Services, Komponenten, Modelle

Markus Völter, voelter@acm.org, www.voelter.de

Service Oriented Architecture (SOA) ist zu einem *der* Hype-Themen geworden. Mehrere meiner Kunden stellen ihre Architekturen derzeit auf „SOA" um oder planen dies zu tun. Interessanterweise machen diese Kunden alle mehr oder weniger unterschiedliche Sachen. Dies deutet darauf hin, dass SOA sehr viele verschiedene Bedeutungen haben kann und ein nicht wirklich scharf definierter Begriff ist. In diesem Artikel möchte ich daher gar nicht erst versuchen, SOA allgemeingültig zu definieren. Stattdessen möchte ich auf einige wichtige Merkmale und Lessons-Learned eingehen. Dabei spielen Komponententechnologien und Modellgetriebene Softwareentwicklung eine zentrale Rolle.

Sichtweisen auf SOA

Wenn man mit verschiedenen Leuten oder Organisationen spricht, stellen sich (mindestens) drei verschiedene Sichtweisen auf SOA heraus. Allen Sichtweisen gemein ist, dass SOA in aller Regel für große, komplexe Systeme bzw. für Komglomerate aus vielen verschiedenen Systemen als Architekturmetapher Verwendung findet.

- *SOA == CBD:* Eine Sichtweise besagt, dass SOA eigentlich nichts anderes ist als „gut gemachte Komponentenarchitektur": Bausteine mit einer wohl definierten Verantwortung bieten bzw. nutzen formal definierte Dienste.
- *SOA == EAI:* Eine andere Sichtweise sieht SOA als Weiterentwicklung von EAI. Damit wird asynchrone, lose gekoppelte (nachrichten basierte) Kommunikation wichtig, Datenstrukturen müssen aufeinander abgebildet, gefiltert und geroutet werden.
- *SOA == BPM:* Die Dritte Sicht auf SOA stellt den Nutzen für die Fachabteilung in den Mittelpunkt. „Business-Driven" ist hier das Schlagwort der Stunde. In dieser Sichtweise rückt die Definition von Geschäftsprozessen in den Vordergrund.

Meines Erachtens passen all diese Sichtweisen ganz gut zusammen und man kann sie durchaus in aufeinander aufbauende Schichten einteilen. Als Basis für SOA dienen dabei wohl definierte Komponenten – irgendjemand muss die Services ja auch implementieren. Darauf aufbauend können nun Geschäftsprozesse definiert werden. Durch entsprechende (Legacy-)Adapter- und Filterkomponenten lässt sich damit EAI betreiben.

Services im Sinne einer SOA sind grobgranular. Sie sollen sich an den Geschäftsprozessen orientieren und aus Sicht der Fachabteilung sinnvolle Funktionalität zur Verfügung stellen. Diese Punkte sind natürlich schwer konkret zu fassen. Es macht des weiteren Sinn zu

unterscheiden zwischen der SOA innerhalb eines Unternehmens und den Services die nach außen, also für externe Kommunikation zur Verfügung stehen.

Zwei weitere Punkte die gerne als wesentliche Vorteile einer SOA hervorgehoben werden sind die Trennung von fachlichen und technischen Belangen sowie Managebarkeit. Ein Beispiel für ersteres ist, dass Serviceinterfaces nur fachlich relevante Geschäftsdaten beinhalten und die Service-Implementierung sich nicht um technische Aspekte (Security, Persistence, Failover, …) kümmern muss. Managebarkeit äußert sich darin, dass man Komponenten verwalten, versionieren und installieren kann, und dass es einen zentralen Überblick über laufende Komponenten (und ggfs. Geschäftsprozesse) gibt. Diese beiden Themen sind zwar wichtig und wesentlich – aber sie sind nichts Neues! Man kann exakt dieselben Punkte auch für Komponenteninfrastrukturen anbringen. Lange Rede, kurzer Sinn: Wenn man über SOAs reden will, kommt man ohne die Diskussion von Komponenteninfrastrukturen nicht herum. Diesen Punkt werde ich später nochmals aufgreifen.

Abbildung 1: Schichtenmodell einer SOA

Abstraktion

Um komplexe Systeme – wie bspw. SOAs – über einen (hoffentlich) langen Zeitraum am Leben zu halten sind einige Voraussetzungen essentiell:

1. Bei der Implementierung von Anwendungslogik will man nicht ständig Artefakte der Laufzeitplattform oder der Datenformate zu sehen bekommen.
2. Die Anwendungslogik muss testbar bleiben – testbar heißt insbesondere, dass man Blöcke von Funktionalität *ohne die technische Infrastruktur* testen kann.
3. Man braucht zwangsläufig eine gewisse Unabhängigkeit von der konkreten Implementierungstechnologie, da diese – wie wir alle wissen – sehr kurzen Änderungs-/Weiterentwicklungszyklen unterworfen ist.

Um den Einsatz einer solchen Architektur in der Praxis eines großen, sich verändernden Unternehmens zu ermöglichen, sind noch weitere Punkte wichtig:

4. Die Beteiligten müssen in der Lage sein möglichst missverständnisfrei miteinander zu kommunizieren – eine gemeinsame Sprache, formale Definitionen, und damit die aussagekräftige Verifikation wichtiger Aspekte sind unerlässlich.

5. Eine gewisse Agilität in der Veränderung der *Fachlichkeit* des Systems muss gewährleistet sein. Wenn die Einführung eines neuen Attributes in ein Interface der Koordination von 8 Parteien bedarf, dauert eine solche Änderung drei Wochen – und wird damit unpraktikabel.

6. Schlussendlich muss man auch noch die eine oder andere Lebenswirklichkeit zur Kenntnis nehmen: beispielsweise die, dass Fachabteilungen oft aufgrund von projektspezifischen Randbedingungen nicht willens oder in der Lage sind, zentral vorgegebene Regeln/Tools/Vorgehen/Konventionen einzuhalten.

Meines Erachtens stellen (formale) Modelle einen Weg dar, viele dieser Probleme zu entschärfen (man beachte: ich schreibe nicht „vollständig lösen"). Die Definition passender Modellierungssprachen sowie die automatische Verarbeitung dieser Modelle stellen meines Erachtens den zentralen Pfeiler einer SOA dar (eine Einführung in die Modellgetriebene Entwicklung findet sich in [SV05]).

