue Testsuite mit dem Namen *suite-unittest.bpts* und machen dieselben Einstellungen wie zuvor. Jedoch erstellen wir vor der Testfalldefinition neue Partner Tracks im rechten oberen Editorbereich. Dafür erstellen wir für jeden der drei Partner Links *FahrschulService*, *SMSService* und *PrintService* einen neuen Partner, der genau den Namen des Partner Links hat und auf die entsprechende WSDL verweist.

Erstellen wir nun einen neuen Test Case, so wird nicht mehr nur ein Partner Track für den Client erstellt, sondern auch für jeden anderen Partner. Der Client Partner Track ändert sich im Vergleich zum ersten Test erst einmal nicht. Dazu kommen nun allerdings Überprüfungen, dass der Prozess auch wirklich alle Partner nach erfolgreicher Prüfung aufruft. Dazu wird nun in jedem Partner Track eine *Receive-Asynchronous*-Aktivität hinzugefügt. Hier können Bedingungen für jede empfangene Nachricht definiert werden. Diese sehen wie in Tabelle 1 aus.

Wird nun der Test gestartet, bietet BPELUnit die Services unterhalb dem angegebenen Base URL ein. So ist der Mock für den Fahrschulservice unter http://localhost:7777/ws/FahrschulService zu erreichen. Damit der Test auch wartet, bis der BPEL-Prozess alle Services aufgerufen hat, muss noch eine *wait*-Aktivität in den *client*-Track hinzugefügt werden. Der Test soll für diesen einfachen Prozess 1500 ms warten.

Nun muss der BPEL-Prozess noch so konfiguriert werden, dass er nicht die ursprünglichen Services, sondern die Mocks aufruft. Dazu gibt es zwei Möglichkeiten: Entweder werden die Endpoint-Einstellungen im Deployment geändert, d. h. im Fall von Apache ODE müssen die Endpunkte in den WSDLs geändert werden (d. h. die *deploy.xml* aus Teil 1 müsste auf die neuen URLs wie http://localhost:7777/ws/FahrschulService angepasst werden) oder aber BPELUnit tut das transparent. Momentan unterstützt BPELUnit das Ersetzen der Endpoints nur mittels des Apache ODE Deployers. Dazu darf der Prozess nicht mehr mittels Eclipse deployt werden, sondern das muss mittels BPELUnit geschehen. Dafür muss der Tomcat mit ODE separat gestartet sein und auf der Optionsseite für das BPELUnit Deployment in den Eclipse-Einstellungen die Option *Automatically modify endpoints to simulated URLs* aktiviert sein. Zudem muss der Deployer in der Testsuite auf *ODE Deployer* gestellt werden und die Deployer-Optionen (Tabelle 2) korrekt gesetzt sein.

Wird nun die Testsuite wie zuvor gestartet, ersetzt BPELUnit die Endpoints in den WSDLs transparent durch die Mock-Endpoints und deployt den Service neu, bevor der eigentliche Testlauf beginnt. Somit sind wir nun in der Lage, den Führerscheinprozess unabhängig von den verwendeten Services zu testen.

Die Tuttiprobe

Mit der Möglichkeit, BPEL-Prozesse einzeln und isoliert oder vollständig mit allen verwendeten Services zu testen, gibt es nun die Möglichkeit, mit dem gesamten Orchester eine „Generalprobe" durchzuführen, bevor die Software ausgeliefert wird und produktiv geht. BPELUnit bietet dafür alle Grundfunktionen. Wenn Sie weitere Informationen brauchen, können Sie sich auf der BPELUnit-Homepage [2] oder der Mailingliste erkundigen.

In Teil 3 werden wir die erkannten Probleme in unserem Prozessmodell beseitigen und um Fehlerbehandlung erweitern. Glücklicherweise können wir hier auf unsere Testsuite für Regressionstests zurückgreifen. ■

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.

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.

Links & Literatur

[1] Schneider, Kurt: „Abenteuer Softwarequalität", dpunkt Verlag 2007

[2] BPELUnit, BPELUnit – The Open Source Unit Testing Framework for BPEL: http://www.bpelunit.net

[3] JMock, A Lightweight Mock Object Library for Java: http://www.jmock.org/