AUF NUMMER SICHER: TESTEN VON AUSFÜHRBAREN GESCHÄFTSPROZESSEN

der autor

Dr.-Ing. Daniel Lübke
(daniel.luebke@innoq.com)
arbeitet zurzeit bei der innoQ Schweiz GmbH als Senior Consultant in SOA- und MDA-Projekten. Zudem ist er Maintainer des Open-Source-Test-Frameworks BPELUnit.

Neue Projekte im Geschäftsprozess-Umfeld stehen häufig vor der Herausforderung, Prozesse zu automatisieren. Dabei müssen Prozesse so formal beschrieben werden, dass ein Computer sie, wie andere Software auch, interpretieren und ausführen kann. Dieses High-Level-Programming verlangt aber auch die Qualitätssicherung, insbesondere von kritischen Prozessteilen. In diesem Artikel werden Möglichkeiten für das Testen der verschiedenen Komponenten von ausführbaren Geschäftsprozessen (Prozess, Datentransformationen) und für das Review (Prozess, Servicedefinitionen) sowie Erfahrungen aus einem Projekt vorgestellt.

Zunächst mit BPEL und nun verstärkt durch die Entwicklungen rund um BPMN2 werden mehr und mehr Geschäftsprozesse automatisiert. Dazu werden Geschäftsprozesse ausführbar modelliert und in einem BPMS zentral ausgeführt und überwacht. Dabei werden typischerweise Web-Services der Geschäftssysteme aufgerufen, um Geschäftsfunktionen auszulösen.

Anfangs stark mit dem Argument verkauft, dass die Fachseite durch die graphische Modellierung in die Lage versetzt würde, selbst Geschäftsprozesse automatisieren zu können, sind die Entwicklung und die Verantwortung für die ausführbaren Geschäftsprozessmodelle schnell zurück in die IT-Organisation gekommen. Letztendlich sind auch graphische, ausführbare BPMN2-Modelle nichts anderes als ein Programm – auch wenn sie in einer für eine spezielle Domäne entwickelten Sprache programmiert wurden. Daher sind alle Praktiken und Methoden aus dem Software-Engineering mit leichten Modifikationen auch in BPEL/BPMN2-Projekten von Vorteil. In diesem Artikel werden Techniken und Methoden rund um die Qualitätssicherung mit besonderem Fokus auf dem Thema Testen vorgestellt.

Qualitätssicherung für automatisierte Geschäftsprozesse ist insofern besonders wichtig, als dass Geschäftsprozesse die Wertschöpfung eines Unternehmens regeln. Sie sind also geschäftskritisch und damit sind Fehler, die dort auftreten, besonders geschäftsschädigend.

Verschiedene Testebenen finden verschiedene Fehler

In klassischen Softwareentwicklungsprojekten gibt es verschiedene Ebenen, auf denen getestet wird: Unit-Tests, Integrationstests und Systemtests. Diese Ebenen versuchen, verschiedene Fehlertypen zu finden. Ein Systemtest kann daher niemals ein enges Netz von Unit-Tests ersetzen. Auch sind die Verantwortlichkeiten jeweils verschieden und die Verfügbarkeit der Umsysteme in den verschiedenen Teststufen ist nicht immer gegeben.

Unit-Tests

Unit-Tests sind immer Aufgabe der Entwickler. Sie sollten gleichzeitig mit der Software – in unserem Fall also den Prozessen – entwickelt werden. Dabei ist es egal, ob die Tests vor dem eigentlichen Prozess (*Test First*) oder danach geschrieben werden. Jedoch sind diese Tests ein sehr hilfreiches und teilweise auch das einzige zuverlässige Hilfsmittel, mit dem ein Entwickler seinen Fortschritt und die Funktionalität der Prozesse bewerten kann, weil diese typischerweise keine Oberflächen direkt bereitstellen oder weil die Oberflächen-Interaktionen über verschiedene Dialoge und Rollen verteilt sind, sodass ein manuelles Ausprobieren in der Software viel zu lange dauert.

