BPM/SOA meets Java EE

Automatisierte Leistungsabrechnung bei der HanseMerkur Versicherung

Jo Ehm, Olaf Fricke

Ende 2008 startete die HanseMerkur Versicherung eine BPM/SOA-Initiative mit der Vision, die vormals monolithische und systemorientierte Sichtweise der hausinternen Informationstechnik gezielt der Prozess- und Serviceorientierung weichen zu lassen, um so die Kundenprozesse End-to-End besser beherrschen zu können. Im Einzelnen bedeutete dies die Bereitstellung einer entsprechenden Infrastruktur, die Entwicklung einer Basisarchitektur und darauf aufsetzend Schritt für Schritt eine weitgehende Ablösung der bestehenden Leistungsabrechnungssysteme durch ablauffähige Prozessbeschreibungen, wiederverwendbare Komponenten und Services auf Basis der Java Enterprise Edition.

Am Anfang des Projekts stand die Frage nach den richtigen Tools und Technologien. Eine der grundlegenden Anforderungen an eine Komponenten- und Servicearchitektur war die Möglichkeit, Services zur Laufzeit dynamisch und kontrolliert austauschen zu können sowie unterschiedliche Versionen des gleichen Service parallel betreiben und gezielt die gewünschte Version eines Service aufrufen zu können. Dazu bot sich zunächst OSGi als Framework an.

Aufgrund der fehlenden Unterstützung von Enterprise-Funktionalitäten, wie z. B. Transaktions-Handling, Security und Skalierbarkeit, fiel die Wahl schließlich aber auf EJB 3 – nicht zuletzt, da im Hause HanseMerkur bereits entsprechendes Know-how vorhanden war und JBoss als Applikationsserver im produktiven Einsatz ist.

Auf dieser Basis galt es nun ein entsprechendes Framework zu entwickeln, das den genannten Anforderungen an eine serviceorientierte Architektur (SOA) gerecht wird.

Die Architektur

Der grundlegende Aufbau der Komponenten- und Servicearchitektur erstreckt sich über mehrere Schichten (s. Abb. 1). Auf der Ebene der Datenbanken sind verschiedene Schemata vorhanden, die im Wesentlichen die Daten einer fachlichen Domäne bereitstellen. Die darüber liegenden Services sind untergliedert in solche, die klar einer Domäne zuzuordnen sind und den Zugriff auf einen fachlich abgegrenzten Bereich der Geschäftsobjekte und Daten im Unternehmen ermöglichen (Domain Services), und solche, die übergreifende, höherwertige Funktionalität anbieten (Business Services).

Die *Domain Services* bieten den sondierenden und modifizierenden Zugriff auf die Entitäten über EJB 3 Stateless Session Beans an. Sie implementieren im Wesentlichen die CRUD-Operationen (Create, Read, Update, Delete) auf alle Entitäten ihrer Domäne und stellen diese als Dienste zur Verfügung. Sofern eine Unterteilung in mehrere Services für eine Domäne sinnvoll erscheint (z. B. nach Zugriffsrollen: Administration vs. Benutzung von Domänen-Entities), können entsprechend auch mehrere Stateless Session Beans innerhalb einer Domänen-Komponente (EAR-Datei) bereitgestellt werden. Die Session Beans innerhalb eines Domänen-EAR können einander gegebenenfalls direkt referenzieren (über die `@EJB`-Annotation).

Zwischen den Services unterschiedlicher Domänen ist jedoch kein direkter Zugriff erlaubt. Dies geht so weit, dass auch zwischen den Entitäten unterschiedlicher Domänen keine JPA-Referenzen erlaubt sind. Natürlich gibt es in der Realität solche domänenübergreifenden Beziehungen. Diese werden jedoch nur über IDs (Primärschlüssel) abgebildet und in der Datenbank über Foreign-Key-Constraints abgesichert. Jede Domäne stellt somit eine in sich geschlossene Einheit dar, deren Dienste nur über die an der öffentlichen Schnittstelle zur Verfügung gestellten Domain Services in Anspruch genommen werden dürfen.

