Mindestumfang eines SOA Test Modells

Gebhard Greiter, Jan 2009

Das Testen von Software, die service-orientierte Architektur hat (SOA Software), muss nach einem etwas anderen Modell erfolgen als das Testen herkömmlicher Software. Vorliegendes Papier dient dazu, dieses neue Modell zu definieren und kurz zu begründen.

1 Zum Test von SOA-Komponenten

Klassische Software gliedert sich hierarchisch in Komponenten und schließlich Module, die i.A. nicht komplett isoliert von der Umgebung, in der sie Verwendung finden, getestet werden können. Sie einzeln zu testen erfordert geeignete Testrahmen, deren Zweck es ist, die Umgebung zu simulieren.

Ganz anders SOA Software: Sie gliedert sich hinreichend fein in Bausteine, die sämtlich in sich abgeschlossene Systeme darstellen, in Bausteine also, die man einzeln und unabhängig von irgend einer bestimmten Umgebung testen kann. Es sind dies ausschließlich Bausteine der Typen

User Interface Client (UIC), **Prozess**, oder **Service**.

Nennen wir sie kurz **SOA-Komponenten**.

Ein Wort zur Klarstellung: SOA-Komponenten sind nicht notwendig disjunkt. Dies einzusehen reicht es, Services zu betrachten, die alle gemeinsam auf einen einzigen, ganz bestimmten Bestand persistenter Daten zugreifen (man nennt ihn die Geschäftsobjekte des Betreibers dieser Services). Sämtliche dieser Services lassen sich dennoch einzeln und unabhängig voneinander testen, da man als **SOA-Service** nur bezeichnet, was eine eigene in sich abgeschlossene Anwendung ist (eine sog. Mini-Anwendung). Erst durch geeignete Orchestrierung einer ganzen Menge von Services zu einem **Prozess** entsteht eine Anwendung aus Business Sicht. Wenigstens dort, wo diese Anwendung nicht als Komponente einer noch größeren auftritt, muss zudem noch ein **User Interface Client** hinzukommen.

Geschäftsobjekte sollten grundsätzlich nur – einzeln oder in geeigneten kleinen Gruppen – über sie abstrahierende SOA Services (sog. **Bestandsservices**) zugänglich sein. Gute Systemarchitekten kennen und beherzigen diese Regel. **Lesender** (und wirklich ***nur*** lesender) SQL-Zugriff auf persistente Daten kann und sollte dem Tester – je nachdem für wen er tätig ist – dennoch erlaubt sein.

Es ist in der Tat wichtig, für jede zu testende SOA-Komponente klar zu sagen, aus wessen Sicht heraus man sie zu testen hat. Es gibt genau drei solcher Sichten:

- die Sicht der **Anwender**,
- die Sicht der **Betreiber** der Komponente, und
- die Sicht der **Entwickler** dieser Software.

Je nachdem, aus welcher Sicht heraus man testet, ist das sog. Implementierungsgeheimnis der Komponente – jener Teil also, den der Tester nur als Black Box zu sehen bekommt – verschieden groß. Es gilt:

- Die **Betreibersicht** umfaßt die Anwendersicht, muss darüber hinaus aber erlauben, zu beobachten, wie dem Anwender und dem Betreiber erlaubte Aufrufe der Komponente persistente Daten fortschreiben.

- Die **Entwicklersicht** besteht aus der Sicht des Betreibers und zudem noch aus dem Source Code der Komponente soweit der nicht Teil einer von der Komponente aufgerufenen anderen SOA-Komponente ist (jene sollte man als Tester oder Entwickler nur aus Betreibersicht betrachten).

Je nachdem, aus welcher Sicht heraus man eine SOA-Komponente zu testen hat, bezeichne man sie als **A-**, **B-** oder **E-Komponente**.

Wenigstens als Testobjekt~~,~~ sollte jede SOA-Komponente einen Bezeichner haben, der aus vier Teilen besteht:

- Teil 1 sei der Buchstabe **A**, **B** oder **E** (die Sicht, aus der zu testen ist).
- Teil 2 sei ein die Funktion der Komponente signalisierender Name.
- Teil 3 sei die Nummer der Version, die zu testen ist.
- Teil 4 sei eine zweite, niemals größere Versionsnummer oder 0.

