SOA revisited: Komponentenorientierung bei Systemlandschaften

Der Autor

Johannes Siedersleben

Prof. Dr. Johannes Siedersleben
Leiter Competence Center System Design
T-Systems Enterprise Services GmbH
Dachauer Str. 651, 80995 München
johannes.siedersleben@t-systems.com

Eingereicht am 2006-08-15,
nach zwei Überarbeitungen
angenommen am 2006-11-14
durch Prof. Dr. E. Sinz.

1 Einleitung

Die Erwartungen an SOA sind hoch: SOA macht komplexe Systemlandschaften beherrschbar, spart Kosten und befreit uns von den Fesseln monolithischer Softwaresysteme. Für jedes Unternehmen, jedes Softwarehaus ist SOA bedeutsam – für die Unternehmen, weil sie ihre Systemlandschaften endlich beherrschen wollen, für die Softwarehäuser, weil sie für ihre Kunden attraktiv sein müssen.

Aber SOA ist nicht die erste Heilslehre. Themen wie 4GL-Sprachen, CASE-Tools, Objektorientierung, Komponentenorientierung und MDA erlitten alle das gleiche Schicksal: Einer Phase der Euphorie folgte die Erprobung, danach die Ernüchterung und schließlich die Besinnung auf einen nützlichen Kern, der dann auch lange Bestand hat. Auf diesem Weg befindet SOA sich je nach Standpunkt irgendwo zwischen Euphorie und Ernüchterung.

Der Beitrag beschreibt den möglichen Nutzen von SOA, aber auch Fehler und Irrwege, und er trägt bei zur Klärung einiger SOA-Begriffe. Der eingenommene Standpunkt ist immer der des Softwarearchitekten: Was bedeutet SOA für die Architektur eines einzelnen Systems, was bedeutet es für die Systemlandschaft insgesamt? Die einschlägigen Managementthemen wie SOA-Governance lassen sich erst dann sinnvoll behandeln, wenn einige grundlegende Fragen der Softwaretechnik geklärt sind. Insofern richtet sich der Beitrag zum einen an Manager, die SOA verstehen wollen, aber auch an Techniker, und zwar vor allem an jene, die SOA unzulässigerweise auf bestimmte Technologien wie XML und Web-Services reduzieren.

SOA als schillerndes, unreifes Thema lässt sich nur mit Mühe in eine sequenzielle Form bringen. Wir versuchen es trotzdem, und zwar folgendermaßen: Nach einer kurzen Darstellung der Unterschiede zwischen Systemen und Systemlandschaften legen wir in Kapitel 3 die gedanklichen, begrifflichen und technischen Grundlagen von SOA. Darauf aufbauend befassen wir uns mit SOA in der Praxis: Wir beschreiben den Weg zu SOA (Kapitel 4), geben Hinweise auf mögliche und häufige Fehler (Kapitel 5) und zeigen zum Schluss (Kapitel 6), dass sich der Aufwand trotz allem lohnt.

2 Systemlandschaften

Objekt- und Komponentenorientierung befassen sich mit dem Entwurf eines einzelnen Systems. SOA befasst sich mit dem Entwurf von Systemlandschaften. Dieser hat mit dem Entwurf eines einzelnen Systems viel gemeinsam, denn letztlich kann man jede Systemlandschaft als ein einziges großes System sehen. Aber es gibt drei wesentliche Unterschiede:

a) Systemlandschaften entstehen im Lauf der Zeit und nicht in einem großen Wurf. Insofern ähneln sie unserem Autobahnnetz: Es wird niemals fertig, immer gibt es Baustellen, und Entwurfsentscheidungen der Vergangenheit erweisen sich oft als falsch. Zudem ändern sich sowohl technische als auch fachliche Anforderungen im Lauf der Zeit dramatisch. Aber im Unterschied zu einem einzelnen System kann man eine Systemlandschaft nicht einfach abschalten und durch eine neue ersetzen. Das bedeutet: Man lebt mit Kompromissen; die ideale Welt gibt es nicht. Das gilt selbst dann, wenn man eine gesamte Systemlandschaft auf der grünen Wiese erstellt (was nur selten vorkommt): Systemlandschaften sind riesengroß, und deshalb wird der erste Entwurf mit hoher Wahrscheinlichkeit Fehler enthalten.
b) Systemlandschaften enthalten Redundanz von Daten und Funktionen. Diese Redundanz ist in einem gewissen Umfang unvermeidbar, denn verschiedene Anwendungen sehen die Dinge häufig unterschiedlich: Die Stückliste aus der Sicht der Konstruktion entspricht nur teilweise der Stückliste aus Sicht der Produktion. Aber viel Redundanz ist einfach ungeplant entstanden, und eine der wichtigsten Aufgaben bei der Gestaltung von Systemlandschaften besteht darin, überflüssige Redundanzen zu erkennen und nach Möglichkeit zu eliminieren oder – wenn das nicht möglich ist – zu kontrollieren.
c) Systemlandschaften sind heterogen: Sie bestehen aus vielen Systemen unterschiedlicher Herkunft, die von verschiedenen Teams zu verschiedenen Zeiten mit verschiedenen Werkzeugen gebaut worden sind. Manche Systeme haben eigene Mitarbeiter erstellt, andere kommen von externen Softwarehäusern, und

wieder andere sind Standardprodukte, die manchmal allerdings erheblich vom Standard abweichen. Gerade Standardprodukte sind oft die Ursache für Redundanz, denn sie bringen ihre Daten und Funktionen ohne Rücksicht auf die vorhandenen Anwendungen einfach mit! So kommt es, dass vielleicht vier Kundenstämme vorhanden sind, obwohl zwei genug wären. Mehrfach vorhandene fachliche Daten (wie die vier Kundenstämme) nennen wir *fachliche* Redundanz, im Gegensatz zur *technischen* Redundanz, die als Optimierungsmaßnahme durchaus üblich ist.

Fachliche Redundanzfreiheit und Konsistenz sind die hehren Ziele des Softwarearchitekten – das hat er gelernt, da will er hin. Aber beim Entwurf von Systemlandschaften muss er umdenken: Dort sind Inkonsistenzen und Redundanz keine vorübergehenden, behebbaren Mängel, sondern charakteristische Eigenschaften.

Real existierende Systemlandschaften sind fast ohne Ausnahme historisch gewachsen. Die Redundanz von Daten und Funktionen ist kaum zu kontrollieren, und die Abhängigkeiten zwischen den verschiedenen Systemen sind komplex. Die Situation insgesamt befindet sich oft an der Grenze der Beherrschbarkeit, und so ist es kein Wunder, dass man seit Langem nach Mitteln und Wegen sucht, um das Chaos zu beherrschen und zu bändigen. Das Ergebnis dieser Suche heißt serviceorientierte Architektur, kurz SOA. Aber die Kluft zwischen den Systemlandschaften, die wir heute betreiben, und einer idealen SOA-konformen Welt ist riesengroß.