In Unit-Tests für den BPMN- bzw. BPEL-Teil eines Geschäftsprozesses wird der Kontrollfluss getestet. Hierbei geht es darum, alle Prozesspfade durch verschiedene Testfälle auszuführen. Dabei sind allerdings die umgebenden Services, die für die Prozessausführung nötig sind, in der Regel noch nicht entwickelt oder stehen gerade früh im Entwicklungszyklus noch nicht zur Verfügung. Daher müssen diese durch Mocks ersetzt (vgl. [Cha02]) werden, d. h. durch simulierte Services ersetzt werden. Das ist insbesondere bei Web-Services gut durchführbar, da diese eine WSDL-Schnittstelle besitzen, die die Kommunikation zwischen dem Prozess und den Partnersystemen definiert. Hier können Frameworks wie „soapUI" (vgl. [soa12]) oder „BPELUnit" (vgl. [BPE12]) den Entwicklern viel Arbeit abnehmen. Teilweise liefern die BPMS-Hersteller selbst Unit-Test-Werkzeuge als Teil der Entwicklungsumgebung mit, die diese Funktionen ebenfalls übernehmen. **Abbildung 1** zeigt hier exemplarisch die BPELUnit-Integration in Eclipse. Mittels dieser Tools wird dann als

BPMN	Business Process Model and Notation
BPEL	Business Process Execution Language
BPMS	Business Process Management System
ERP	Enterprise Resource Planning
JAX-WS	Java API for XML Web Services
QA	Qualitätssicherung
WSDL	Web Service Definition Language
WS-HT	WS-HumanTask
URN-Mappings	Uniform Resource Name Mappings
UUID	Universally Unique Identifier
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformations

Kasten 1: Glossar.

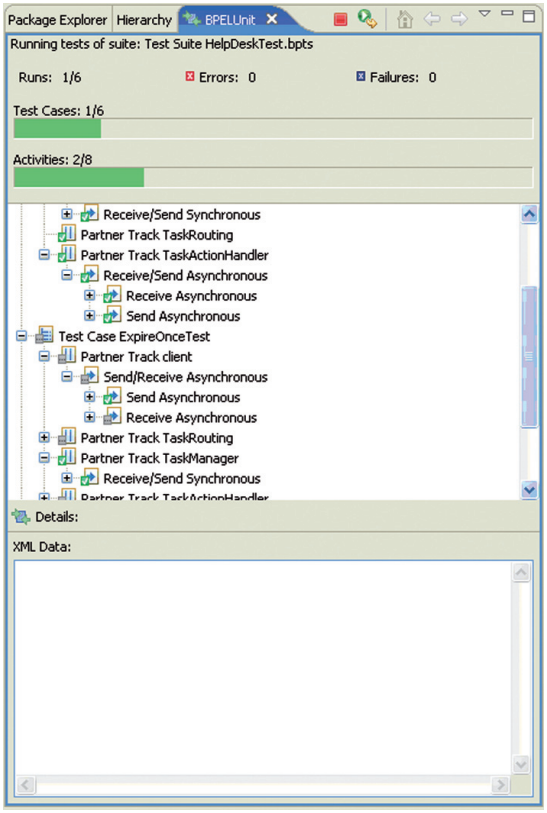

Abb. 1: *Gute Test-Tools lassen sich in die Entwicklungsumgebung integrieren (Screenshot von [BPE12]).*

Unit-Test die Kommunikation zwischen den Prozess- und den Umsystemen beschrieben. Aufgrund der gesendeten SOAP-Nachrichten durchläuft der Prozess verschiedene Prozesspfade und versendet unterschiedliche Nachrichten an die Service-Mocks. Auf Basis der empfangenden Nachrichten können Bedingungen (*Assertions*) erstellt werden, die überprüfen, ob die SOAP-Nachrichten die erwünschten Informationen beinhalten.