Um es mit den Begrifflichkeiten der MDA zu beschreiben: man wird eine Art „Architektur-PIM" definieren. In diesem PIM[1] finden sich die wesentlichen Bausteine der SOA – also Dinge wie Services, Komponenten, Systeme oder Quality-of-Service-Constraints. Dieses PIM ist eben gerade *unabhängig* von der konkreten Technologieplattform. Dies ist besonders wichtig, wenn man seine SOA auf Webservice-Standards aufbauen will, da die betreffenden Standards noch nicht vollständig definiert sind bzw. noch wenig Praxiserfahrung mit den betreffenden Technologien vorhanden ist.

Metamodelle

Um Modelle – beispielsweise obiges PIM - erstellen zu können braucht man zunächst Metamodelle. Ein Metamodell definiert konzeptionell die Modellelemente (Sprachbauchsteine), die zur Erstellung eines Modells zur Verfügung stehen. Im UML Metamodell finden sich daher beispielsweise Dinge wie Klasse, Attribut, Operation, Vererbung oder Zustand. In UML Modellen – technisch gesehen, Instanzen des UML Metamodells – findet sich also beispielsweise eine Klasse mit dem Namen Person die ein Attribut mit dem Namen *name* und dem Typ *String* besitzt. Entsprechend wird man nun Metamodelle definieren, die die relevanten Konzepte einer SOA definieren. Um den untersten Layer – also Komponenten und Services – definieren zu können benötigen wir mindestens drei Viewpoints. Diese sind im Folgenden beschrieben.

[1] PIM steht für Platform Independent Model und ist der Begriff der in der MDA verwendet wird, um Modelle zu benennen, die unabhängig von der Implementierung der modellierten Sachverhalte auf einer bestimmten Plattform sind.

Typmodelle

Im Typmodell definieren wir als zentralen Bestandteil den Service. Ein Service hat eine Reihe von Operationen[2]. Diese verwenden Geschäftsdatentypen in ihren Signaturen. Services definieren auch ein Protokoll (oft mittels einer Protokollzustandsmaschine) um vorzugeben, wie gültige Interaktionen mit dem Service aussehen. Komponenten können nun mittels ProvidedPorts Services anbieten und mittels RequiredPorts Services nutzen. Bei dieser Nutzung ist anzugeben, ob sie synchron oder asynchron (und wenn letzteres, wie genau) verwendet werden sollen. In dieser Sicht auf die Welt stellen Services Interaktions-Verträge dar. Komponenten kapseln Verhalten um solche Services zu implementieren oder zu nutzen. Da komplexe Datenstrukturen oft als XML Schema definiert werden habe ich diesen Teil des Metamodells einfach weggelassen und mit dem „Proxy" XSD versehen.

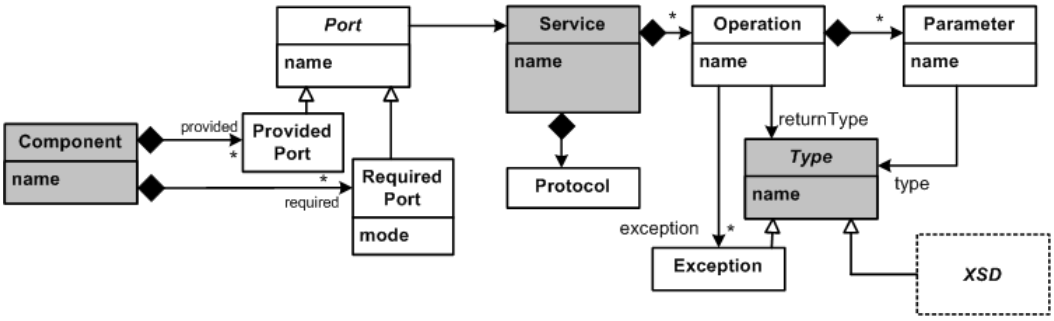

Abbildung 2: Logisches Komponentenmodell

Man beachte dass man aus diesem Modell nun alles generieren kann was nötig ist um die einzelnen Komponenten zu mit der Middlewareinfrastruktur zu verbinden. Die Strukturen in diesem Metamodell entsprechen übrigens sehr gut den in WSDL 2 definierten Abstraktionen, insofern ist eine Transformation in/von diesem Format recht einfach. Was bleibt ist die Implementierung der Geschäftslogik; diese wird üblicherweise von Hand durchgeführt. Dazu muss ein Programmiermodell festgelegt werden welches Technologieunabhängig ist. Mit anderen Worten: Im Implementierungscode darf man nichts verwenden, was irgendwelche Abhängigkeiten zu einer bestimmten Technologie bringen würde. Dies ermöglicht auch die leichte, infrastrukturunabhängige Testbarkeit der Komponenten.

[2] Man kann statt des auf Operationen basierenden Metamodells auch ein Nachrichtenorientiertes verwenden. Dann wären Services definiert als Mengen von Nachrichten, die in vorgegebenen Interaktionsmustern (*Request/Reply, Publish/Subscribe* oder andere Protokolle) verwendet werden dürfen.

Kompositionsmodelle

Der zweite Viewpoint dient nun dazu Instanzen der oben definierten Komponenten zu definieren und diese miteinander zu verbinden. Eine solche Konfiguration definiert nun also eine (Use-Case spezifische) Verdrahtung von Komponenteninstanzen die zusammenarbeiten. Konnektoren verbinden dabei einen ProvidedPort mit (einem oder mehreren) RequiredPorts. Dabei gibt es natürlich ein paar im Diagramm nicht gezeigt Constraints: Beispielsweise müssen zwei miteinander verdrahtete Ports mit den gleichen Services (oder allgemeiner: mit *kompatiblen* Services) verbunden sein.

Aus einem Composition-Modell kann an sich noch kein weiterer Code generiert werden, da noch keine Deploymentinformationen vorhanden sind. Diese zu liefern ist Aufgabe des dritten Viewpoints. Es können allerdings durchaus aussagekräftige und wichtige Verifikationen durchgeführt werden. Dazu zählt die Vollständigkeit eines Systems (Sind alle *RequiredPorts* mit einem *ProvidedPort* verbunden?) oder die Kompatibilität zwischen Versionen (Ist das System noch korrekt wenn ich eine neue Version der Komponente X einsetze?) – für letzteres ist es allerdings nötig, formal zu definieren, was „kompatibel" bedeutet.