3 Was ist SOA?

SOA ist eine Methode zum Entwurf von Systemlandschaften. Jede Systemlandschaft ist mehr oder weniger SOA-konform: Abschnitt 3.1 befasst sich mit der Frage der Messbarkeit von SOA-Konformität. SOA bedeutet wörtlich nicht mehr als das Folgende:

Die Elemente eines Systems (Komponenten, Module, Subsysteme o. ä.) kommunizieren über Services.

Diese Definition ist in ihrer Allgemeinheit fast wertlos, zumal der Begriff Service in wohl jedem SOA-Projekt Gegenstand heftiger Diskussionen ist. In [IBM05] findet man die Definition:

A service is a repeatable task within a business process.

Damit kann man im Grunde jede Systemlandschaft als SOA-konform betrachten, denn: Jede Systemlandschaft unterstützt Geschäftsprozesse wie Auftragseingang, Auftragsabwicklung, Beschwerdemanagement in irgendeiner Weise. Jeder Geschäftsprozess läuft als Szenario durch die beteiligten Systeme, und jedes beteiligte System tut dabei etwas – und genau das ist *a repeatable task within a business process*, mithin ein Service.

3.1 Drei Ideen, drei Modelle

Historisch gewachsene Services geben oft Anlass zu Klagen: Sie sind ungeschickt geschnitten, technisch umständlich oder unsicher implementiert, schlecht dokumentiert und insgesamt Ursache des Chaos. An dieser Stelle kommt SOA mit drei Ideen zu Hilfe:

a) *Komponentenorientierung*: Mit SOA kommunizieren alle Systeme über definierte Schnittstellen. Jedes beteiligte System wird als (möglicherweise sehr große) Komponente betrachtet, jedes System sieht seine Partner nur über definierte Schnittstellen und jedes System implementiert selbst eine beliebige Anzahl von Schnittstellen, die es seinen Partnern zur Verfügung stellt.
b) *Lose Koppelung*: Mit SOA werden Systeme entkoppelt. Sie kommunizieren asynchron und reagieren robust auf Fehler oder gar den Ausfall anderer Systeme.
c) *Workflow*: Mit SOA liegt die Ablaufsteuerung in einer eigenen Workflow-Komponente. So ist gewährleistet, dass die einzelnen Systeme möglichst wenig voneinander wissen.

Diese drei Elemente sind die kennzeichnenden Merkmale einer SOA-konformen Systemlandschaft, und sie sind die Grundlage für die Messung der SOA-Konformität. Technologien wie XML oder Web-Services werden in Verbindung mit SOA oft genannt, aber man kann sich eine SOA-konforme Landschaft ohne weiteres ohne Web-Anbindung vorstellen, und XML ist als Format der Datenübertragung sicher der Standard, aber nicht die einzige Option. So gelangen wir zu folgender Definition:

SOA ist Komponentenorientierung in Verbindung mit loser Koppelung und ausgelagerter Ablaufsteuerung.

SOA manifestiert sich in *drei Modellen*: Prozessmodell, Servicemodell und Technikmodell (siehe Bild 1). Das *Prozessmodell* ist natürlich keine Neuerung, denn Geschäftsprozesse beschreiben wir schon lange. Aber ohne genaue Kenntnis der Geschäftsprozesse kann man keine Services definieren. Die Services und ihr Bezug zu den Geschäftsprozessen sind im *Servicemodell* festgehalten. Damit ist festgelegt, was die Systemlandschaft insgesamt leistet. Im Idealfall bestimmt das Prozessmodell das Servicemodell aber nicht umgekehrt. Aber Standardsoftware und Altsysteme nehmen nicht immer Rücksicht auf die vorhandenen Geschäftsprozesse, und deshalb gibt es auch Abhängigkeiten in die andere Richtung.

Das *Technikmodell* oder *plattformabhängige Modell* beschreibt, mit welcher Technik Services zu Verfügung gestellt und implementiert werden. Die Auswahl ist groß, von Sockets und SOAP bis zum Enterprise Service Bus (ESB). Das Technikmodell ermöglicht die effiziente Implementierung der Services; das Servicemodell ermöglicht die effiziente Durchführung der Geschäftsprozesse. So entkoppelt das Servicemodell Anwendung und Technik: Wir ändern einen Geschäftsprozess, um den Kunden besser zu bedienen, aber nicht wegen einer neuen Version von EJB oder .NET.

Über Ideen und Modellen steht die *Governance*. Die Regel sind bisher Großprojekte mit verschiedenen Auftraggebern, die erst weit oben in der Linie zusammenlaufen, und deren fachliche und technische Koordination oft mangelhaft ist. SOA ersetzt schlecht koordinierte Großprojekte durch viele kleine Projekte, die insgesamt das einzig verbleibende Großprojekt *SOA* bilden. Governance ist nichts anderes als die Projektleitung dieses Großprojekts, das im Unterschied zu herkömmlichen Großprojekten schon durch seine Konstruktion in handhabbare Teilprojekte zerlegt ist. In der Governance finden wir alle Rollen der herkömmlichen Projektleitung, insbesondere die des Chefarchitekten.

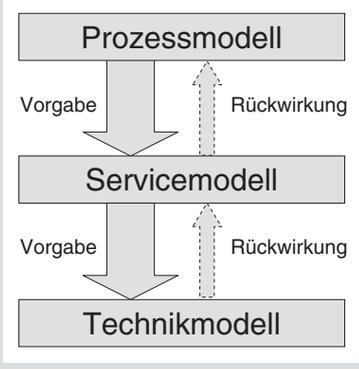

Bild 1 Die drei Modelle von SOA

3.2 Komponentenorientierung

Ein System ist komponentenorientiert, wenn es aus Komponenten zusammengesetzt ist. Jede Komponente stellt mindestens eine Schnittstelle zur Verfügung (*Export*), und sie benötigt eine beliebige Anzahl (größer gleich null) von Schnittstellen (*Import*). Der Komponentenbegriff ist hierarchisch: Eine Komponente kann aus beliebig vielen Komponenten zusammengesetzt sein. Entscheidend ist die Unabhängigkeit von der Implementierung: Der Importeur sieht nur die importierte Schnittstelle, aber macht keine Annahmen über die exportierende Komponente.

SOA bedeutet Komponentenorientierung im Großen. Komponenten und Schnittstellen sind die Grundlage von SOA. SOA ist die Verallgemeinerung der Komponentenorientierung von Systemen auf Systemlandschaften: Jedes System exportiert Schnittstellen zur Verwendung durch die Partnersysteme und es importiert Schnittstellen, die andere Systeme zur Verfügung stellen. Alle Abhängigkeiten ergeben sich aus den Benutzt-Beziehungen der Schnittstellen; es gibt keine Abhängigkeit von der konkreten Implementierung.