So wie für andere Programmiersprachen gibt es auch für BPEL und BPMN2 Testabdeckungsmetriken. Diese messen, wie viele Aktivitäten und Pfade von den Testfällen ausgeführt wurden. Allerdings sind diese Metriken typischerweise viel höher als in den klassischen Programmiersprachen, weil allein schon ein Testfall, der einen Prozess von Anfang bis Ende ausführt, typischerweise eine Überdeckung von 60 bis 80 % erreicht. Ein großer Nachteil dieser Metriken ist, dass sie nicht feststellen können, ob der Datenfluss überprüft wird. Auch wenn ein Testfall keine *Assertions* definiert, werden die ausgeführten Aktivitäten gezählt. Ein solcher Test würde zwar immerhin zeigen, ob die gesendeten SOAP-Nachrichten XML-Schema-konform sind, aber er könnte keinerlei Aussage über die fachliche Richtigkeit der Daten treffen. Darum sind gerade *Assertions* wichtig, denn diese zeigen in einem Testfall erst an, ob ein Service später mit diesen Nachrichten umgehen kann. Gerade bei komplexen Nachrichten lohnt es sich sogar, Referenznachrichten mit Experten für den aufgerufenen Service zu definieren und diese dann später im Test direkt als Sollwert zu verwenden.

Typischerweise beinhalten Prozesse auch komplexe Datentransformationen. Das gilt umso mehr, wenn das Domänenmodell komplexer ist. Zwar könnten die Datentransformationen auch als Teil des Prozesses getestet werden, indem ein kompletter Prozess für die jeweilige Kombination der Eingabedaten ausgeführt wird. Das ist allerdings sehr ineffizient, weil die Ausführung in der Regel lange dauert. Hier empfiehlt es sich dann, die Datentransformationen separat zu testen. In BPEL sind diese typischerweise als XSLT realisiert. Der BPMN2-Standard erlaubt es dagegen den Herstellern, dieses Feld weiter auszugestalten. So können auch Java-Mappings etc. verwendet werden, um die Datentransformationen zu entwickeln. Hier empfehlen sich separate JUnit-Test-Suiten, die im Fall von XSLT auch einen XSLT-Interpreter aufrufen und die Ergebnisse mit Sollwerten vergleichen können. So ist es möglich, die Datentransformationen separat und von der Ausführungsdauer viel schneller zu testen. Neben Datentransformationen gibt es anderen Code, der ebenfalls sehr eng an den Prozess gekoppelt ist, wie z. B. eigene XPath-Funktionen. Auch für diese sollten separate (J)Unit-Test-Suites erstellt werden, die die Unit-Tests für den eigentlichen Prozess entlasten.

Eine mögliche Struktur der Tests zeigt **Abbildung 2**. Blau ist das eigentliche Prozess-Deployment dargestellt, das den ausführbaren Prozess sowie die WSDLs, XSLT-Stylesheets und ähnliches beinhaltet. Violett sind die BPELUnit-Artefakte dargestellt. Während ein Test läuft, übernimmt BPELUnit die Rolle aller umliegenden Systeme und interagiert mittels SOAP-Nachrichten mit dem Prozess. *Assertions* überprüfen dabei den Inhalt der empfangenden Nachrichten. Grün dargestellt sind die JUnit-Tests, die alle nötigen Eingaben an die XSLT-Stylesheets schicken und die generierten Datenstrukturen mit den Solldaten vergleichen.

Wie in anderen Softwareprojekten gibt es auch in BPMN2/BPEL-Projekten Unterschiede in der Testbarkeit der Software. So sollte der Prozessfluss lediglich von den an den Prozess gesendeten SOAP-Nachrichten abhängen. Ist das nicht der Fall, weil z. B. eine Java-Klasse direkt aufgerufen wird, die auf die Datenbank zugreift, oder weil eine eigene XPath-Funktion Zufallszahlen generiert, wird das Testen viel aufwändiger und zuweilen sogar unmöglich. Genauso sollte von der Möglichkeit Gebrauch gemacht werden, Datentransformationen in eine eigenen Datei oder Klasse auszulagern. Dies erlaubt das separate Testen, wie oben beschrieben. Ebenso können in einigen BPMS beim Deployment die Endpunkte der aufgerufenen Services durch Platzhalter definiert werden (z. B. in „ActiveVOS" durch URN-Mappings, vgl. [XML12]), die erst auf dem Server in konkrete Endpunkte aufgelöst werden. Durch diesen Mechanismus entfällt das neue Paketieren der Deployment-Einheiten beim Wechsel zwischen verschiedenen Testumgebungen und/oder der Produktionsumgebung.