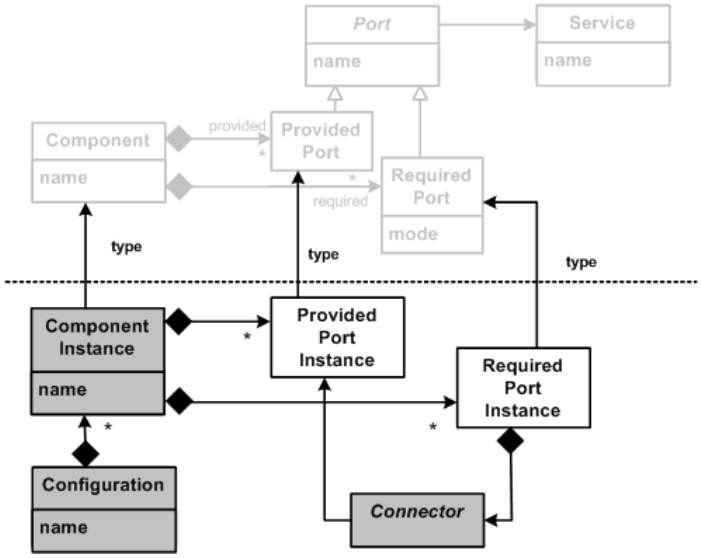

Abbildung 3: Instanzen, Konfigurationen und Konnektoren

Der Ansatz funktioniert übrigens genauso gut in Umgebungen wo zur Zeit des Modellierens noch nicht fest steht, mit welcher Komponenteninstanz (Serviceprovider) ein RequiredPort verdrahtet werden soll. In diesem Fall gibt man im Konnektor eben nicht die ID der Zielinstanz an, sondern Kriterien, nach denen die Instanz zur Laufzeit dynamisch gefunden werden kann: Interface, QoS Kriterien oder Middleware-Partition. Daraus wird dann (Konfigurations-)Code generiert, der zur Laufzeit einen passenden Serviceprovider

findet. Natürlich wird dann zur Laufzeit eine Service-Registry benötigt (CORBA Naming oder Trading, UDDI, etc.)

Deploymentmodelle

Der dritte essentielle Viewpoint ist das Deployment. Komponenten und ihre Implementierungen, sowie deren Instanziierung und Verdrahtung hilft uns noch nicht wirklich viel, denn was noch fehlt ist die Abbildung auf Systeme – also Prozesse, Hardware, Netzwerke.

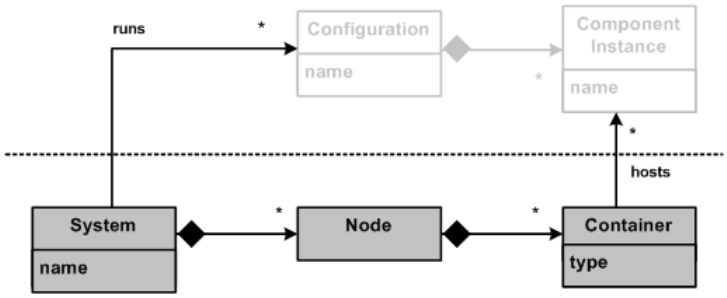

Abbildung 4: Das Deployment: System, Nodes und Container

Genauer gesagt beschreibt das Modell Systeme die aus mehreren Knoten (Maschinen) bestehen. Auf jeder dieser Maschinen können ein oder mehrere Container laufen. Ein Container beherbergt Komponenteninstanzen und stellt (mehr oder weniger mächtige) technische Dienste zur Verfügung. Im Modell hat jeder Container einen Typ: Containertypen wären bspw. CORBA, EJB oder Spring.

Ausgehend von diesem Modell – zusammen mit den anderen beiden bereits eingeführten Modellen – lassen sich komplette Systeme generieren. Dies umfasst

- Glue Code der die Komponentenimplementierung auf dem betreffenden Container ausführbar macht und an die entspr. Middleware anbindet.
- Konfiguration der Container, insbes. die Konfiguration für die SOA Infrastruktur zwischen den Containern auf den verschiedenen Knoten
- Buildskripte die den Bau und das Packaging der Komponenten/Systeme übernehmen

Natürlich haben wir auch in diesem Metamodell einige Dinge vereinfacht: beispielsweise haben wir Netzwerkverbindungen zwischen den Knoten nicht explizit beschrieben und auch nicht gesagt, dass wir einzelne Konnektoren über bestimmte Protokolle (CORBA, Webservice, Tibco) routen wollen. Auch haben wir nicht zum Ausdruck gebracht, dass Komponenten möglicherweise bestimmte technsiche Dienste vom Container benötigen und daher bestimmte Komponenten nicht auf bestimmten Containertypen laufen können.

Bevor wir uns um die höheren Schichten einer SOA kümmern sei noch eine Sache angemerkt: Um zu ermöglichen, dass wir die gleichen Komponenten in verschiedenen Compositions verwenden können, und um diese wiederum auf verschiedenen Systemen deployen zu können muss man sicherstellen, dass die Abhängigkeiten nur in die richtige Richtung zeigen (siehe Abbildung 5). Ein Blick auf die Metamodelle oben zeigt, dass dem auch so ist.

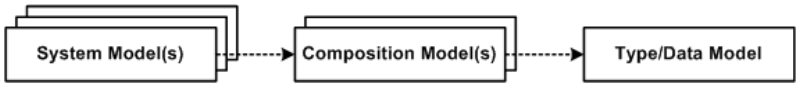

Abbildung 5:Abhängigkeiten zwischen den Modellen

Zwischenergebnis

Die oben gezeigten Metamodelle erlauben es uns – nachdem man sie noch um viele Details erweitert hat damit das ganze in der Praxis funktioniert – saubere komponentenbasierte Software zu beschreiben (siehe Punkt 4) . Die Komponenten tauschen ihre Daten mittels wohl definierten Interfaces aus. Komponenteinstanzen leben in Containern die sich um die technischen Aspekte des Systems (Security, Lastverteilung, Failover, Persistenz) kümmern. Aus den Modellen lassen sich alle relevanten technischen Artefakte (Glue Code, Konfigurationsfiles, Build- und Packagingskripte, etc) generieren. Die Anwendungslogik entwickeln wir weiterhin von Hand (Punkt 3). Das Programmiermodell bleibt dabei unabhängig von den gewählten Technologien (Punkt 1). Testbarkeit wird gewährleistet unter anderem dadurch, dass man die Komponenten – sie sind ja technologiefrei – auch lokal in der IDE per Testframework (bspw. JUnit) testen kann (Punkt 2).