Domänenübergreifende Geschäftslogik wird über eine darüber liegende Schicht von *Business Services* realisiert. Diese arbeiten mit den Entitäten, die ihnen von den verschiedenen Domain Services zur Verfügung gestellt werden, stellen die domänenübergreifende Konsistenz sicher und implementieren die fachlichen Dienste, die schließlich in den Geschäftsprozessen Verwendung finden. Hierfür bieten sie neben den üblichen RMI Remote Interfaces auch Webservice-Schnittstellen an, die von der Process Engine genutzt werden.

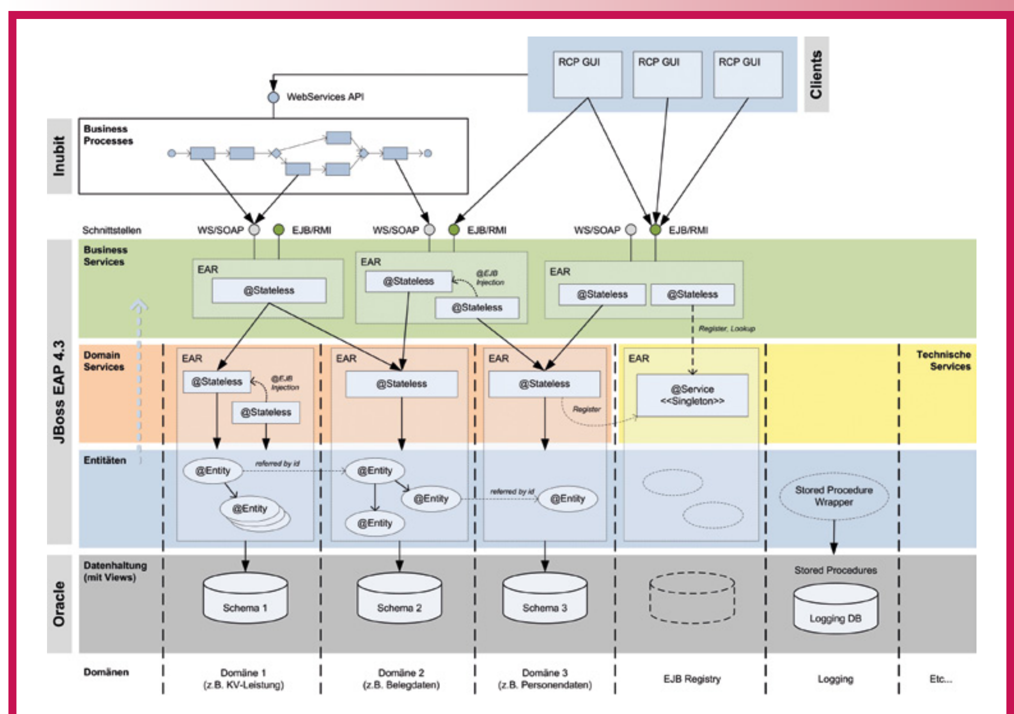

Abb. 1: Schichten der Komponenten- und Servicearchitektur

Mehrversionsfähigkeit

Um parallel mehrere Versionen einer Komponente im JBoss laufen lassen zu können, muss jedes EAR einen eigenen Classloader verwenden. Dazu wird über das **maven-ear-plugin** beim Build ein entsprechender Eintrag in der Datei **jboss-app.xml** erzeugt. Hierbei werden einfach der Name und die Versionsnummer der Komponente als Teil des eindeutigen Namens des Classloaders verwendet:

```
<plugin>
  <artifactId>maven-ear-plugin</artifactId>
    <configuration>
      <defaultLibBundleDir>lib</defaultLibBundleDir>
      <jboss>
        <loader-repository>de.hansemerkur:loader=
          ${project.artifactId}-${project.version}</loader-repository>
    </jboss>
  </configuration>
</plugin>
```

Gleiches gilt auch für die JNDI-Namen der Session Beans. Erreicht wird dies dadurch, dass die Versionsnummer auch als Namensbestandteil der erzeugten EAR-Komponente verwendet wird. Über ähnliche Konfigurationen wird weiterhin sichergestellt, dass auch alle Persistence Units und MBeans eindeutige Namen erhalten.