Ein wichtiges Merkmal der Unit-Tests ist ihre vollständig automatisierte Ausführung. Dabei können die Frameworks (wie BPELUnit) sogar das Deployment selbst vornehmen, sodass der Prozessdesigner

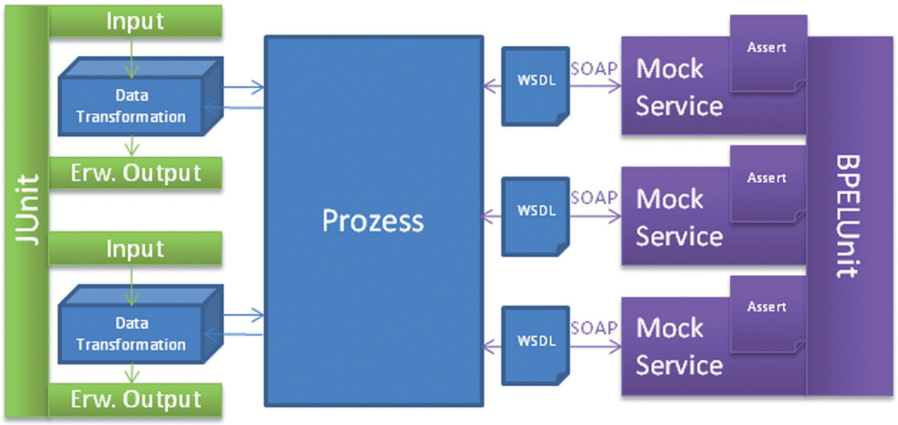

Abb. 2: Verschiedene Test-Frameworks für verschiedene Funktionalitäten.

lediglich den Test starten muss und nach einiger Zeit hoffentlich den beliebten grünen Balken zu sehen bekommt. Sind die Test-Suites so aufgesetzt, ergänzen und ersetzen sie viele Einsatzzwecke der in den Designern enthaltenen Simulationswerkzeuge. Zusätzlich werden bei den Testfällen auch die Serverumgebung inklusive der Konfiguration und anderen installierten Softwarekomponenten getestet.

Die Automatisierung der Tests ist dank der Entkoppelung über SOAP-Nachrichten in der Regel sehr einfach und zahlt sich insbesondere bei Änderungen aus. So werden Prozesse ständig angepasst – sei es, weil sich das Geschäft ändert, weil eine Firma umstrukturiert wird oder aus anderen Gründen. In solchen Fällen ist es gut, wenn es über die Testfälle einen Mechanismus gibt, mit dem geprüft werden kann, ob eine kleine Änderung irgendwo im Prozess nicht ungewollte Nebenwirkungen hat. Auch können dann Prozesse in *Continuous-Integration*-Umgebungen gebaut, getestet und ausgeliefert werden.

Allerdings gibt es auch Aspekte, die sich mit Unit-Tests nur schwer testen lassen. Hierunter fallen insbesondere zeitliche Abhängigkeiten. So gibt es in den Prozessen oft definierte Wartepausen und *Time-Outs*, die meistens mehrere Tage oder gar Wochen betragen, z. B. wenn eine Rechnung nach vier Wochen angemahnt wird. In einem Unit-Test sind derart lange Wartezeiten nicht akzeptabel. Deswegen werden solche Aspekte leider oft nicht getestet. Man kann allerdings den Prozess konfigurierbar machen, sodass die Wartezeit im Test heruntergesetzt wird. Allerdings wird dann ein Prozess getestet, der so später nicht fachlich korrekt wäre. Somit muss später sichergestellt sein, dass die Konfiguration im produktiven System korrekt ist.