Weitere Herausforderungen und einige Lösungen

Um die Agilität im Zusammenhang mit der fachlichen (Weiter-)Entwicklung zu gewährleisten (Punkt 5 und 6) sind einige Dinge zu beachten – auch hier können Modelle Vorteile bringen.

Datenstrukturen und ihre Besitzer

Ein sehr zentraler Aspekt ist die Frage, wem Datenstrukturen gehören und wer sie verwenden darf. Wenn man versucht, sich in einem großen Unternehmen auf ein zentrales „Geschäftsobjektmodell“ zu einigen dann dauert das in aller Regel ewig – meist einigt man sich nicht wirklich. *Falls* man sich jedoch einigt hat das zwei Dinge zur Folge:

- Zum einen wird man die Datenstrukturen nie wieder (in endlicher Zeit) ändern können falls sich (bei einem der nutzenden Services) der Bedarf ergibt.
- Zum anderen sind solche Datenstrukturen üblicherweise groß und komplex, da sie die Bedürfnisse der verschiedenen Beteiligten erfüllen müssen.

Die Arbeit mit diesen Datenstrukturen ist also nicht besonders angenehm und außerdem sind die Strukturen änderungsresistent – von Agilität keine Spur.

Eine (extreme) Lösung dieses Problems besteht darin, Datenstrukturen direkt dem Service zuzuordnen, sodass sie nur von Anbietern und Nutzern des betreffenden Service verwendet werden können. Damit müssen sich auch nur die direkt am Service beteiligten Parteien auf Änderungen einigen. Im einfachsten Fall legt man fest, dass der Serviceanbieter den Service samt Datentypen definiert, und die Konsumenten sehen müssen wie sie damit klarkommen. Ein Nachteil ist natürlich, dass sich in großen Firmen dann wahrscheinlich 20 ähnliche, aber nicht gleiche Datenstrukturen *Kunde* entstehen. Ein in der Praxis ganz nützlicher Kompromiss besteht darin, ein weiteres Konzept – ich nenne es hier Domäne – einzuführen.

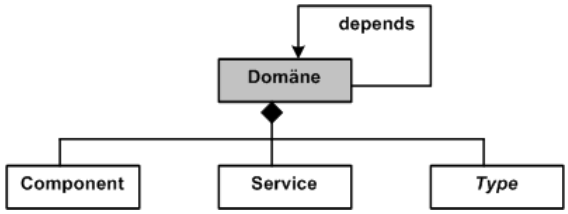

Abbildung 6: Subsystem als Besitzer der anderen Artefakte

Grundsätzlich ist die Sichtbarkeit von Datenstrukturen sowie von Services und Komponenten auf die Domäne beschränkt. Wenn ein Service aus Domäne A einen Datentyp von Domäne B verwenden möchte, muss eine *depends* Beziehung zwischen Domäne A und B erstellt werden. Damit sind Abhängigkeiten zwar nicht aufgehoben; sie sind aber zumindest explizit im Modell beschrieben, was den Umgang mit ihnen erheblich vereinfacht (Erkennung von Abhängigkeiten, verbieten unerlaubter Abhängigkeiten mittels Constraints).

Typisierung der Datenstrukturen

Um das Problem mit der Abstimmung der Datentypen noch weiter zu entschärfen, bietet es sich an, die Datentypen technisch so zu definieren, dass sie interpretativ verarbeitet werden und nicht mittels statisch generierten Codes. In diesem Zusammenhang wird auch der Vorteil von XML gegenüber z.B. Java Serialisierung oder CORBA Structs deutlich: wenn sich die Definition der Datenstruktur weiterentwickelt, bricht das den in Java oder IDL definierten Vertrag, es kommt zu einem Laufzeitfehler. Wird die Datenstruktur interpretiert (also bspw. mit einem XML Parser) so können fehlende Attribute/Substrukturen mit Defaults belegt werden und neue (aus Sicht eines alten Service überflüssige) Attribute/Substrukturen werden einfach ignoriert.

Wiederverwendung von Services

Der Aufbau einer SOA ist ja oft mit dem Drang zur Vereinheitlichung und Standardisierung verbunden. Ein solches Unterfangen ist auf der Metaebene durchaus sinnvoll – siehe die Metamodelle oben. Auf dem Level der konkreten Artefakte – bspw. der Service-Definition – wird es aber schon erheblich schwieriger. Angenommen, sie wollen sich konzernweit auf einen Service einigen, der zu einer Kunden-ID die Kundendaten liefert. Dann haben sie dabei zum einen das Problem, dass Sie sich darauf einigen müssen, was denn *die Kundendaten* sind – sie müssen also die Datenstruktur standardisieren (Problem siehe letzter Abschnitt). Sie haben aber auch noch ein anderes Problem. Verschiedene Use-Cases haben verschiedene Anforderungen an Antwortzeiten, Durchsatz oder andere Aspekte der Quality of Service. Beispiele:

- Das Call-Center benötigt die Daten bspw. extrem schnell (der Kunde ist am Telefon …), allerdings werden nur wenige Daten über den Kunden benötigt (die weiteren Dinge werden bei Bedarf nachgeladen).
- Andere Use-Cases brauchen mehr Daten, sind daher auch bereit, etwas längere Abfragezeiten zu tolerieren.

Selbst wenn sich also die verschiedenen Parteien auf gemeinsame Datenstrukturen einigen *könnten*, gäbe es bzgl. der nötigen Performance und anderer Nichtfunktionaler Anforderungen mit Sicherheit unterschiedliche Anforderungen.