Der zweiten Versionsnummer kommt fundamentale Bedeutung zu: Ist nämlich **Vn** die Nummer der zu testenden Version, so muss die zweite Versionsnummer **Va** so gewählt werden dass gilt:

Wo immer man die Komponete zeitlich parallel in Versionen **V** mit **Va** <= **V** <= **Vn** arbeiten lässt, muss die Wirkung all dieser Aufrufe so sein, als hätten sie in mindestens einer Reihenfolge streng sequentiell hintereinander gearbeitet.

Man erkennt: In einer Umgebung, in der die neue Version **Vn** der Komponente aufrufbar sein soll, dürfen Versionen kleiner als **Va** nicht mehr installiert (sprich: aufrufbar) sein.

Die Nummer **Va** setzen zu können ist notwendig, da man – in Abhängigkeit der Implementierung älterer Versionen – nicht immer garantieren kann, dass wirklich alle Versionen einer SOA-Komponente parallel zueinander in derselben Umgebung einsetzbar sind.

Zudem wird es nicht immer möglich sein, jeweils nur die letzte Version produktiv zu haben: Wenigstens dort, wo SOA-Komponenten als Webservice publiziert sind, müssen Versionen, die sich aus Sicht ihrer Anwender unterscheiden – wie wenig auch immer –, auf jeden Fall als voneinander verschiedene Komponenten gesehen werden. Ganz besonders dort, wo auch Geschäftspartner des Betreibers eine Komponente nutzen, wird er eine einmal existierende Komponentenversion i.A. nicht ohne weiteres vom Markt nehmen können. Daher also die oben genannte Regel.

Sofern der Tester für den Betreiber oder die Entwickler der Komponentenversion **Vn** arbeitet, gehört es mit zu seinen Aufgaben, zu testen, ob **Va** hinreichend groß gewählt ist. Er muss insbesondere eine Testumgebung zur Verfügung haben, in der er sämtliche Versionen der Komponente – bis hinunter zu **Va** – auch wirklich parallel zueinander aufrufen kann.

Zudem sollte der Tester gehalten sein, ein Komponentenprofil zu erstellen, aus dem hervorgeht, welchen Resourssenbedarf eine Nutzung der dem Test unterworfenen Komponente impliziert.

Nur wenn der Tester auch für den Entwickler arbeitet, kann gewünscht werden, dass dieses Komponentenprofil die Komponente zudem noch charakterisiert im Lichte gängiger Metriken für realisierungstechnische Software-Qualität.

2 Mehr zum Test von SOA-Services

Alle bisher gemachten Aussagen gelten für jeden Typ von SOA-Komponenten. Wer nun aber konkret zu testen beginnt, wird sofort feststellen, dass man – je nach Komponententyp – im Einzelnen recht verschieden vorgehen muss.

Besonders einfach zu testen sind Services: Sie haben eine Programmierschnittstelle, sind also auf jeden Fall testbar über automatisch arbeitende Testtreiber. Auch SOLL/IST-Vergleich ist einfach, sobald man geeignete Testdaten gefunden hat. Jeder Test Case ist ein einzelner Serviceaufruf.

3 Mehr zum Test von SOA-Prozessen

Jeder SOA-Prozess realisiert Vorgangsbearbeitung. Insbesondere haben SOA-Komponenten des Typs **Prozess** i.A. mehr als nur eine Aufrufschnittstelle: Sie stellen ihrem Anwender Operationen bereit, über die er

- Vorgänge ins Leben rufen sowie
- Vorgangsdaten einsehen und fortschreiben

kann. Insbesondere muss schon die Anwendersicht auf den Prozess ein Statusmodell für Vorgangsdaten enthalten. Es ist Aufgabe des Tester zu prüfen, ob

- Vorgänge korrekt ins Leben gerufen werden,
- Vorgangsdaten (Geschäftsobjekte also) korrekt initialisiert werden,
- jede auf den Vorgang zugreifende Operation den Vorgangszustand in Abhängigkeit aktueller Parameter korrekt fortschreibt,
- jeder eintretende Zustandsübergang eventuell zu triggernde andere Prozesse korrekt anstößt,
- bzw. durch den Zustandsübergang abgeschlossene Vorgänge geeignet archiviert (was i.A. nur aus Betreibersicht prüfbar sein muss).

Sofern der zu testende Prozess gewisse Zustandsübergange zeitgesteuert anzustoßen hat, ist mit zu prüfen, ob dies auch tatsächlich erfolgt (und ob hierbei die gewünschte Wirkung eintritt).