Doch wie lässt sich nun gezielt auf eine bestimmte Version eines Service zugreifen? Hierzu wurde eine zentrale Registry-Komponente entwickelt, die im globalen Classloader des JBoss läuft und somit für alle Service-Komponenten zugänglich ist. Im Wesentlichen handelt es sich hierbei um eine Singleton Stateless Session Bean, die in einer verschachtelten Map zu jedem Remote Interface der deployten Service-EJBs für alle verfügbaren Versionen das zugehörige Proxy-Objekt, wie man es sonst über JNDI bekommen würde, bereithält. Die Registrierung eines Service bei dieser Component Registry geschieht automatisch beim Deployment über eine JMX MBean, die auf die Start-Events des JBoss-ServiceControllers horcht (s. Abb. 2). Die Deregistrierung von undeployten EJBs erfolgt analog beim Auftreten eines Stop-Events.

Für den Zugriff auf die so registrierten Services bietet die Component Registry eine Reihe von **lookup()**-Methoden. Bei der Benutzung der Component Registry durch serverseitige Komponenten ergibt sich allerdings das Problem, dass das Proxy-Objekt, das in der Registry gespeichert ist, zu dem Classloader des EAR gehört, das die entsprechende EJB bereitstellt. Will eine andere Komponente diesen Proxy benutzen, so muss dieser für den entsprechenden Ziel-Classloader bereitgestellt werden – ansonsten kommt es zu einer **ClassCastException**. Die Lookup-Methoden der Component Registry ermitteln daher den Classloader der aufrufenden Komponente und die Component Registry überprüft *bei einem davon*, ob der angefragte Classloader dem des Proxy-Objekts aus ihrer primären Liste entspricht. Ist dies nicht der Fall, so wird das Proxy-Objekt für den Ziel-Classloader serialisiert und in einer weiteren, nochmals verschachtelten Map zwischengespeichert (dem „Classloader Cache"). Bei nachfolgenden Lookups wird dann in diesem Classloader Cache nachgesehen, ob ein passender Proxy vorhanden ist.

Über die **lookup()**-Methoden können nun – angelehnt an OSGi – gezielt konkrete Versionen angefragt werden oder auch Versionsbereiche, aus denen dann die höchste verfügbare Version geliefert wird. In der Praxis ist dies jedoch von untergeordneter Bedeutung. In der Regel weiß eine Komponente genau, mit welcher Version eines anderen Service sie arbeiten kann und will. Diese Information ist sogar immanent, da sie die API-Klassen der verwendeten Services ja als Bibliotheken quasi „im Bauch" hat. In der Datei MANIFEST.MF dieser API-JARs findet sich wiederum die Version des verwendeten Interfaces.

Dies macht sich die Superklasse aller Service-Beans zunutze: Sie kapselt nicht nur die Component Registry an sich (deren lokales Interface über die **@EJB**-Annotation dort injiziert wird), sondern auch die verschiedenen **lookup()**-Methoden. Über die Superklasse wird den Service-Beans nur eine einzelne **lookup()**-Methode bereitgestellt, die ohne Versionsangabe und nur mit der angefragten Interface-Klasse arbeitet. In dieser Methode wird die Version des angefragten Interfaces aus dem zugehörigen Manifest gelesen und die entsprechende Version bei der Component Registry angefragt.

Darüber hinaus findet noch eine weitere Abstraktion statt. Bei einem Versionsschema *<major>.<minor>.<patch>* ist es in der Regel egal, welches Patch-Level verwendet wird, denn per Definition bedeutet ein Patch keine Änderung an den Schnitt-