Dieses Beispiel zeigt, dass der Versuch, Services Enterprise-weit zu standardisieren in der Praxis oft nicht klappen wird. Die Weiterentwicklung – als Varianten- und Versionsbildung über die Zeit – ist ein weiteres nicht zu unterschätzendes Problem. Wenn man dieses Thema systematisch angehen will, muss man die verschiedenen Services als Produktlinie betrachten und die Varianten explizit managen. Featuremodelle können die nötige Variabilitätsanalyse dabei deutlich vereinfachen. Insbesondere lassen sich damit auch die Datenmodelle und Schnittstellen bzgl. ihrer Variabilität beschreiben. Wie üblich kann durch Generierung aus diesen Modellen die Implementierung der Services deutlich vereinfacht werden. Die folgende (zugegebenermaßen etwas unübersichtliche) Abbildung zeigt ein Beispiel, in dem die Datenstrukturen abhängig von den gewählten Features modifiziert werden (für eine detailliertere Erläuterung von Featuremodellen siehe [CE00])

- Der Service *GetCustomerInfo* liefert defaultmäßig nur die Kundenidentität (*Customer*) sowie seine Basisdaten und die Rechnungsadresse.
- Optional kann man das Merkmal *ShippingAddress* auswählen, was zur Folge hat, dass nun die Assoziation zur Lieferaddresse „aktiviert“ wird – sie betreffenden Daten sind nun also im Ergebnis des Aufrufs enthalten.
- Optional kann man Rechnungsinformationen erhalten (*Billing*). Man muss sich dann dafür entscheiden, ob man nur das *CreditRating* benötigt, oder die gesamte Rechnungshistorie. Beides zugleich ist nicht möglich.

- Man kann sich zusätzlich Daten über die offenen Aufträge liefern lassen. Dabei kann man optional auch die einzelnen Bestellpositionen bzw. die Rechnung miterfragen.

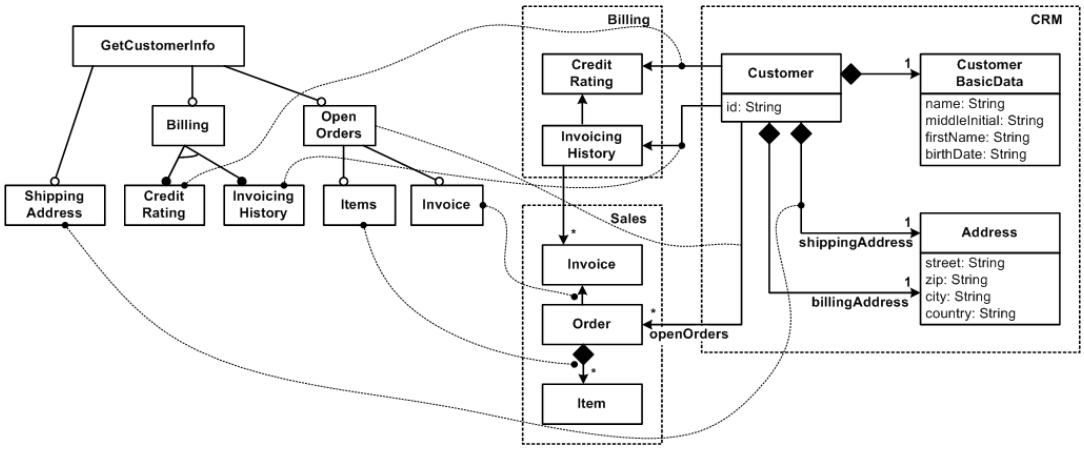

Abbildung 7: Servicevarianten und deren Datenstrukturen

Aus diesen Modellen kann man nun entweder verschiedene statische Serviceinterfaces und Datenstrukturen generieren oder (besser!) die entsprechenden XML Schemata für die Datenstrukturen der betreffenden Services (bzw. Servicevarianten). Durch Annotation der Features in der linken Hälfe des obigen Diagramms mit Kosten kann auch etwas über die Performance der Servicevariante ausgesagt werden.

Prozess: zentral vs. dezentral

Der Drang zur Zentralisierung hat meistens auch andere Auswirkungen. Oft wird bspw. ein zentrales Service Repository angestrebt. Oft ist das eine Datenbank. Das bedeutet, dass Entwickler die Services definieren oder implementieren wollen mit der zentralen Datenbank kommunizieren müssen. Dies ist zum einen schlicht unpraktisch, da man dann immer online arbeiten muss. Zum anderen ist dies aber oft auch ein Prozess-Problem. Wenn ich, um im Rahmen einer iterativen Entwicklung – den Service ändern will und ich dabei sofort mit Enterprise-weiten Standards und Prozessen konfrontiert bin, nimmt die Produktivität deutlich ab. Es ist also essentiell, dass man – technisch gesehen – offline arbeiten kann.

Wichtig ist auch, dass man, um Services zu definieren und zu implementieren mit niemand anderem reden muss als mit den am Service beteiligten (Provider/ Konsument(en)). Erst wenn man den Service in Produktion bringen will (also betrieben vom Rechenzentrum, skalierbar, ausfallsicher, etc). möchte ich den Service „einchecken“. Hier wird nun auch überprüft in wiefern mein neuer Service irgendwelchen Konzernrichtlinien entspricht. Man benötigt also ein Status/Prozessmodell, bei dem

Servicedefinitionen bestimmten Kriterien entsprechen müssen wenn sie von „ausprobieren“ in „Produktion“ gehen sollen.

Infrastruktur vs. Anwendungsentwicklung

In vielen Konzernen werden SOAs von einer Zentralabteilung initiiert mit dem Ziel, die IT Infrastruktur zu vereinheitlichen, und diese besser managen und monitoren zu können. Eine Folge dessen ist oft ein Fokus auf Middleware und Infrastruktur-Technologien (sowie ein starker Drang zur Zentralisierung, s.o.). Diese Ziele entsprechen nun leider gar nicht den Zielen der Fachabteilungen (bzw. der von ihnen gesponserten IT Projekte). Deren Ziel ist es, pragmatisch in kürzester Zeit Anwendungen in Produktion bringen. Aus dieser Situation entsteht fast immer ein Zielkonflikt: Die Fachabteilungen sind nicht willens, den Vorgaben der zentralen IT (oder wem auch immer) zu folgen, da sie die Vorteile für sich nicht sehen. Und um es klar zu sagen: dieser Standpunkt ist völlig legitim!