Ein anderes „Problem" stellen die menschlichen Anwender dar. Über „BPEL4People" oder „People Tasks" können auch menschliche Akteure in den Prozess eingebunden werden. Dabei werden so genannte *Tasks* erzeugt, die dann in einem Portal oder einer anderen Anwendung bearbeitet werden können. Die meisten BPMS bieten dazu eine WS-HT-Schnittstelle an. Über diese können auch die *Tasks* bearbeitet und abgeschlossen werden, ohne dass dazu eine andere Anwendung nötig wäre. Die aktuelle Entwicklungsversion von BPELUnit bietet z. B. WS-HT-Unterstützung an, aber auch die Ansteuerung über JAX-WS in (J)Unit-Tests stellt kein Problem dar.

Gar nicht möglich sind dagegen Tests, die Aussagen über die meisten nicht-funktionalen Eigenschaften – wie beispielsweise Performance oder die Anzahl gleichzeitiger Prozessinstanzen – machen, da diese stark von den darunter liegenden Softwareschichten und ihrer Konfiguration (z. B. Datenbank, JVM, Betriebssystem) sowie der Hardware (z. B. Arbeitsspeicher, CPUs, Anzahl der Server, Netzwerk) abhängen. Diese können erst später in der Zielumgebung oder einer dieser ähnlichen Umgebung getestet werden.

Integrationstests

Ist der Prozess in einem ausführbaren Zustand, so kann er mit den anderen Umsystemen integriert werden. Dazu werden Integrationstests in einer eigenen Integrationsumgebung, in der gezielt die Kommunikation zwischen den einzelnen Serviceanbietern und -konsumenten getestet wird, durchgeführt. Dabei sollte sichergestellt sein, dass nur Komponenten in den Integrationstest kommen, für die es bereits Unit-Tests gibt. Werden haufenweise funktionale Fehler in Integrationstests gefunden, steigt der Aufwand beträchtlich und zehrt vor allem an den Nerven der Entwickler der Gegenpartei, was im weiteren Projektverlauf oftmals weitere (soziale und organisatorische) Probleme nach sich zieht.

In die Integrationstests kann der Prozess mit den anderen Komponenten integriert werden. Hierbei ist es oftmals nötig, andere umgebende Services weiterhin durch Mocks ersetzent zu lassen. Gerade in Integrationsszenarien, in denen verschiedene Partnersysteme aus unterschiedlichen Abteilungen oder Unternehmen angesprochen werden, ist dies nötig. Arbeit kann man sich sparen, wenn man dazu die Testdaten aus den Unit-Tests wiederverwendet. Dazu sind dann die Integrationstests so zu definieren, dass z. B. in einem Portal die Werte eingegeben werden, mit denen man im Unit-Test vorher den Prozess gestartet hat, oder dass im Unit-Test die Mocks bereits die Daten entsprechend den Testdaten-Beständen im ERP-System zurückliefern.

Nach Möglichkeit sollten hier die Entwickler schon versuchen, die zeitabhängigen Tests, die in den Unit-Tests noch nicht zufriedenstellend vollzogen werden konnten, hier nachzuholen. Das ist nicht immer möglich, da auch Integrationstests nicht Wochen dauern sollen. Jedoch hat man eine eigene Serverumgebung, die unabhängig von der Entwicklungsumgebung läuft, und kann dort Prozesse starten und dann später nachsehen, ob alles richtig funktioniert hat. Funktioniert die Kommunikation zwischen den zu integrierenden Teilen, sollten die Deployment-Einheiten eingefroren und nicht mehr geändert werden. Am besten kommen nur Deployments aus einem zentralen Build, die auch eindeutig über eine Build-Nummer oder ein *Tag* in der Versionskontrolle identifiziert werden können, in die Integrationsumgebung. So weiß man genau, welche Entwicklungsstände getestet wurden. Diese können nach erfolgreichem Test dann weiter in den Systemtest geschoben werden.