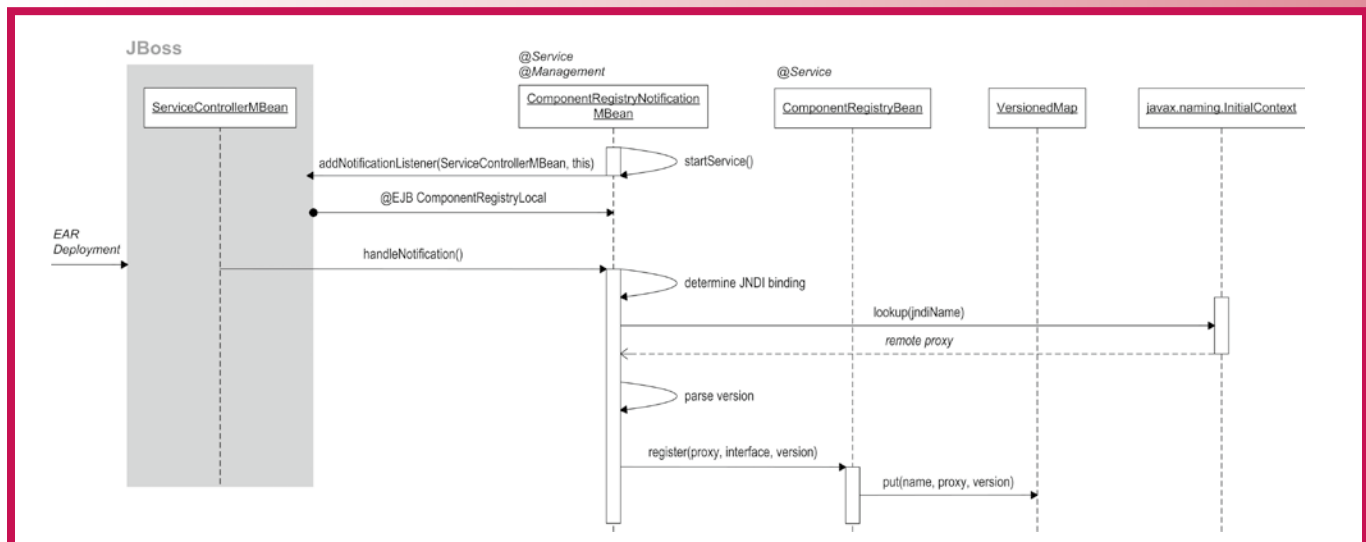

Abb. 2: Funktionsweise der automatischen Registrierung beim Deployment

stellen oder der gr undlegenden Geschäftslogik. Es ist dementsprechend wünschenswert, immer stets die höchste verfügbare Patch-Level-Version eines Service zu bekommen. Genau dies leistet die `lookup()`-Methode der Superklasse der Service-Beans, indem sie bei der Component Registry den entsprechenden Versionsbereich nachfragt.

Für den Entwickler stellt sich der Zugriff auf Services anderer Komponenten dank dieses Frameworks denkbar einfach dar:

```
@Stateless
@Remote(MyBusinessService.class)
public class MyBusinessService extends BusinessServiceBase
  implements MyBusinessService {
  public void accessDomainServices() {
    DomainService1 ds1 = lookup(DomainService1.class);
    DomainService2 ds2 = lookup(DomainService2.class);
    // ...
  }
}
```

Die Component Registry ist also nicht nur ein Verzeichnis verfügbarer Services, sondern gleichzeitig auch ein Cache für Proxy-Objekte. Nach dem erstmaligen JNDI-Lookup eines Objekts wird kein weiterer JNDI-Zugriff benötigt (solange das Objekt nicht undeployed wird) und die Proxys im Classloader des Dienstnehmers müssen nur ein einziges Mal für den entsprechenden Classloader serialisiert werden.

Dies bedeutet für das Programmiermodell der Services, dass vor jedem Aufruf einer anderen Komponente immer ein Lookup erfolgen muss und keine Referenzen auf andere Services lokal vorgehalten werden dürfen. Erst so ist es möglich, Services auf der Ebene von Patch-Level-Releases zur Laufzeit auszutauschen. Dieses Vorgehen behebt das JEE-Manko, dass `@EJB`-Referenzen nur einmalig beim Deployment injiziert werden, später aber weder aktualisiert noch geleert werden.

Die Prozesse

Auf die oberste Schicht der Business Services setzen nun die eigentlichen Geschäftsprozesse auf – vom Input-Management über die fachlichen Prozesse, bestehend aus der gebührenrechtlichen und der tariflichen Leistungsprüfung sowie der Leistungsabrechnung innerhalb des Krankenleistungssystems, bis hin zum Output-Management.