Man muss also dafür sorgen, dass eine SOA für die Anwendungsentwickler spürbare Vorteile bringt. Aus Prozesssicht bedeutet das, dass die Prozesse die Entwickler nicht bei der Arbeit behindern dürfen (siehe voriger Abschnitt). Es bedeutet aber auch, dass die technische Infrastruktur beherrschbar sein muss. Man darf sich bei der Einführung einer SOA also nicht nur auf die Infrastruktur beschränken („ESB von Firma X ist nun Standard“) sondern muss insbesondere eine gute Toolunterstützung für die Anwendungsentwicklung bieten.

Die Chance dazu ergibt sich, wenn man konsequent modellgetrieben arbeitet fast von selbst: Wenn man eine schöne IDE baut, die basierend auf den Metamodellen und den dazu definierten, technologiefreien Programmiermodellen die Entwicklung vereinfacht, ist man schon einen großen Schritt weiter. Man kann typischerweise fast alle Aspekte der Anbindung an die SOA automatisieren und den Anwendungsentwicklern die Arbeit damit wesentlich erleichtern.

Dieses Vorgehen ist natürlich besonders geeignet bei der Implementierung neuer Services. Die Integration von Legacy Systemen lässt sich aber mit denselben Mitteln bewerkstelligen:

- Ein Ansatz besteht darin, die Serviceschnittstelle mit Hilfe der oben erläuterten Modelle zu definieren und mit Hilfe der gerade erwähnten IDE die entsprechende Infrastruktur zu generieren. Die Anbindung an den Legacy geschieht dann durch manuelle Programmierung im Rahmen der Komponentenimplementierung.
- Der andere Ansatz besteht in der automatischen Generierung von Services und Komponenten basierend auf den Schnittstellen der Legacysysteme. Oft müssen dabei noch zusätzliche Modelle definiert werden, die beschreiben, wie bestimmte Legacy-Systeme auf Serviceschnittstellen abgebildet werden können.

Das Spaghetti-Missverständnis

Das folgende Bild kennen Sie wahrscheinlich alle. SOA löst alle ihre Infrastrukturprobleme, weil sie nun einen Enterprise Service Bus (ESB) einsetzen, an den Sie ihre Anwendungen andocken. Point-to-Point Verbindungen und der Infrastruktur-Wildwuchs gehören damit der Vergangenheit an.

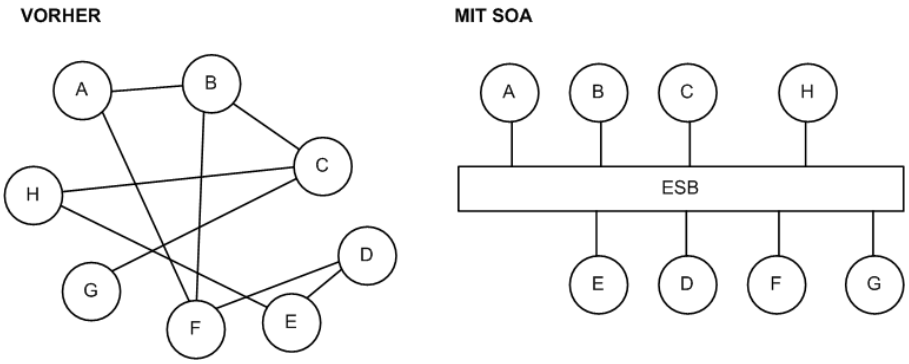

Abbildung 8: Spaghetti

Leider ist auch das nicht ganz so einfach. Mal abgesehen von der Tatsache, dass jeder Hersteller den Begriff ESB anders definiert (und demnach unterschiedliche Features anbietet), ist es unrealistisch und auch unvernünftig alles über die gleiche Middleware abwickeln zu wollen. Wie auch bei der Deploymentplattform wird man je nach benötigter QoS verschiedene Transportprotokolle einsetzen. Innerhalb eines Konzerns werden oft Infrastrukturen wie bspw. die Produkte von Tibco eingesetzt. Zwischen verschiedenen Unternehmen macht sicherlich eine offene, auf Webservices basierende Kommunikationsinfrastruktur Sinn.

Entscheidend ist also *nicht* die Tatsache, dass man sämtliche Kommunikation über dieselbe Middleware abwickelt, sondern, dass man in der Lage ist, potentiell jeden mit jedem reden zu lassen: man muss also die Services auf Modellebene – technologiefrei! – beschreiben, sodass man sie auf verschiedene, für die QoS Anforderungen passende Infrastrukturen abbilden kann. Dass es dabei sinnvoll ist, sich konzernweit auf eine begrenzte Anzahl zu beschränken ist natürlich auch klar.

Dieses Vorgehen erlaubt es insbesondere auch, Serviceprovider und –Consumer im selben Prozess laufen zu lassen (wenn sie mit der gleichen Technologie implementiert sind). Damit können performanzkritische Interaktionen ohne die Beteiligung einer Middleware umgesetzt werden. Auch funktionale Tests werden vereinfacht, weil man alle Komponenteninstanzen die für fachlichen Test notwendig sind in einem Prozess (innerhalb der IDE) laufen lassen kann.

Business Process Management

Geschäftsprozesse sind Abläufe, die in aller Regel lange laufen (Stunden, Tage, Wochen, Monate). Die Ausführung eines solchen Geschäftsprozesses schließt in aller Regel den Zugriff auf Services ein. Geschäftsprozesse lassen sich in vielen Formen beschreiben, die einfachste (wenn auch nicht immer ausreichende) Beschreibungsform sind Zustandsmaschinen, eine andere weit verbreitete Notation ist BPEL. Um Geschäftsprozesse flexibel zu halten bietet es sich an, deren Definition interpretativ zu verarbeiten und nicht durch Codegenerierung starre Strukturen zu erstellen (man generiert dann aus den graphischen Beschreibungen eben die vom Interpreter zu verarbeitende Spezifikation).

Integration mittels Prozess-Komponenten

Es gibt zwei grundlegend verschiedene Möglichkeiten die Geschäftsprozesse mit den Services bzw. den Komponenten zu verbinden. Die eine besteht darin einen speziellen Komponententyp einzuführen, dessen Implementierung mittels einer Zustandsmaschine (oder einer anderen der sinnvollen Prozessbeschreibungssprachen) definiert wird. Hier das entsprechend erweiterte Metamodell.