Der Begriff *Service* wird im Sprachgebrauch manchmal im Sinn von *Operation einer Schnittstelle*, dann im Sinn von *Schnittstelle*, manchmal im Sinn von *Komponente* verwendet. Das ist unglücklich. Es wird deshalb vorgeschlagen, die Begriffspaare *Schnittstelle/Komponente* und *Service/System* nebeneinander zu verwenden: Jedes System ist nichts anderes als eine besonders große, eigenständige Komponente und jeder Service nichts anderes als eine Schnittstelle, die sich für die systemübergreifende Koppelung besonders eignet. Jeder Service (jede Schnittstelle) enthält Operationen. Auf der Ebene der Realisierung stellt sich jeder Service letztlich dar als Interface im Sinn von Java, C# oder irgendeiner anderen geeigneten Programmiersprache. Wir sprechen kurz vom *Aufruf eines Service* (einer Schnittstelle), wenn eigentlich der Aufruf von Operationen dieses Service gemeint ist.

SOA als Fortsetzung der Komponentenorientierung besitzt alle Vorzüge der Komponentenorientierung. Der wichtigste Vorzug ist die Unabhängigkeit von der Implementierung: Jeder Service kann von verschiedenen Systemen exportiert werden, und der Austausch des exportierenden Systems ist transparent für alle Importeure. Auch wenn diese Aussage für manche Ohren nach akademischer Informatik klingt: Wer jemals in der Praxis ein großes, zentrales Altsystem durch eine Neuentwicklung ausgetauscht hat, kann ermessen, wie viel Ersparnis an Schweiß und Geld diese Austauschbarkeit bedeutet.

Aber SOA besitzt auch alle Probleme der Komponentenorientierung. Hier sind vor allem zwei Punkte zu nennen: *Definition von Services* und *Seiteneffekte* (siehe [Sied04]).

Bei der *Definition von Services* (oder Schnittstellen) geht es um die schlichte Frage, was der Aufruf eines Dienstes genau bewirkt. Die zahlreichen Verfahren, die zu diesem Thema entwickelt wurden, können nicht befriedigen: Die einen sind formal, präzise, aber nicht praktikabel, die anderen informell, unklar und mehrdeutig, aber wenigstens praktikabel. Mit diesem Mangel lebt die Komponentenorientierung seit Jahren, und auch SOA scheint sich damit abzufinden.

Seiteneffekte sind ein Kernproblem der Komponentenorientierung: Jede Komponente ruft abhängig von der Implementierung nach Belieben andere Komponenten über deren Schnittstellen auf und kann deren Zustand ändern. Im Allgemeinen ist also kaum vorherzusagen, welche Komponenten sich bei der Abarbeitung eines Anwendungsfalls ändern, oder anders formuliert: Jeder Anwendungsfall kann ungeplante Seiteneffekte in beliebiger Anzahl produzieren. Diese unbefriedigende Situation wird beherrschbar, wenn gewährleistet ist, dass jede Komponente im Verlauf einer Verarbeitung von höchstens einem Aufrufer angestoßen wird, oder in anderen Worten: Aufrufpfade enthalten keine Kreise; sie bilden einen Baum. Dies zu erreichen ist leichter gesagt als getan, aber Softwarearchitekten werden ja auch gut bezahlt.

SOA ist also mindestens so schwer wie die Komponentenorientierung; alle Probleme, die dort ungelöst sind, begegnen uns auch bei SOA.

3.3 Lose Koppelung

Wir nennen System A *eng gekoppelt* an System B, wenn A die Verfügbarkeit von B voraussetzt. Zwei Systeme sind *eng gekoppelt*, wenn enge Koppelung beide Richtungen besteht; sie sind *lose gekoppelt*, wenn enge Koppelung in keine Richtung besteht.

Synchrone Aufrufe sind oft die Ursache enger Koppelung. Sie sind das vorherrschende Denkmodell beim Bau eines einzelnen Systems: Der Architekt unterstellt, dass jeder Aufruf ganz oder gar nicht stattfindet, unabhängig von der Komplexität der ausgelösten Aufrufhierarchie. Das ist das ACID-Prinzip der Datenbanken (*Atomicity*, *Consistency*, *Integrity*, *Durability*). Komponentenorientierung verstehen wir vor allem im Kontext eines einzelnen Systems, von dem wir zu jedem Zeitpunkt Konsistenz und fachliche Redundanzfreiheit verlangen. Die beteiligten Komponenten sind konsistent, aber wenn auch nur eine Einzige versagt, versagt das ganze System.

Enge Koppelung ist ungeeignet für die systemübergreifende Kommunikation. Deshalb findet man in der Praxis viele primitive, oft sogar merkwürdige Techniken, die enge Koppelung vermeiden, aber schlecht abschneiden in Bezug auf Robustheit, Wiederaufsetzbarkeit und andere nichtfunktionale Eigenschaften: Filetransfer, E-Mails und sogar SMS von einem System zum anderen sind hier zu nennen. Alle Versuche, enge Koppelung systemübergreifend herzustellen, können heute als gescheitert betrachtet werden. Die bekannten Verfahren der engen Koppelung (2-Phasen-Commit, Remote Procedure Call, CORBA) werden heute nur noch verwendet, um Verfahren der losen Koppelung zu implementieren. Ihr Einsatz für systemübergreifende Kommunikation entspricht nicht mehr dem Stand der Kunst.

SOA verbindet Komponentenorientierung mit loser Koppelung: Die Systeme einer serviceorientierten Systemlandschaft kommunizieren asynchron über Nachrichten. Dies ist ein wesentliches SOA-Merkmal, und erst in zweiter Linie stellt sich die Frage, mit welcher Technik diese Nachrichten versandt werden. Die Wirklichkeit, die unsere Software modelliert, verhält sich asynchron, und deswegen sollten sich auch unsere Softwaresysteme asynchron verhalten. Das synchrone Denkmodell eignet sich nur für atomare Abläufe innerhalb einer Komponente.

In fast jeder Programmiersprache kann man objektorientiert programmieren, und mit fast jeder Technik (auch mit den gerade genannten) kann man serviceorientierte Architektur betreiben. Aber das Werkzeug der Wahl ist ein *Enterprise Service Bus* (ESB), der die Kommunikation zwischen den verschiedenen Komponenten durchführt, verwaltet und überwacht.

3.4 Der ESB (Enterprise Service Bus)

ESBs arbeiten asynchron: Sie übermitteln Nachrichten von einem System zum anderen; der Sender wartet nicht auf die Rück-

meldung des Empfängers. Er setzt seine Arbeit ohne Verzögerung fort, aber er darf über die Laufzeit der Nachricht keine Annahmen machen. Der ESB ist integraler Bestandteil einer SOA-konformen Systemlandschaft: Er exportiert Services wie jedes andere System auch; es wäre falsch, ihm eine Sonderrolle zuzuweisen. Hier die Services eines ESBs im Einzelnen:

1) *Routing*. Der ESB leitet Nachrichten an die richtigen Adressaten weiter. Diese Eigenschaft ist keine Neuerung gegenüber den bekannten Brokern (CORBA). Der ESB erlaubt selbstverständlich, dass eine Nachricht eine andere überholt. Dies fördert den Durchsatz, kann aber die Konsistenz der beteiligten Systeme beeinträchtigen.
2) *Persistenz*. Der ESB garantiert, dass keine Nachricht verloren geht. Jede gesendete Nachricht kommt irgendwann bei allen vorgesehenen Empfängern an. Allerdings besitzen Sender und Empfänger während der Laufzeit der Nachricht einen unterschiedlichen Informationsstand und sind daher im Allgemeinen inkonsistent, solange eine schreibende Nachricht verarbeitet wird. Die Persistenz von ESBs ist auf keinen Fall mit Datenbanktransaktionen zu verwechseln.
3) *Fachliche Transformation*. Im Allgemeinen hat jedes beteiligte System seine eigene Begriffswelt. Der ESB hat die Aufgabe, diese Begriffswelten zu isolieren, indem er jeden Gesprächspartner in seiner Sprache anspricht. Früher war es häufig so, dass sich kommunizierende Systeme gegenseitig mit ihren Fachbegriffen infizierten: Im Altsystem spricht man etwa vom *Status30*, aber neue Systeme sollten stattdessen sprechende Namen verwenden (siehe auch Abschnitt 5.3)
4) *Technische Transformation*. Es gibt zahllose technische Möglichkeiten der Kommunikation zwischen Systemen – von gemeinsam benutzten Datenbanktabellen über SOAP bis zu E-Mail und SMS. Der ESB versteckt die Technik vor der Anwendung. Dieser Punkt ist kaum zu überschätzen, denn so wird es möglich, dass Systeme kommunizieren, die auf unterschiedlichen Plattformen laufen und in unterschiedlichen Programmiersprachen erstellt sind.
5) *Workflow*. Der ESB übernimmt die Ablaufsteuerung, wenn mehrere Services in einer bestimmten Reihenfolge aufzurufen sind. Die vorhandenen Produkte besitzen unterschiedliche Fähigkeiten; der Standard ist BPEL.
6) *Fehlerbehandlung*. Je größer die Systemlandschaft, desto aussichtsloser der Versuch, alle möglichen Fehler vorausschauend zu behandeln. Der ESB rechnet damit, dass jeder Aufruf eines Service fehlschlagen kann. Wenn dies geschieht, informiert er alle betroffenen Systeme, und es ist deren Verantwortung, angemessen zu reagieren.
7) *Monitoring, Protokollierung*. Der ESB ist hervorragend geeignet, um das gesamte Geschehen zu protokollieren: Er merkt sich, welche Services in welcher Häufigkeit, mit welchen Parametern und mit welchem Erfolg aufgerufen wurden. Auf diese Weise lässt sich die Einhaltung oder Verletzung von SLAs nachweisen.
8) *Lastverteilung, Ausfallsicherheit und Skalierbarkeit*. Dies ist vielleicht die stärkste Seite moderner ESBs. Die nichtfunktionalen Eigenschaften *Performance*, *Skalierbarkeit* und *Ausfallsicherheit* verursachen Schwierigkeiten, seitdem wir Softwaresysteme bauen, und ESBs bieten zwei Funktionen zur systematischen Herbeiführung dieser Eigenschaften: Erstens lässt sich jeder Service mehrfach registrieren; der ESB übernimmt die Lastverteilung. Zweitens können moderne ESBs in mehreren Exemplaren laufen; die Lastverteilung liegt bei einem Masterexemplar. Das ist gut für die Performance wegen der Parallelisierung, für die Skalierbarkeit wegen der Option, Service- und ESB-Exemplare nach Belieben zu ergänzen und für die Ausfallsicherheit wegen der Redundanz.

Ein ESB besitzt also zahlreiche, mächtige Funktionen, die wenig miteinander zu tun haben. Das hat seinen Preis, nicht nur in Form von Lizenzkosten. ESBs haben eine ganze Weile gebraucht, bis sie in allen Situationen stabil laufen, und sie kosten auch heute erhebliche Rechenzeit: Die Latenzzeit beim Aufruf eines ESB-Service ist um Größenordnungen länger als die eines Unterprogramm-Aufrufs. Somit kann man jedes ESB-basierte System durch ungeschickte ESB-Nutzung lahmlegen.

3.5 Workflow als eigene Komponente

Jedes System einer Systemlandschaft sollte möglichst wenig von den anderen Systemen wissen. Daher liegt es nahe, die systemübergreifende Steuerungslogik in eine eigene Workflow-Komponente zu verlagern. Jede SOA-konforme Systemlandschaft stellt sich nach außen als ein einziges großes System dar, das Aufträge von menschlichen Benutzern oder von externen Systemen entgegennimmt und erledigt. Dabei stellt man sich am besten vor, dass jeder Auftrag mit einem Laufzettel versehen wird. Dieser Zettel enthält zu Beginn der Bearbeitung alle Stationen, die der Auftrag durchlaufen wird, und am Ende all diejenigen, die er tatsächlich durchlaufen hat.

Moderne ESBs (siehe Abschnitt 3.4) enthalten eine solche Workflow-Komponente. In der Regel dient BPEL [BPELOJ] oder eine vergleichbare Sprache zur Definition der Abläufe.

Jedes Werkzeug verursacht Schaden bei Missbrauch. Dies gilt für Workflow-Komponenten in besonderem Maß: Manche fürchten, dass mit der Auslagerung der Steuerungslogik die *if*-Anweisungen in der Programmierung verboten sind. Das ist ein Missverständnis, denn die Workflow-Komponente steuert nur systemübergreifend. SOA kümmert sich nicht um die Ablaufsteuerung innerhalb eines einzelnen Systems. Viele Architekten unterscheiden Makro- und Mikro-Abläufe: Makro-Abläufe betreffen mehrere Systeme und sind Sache des ESB, Mikro-Abläufe finden innerhalb eines Systems statt. Dies ist aber nur ein erster Schritt zur Darstellung von komplexen Abläufen durch eine Hierarchie von immer feineren Workflows. Der Architekt hat zwei Optionen:

a) Ein einziger ESB verwaltet alle Workflow-Ebenen. Die zugehörige Workflow-Komponente kennt alle Workflow-Definitionen.
b) Er verwendet einen Baum von ESBs, dessen Struktur dem Workflow-Baum entspricht; jeder ESB verwaltet einen Knoten des Workflow-Baums.

Es gibt keine allgemeine Regel für (a) oder (b). Allerdings hat Option (b) den Vorzug, dass sie die Zuständigkeiten optimal trennt.

4 Der Weg zu SOA

Die vollkommene SOA-konforme Systemlandschaft ist ein unerreichbares Ideal, und die Vorstellung, SOA auf einen Schlag mit einem riesigen Projekt einzuführen, ist in den meisten Fällen abwegig. Eine der wichtigsten Eigenschaften von SOA ist die Möglichkeit der weichen Einführung.

4.1 Weiche Einführung

Die Idee besteht darin, zwei parallele Welten zu betreiben: Auf der einen Seite gibt

es die moderne SOA-Welt, die zu Beginn leer ist und mit der Zeit immer mächtiger wird. Auf der anderen Seite gibt es die Prä-SOA-Welt der vorhandenen Systeme, die schrittweise zurückgefahren wird. So gelangt ein System nach dem anderen von der alten Welt in die neue. Bei manchen Systemen werden erhebliche Änderungen und Anpassungen nötig sein, bei anderen nicht.