Als *Business Process Mangement System* (BPMS) kommt bei der HanseMerkur die inubit Suite zum Einsatz. Hier modellieren zunächst die Mitarbeiter der Fachbereiche ihre Leistungsprozesse in BPMN. Diese Vorgaben werden dann von der Entwicklung als ausführbare Technical Workflows umgesetzt, in denen die fachlichen Daten als XML-Strukturen durch den Prozess geführt werden und mittels XSLT, XPath usw. manipuliert werden können. Besonders komplexe Entscheidungslogik wird aus den Prozessen ausgelagert, mittels Visual Rules realisiert und als Service bereitgestellt.

Aus den Prozessen heraus erfolgt der Zugriff auf die Business Services über deren Webservice-Schnittstellen mittels entsprechender Connectoren. Um auch auf diesem Weg gezielt auf eine gewünschte Version zugreifen zu können, wird in den Service-Klassen beim Maven-Build über die Annotation `@WebContext` die Versionsinformation als Teil der Webservice-URL generiert. Auch hier wird von der dritten Stelle der Versionsnummer (Patch-Level) abstrahiert, sodass Patches von Services ohne Änderung an den aufrufenden Prozessen eingespielt werden können.

Wesentlich für die technische Ausgestaltung der Prozesse ist die Frage, welche Menge an Daten zwischen Prozess und den darunterliegenden Services ausgetauscht wird. Welche Daten werden allein durch Services verwaltet und welche als sogenannte Payload durch den Prozess geführt? Dabei sind der Durchsatz, die Aktualität der Daten und die Anzahl der Serviceaufrufe gegeneinander abzuwägen.

Führt man alle ermittelten Informationen stets durch alle nachfolgenden Prozessschritte, so erspart man sich vielleicht wiederholte Anfragen an die Serviceschicht, hantiert aber unter Umständen mit Daten, die auf Ebene des Prozesses gar nicht benötigt werden, hat damit gegebenenfalls erhöhte Konvertierungsaufwände oder der Durchsatz wird nachteilig beeinflusst. Bei langlaufenden Prozessen muss ferner darauf geachtet werden, dass die Daten noch aktuell sind und nicht zwischenzeitlich durch andere Systeme geändert wurden.

Aus diesen Gründen wurden die technischen Prozesse so ausgelegt, dass die Payload auf das Nötigste reduziert wird: Anstelle von kompletten Datensätzen werden im Wesentlichen nur IDs und Statusinformationen durch den Prozess geführt. Die Manipulation der zugrunde liegenden Daten und Geschäftsobjekte erfolgt rein auf der Ebene der Services. Auf diese Weise wird auch eine saubere Trennung von Ablauf- und Geschäftslogik erzielt – ein wichtiger Grundstein, um Prozesse auf Basis wiederverwendbarer Services aufbauen zu können und redundant implementierte Geschäftslogik zu vermeiden.

Um zur Laufzeit die prozessrelevanten Geschäftsobjekte zusammenhalten zu können, wurde – in Analogie zur realen Welt – das Konzept einer Arbeitsmappe (s. Abb. 3) realisiert. Eine Arbeitsmappe ist eine Gruppierung von Elementen, die gemeinsam durch einen Prozess geführt werden sollen. Sie repräsentiert die Verarbeitungseinheit für einen Prozess, d. h., pro Prozessinstanz wird initial dessen Arbeitsmappe erzeugt und darin werden die zu bearbeitenden Elemente angelegt. Diese Arbeitsmappenelemente sind die (unteilbaren) Einheiten, auf denen der Prozess letztendlich operiert. Dabei handelt es sich jedoch nicht direkt um konkrete Geschäftsobjekte, sondern um Stellvertreter in Form