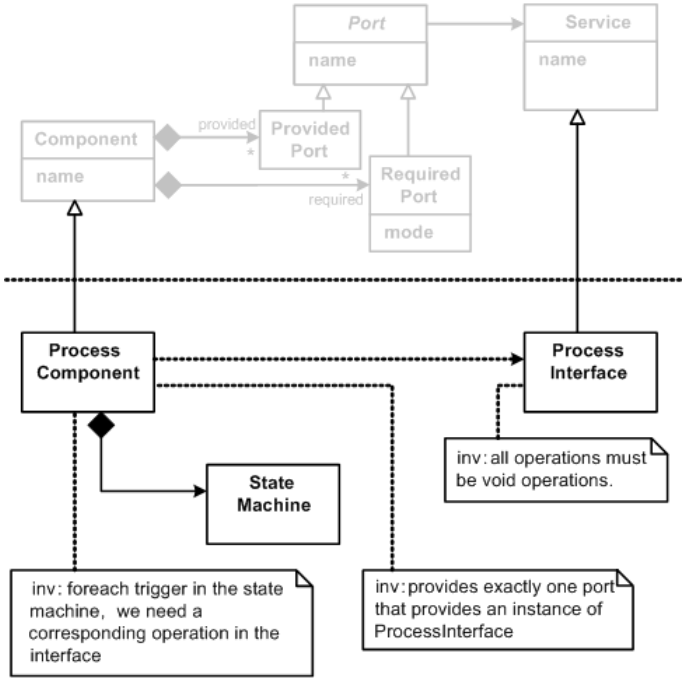

Abbildung 9: Das Metamodell erweitert für BPM

Prozesskomponenten sind zunächst ganz normale Komponenten im Rahmen der SOA. Sie können also insbesondere Abhängigkeiten zu anderen Services haben, sie können

instanziiert, verdrahtet und deployed werden. Allerdings gibt es auch einige Besonderheiten: sie bieten genau einen Service an der für jeden Trigger der Zustandsmaschine genau eine Operation besitzt. Da Zustandsmaschinentrigger immer asynchron sind, müssen die Operationen *void*-Operationen sein. Mittels dieser Operationen können Clients Events in den Prozess „einspeisen". Im Rahmen von Actions oder Guards – die man bspw. mittels manuell zu schreibendem Code implementiert – kann auf andere Services zugegriffen werden. Das folgende zeigt ein Beispielmodell, hier komplett in UML dargestellt. Zu beachten ist, wie aus der Operation die die *sendReminder()* Action implementiert, mittels des RequiredPorts auf andere Services zugegriffen wird.

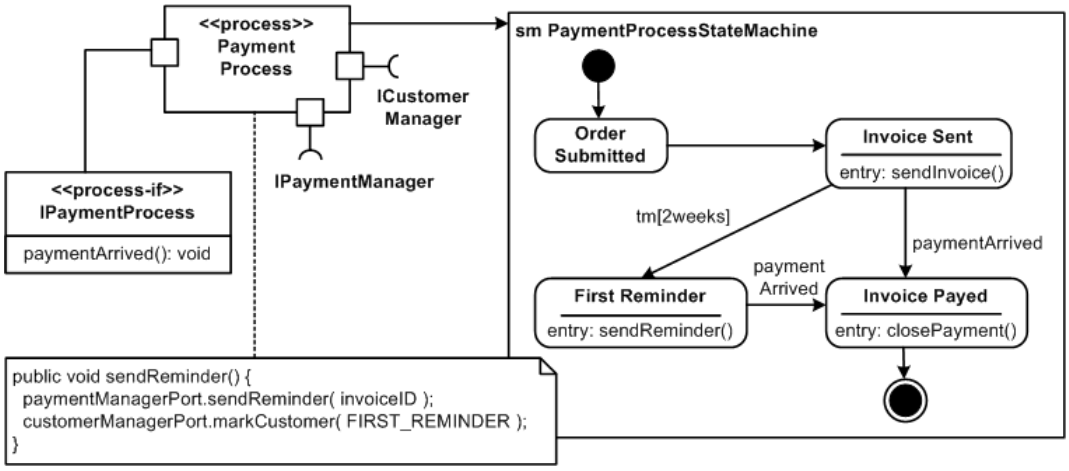

Abbildung 10: Beispielmodell für eine Prozesskomponente

Integration mittels separater Prozessschicht

Eine andere Möglichkeit zur Integration von Geschäftsprozessen besteht darin, die Prozessschicht komplett von der Serviceschicht zu trennen. Dabei werden die Prozesse nicht mittels Komponenten implementiert sondern mittels 3rd-Party Werkzeugen zur Geschäftsprozessmodellierung und –Ausführung. Diese Werkzeuge bieten in aller Regel komfortable Editoren zur Definition von Geschäftsprozessen sowie skalierbare Engines um die Prozesse auszuführen. Adapter für verschiedene Middlewaretechnologien erlauben den Zugriff auf Services. Auch in diesem Fall spielen Modelle – die Definition der Prozesse eben – die zentrale Rolle.

Hauptgrund für die Verwendung dieses Ansatzes ist, dass die Geschäftsprozesse dadurch von Mitarbeitern der Fachabteilung leichter zu entwickeln und zu verändern sind sowie die Annahme dass sich die Geschäftsprozesse öfter ändern werden als Services. Services und ihre Implementierung ist sozusagen Sache der IT und stellt das stabile Fundament der SOA dar. Die Definition der darauf aufbauenden Prozesse wird (mehr oder weniger) direkt von der Fachabteilung durchgeführt.

Technologien und Standards

Wie sie sicher schon bemerkt haben, will ich in diesem Artikel eben genau nicht über bestimmte Technologien für die Implementierung einer SOA sprechen; der Punkt ist ja gerade, dass die Konzepte wichtiger sind als die konkrete Technologie. Damit will ich natürlich nicht zum Ausdruck bringen, dass die Wahl der Technologie irrelevant ist – die Technologie bestimmt letztendlich zum Gutteil wie performant, skalierbar, offen und interoperabel die SOA sein wird. Nichtsdestotrotz ist die Wahl der Technologie der zweite Schritt, nicht der primäre. Ich möchte daher zum Abschluss doch noch ein paar Worte zu Technologien sagen.