Systemtests

Im letzten Schritt wird das System als ganzes getestet. Hierbei sollte die Testumgebung möglichst der produktiven Umgebung entsprechen. Spätestens hier

sollten dedizierte Tester und nicht mehr die Entwickler und Prozessdesigner die Tests durchführen, da sie einen anderen Blickwinkel auf das System haben und nicht während des Projekts betriebsblind geworden sind. Die Tester arbeiten auch strikt gegen die ursprünglich definierten Anforderungen und finden so falsch oder nicht implementierte Features, die typischerweise von den Entwicklern in den Unit-Tests nicht gefunden worden sind.

Notwendigerweise sollten die Tester alle Abnahmetestfälle hier bereits einmal durchführen, bevor sie das bei der endgültigen Abnahme tun. Außerdem können zum ersten Mal Last-, Performance- und Sicherheitstests durchgeführt werden, weil hier eine finale und fertig konfigurierte Testumgebung zur Verfügung steht, die der Produktivumgebung ähnlich ist. Die Testumgebung sollte auch getrennt von der Integrationsumgebung sein, damit sich Entwickler/Prozessdesigner und Tester nicht gegenseitig behindern.

Die Tester bzw. die Testwerkzeuge interagieren nur mit den für die späteren Akteuren zur Verfügung gestellten Schnittstellen, wie z. B. Portale oder Systemschnittstellen für externe Systeme.

Reviews

Neben Tests gibt es noch Reviews, um die Qualität der Prozesse zu gewährleisten. In Reviews können alle Aspekte eines Prozesses durch ein Review-Team überprüft werden, die schlecht oder gar nicht zu testen sind.

Hierzu gehört einerseits vor allem die Einhaltung der Modellierungsrichtlinien, um die spätere Weiterentwicklung sicherzustellen und die Vergleichbarkeit und Konsistenz der verschiedenen Geschäftsprozessautomatisierungsprojekte untereinander zu gewährleisten. Andererseits fallen auch technische Aspekte darunter, wie zum Beispiel:

- Die korrekte Definition der *Correlation Sets*, da inkorrekte Definitionen manchmal nicht in den Tests gefunden werden.
- Die korrekte Konfiguration in den Deployment-Deskriptoren, wie z. B. Prozess-Persistenz für Auswertung, Reporting und manuelle Fehlerbehandlung.
- Das Review des Fehlermanagements, z. B. *Exception Flows* bzw. *Fault Handler*.

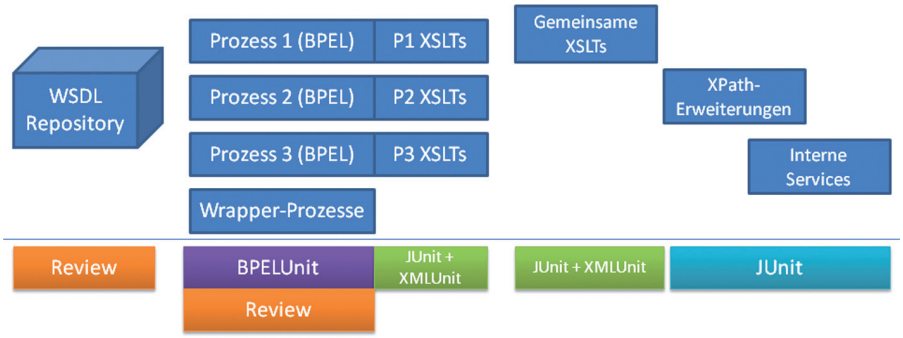

Abb. 3: Projekt- und Teststruktur.

- Die korrekte Definition von Zeitspannen und Timeouts, sofern nicht getestet.

Je nach Projekt und Organisation werden diese Punkte durch andere Aspekte ergänzt.

Erfahrungen aus einem Projekt

Um diese abstrakte Beschreibung ein wenig mit Leben zu füllen, stelle ich im Folgenden ein reales annonymisiertes Projekt vor, das viele Probleme und Entscheidungen beinhaltet.

