Gernot Starke · Stefan Tilkov (Hrsg.)

SOA-Expertenwissen

Methoden, Konzepte und Praxis serviceorientierter Architekturen

dpunkt.verlag

27 Kooperation statt Kommandoton

Was kommt nach RPC und nachrichtenorientierter Kommunikation?

Ralf Westphal

Zusammenfassung

Grundlage für die Serviceorientierung ist die Verteilung von Software und damit die Kommunikation über Prozessgrenzen hinweg. Trotz jahrzehntelanger Anstrengungen bereitet es allerdings dem Gros der Softwareentwickler wie auch vielen Softwarearchitekten immer noch erhebliche Schwierigkeiten, diese Kommunikation zu planen und zu implementieren. Zwar hat inzwischen ein Umdenken von RPC-Ansätzen und Objektorientierung hin zur Nachrichtenorientierung stattgefunden – aber letztlich ist das Ziel immer noch, im Programmiermodell den Graben zwischen kommunizierenden Softwareteilen in unterschiedlichen Prozessen so weit wie möglich zu kaschieren. Softwareentwickler wollen und sollen Aktionen in entfernten Programmteilen mit Methodenaufrufen anstoßen können. Das ist ein verständlicher Wunsch, aber seine Erfüllung widerspricht der schlichten physikalischen Realität der Kommunikation über Prozess- oder gar Rechnergrenzen hinweg.

Der Beitrag vertritt die These, dass genau diese Entfremdung von der Realität essenziell kontraproduktiv für die Planung und Implementation verteilter Softwaresysteme ist. Er geht von der Prämisse aus, softwaretechnische Abstraktionen sollten nicht beliebig weit getrieben werden, um zu vereinfachen, sondern sollten ihre Form immer auch dem Zweck anpassen. Und der Zweck verteilter Softwaresysteme ist die Kooperation, nicht die gegenseitige Befehlserfüllung. Für eine Kooperation aber sind sowohl Methodenaufrufe wie Nachrichten ungenügende Abstraktionen.

27.1 Kommunikation als Herzstück der Serviceorientierung

Zentral für das Konzept serviceorientierter Softwaresysteme ist die Kommunikation zwischen den Services. Services ohne Kommunikation mit anderen haben schlicht keinen Wert. Wo früher mehrere monolithische Programme unverbunden nebeneinander stehend die Unternehmensziele zu erreichen halfen (Abb. 27–1), da soll das heute durch die Kooperation von Services in einem Netzwerk geschehen (Abb. 27–2).

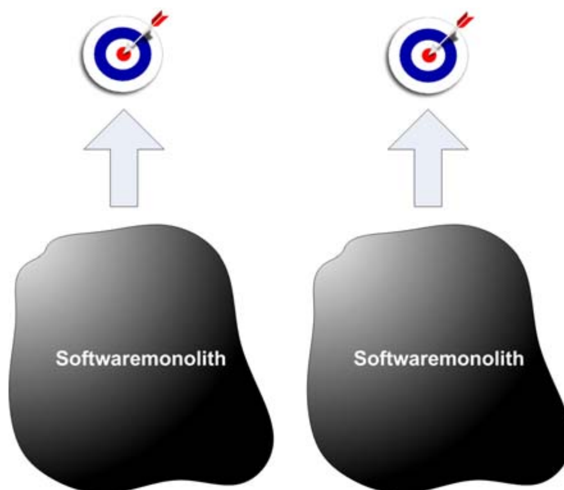

Abb. 27–1 *Früher dienten unverbundene monolithische Applikationen der Erreichung der Ziele eines Unternehmens.*

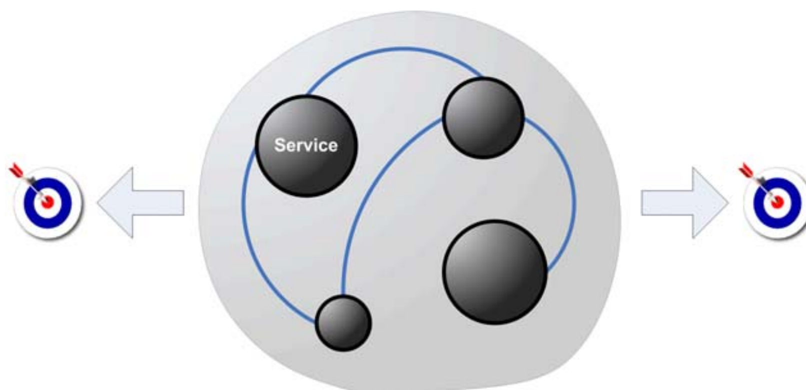

Abb. 27–2 *Heute sollen kooperierende Services ein Netzwerk aufspannen, das in seiner Gesamtheit möglichst vielen Unternehmenszielen dient.*

Die Vernetzung, d.h. die Relationen zwischen Services, machen serviceorientierte Softwaresysteme aus. Sie sind die Träger der Leistungsdefinitionen der Services in funktionaler wie nichtfunktionaler Hinsicht. Aus großer Entfernung betrachtet, schrumpfen die Services selbst zu atomaren Codeeinheiten, deren Sinn erst in der Kopplung mit anderen entsteht (Abb. 27–3).