Der entscheidende Punkt ist dabei die Schnittstelle zwischen alter und neuer Welt: Die neue Welt sieht die alte nur über Services; in der anderen Richtung werden im Allgemeinen auch nicht serviceorientierte Kommunikationspfade bestehen. Die alte Welt erscheint aus Sicht der neuen als ein einziges großes System. Der Weg zu SOA folgt den drei Modellen aus Abschnitt 3.1, die natürlich parallel entwickelt werden können:

1. *Prozessmodell*: Welche Geschäftsprozesse gibt es, und wie werden sie von der Systemlandschaft unterstützt?
2. *Servicemodell*: Die Services ergeben sich aus der Analyse der *Geschäftsdaten* und der *Geschäftsfunktionen*: Welche Geschäftsobjekte gibt es, und welche Anwendungen verwenden sie? Welche Redundanzen gibt es; welche davon sind vermeidbar? Systemlandschaften enthalten Redundanz, zum Teil vermeidbar, zum Teil aber auch nicht. Dies hat für Datenmodellierung auf der Ebene von Systemlandschaften eine wichtige Konsequenz: Wir unterscheiden zwischen abstrakten Geschäftsobjekten (der Kunde als solcher, die Stückliste als solche) und ihrer Implementierung in verschiedenen Systemen: Jedes System hat sein eigenes Datenmodell; das Geschäftsobjekt *Kunde* besitzt 23 Attribute im System A, 34 im System B. Davon sind 12 namensgleich und bedeuten dasselbe, 3 haben denselben Namen, aber verschiedene Bedeutungen, 2 haben verschiedene Namen, aber dieselbe Bedeutung, und der Rest hat nichts miteinander zu tun. Die Erkennung, Dokumentation und Kontrolle von Abhängigkeiten dieser Art ist eine Voraussetzung für erfolgreiche SOA-Projekte. Parallel zu den Geschäftsdaten analysiert man die Geschäftsfunktionen: Welche gibt es heute schon, welche Systeme bieten sie an, und welche Systeme verwenden sie? Welche Redundanzen gibt es; welche davon sind vermeidbar? Welche Geschäftsfunktionen fehlen?
3. *Technikmodell*: In jeder Systemlandschaft stellen sich technische Fragen, die systemübergreifend zu beantworten sind: Ausnahmebehandlung, Monitoring oder Rücksetzbarkeit von Operationen (siehe Abschnitt 5.4 über nichtfunktionale Eigenschaften). Das Technikmodell beschreibt die Konzepte und die Werkzeuge (allen voran der ESB), mit deren Hilfe nichtfunktionale Eigenschaften systemübergreifend herbeigeführt werden.

Auf der Basis dieser Analysen ist es möglich, die Reihenfolge festzulegen, in der Systeme von der alten Welt in die SOA-Welt überführt werden.

4.2 Die richtigen Services

Eine oft gestellte Frage ist die nach den richtigen Services. Diese sucht man im ersten Schritt ohne Rücksicht auf technische Restriktionen aus fachlicher Sicht: Jeder Service, der über den ESB geht, hat eine fachliche Bedeutung und ist auch für IT-Laien verständlich. Neben dieser Grundregel empfehlen wir eine Reihe von Kriterien für die Definition von Services zur Verbesserung der Performance, der Robustheit und der Entkoppelung:

- *Mächtige Operationen*: Lose Koppelung geht einher mit der Regel, dass wenige Aufrufe, die viel bewirken, besser sind als viele Aufrufe, die erst zusammen den gewünschten Effekt erzielen. Daher sind wenige mächtige Operationen besser als viele kleine, und zwar nicht nur aus Gründen der Performance, sondern auch zur Reduzierung der durch Asynchronität möglichen Inkonsistenzen. So ist es sinnvoller, alle Kundendaten zu einer Kundennummer mit einer einzigen Operation zur Verfügung zu stellen, als dem Aufrufer zuzumuten, Adresse, Bonität usw. einzeln abzuholen. Ein anderes Beispiel ist die Bestellung mit ihren Bestellpositionen: Auch hier ist es sinnvoll, alle Bestellpositionen auf einmal zu übergeben.
- *Idempotenz*: Idempotente Operationen zeichnen sich dadurch aus, dass ein mehrmaliger Aufruf mit denselben Parametern denselben Effekt hat wie der einmalige. Beispiel: Einmal stornieren genügt; ein zweiter Aufruf der Storno-Methode bleibt ohne Wirkung, und richtet daher keinen Schaden an. Die Idee kommt aus dem Stapelbetrieb: Man baut Batches nach Möglichkeit so, dass ein versehentlich angestoßener zweiter Lauf kein Unheil anrichtet (z. B. beim Einlesen einer Einzahlungsdatei).
- *One Shot*: One-Shot-Methoden zeichnen sich dadurch aus, dass sie in einem einzigen System in einer einzigen Transaktion ganz oder gar nicht stattfinden. Die Beschränkung auf One-Shot-Methoden, die dringend empfohlen wird, bedeutet den Verzicht auf systemübergreifende Transaktionen: Jeder systemübergreifende Ablauf wird in mehreren unabhängigen Transaktionen durchgeführt – mit den entsprechenden Möglichkeiten der Inkonsistenz, die sich nicht mehr durch einen schlichten Rollback reparieren lässt, sondern nur durch kompensierende Methoden (Einzahlen kompensiert Abheben, nochmaliges Ändern auf den alten Wert kompensiert die Änderung). Aber die bewirkte Entkoppelung rechtfertigt den Aufwand für die Implementierung kompensierender Methoden.
- *Kontextfreiheit:* Das gerufene System kennt keine Benutzer und keine Sitzungen. Der Aufrufer stellt – falls erforderlich – die Verbindung zwischen aufeinanderfolgenden Aufrufen her, nicht die gerufene Anwendung. Auch die Kontextfreiheit hat keinen anderen Zweck, als Rufer und Gerufenen zu entkoppeln (siehe [Evans2003]).
- *Die richtige Abstraktionsebene:* Dies ist die Abwägung von Typsicherheit gegen Flexibilität. Beispiel: Ein Service zur Auftragserfassung sollte so gestaltet sein, dass er bei der Einführung eines neuen Produkts unverändert bleiben kann. Das einzelne Produkt darf er also nicht kennen, muss aber sinnvolle Annahmen machen über Eigenschaften und Verhalten aller vorhersehbaren Produkte. Die Empfehlung lautet: Jeder Service sollte so konkret sein wie nötig, aber nicht konkreter. Ein Zuviel an Konkretion erkennt man durch die Analyse von Änderungsszenarien, etwa der Einführung eines neuen Produkts. Eine generischer Service der Bauart *execute(Object)* ist in jedem Fall zu abstrakt.