Wo man aus Anwendersicht zu testen hat, kann erschwerend hinzukommen, dass der Tester auf den Prozess nur Zugriff über eine graphische Benutzeroberfläche hat (der Testling dann also nicht der nackte Prozess ist sondern eine komplette Dialoganwendung im klassischen Sinne – die i.A. schwierigste Situation).

4 Mehr zum Test von User Interface Komponenten

Dieser Test ist der im SOA-Umfeld wohl einfachste, wird aber i.A. nur möglich sein, wenn man aus Entwickler-, oder wenigstens noch aus Betreibersicht testet, aus Sicht eines Teams also, welches die zu prüfende SOA-Komponente isoliert von anderen zur Verfügung stellen kann.

5 Voraussetzungen für kostengünstige Fehlerdiagnose

Insbesondere dann, wenn Prozesse zu testen sind, wird es für den Tester ohne geeignete Loggingfunktionalität extrem schwierig, zu beurteilen, was Aufrufe der Prozessoperationen tatsächlich bewirken.

Mindestanforderung an solche Loggingmöglichkeit ist, dass Trace entsteht, der zweifelsfrei zeigt, welche Services in welcher Verschachtelung durch den jeweiligen Aufruf der Prozessoperation aktiviert wurden und – das ist wichtig –, welche Parameterwerte ihnen mitgegeben wurden.

Wo Logging, welches diese Anforderung erfüllt, nicht möglich ist, wird der Tester nur allzu einfache Test Cases konstruieren. Im Extremfall wird solcher Test dann zur Geldverschwendung (weil er kaum Fehler zu finden in der Lage sein wird, aber doch hohen Aufwand verursacht).

Umgekehrt ist eine optimale Situation dann gegeben, wenn bei der Entwicklung von SAO-Software sichergestellt wurde, dass jeder Serviceaufruf einen Parameter hat, der die den Aufruf direkt aktivierende SAO-Komponente nennt oder – noch optimaler – den Aufrufstack beschreibt.

Der Aufrufstack lässt sich am besten beschreiben als String, der sich zusammensetzt aus Paaren (**Z**,**S**), die Serviceaufrufe benennen:

Z der Zeitpunkt, zu dem Service **S** aufgerufen wurde.

Dieser String – die sog. **Stack ID** – wächst und schrumpft entsprechend der zu jedem Zeitpunkt gerade ineinandergeschachtelten Serviceaufrufe. Er zerlegt das Log File in Abschnitte, deren jeder mit einer Stack ID beginnt und im Anschluss daran die aktuellen Parameter der an dieser Stelle aufgerufenen Operation enthalten kann. Ein ganz primitives Werkzeug des Testers (oder des Entwicklers, dem solche Logs ebenfalls eine grosse Hilfe sein werden) kann dann über einen einzigen Aufruf einen Auszug des Log Files erstellen, welcher alles – und *nur* das – enthält, was zeigt, wie der fragliche Prozessaufruf bis in alle Details hinein gewirkt hat. Auch Performanzprofile lassen sich auf diese Weise mühelos und sehr aussagekräftig erstellen.

Da die Entwickler von Services i.A. auch eingekaufte Services nutzen werden, kann man nicht davon ausgehen, dass jeder Service direkt den fraglichen Parameter **Stack ID** schon hat. Einer der Vorteile service-orientierter Architekur besteht nun aber gerade darin, dass Serviceaufrufe ganz grundsätzlich nicht hart codiert sondern per XML konfiguriert werden. Zudem werden sie dem angesprochenen Service stets nur mittelbar – über einen sog. Enterprise Service Bus – zugestellt.

Bei der Bereitstellung von SOA-Komponenten zum Test kann deswegen eine für diesen Zweck maßgeschneiderte Version der Konfigurationsdatei zum Einsatz kommen: Über einen passenden, einfach zu erstellenden Generator werde jeder Service, den auch der Entwickler der zum Test vorgelegten SOA-Komponente nur als Black Box kennt, in einer Schale gekapselt, die den neuen Parameter berücksichtigt. Das reicht aus, denn selbst wenn die fragliche Komponente aus Entwicklersicht getestet wird, muss man durch sie aufgerufene Fremdservices auf

jeden Fall als nicht weiter zerlegbar betrachten: Ihr Implementierungsgeheimnis ist ja nur Personen bekannt, für die der Tester *nicht* arbeitet.