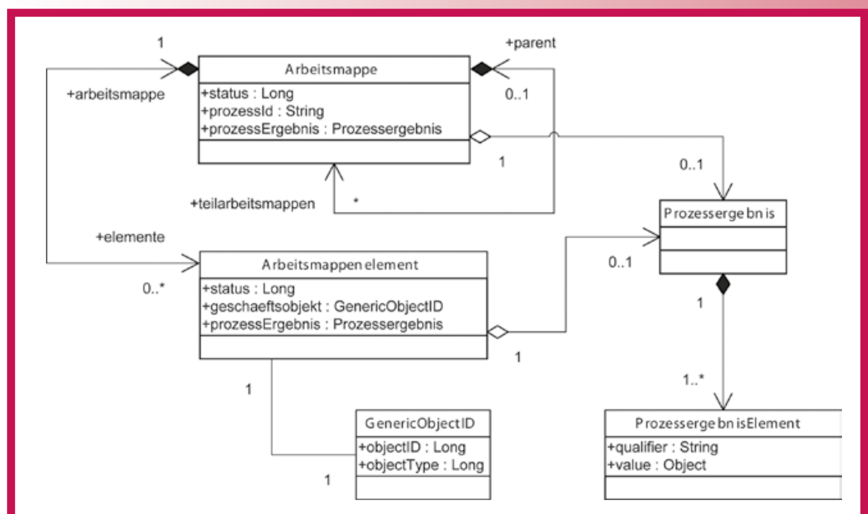

Abb. 3: Arbeitsmappe

sogenannter `GenericObjectID`s, die über eine ID (Primärschlüssel) und eine Typangabe auf konkrete Geschäftsobjekte verweisen.

Sowohl Arbeitsmappe als auch Arbeitsmappenelemente unterliegen einem Lebenszyklus-Statusmodell, das durch den Prozess bestimmt wird. Gleichzeitig können durch die Prozesse oder Services ermittelte Ergebnisse in Form von Key/Value-Paaren an eine Arbeitsmappe oder einzelne Arbeitsmappenelemente „angehängt“ werden. Bei den Werten der ermittelten Prozessergebnisse kann es sich sowohl um einfache Strings, numerische, boolesche oder Datumswerte als auch um Referenzen auf komplexere Geschäftsobjekte (via `GenericObjectID`) handeln.

Dies ist (neben möglichen Rückgabewerten einzelner Service-Aufrufe) die Datenstruktur, auf deren Basis die Prozesse ihre Ablaufentscheidungen treffen. Die Manipulation der eigentlichen Geschäftsobjekte erfolgt, wie bereits erwähnt, auf der Ebene der fachlichen Services, denen seitens der Prozesse dann jeweils nur die entsprechenden Objekt-IDs übergeben werden.

Das Testen

Eine weitere Herausforderung in einer derartig verteilten Architektur ist immer wieder das Testen. Für die Unit-Tests von EJBs werden bei der HanseMerkur JUnit und Mockito eingesetzt. Für sinnvolle und vor allem für wiederholbare Tests von Services, die hauptsächlich auf der Datenbank operieren, ist es jedoch oft auch nötig, entsprechende Datenlagen als Vorbedingung für die Testdurchführung zu schaffen bzw. nach einem Test wieder zurückzusetzen.

Für die Unit-Tests auf Ebene der Domain Services wurden daher als Teil des Infrastruktur-Frameworks eigene Klassen für Test-Cases entwickelt, mit denen es unter anderem möglich ist, einzelne Testmethoden mit speziellen Annotationen auszuzeichnen, um so für die jeweiligen Tests vor- und nachbereitende SQL-Statements (auch dateibasiert und parametrisiert) deklarieren zu können.

Auf der Ebene der Business Services erwies es sich als großer Vorteil, dass der Zugriff auf benötigte andere Services ausschließlich über die eigens entwickelte Component Registry erfolgt. Da diese Komponente selbst auch ein POJO ist, war es einfach möglich, entsprechende Unit-Test-Superklassen bereitzustellen, mit denen den zu testenden EJBs ein ComponentRegistry-POJO injiziert werden kann, um dort dann Mock-Objekte der benutzten Services zu registrieren.