4.3 Die richtigen Systeme

Die Architektur eines einzelnen Systems folgt dem bekannten Schichtenmodell, das in einer gängigen Version drei Schichten besitzt: Die oberste Schicht analysiert die Eingaben des Benutzers und beauftragt nach deren erfolgreicher Prüfung die nächste Schicht, den Anwendungskern, mit der Durchführung der verlangten fachlichen Funktionen. Die Datenschicht ist zuständig für die Verwaltung der Daten und stellt die Verbindung zum Datenbanksystem her.

In einem solchen Schichtenmodell erfolgen die Aufrufe von oben nach unten; Komponenten der unteren Schichten sind

unabhängig von den Komponenten weiter oben. Es ist somit ein Beitrag zur Klarheit und Beherrschbarkeit.

Dieser Abschnitt beschreibt ein Schichtenmodell auf der Ebene von Systemlandschaften: So wie jede Komponente eines einzelnen Systems einer bestimmten Schicht angehört und die entsprechenden Aufgaben erledigt, so lässt sich jedes System einer SOA-konformen Systemlandschaft einer von fünf Kategorien zuordnen (siehe Bild 2). Alle Systeme einer Kategorie übernehmen bestimmte, feststehende Aufgaben. Diese Kategorien sind eine Hilfe beim Entwurf neuer Systeme und bei der serviceorientierten Umgestaltung vorhandener Systemlandschaften, weil sie die möglichen Abhängigkeiten reduzieren und deren Kontrolle erleichtern:

Die durchgezogenen Pfeile in Bild 2 beschreiben die Standardaufrufe: Portalsysteme verschaffen den Zugang zu Prozessen und zu Analysesystemen. Prozesse bedienen sich der Geschäftsfunktionen, und diese greifen auf die Geschäftsdaten zu. Analysesysteme haben selbstverständlich Zugang zu den Daten, die sie analysieren. Neben den Standardaufrufen gibt es noch die geduldeten Aufrufe (gestrichelte Pfeile in Bild 2). Der direkte Zugriff vom Portal auf Daten oder Funktionen kann betroffene Geschäftsprozesse stören; trotzdem ist dieser Weg in der Regel offen. Für den direkten Zugriff von Prozessen auf die Daten gilt die analoge Aussage.

Im Weiteren werden die Systemkategorien im Einzelnen beschrieben:

a) *Portale*: Jedes Portal verschafft den Zugang zu einer oder (meistens) mehreren Anwendungen, also einer ganzen Systemlandschaft. Sie bietet nach außen ein einheitliches Bild; Single-Sign-On ist Sache des Portals. Zu einer Systemlandschaft gibt es oft nur ein Portal, möglicherweise aber auch mehrere für verschiedene Benutzergruppen (Mitarbeiter und Kunden). Das Portal ist der technische Rahmen einer Systemlandschaft, es enthält nur wenig fachliche Logik.
b) *Prozesse*: Jedes Prozesssystem unterstützt einen oder mehrere Geschäftsprozesse. Es kann direkt im ESB implementiert sein, aber das ist nicht notwendigerweise der Fall. Seine Aufgabe ist die Steuerung des Workflows gemäß den Geschäftsregeln über die verschiedenen Anwendungen hinweg. Prozesssysteme bilden den fachlichen Rahmen einer Systemlandschaft. Prozesssysteme verwenden Geschäftsfunktionen und Geschäftsdaten. Ihr Entwurf orientiert sich an den abzubildenden Geschäftsprozessen. Deren Hierarchie wird die Struktur der Geschäftsprozesssysteme beeinflussen (siehe Abschnitt 3.5).
c) *Funktionen*: Systeme dieser Kategorie implementieren die Services, die auf der Ebene darüber zu Workflows verknüpft werden. Der Entwurf dieser Systeme orientiert sich an den Geschäftsprozessen: Jeder Geschäftsprozess sollte mit möglichst wenigen verschiedenen Funktionssystemen auskommen, um die Abhängigkeiten zu minimieren.
d) *Daten*: Datensysteme kennen die Konsistenzbedingungen und überwachen diese bei jeder Änderung. Sie bieten die bekannten Funktionen Anlegen/Ändern/Löschen für die Geschäftsobjekte, die sie beherbergen; darüber hinaus gehendes fachliches Wissen besitzen sie nicht. Datensysteme entwirft man durch eine sinnvolle Aufteilung der gesamten Datenwelt in überschaubare und möglichst disjunkte Portionen.
e) *Analyse*: Analyseanwendungen verdichten und analysieren die laufend anfallenden Geschäftsdaten. Business-Information-Systeme gehören zu dieser Gattung.

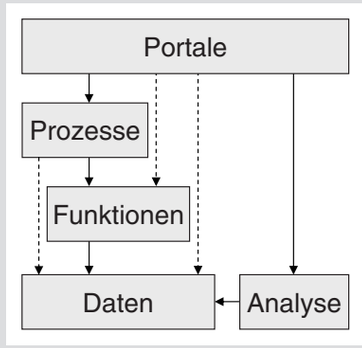

Bild 2 Fünf Systemkategorien

5 Warum ist SOA schwer?

Dieser Abschnitt behandelt Tücken und Fallstricke von SOA, die uns in verschiedenen Projekten aufgefallen sind. Er ist ein Appell an die Wachsamkeit der SOA-Verantwortlichen, denn leider gibt es kein Regelwerk, mit dessen Hilfe die genannten Schwierigkeiten sicher zu umgehen wären.

5.1 Sicherstellung der Performance

Wohl jedes SOA-Vorhaben kämpft mit Performance-Problemen. Der erste Schritt zu deren Vermeidung sind mächtige Services (siehe Abschnitt 4.2). Aber es gibt ein weiteres Problem: Viele Unternehmen erlassen die Regel, dass jede systemübergreifende Kommunikation nur und ausschließlich über den ESB stattfindet. Nun ist der Versand einer Nachricht über den ESB um Größenordnungen langsamer, und er ist asynchron. Deshalb kann die ausschließliche Verwendung des ESB zu Schwierigkeiten führen.

Zur Veranschaulichung betrachten wir einen Kundenserver, der alle Kundenadressen enthält, und nehmen an, dass alle Adressabfragen über den ESB laufen. Hier wird man Folgendes beobachten: Jeder, der Adressen braucht, diese über den ESB aber nicht in der gewünschten Geschwindigkeit bekommt, wird sich seinen privaten Cache bauen. Das Ergebnis sind zahlreiche Caches, deren Aktualität wahrscheinlich niemand genau kennt, hoher Aufwand für Entwicklung und Pflege dieser Caches sowie unkontrollierte Redundanz. In diesem Beispiel ist es sinnvoller, die Caches zu vermeiden, indem man *lesende* Direktzugriffe vorbei am ESB zulässt – jede Adress*änderung* erfolgt selbstverständlich über den Bus.