Das Projekt bestand aus mehreren Prozessen, die alle mit den gleichen Service-Definitionen arbeiteten, dabei aber unterschiedliche fachliche Abläufe implementierten. Die Prozesse hatten dabei eine unterschiedliche Komplexität: Einige waren sehr linear – mit Ausnahme von Bewilligungen, die aber direkt zum Prozessende führten –, ein anderer war aber sehr komplex und hatte alleine fünf Hauptzweige. Die Projektstruktur ist in Abbildung 3 illustriert.

Gewisse Bewilligungslogiken und Benachrichtigungen an umliegende Partnersysteme waren ebenfalls gleich und die nötigen Services dazu in den gemeinsamen WSDLs definiert. Da WSDLs nicht getestet werden konnten, wurden sie innerhalb eines definierten QA-Prozesses einem Review unterzogen. Da die Prozesse teilweise die gleichen Operationen aufgerufen haben, wurden zentral in XSLTs die entsprechenden Transformationen definiert, die die entsprechenden SOAP-Nachrichten erzeugen. Alle XSLTs wurden über JUnit- und XM-Unit-Tests (vgl. [XML12]) getestet. Neben den wiederkehrenden Geschäftsbenachrichtigungen wurden ebenfalls technische XSLTs für kundenspezifische Header etc. definiert. Die Tests wurden anhand des Ziel-XML-Schemas entwickelt. So musste z. B. für ein optionales Feld ein Testfall existieren, der dieses Feld setzt, und einer, der das Feld nicht setzt. Für Listen gab es zusätzlich noch einen Testfall mit mehreren Elementen.

Zum Testen entstanden hier ein paar Java-Hilfsklassen, die im Wesentlichen ein spezifiziertes XSLT mit gegebenen Eingabe-XML-Daten ausführten und das entstehende XML-Dokument mit einem Soll-Dokument verglichen. Diese Klassen wurden dann auch für die XSLTs, die prozessspezifisch entwickelt wurden, verwendet. Die so implementierten Tests sind in der Abbildung grün gekennzeichnet. Das Praktische an dieser Struktur war die Separierung der Testlogik von den Testdaten. Die Soll-XML-Dokumente konnten von den Serviceverantwortlichen einem Review unterzogen werden, ohne dass diese den Testcode verstehen mussten.

Neben den wiederverwendeten XSLTs gab es noch eine Bibliothek von XPath-Erweiterungen. Bis auf eine Funktion für die Generierung von UUIDs waren alle Funktionen zustandslos und deterministisch, sodass sie die Ausführung der Prozesse nicht beeinflussen konnten. Auch die Generierung der Geschäftsfall-Nummern wurde als eigener Service implementiert, da diese die Grundlage für die Nachrichtenkorrelation waren. Somit konnten die Unit-Tests die Schlüsselwerte über einen Service-Mock definieren und die Nachrichten mussten nicht mit viel Aufwand dynamisch an den Prozess angepasst werden. Die XPath-Erweiterungen wurden als Java-Klassen implementiert, die mit normalen JUnit-Tests getestet werden konnten.

Für das Testen des Kontrollflusses wurden BPELUnit-Test-Suites erstellt. Diese referenzierten jeweils einen Satz von SOAP-Nachrichten, die als externe Dateien im Projekt abgelegt waren. Diese Dateien waren entweder direkt aus den Spezifikationsbeispielen für die Services entnom-

men oder leicht abgeändert. Die Tests wurden als Teil eines Maven-Builds ausgeführt. Dieser Build kopierte die WSDLs und XSDs aus dem zentralen Repository genauso wie die wieder verwendeten XSLTs in das Prozessprojekt. Der Maven-Build konnte dann lokal auf den Entwicklerrechnern inklusive Deployment auf ein lokales BPMS oder als Teil des zentralen Builds ausgeführt werden. Somit war eine hohe Standardisierung des Builds und des Tests erreicht.