Nebenbei: Sind die Entwickler der zu testenden SOA-Software identisch mit den Autoren des den SOA-Bus (ESB) darstellenden Codes, so braucht man auch eingekaufte Services nicht mit einer Schale versehen: Das ansonsten durch diese Schalen zu erledigende Logging kann (und sollte) dann im ESB selbst implementiert werden.

Ist ein Prozess zu testen, der ausschließlich Fremdservices aufruft, so werden die Werte der **Stack ID** stets nur aus genau zwei Paaren (**Z**,**S**) bestehen. Selbst in diesem Extremfall aber ist die entstehende Logdatei ausreichend, genau zu erkennen, welche Serviceaufrufe der zu testende Teil der Service-Orchestrierung nach jedem Prozessaufruf denn nun eigentlich getätigt hat. Wenn maximaler Logging Level aktiviert ist, sind dem Trace zudem die Werte der Parameter all dieser Aufrufe zu entnehmen.

Logging ganz ohne **Stack ID** ist nur ausreichend, wo

- Prozesse zu testen sind, bei denen die Eingabeparameter jedes dem Tester sichtbaren Serviceaufrufs eine den jeweils betroffenen Vorgang eindeutig identifizierende ID enthalten
- und man zudem dem Log File für jeden Serviceaufruf entnehmen kann, ob der Aufrufer ihn synchron oder asynchron gestartet hat.

Es sollte dann der Bus selbst alle Serviceaufrufe im Log File beschreiben über Paare (**x.Z**,**S**), wo **x** ein Buchstabe ist, welcher sagt, ob hier der Serviceaufruf

- **asynchron gestartet**,
- **synchron gestartet** oder
- **synchron abgeschlossen** wurde.

Solche Beschreibung der Serviceaufrufe ist deswegen ausreichend, da – wenn zudem stets auch die Werte der Eingabeparameter abgebildet werden – die dem Tester wichtige **Stack ID** über die Utility errechnet werden kann, welche das Log File reduziert auf den Teil, der beschreibt, wie der Vorgang, den der Tester zu analysieren wünscht, bearbeitet wurde.

Diese Form des Loggings hat zudem noch den Vorteil, das jeweilige Geschehen auch dann noch gut verständlich zu machen, wenn mehrere Benutzer denselben Vorgang parallel zueinander bearbeiten.

In der Summe ist festzuhalten:

Optimales Logging zu gewährleisten, kann in Abhängigkeit der als SOA-Plattform genutzten Produkte ganz unterschiedlich schwierig sein. Wo Testcenter aufgebaut werden, sollte es deswegen eine der ersten Aufgaben der Tester sein, mit dem Auftraggeber zu klären, in welchem Umfang er bereit ist, eventuell noch fehlende Logging-Möglichkeiten mit zu schaffen. Letzteres kann Mitarbeit der SOA-Entwickler erfordern. Was erreichbar ist, wird oft projektspezifisch sein.

6 SOA impliziert Chancen, den Testprozess zu verbessern

Im Falle klassischer Client Server Software (etwa in C++ geschrieben) können interne Schnittstellen von Applikationssystemen eigentlich nur durch die Entwickler selbst erfolgreich getestet werden.

Ganz anders bei SOA-Software: Da dort nicht nur die Anwendung insgesamt sondern zudem auch jede einzelne SOA-Komponente eine logisch in sich abgeschlossene, ausreichend gut dokumentierte, ohne Kontext arbeitsfähige (Mini-) Applikation darstellt, können hier vom Entwickler unabhängig arbeitende Tester auch noch in großem Umfang White Box Test übernehmen.

Man erkennt: Im SOA-Umfeld ist die Arbeit der Entwickler und Tester weit mehr parallelisierbar, als das bei klassisch strukturierter Software der Fall war.

Zudem gilt: Während bei klassischer Client Server Software unabhängige Tester den Testling nur im Ausnahmefall über eine Programmierschnittstelle aufzurufen in der Lage sind, werden sie beim Test von SOA-Software das in weit höherem Maße tun können. Dies bedeutet, dass SOA-Test weit besser automatisierbar ist als der Test klassisch strukturierter Client Server Software – und das nicht nur deswegen, weil Komponententest sich nun mehr und mehr vom Entwickler weg hin zum unabhängigen Tester verlagert.

Test Center einzurichten und umfangreiche, komplett automatisch arbeitende **Suiten für Regressionstest** aufzubauen, macht so mehr und mehr Sinn.