5.2 Abstimmung von Services

Man kann sich zu Tode abstimmen. Ein unternehmensweites Servicemodell hätte genauso wenig Erfolgsaussichten wie das mittlerweile begrabene unternehmensweite Datenmodell, egal wie gut die Werkzeugunterstützung ist. Die Abstimmung von Services ist deshalb schwer, weil jeder Service zunächst einmal entweder aus Sicht des Exporteurs (*das biete ich an*) oder aus Sicht des Importeurs (*das brauche ich*) entworfen wird. Aber die Vorstellungen von Exporteur und Importeur passen auf Anhieb nur selten zusammen. Die Verwendung eines Service durch mehrere unabhängige Importeure macht die Abstimmung noch schwerer. Nun steht immer der Weg offen, langwierige Abstimmungen durch geeignete Adapter abzukürzen. Aber zu viele Adapter produzieren Aufwand, erschweren die Verwaltung und beeinträchtigen möglicherweise sogar die Performance. Die Kunst besteht darin, auch mühsame Abstimmungen durchzusetzen, wo Adapter stören, und Adapter dort zuzulassen, wo sie nicht schaden.

5.3 Umgang mit Altsystemen

Ein wichtiger Vorzug von SOA ist die Integration von Altsystemen durch geeignete Adapter. Allerdings gibt es dabei mindestens zwei Probleme, die der Erwähnung bedürfen, und zwar die *Entflechtung der Daten* und die *Kontrolle der Begriffswelten*.

Zunächst zur *Entflechtung der Daten*. Viele Systeme kommunizieren über gemeinsam verwendete Datenbanktabellen: System A schreibt Informationen in Tabelle T, System B liest daraus. Zum Beispiel war in der Standardarchitektur der Versicherungen (VAA) dies der einzig legale Weg der Kommunikation zwischen verschiedenen Systemen. Die Verflechtung der Daten verursacht folgende Probleme:

a) Die beteiligten Systeme kommunizieren nicht in den Begriffen der Anwendung, wie das bei Services der Fall ist, sondern in den Begriffen der Datenbank. Datenbanktabellen und -attribute haben nur in Lehrbüchern sprechende Namen; in der Praxis findet man viele technische Namen und viele Tabellen und Attribute, die nur technischen Zwecken dienen. Deshalb sind die Begriffe der Datenbank ungeeignet für die Formulierung von fachlichen Services.
b) Die beteiligten Systeme machen undokumentierte Annahmen darüber, wann das Partnersystem Daten schreibt oder liest. Das lesende System kennt nicht die Aktualität des Gelesenen.

Es ist also mühsam festzustellen, in welchen Situationen sich die Tabelle T ändert, wer da hineinschreibt, und was die Änderungen jeweils bedeuten. Deshalb hat ein Adapter, der ein Altsystem abschirmt, nur einen begrenzten Nutzen, denn jeder angebotene Service kann kaum nachvollziehbar Seiteneffekte verursachen. Zur Entflechtung ersetzt man unkontrollierte Zugriffe auf nackte Datenbanktabellen durch Aufrufe von Datenkomponenten, die Status und Konsistenz der Datenbanktabellen kennen und überwachen.

Mit der *Kontrolle der Begriffswelten* ist folgendes gemeint: Jedes Altsystem hat seine eigene, historisch gewachsenen Begriffswelt. So bezeichnet bei einem großen Versicherer die *Abgangsart 99* den Tod des Versicherten. Kryptisch verschlüsselte Statusfelder und Satzarten sind typisch für Altsysteme, aber auch jedes neue System entwickelt seine eigene Begriffswelt – so wie in jedem größeren Unternehmen Begriffe und Wendungen existieren, die für Außenstehende unverständlich sind. Die Gefahr besteht, dass solche Begriffe nicht isoliert werden, sondern nach und nach die gesamte Systemlandschaft verseuchen. Deshalb ist bei jedem Service sicherzustellen, dass er in Begriffen formuliert ist, deren Verwendung allen beteiligten Systemen (Importeure und Exporteur) zuzumuten ist. Jeder Service wird in der einfachstmöglichen Sprache verfasst. Die Softwarekategorien in [Sied04] können bei der Überwachung solcher Begriffswelten helfen.

5.4 Nichtfunktionale Eigenschaften

Nichtfunktionale Eigenschaften verursachen schon bei einzelnen Systemen regelmäßig Probleme, und für Systemlandschaften gilt dies erst recht. Wir betrachten als Beispiel die Behandlung von Ausnahmen, einer Voraussetzung für Robustheit: Es ist ein weiter Weg von isolierten Systemen mit jeweils eigener Ausnahmebehandlung hin zu einer integrierten Systemlandschaft mit einer übergreifenden Ausnahmebehandlung. Leider gibt es keine Patentrezepte, und jede der folgenden nichtfunktionalen Eigenschaften würde Stoff für einen eigenen Artikel liefern.

- *Ausnahmebehandlung*: Der ESB informiert jeden beteiligten Service über Erfolg oder Misserfolg der gesamten Anforderung. Die Verantwortung für die richtige Reaktion liegt beim einzelnen Service – und dies erfordert nicht selten die Implementierung eigener, in der Prä-SOA-Welt nicht vorgesehener Anwendungsfälle.
- *Authentifizierung und Autorisierung*: Jedes beteiligte System hat seine eigene Vorstellung von den Benutzern und deren Rechten. Diese lassen sich nur mit Mühe in Einklang bringen.
- *Logging*: Jedes System schreibt seine Log-Informationen nach eigenen Regeln, in eigenen Formaten in verschiedene Dateien. In dieser Form ist Log-Information nahezu wertlos. Die Vereinheitlichung der Logging-Strategie aller Systeme einer Landschaft ist ein Projekt, das nicht zu unterschätzen ist.
- *Mandantenfähigkeit.* Dies ist ein Beispiel für eine nichtfunktionale Eigenschaft, die schon bei einem einzelnen System regelmäßig Schwierigkeiten macht.

5.5 Dominanz der Technik

Softwarewerkzeuge sind hilfreich, dürfen aber nie im Vordergrund stehen. Leider entwickeln sich serviceorientierte Architekturen manchmal um das gewählte Produkt herum. Aber die Unabhängigkeit der Anwendung von der Technik ist eine der Grundlagen von SOA. Dies zeigt bereits die Trennung der beiden Modelle für Services und Technik: Jeder Service existiert zunächst einmal abstrakt, ohne Bezug auf die technische Implementierung: Eine Kundenverwaltung liefert die Kundendaten, und es ist völlig egal, ob der Zugriff über Sockets, RMI oder einen ESB erfolgt. Ein Berechtigungsmanager exportiert die bekannte Methode „darf der Anwender, was er vorhat“, und auch hier spielt es keine Rolle, wie der Zugriff technisch realisiert ist. So ist es möglich, ein und denselben Service auf verschiedenen technischen Kanälen zu publizieren – von Sockets bis SOAP. Erst in der vollkommenen SOA-Welt sind alle diese Techniken überholt – aber dieser Zustand ist unerreichbar.

6 Der Nutzen von SOA

Es ist schwer, den Nutzen einer Methode oder einer Technologie plausibel zu machen. Die üblichen Behauptungen zu Flexibilität und Effizienz können selten überzeugen. Auch SOA leistet einen Beitrag zu Effizienz und Flexibilität, der freilich schwer zu quantifizieren ist. Aber es gibt mindestens drei konkrete, nachprüfbare Argumente für den Nutzen von SOA.