Abb. 27–3 *Die Relationen zwischen Services machen den Wert eines serviceorientierten Softwaresystems aus.*

Die Relationen in einem serviceorientierten Softwaresystem sind uniform: Es sind gerichtete Beziehungen zwischen Funktionseinheiten. Das Verhältnis der Funktionseinheiten ist bezogen auf eine einzelne Relation somit asymmetrisch, da eine Funktionseinheit – der Konsument – Kontrolle über die andere – den Produzenten – ausübt (Abb. 27–4). Damit unterscheidet sich die Serviceorientierung aber natürlich noch nicht von bisherigen Architekturkonzepten im Kleinen wie im Großen. Bei der Modellierung von Software geht es zunächst immer um Konsument-Produzent-Verhältnisse von Codeeinheiten, seien es Methoden, Klassen, Komponenten, Prozesse oder Services. Ohne nähere Angaben kann man einem Konsument-Produzent-Paar nicht ansehen, auf welcher physischen Aggregationsebene es sich befindet (Abb. 27–5).

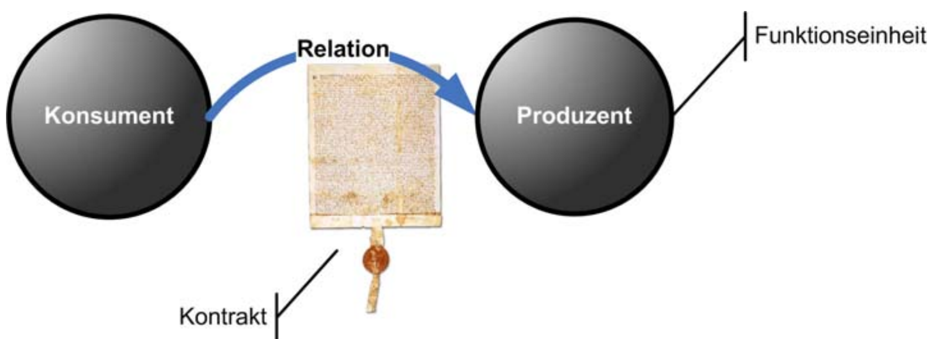

Abb. 27–4 *Relationen zwischen Funktionseinheiten sind gerichtet und durch einen Kontrakt definiert.*

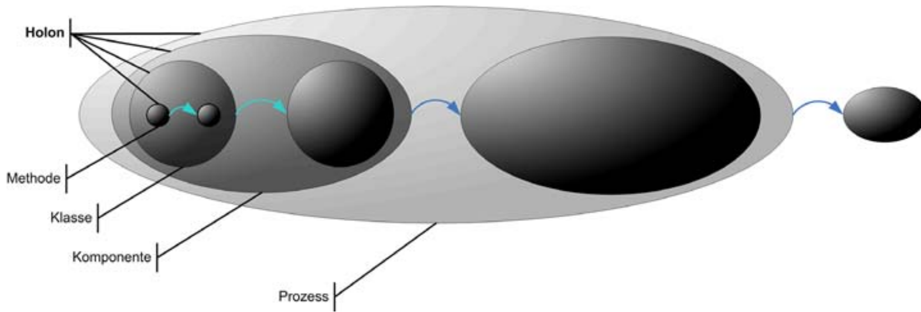

Abb. 27–5 *Ineinandergeschachtelte Codeeinheiten können als Hierarchie von Holons (Holarchie) verstanden werden.*

Codeeinheiten können daher ganz allgemein als Funktionalitäten betrachtet werden, die gleichzeitig Ganzes in Bezug auf ihre Teile sind wie auch Teil von etwas größerem Ganzen. Der in [Koestler 1990] eingeführte Begriff *Holon* – gebildet aus *holos* (griechisch für »das Ganze«) und dem Suffix *-on* (griechisch für »ein Teil von«) wie in Proton – fasst diese Dualität griffig zusammen und führt zur *Holarchie* als der Hierarchie ineinandergeschachtelter Holons.