Für die Integrationstests über die verschiedenen Schichten der Services hinweg wird weiterhin das FIT-Framework eingesetzt. Standardmäßig können mit FIT z. B. in Excel-Dateien zu definierten Eingabewerten erwartete Ausgabewerte angegeben und verglichen werden. Für komplexere Tests muss aber auch hier oft erst als Vorbedingung datenbankseitig eine bestimmte Datenlage geschaffen werden, oder der Test ist erst dann wirklich aussagekräftig, wenn auch die datenbankseitig erzeugten Ergebnisse, die nicht zwingend Teil der Rückgabewerte sind, überprüft werden können. In diesem Sinne wurde auch das FIT-Framework entsprechend erweitert, um in einem Vorlauf Geschäftsobjekte in der Datenbank erzeugen zu können und diese (wie auch die durch den Test entstandenen persistenten Objekte) in einem Nachlauf automatisch wieder zu löschen.

Dieses Prinzip lässt sich nicht nur für Integrationstests von Services anwenden, sondern auch für einzelne Teilprozesse oder gesamte Prozessabläufe. Dabei werden dann die im Verlauf eines Prozesses erwirtschafteten Prozessergebnisse ausgelesen und verglichen. Die Information bzw. die Erwartungshaltung, welche Prozessergebnisse in welchen Ausprägungen vorhanden sind, gibt dabei Aufschluss über den tatsächlichen Prozessverlauf. Diese Arten von FIT-Tests werden zum einen in der Entwicklung für technische Integrationstests genutzt, zum anderen wurden sie speziell für den Fachbereich entwickelt, um die umfangreichen Abnahmetests zu automatisieren und so eine immer größer werdende Menge von Regressionstests aufbauen zu können.

Fazit

Mit ihrer BPM/SOA-Initiative hat die HanseMerkur einen wesentlichen Schritt in Richtung „Versicherungsbetrieb der Zukunft“ getan. Mit den bisherigen Ausbaustufen konnte die Automatisierungsquote über alle Leistungsabrechnungen von anfänglich rund 8 % in 2009 auf aktuell über 25 % gesteigert werden. Dieser hohe Anteil an voll maschinell erbrachten Leistungsabrechnungen gibt den Mitarbeitern der Leistungsabteilungen einerseits die nötige Zeit, komplexe Fälle gewissenhaft manuell zu prüfen und andererseits direkte Beratungsleistungen im Rahmen des Kundenservice in den Vordergrund der Arbeit zu stellen.

IT-seitig hat sich mit der beschriebenen Komponenten- und Servicearchitektur und den damit verbundenen Vorgehensmodellen der Prozessorientierung ein neues Paradigma etabliert, das auch in Zukunft die Entwicklung der hausinternen Anwendungssysteme bestimmen wird. Insbesondere die beschriebene Component Registry hat sich mittlerweile in der Praxis bewährt. Nicht nur, weil sie die Ad-hoc-Austauschbarkeit von Services und deren Parallelbetrieb ermöglicht, sondern auch, weil sie ein für den Entwickler gut handhabbares Programmiermodell liefert und die Testbarkeit der Komponenten begünstigt. Sie ist ein entscheidender Teil des technologischen Rückgrats der unternehmensweiten Komponenten- und Servicelandschaft, die es durch kommende Projekte weiter aufzubauen gilt.

Jo Ehm ist Senior Consultant bei der Holistcon AG in Hamburg und dort u. a. verantwortlich für das Geschäftsfeld BPM/SOA. Er ist Certified Scrum Master und seit über zehn Jahren als Business Analyst, Softwarearchitekt, Trainer und Coach im Bereich objekt-, komponenten- und serviceorientierter Architekturen tätig. Als BPM/SOA-Berater und Scrum-Coach war er mitverantwortlich für den Aufbau der BPM/SOA-Architektur bei der HanseMerkur.
E-Mail: jo.ehm@holisticon.de

Olaf Fricke ist Diplom-Informatiker und seit mehr als zwölf Jahren in der Softwareentwicklung tätig. Bei der HanseMerkur Versicherung AG verantwortet er die technischen Portale des Unternehmens und steht in Entwicklungsprojekten als Architekt und Java-Spezialist beratend zur Seite. Die Komponenten- und Service-Architektur der HanseMerkur hat er maßgeblich mitgestaltet.
E-Mail: olaf.fricke@hansemerkur.de