6.1 Keine Jumbo-Projekte

Entwicklungsprojekte werden immer größer – aber die Wahrscheinlichkeit des Projekterfolgs ist umgekehrt proportional zur Projektgröße. Jumbo-Projekte sind zum Scheitern verurteilt, wenn es nicht gelingt, sie in machbare Teilprojekte zu zerlegen. SOA-konforme Systemlandschaften machen Jumbo-Projekte überflüssig, weil sie ein inkrementelles Vorgehen ermöglichen. Mit SOA hat der Architekt die folgenden Optionen:

a) Er kann bei unveränderten Services vorhandene Workflows ändern oder neue definieren.
b) Er kann die Implementierung vorhandener Services ändern oder austauschen.
c) Er kann neue Services kaufen oder erstellen und diese schrittweise in den vorhandenen Workflows einsetzen.
d) Er kann veraltete Services schrittweise zurückbauen und schließlich abschalten, wenn sie nicht mehr gebraucht werden. Rückbau bedeutet, dass die Importeure des veralteten Service nach und nach auf einen neuen Service umgestellt werden (siehe auch [Moor05]).

Die Änderung eines vorhandenen Service (also die Schnittstelle, nicht die Implementierung) wird zurückgeführt auf die Einführung des geänderten Service als neuer Service (c) und den schrittweisen Rückbau der alten Version (d). In einer idealen SOA-Welt lässt sich mit diesen vier Optionen jedes Jumbo-Projekt auf eine planbare Folge von Inkrementen zurückführen.

6.2 Entwicklungsprozess

SOA entkoppelt nicht nur die Systeme einer Systemlandschaft, sondern erleichtert auch die Arbeit der Entwickler, in erster Linie durch das im letzten Abschnitt propagierte inkrementelle Vorgehen. Aber SOA bringt den Entwicklern noch mehr Vorteile:

a) *Verbundtest*. Beim Verbundtest testet man das geänderte oder neu erstellte System im Verbund seiner Nachbarsysteme. In der Praxis ist das schwierig, denn es ist so gut wie unmöglich, eine komplette Systemlandschaft in zwei identischen Exemplaren für den Wirkbetrieb und für den Verbundtest zu betreiben. Mit SOA hat man zunächst die Möglichkeit, den oder die ESBs mit identischen Workflow-Definitionen auch in einer Testumgebung einzusetzen. Ferner lässt sich – wie bei der Komponentenorientierung – jeder Service durch einen mehr oder weniger intelligenten Dummy ersetzen. So erleichtert SOA die Simulation des Wirkbetriebs.

b) *Weiche Einführung (soft rollout).* Mit SOA wird man neue Services nur schrittweise freigeben. So könnte man einen neuen Service in einer ersten Stufe nur leer mitlaufen lassen. Im zweiten Schritt würde er nur unkritische Testaufgaben wahrnehmen und erst dann seine eigentliche Arbeit aufnehmen. Dieser *Test im Wirkbetrieb* wäre in einer klassischen Systemlandschaft völlig undenkbar.

c) *Entkoppelung.* Die Entkoppelung der Systemlandschaft ermöglicht auch die Entkoppelung der Entwicklungsarbeiten. Die parallele Arbeit an verschiedenen Systemen durch Projektteams verschiedener Herkunft und Standorte wird beherrschbar.

6.3 Dynamic Computing

Dynamic Computing ist eine der wichtigsten Neuerung für den kostengünstigen, industriellen Betrieb von Softwaresystemen, macht aber Annahmen über die zu betreibende Software. Es zeigt sich, dass SOA-konforme Anwendungen für Dynamic Computing besonders geeignet sind.

Dynamic Computing bedeutet, dass Rechenleistung und Basissoftware aus der Steckdose kommen. Sie werden als Commodity betrachtet: als eine Ware, die wie Strom oder Wasser in beliebiger Menge verfügbar ist (siehe [Carr04]). Niemand interessiert sich für das Kraftwerk, das den Strom produziert, und genauso wenig interessiert der Rechner, der die benötigten MIPS zur Verfügung stellt. Dynamic Computing basiert vor allem auf der Verwendung von Blades (leistungsfähigen, preisgünstigen Rechnern ohne Kühlung und ohne Netzteil, die in großer Anzahl in Rahmen eingesteckt werden). Jedes Blade beherbergt ein Betriebssystem (oft Linux), möglicherweise auch mehrere unter Verwendung von VMware oder Xen. Der Benutzer sieht keine Hardware, sondern Betriebssystemexemplare, die sich nur in der zugeordneten Rechenleistung und im Speicherplatz unterscheiden, aber sonst nicht unterscheidbar sind. Hier sei auf die Analogie zur Objektorientierung hingewiesen: Objekte einer Klasse (etwa der Klasse *String*) unterscheiden sich in der Größe, sind aber in ihrem Verhalten nicht unterscheidbar. Die Idee nicht unterscheidbarer Betriebssystemexemplare lässt sich verallgemeinern – zum Beispiel auf ein Datenbanksystem oder einen Application Server.

Dynamic Computing entfaltet seinen Nutzen erst, wenn sich die zu betreibende Anwendung für Parallelisierung eignet. SOA schafft durch Asynchronität die idealen Voraussetzungen, um parallele Rechenleistung mit Vorteil zu verwenden: Der ESB ermöglicht eine n:m-Beziehung zwischen Betriebssystemexemplar und Service: Rechenintensive Services werden in zahlreichen Exemplaren registriert; jedes Serviceexemplar hat sein eigenes Betriebssytemexemplar. Andere Services, die weniger Rechenleistung brauchen, teilen sich ein Blade mit anderen Services. Dies garantiert ausgezeichnete Skalierbarkeit, denn zusätzliche Blades können in fast beliebiger Anzahl hinzugefügt werden. Manche Hersteller behandeln den ESB selbst als Service, der dann seinerseits in mehreren, sich gegenseitig überwachenden Exemplaren läuft – und dies wieder auf einer beliebigen Anzahl von Blades.

Literatur

[BPELoJ] http://www.bpelsource.com/, Abruf am 2006-10-02.

[Carr04] *Carr, Nicholas G.:* Does IT Matter? Harvard Business School Press, 2004.

[Chap04] *Chappell, David:* Enterprise Service Bus. O'Reilly, 2004.

[Erl04] *Erl, Thomas:* Service-Oriented Architecture. Prentice Hall, 2004.

[Evan03] *Evans, Eric:* Domain Driven Design. Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

[IBM05] *IBM's SOA Foundation:* An Architectual Introduction and Overview. IBM, 2005.

[KrBS04] *Krafzig, Dirk; Banke, Karl; Slama, Dirk:* Enterprise SOA. Prentice Hall, 2004.

[Moor05] *Moore, Geoffrey:* Dealing with Darwin. Portfolio-Penguin Group, 2005.

[Sied04] *Siedersleben, Johannes:* Moderne Software-Architektur. dPunkt Verlag, 2004.

[Szyp02] *Szyperski, Clemens:* Component Software. Addison-Wesley, 2002.