Wie dann die Relation zwischen Konsument-Holon und Produzent-Holon konkret aussieht, beschreibt ein Kontrakt. Der Konsument kennt ihn und kann ihm gemäß Kontrolle über den Produzenten ausüben. Der Produzent implementiert ihn und verspricht damit seine Erfüllung in funktionaler wie nichtfunktionaler Hinsicht.

Die Besonderheit der Serviceorientierung in Abgrenzung zur Objektorientierung oder auch verteilten Softwaresystemen zeigt sich nun in ihrem Blick auf die Relationen zwischen Holons. [Box 2004] definiert vier Grundsätze serviceorientierter Softwaresysteme:

1. Services sind autonom.
2. Services haben explizite Grenzen.
3. Services beziehen sich auf gemeinsame Schemata und Kontrakte, nicht Klassen.
4. Servicekompatibilität wird durch Policies bestimmt.

Eine Bezugnahme auf eine bestimmte Ebene in der Holarchie der Codeeinheiten ist darin allerdings nicht enthalten. Services mögen heute zwar meist als Prozesse oder Gruppen von Prozessen realisiert werden – strenggenommen ist das aber nicht nötig. Auch eine Komponente innerhalb eines Prozesses kann als Service ausgelegt werden, wenn sie den obigen vier Grundsätzen folgt.

In den Grundsätzen ist nun die einzige Aussage in Bezug auf Service-Holons, sie sollten autonom sein. Das heißt beispielsweise, sie verrichten ihre Arbeit selbstbestimmt, lassen sich unabhängig von anderen Services deployen/austauschen und begrenzen bewusst die Auswirkungen interner Fehler auf ihre Konsu-

menten. Services sind für ihre Umwelt schlicht Black Boxes. Sie realisieren quasi idealtypisch die softwarearchitektonischen Konzepte von Kapselung und hoher Kohäsion.

Alle anderen Aussagen beziehen sich auf die Relationen zwischen Holons. Die syntaktischen Aspekte werden durch Schemas/Kontrakte spezifiziert, wobei diese ausdrücklich in Form und Beschreibungsort unabhängig von Konsument wie auch Produzent sind. Für serviceorientierte Softwaresysteme gilt, dass Kontrakte vor den Implementationen der kommunizierenden Holons festzulegen sind (*Contract-First Design,* [Skonnard 2005]). Die semantischen Aspekte von Relationen werden hingegen durch Policies spezifiziert.

Die expliziten Grenzen scheinen sich zwar auf die Holons zu beziehen, indem sie physische Separation von Services nahelegen. Die wahre Umfriedung des Codes, der einen Service implementiert, findet jedoch nicht durch seine Verpackung in ein Holon einer bestimmten Holarchieebene statt. Prozesse oder die AppDomains des .NET Frameworks mögen helfen, den »Schutzwall« um die Autonomität von Servicecode höher aufzuschütten. Letztlich erfolgt die explizite Grenzziehung aber vor allem durch die explizite Definition von Kommunikationspunkten. Zu jeder Mauer gehören eben Tore. Oder umgekehrt: Alles, was nicht Tor ist, ist Mauer. Nur diese Tore sind dann die Ansatzstellen für Relationen (Abb. 27–6).

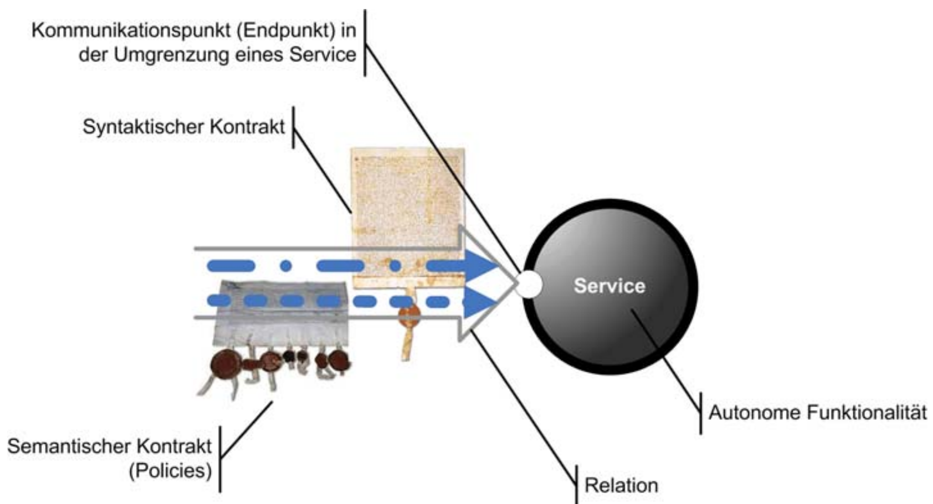

Abb. 27–6 *Die Grundsätze serviceorientierter Softwaresysteme beziehen sich vor allem auf die Relationen zwischen Services.*