Neben den Hauptprozessen gab es noch kleine Wrapper-Prozesse, die die menschliche Interaktion, die über ein Portal stattfand, auf WS-HT abgebildet hat, sodass aus Prozesssicht ein einfacher Web-Service bereitstand. Auch für diese Wrapper-Prozesse wurden BPELUnit-Test-Suiten erstellt. In diesem Zuge zeigte sich der Vorteil von Open-Source, weil die WS-HT-Funktionalität ursprünglich nicht vorhanden war, aber so durch das Projekt ergänzt werden konnte.

Während der Integration war die größte Herausforderung die Integration mit einem individuell entwickelten Portal, das leider keine Unit-Tests hatte und so schon die meisten SOAP-Messages nicht XML-Schema-konform waren. Das kostete einige Zeit und vor allem einige zeitraubende Anläufe, bis die Kommunikation zwischen dem Portal und den Prozessen einwandfrei funktionierte. Diesen Problemen hätte man mit Unit-Tests begegnen können, die unter dem Strich auch weniger Zeit gekostet hätten als der Aufwand, den man investieren musste, um Fehler aufwändig zu beheben und neu zu testen.

Links

[BPE12] BPELUnit, Projekt-Homepage, abgerufen am: 08.03.2012, siehe: bpelunit.net

[Cha02] A. Chaffee, W. Pietri, Unit Testing with Mock Objects, 2002, siehe: ibm.com/developerworks/java/library/j-mocktest/index.html

[soa12] soapUI, Projekt-Homepage, abgerufen am: 08.03.2012, siehe: soapui.org

[XML12] XMLUnit, Projekt-Homepage, abgerufen am: 08.03.2012, xmlunit.sourceforge.net

Daneben gab es Reviews der Prozessmodelle, ob Timeouts korrekt implementiert und ob die Modellierungsrichtlinien eingehalten worden waren. Insgesamt entstanden so für das Testen der Prozesse 74 BPELUnit-Testfälle in 12 Test-Suiten, die 884 SOAP-Nachrichten senden. Für die Datentransformationen entstanden 83 JUnit-Testfälle für 2.398 Zeilen in XSLT- und 864 Zeilen in XQuery-Code sowie 161 XML-Dateien für Eingaben und erwartete Ausgaben. Durch die Hilfsklassen, die in allen Testfällen wiederverwendet werden, besteht ein Testfall nur noch aus ca. 10 Zeilen Java-Code und den zugehörigen XML-Dateien und ist somit leicht und schnell zu erstellen.

Das Erstellen einer Test-Suite dauert dabei ungefähr einen Arbeitstag. Das Beheben eines Fehlers, was die Analyse und das Hinzufügen mindestens eines Testfalls beinhaltet, dauert in der Regel nicht länger als einen halben Tag. Das erleichtert auch die Integrationstests, da ein Fehler schnell behoben werden kann und das weitere Testen nicht lange verzögert wird. Durch den automatisierten Build können schnell Fehlerbehebungsversionen in die verschiedenen Test- und Produktionsumgebungen ausgerollt werden. In der Pilotphase des Systems zeigte sich der Nutzen der Tests: Es gab bislang lediglich einen Fehler in dem Prozessteil, der auf fehlerhafte Anforderungen zurückzuführen war.

In dem Projekt wurden keine Testabdeckungsmetriken auf Codeebene verwendet. Stattdessen wurde eine Facettenklassifikation eingesetzt, die Prozessattribute (z. B. Anzahl von Artikeln) in verschiedene Klassen (z. B. keiner, einer, mehrere) unterteilt. Jede Klasse für jedes Attribut musste dabei in mindestens einem Testfall ausgeführt worden sein. Die Testabdeckungsmessung erfolgt so auf der fachlichen statt auf der technischen Ebene und ist maßgeblicher für Aussagen bezüglich der implementierten Funktionen.

Insgesamt hat sich der hier präsentierte Ansatz zur Qualitätssicherung in dem Projekt bewährt. Er wurde während der Projektdauer nach und nach verfeinert und ist nun in einem Zustand, wo es leicht ist, neue Funktionen zu ergänzen und zu überprüfen, ob diese und alle alten Funktionen weiterhin funktionieren. ■