Zunächst wäre da die Frage der Middleware. Natürlich liegt es nahe, für die Teile der SOA, bei denen Interoperabilität wichtig ist Webservices zu verwenden. Eine Abbildung der obigen Modelle auf WSDL ist straight forward, insbesondere, wenn man zur Datenstrukturbeschreibung sowieso schon auf XML Schema setzt. Andere Technologien wie HTTP/Rest-basierte Protokolle, .NET Remoting, CORBA oder JMS können allerdings genauso Sinn machen. Da diese vom impliziten Metamodell her alle ähnlich sind, ist eine automatische Generierung der nötigen Artefakte auch hier kein besonders großer Aufwand.

Ein zweiter Aspekt ist die Komponentenplattform. Dort bietet sich in der Java Welt der alte Bekannte J2EE an. Auch Spring ist ein interessanter Kandidat, wobei man einschränkend sagen muss, dass die Vorteile des schöneren Programmiermodells von Spring hier nicht so stark zur Geltung kommen, da man die Technologieabbildung durch Generierung automatisiert und man das Programmiermodell selbst festlegt. Ein an Spring angelehntes Programmiermodell macht – aufgrund seiner wenig invasiven API – allerdings durchaus Sinn. In der .NET Welt ist WCF (aka Indigo) ein heißer Kandidat.

Zur Beschreibung von Geschäftsprozessen gibt es neben der UML natürlich noch eine ganze Reihe von Alternativen: BPEL, BPMN, etc.

In der letzten Zeit haben sich in der Java-Welt zwei weitere interessante Standards entwickelt (bzw. entwickeln sich noch): JBI und SCA[3]. Dabei ist es interessant zu sehen, dass sich JBI mehr der Infrastrukturseite widmet, während SCA eher einen programmiermodell-zentrierten Ansatz verfolgt.

JBI (Java Business Integration) ist dabei quasi eine "Middleware Middleware" (siehe [JBI]). JBI bietet eine einheitliche Sicht auf verschiedene Kommunikations-Middlewares in dem es die Kommunikation auf so genannte Normalized Messages abbildet und diese zwischen den Komponenten eines JBI Containers hin und her routet (daher als zentrale Komponente der so genannte Normalized Message Router, NMR). Die Komponenten in einem JBI

[3] Streng genommen ist SCA kein Java-spezfischer Standard, obwohl de-facto ein Schwerpunkt auf Java liegen wird.

Container werden untergliedert in Service Engines und Binding Components. Erstere implementieren Geschäfts- oder Transformationslogik, letztere dienen zur Anbindung externer Systeme (oder anderer JBI Container), sie dienen also als Protokolladapter. Services werden mittels WSDL (genauer: den Abstract Message Definitions von WSDL 2) definiert. Aus meiner Sicht fehlt bei JBI leider der Systemgedanke. Es gibt keine ausreichenden Beschreibungsmöglichkeiten für die Aspekte, die ich oben mittels der Kompositions- und Deploymentmodelle beschrieben habe.

Bei der SCA (Service Component Architecture) sieht das etwas anders aus. Die Service-Implementierungs-Komponenten können in verschiedenen Sprachen definiert werden. Daraus werden dann Interface-Beschreibungen abgeleitet, die sich auf verschiedene Kommunikationstechnologien abbilden lassen. Mittels verschiedener XML Dateien lassen sich diese Komponenten dann instanziieren und mit einander mittels Konnektoren verbinden. Weiterhin lassen sich Subsysteme und Systeme definieren, die aus verschiedenen Maschinen und Netzwerkverbindungen bestehen. Was mir an SCA sehr gut gefällt ist die Tatsache, dass man die obigen drei Viewpoints (Logische Definition, Komposition, Systemmodell) wieder finden kann. Im Ergebnis bekommt man also eine wirkliche *System*sicht. Der Standard ist noch nicht final, es gibt aber bereits Implementierungsprojekte bei Apache (Tuscany, siehe [TUS]) und bei Eclipse (STP, siehe [STP]). SCA ist also definitiv etwas, das man weiter beobachten und verfolgen sollte.

Zusammenfassung

SOA ist zunächst mal ein Konzept, ein Architektur-Blueprint. Es hat zunächst nichts mit (Kommunikations-)Middleware zu tun. Um die nötige Agilität zu erzielen, ist es vorteilhaft die entsprechenden Services, Datenstrukturen, Komponenten sowie deren Deployment mittels Modellen zu beschreiben. Diese dürfen sie in Ihrem Konzern auch ruhig selbst definieren – sie sind schließlich Ihr zentrales Asset! Dabei gehen diese Modelle weit über eine Schnittstellenmodellierung bzw. die Beschreibung von Endpoints (wie es WSDL ermöglich) hinaus. Basierend auf den Modellen die die verschiedenen Viewpoints beschreiben, lassen sich mit erträglichem Aufwand die nötigen Artefakte zum Betrieb einer SOA generieren. Dabei setzen sie natürlich 3rd-Party Middleware und Komponentenplattformen ein. Durch die daraus folgende Kapselung und Automatisierung der Technologieanbindung ist eine Migration auf sich entwickelnde SOA Standards und Protokolle mit wenig Aufwand möglich – ein wichtiger Aspekt in Zeiten sich rasant ändernder Technologien. Außerdem bleibt das Programmiermodell weitgehend frei von Technischen Aspekten und macht die Serviceimplementierung damit sehr effektiv.

Dank an Nicolai Josuttis, Arno Haase, Alexander Schmid und Eberhard Wolff für das Feedback zu dem Artikel!

Referenzen

[SV05] Stahl, Völter, *Modellgetriebene Softwareentwicklung*, dPunkt Verlag, 2005

[CE00] Czarnecki, Eisenecker, *Generative Programming*, Addison-Wesley, 2000

[JBI] Java Community Process, *Java Business Integration*, http://www.jcp.org/en/jsr/detail?id=208

[SCA] IBM u.a., *Service Component Architecture*, http://www.ibm.com/developerworks/library/specification/ws-sca/

[TUS] Apache Software Foundation, *Tuscany Project*, http://wiki.apache.org/incubator/TuscanyProposal

[STP] Eclipse Foundation, *SOA Tools Platform*, http://www.eclipse.org/stp/