Wenn drei von vier Grundsätzen servicerorientierer Softwaresysteme sich auf die Relationen von Services beziehen, dann ist nicht von der Hand zu weisen, dass Kommunikation zentral für die Serviceorientierung ist. Daran ändert auch ein

erweiterer Kritierienkatalog wie der in [Erl 2006] nichts. Auch dort bezieht sich die Mehrheit der Eigenschaften auf die Relationen zwischen Services. Es sollte daher selbstverständlich sein, Relationen im Programmcode möglichst verständlich und klar zu formulieren.

27.2 Die trügerische Form des Kommandotons

Services sind Software-Holons in einer Konsument-Produzent-Beziehung. Der eine wünscht sich Dienstleistungen vom anderen. Das bedeutet, die syntaktischen und semantischen Kontrakte, die an einem Tor in der Mauer um einen Produzenten gelten, durch das seine Konsumenten mit ihm kommunizieren dürfen, werden weitgehend durch eben diese Konsumenten bestimmt. Der Kunde ist auch hier König: Erfüllt ein Service nicht die Wünsche seiner Kunden in Form und Inhalt, dann wird er unattraktiv. Contract-First Design ist also Verhandlungssache zwischen den (potenziellen) Konsumenten und dem Produzenten bzw. ihren jeweiligen Entwicklern, um die Services maximal nah an den Wünschen ihrer Nutzer zu orientieren.

Während beim Serviceentwurf aber noch Diplomatie und Konsens zu herrschen scheinen, ist es damit zur Laufzeit vorbei. Dann herrscht meist kindlich-trotziger Kommandoton. »Ich will!« ist die wiederkehrende Formulierung für Wünsche an die Dienstangebote eines Produzenten innerhalb seiner Konsumenten:

```
Produzent p;
…
Arbeitspaket a;
Ergebnis e;
…
e = p.TueEtwas(a);
```

Die Aufforderungen eines Konsumenten – *TueEtwas()* – sind unmissverständlich und wenden sich direkt an den Produzenten (*p*). Es sind platte Befehle, die in den gängigen objektorientierten Hochsprachen die Form eines Methodenaufrufs haben. Ob dabei das Arbeitspaket für den Produzenten nun in Form eines Nachrichtenobjektes oder mittels mehrerer Parameter übergeben wird, ist zweitrangig. Die grundsätzliche Nachrichtenorientierung, die die Form der Kommunikation zwischen Services haben soll (vgl. [Box 2004]), ist am Aufrufort nicht wirklich erkennbar.

Die Serviceorientierung macht über die konkrete Formulierung der Kommunikation zwischen Services bewusst keine Aussage. Ihr sind Implementationsdetails einerlei. Schemata, Kontrakte, Policies sind plattformunabhängig. Ob ein Service zur Laufzeit Objekte instanziiert oder wie er seine Leistungen realisiert, soll für seine Konsumenten unerheblich sein.

Wenn also Konsumenten Methodenaufrufe benutzen, um sich ihre Wünsche von Produzenten-Services erfüllen zu lassen, dann ist das eine Entscheidung derjenigen, die die Konsumenten implementieren.

Ist das aber eine gute Entscheidung? Diese Frage ist berechtigt, denn nur weil diese Form des Kommandotons zwischen Holons Tradition hat und naheliegt, muss sie nicht auch automatisch »gut« sein. Gut wäre sie, wenn die Form der Funktion angemessen wäre.

Sind Methodenaufrufe auf Services also eine angemessene Form für die Kommunikation zwischen autonomen, lose gekoppelten Holons? Um diese Frage zu beantworten, ist eine Analyse nötig, wofür die Form der direkten Methodenaufrufe eigentlich steht. Ihre Konnotationen sind nämlich vielfältig und selten bewusst. Aus der Geschichte von Unterprogramm- und Methodenaufrufen ergibt sich dennoch leicht, welche Erwartungen an ihre Form geknüpft werden. Ein Befehl wie

```
z = TueEtwas(x, y);
```

verspricht:

- Verzögerungsfreier Beginn der Ausführung und möglichst verzögerungsfreie Ablieferung des Resultats
- Komplette Abgabe der Kontrolle an den Produzenten, d.h., dieser arbeitet auf demselben Thread wie der Konsument
- 100%ige Verfügbarkeit des Produzenten
- 100% sichere Kommunikation zwischen Konsument und Produzent
- 100% ausfallsichere Kommunikation

Daran ändert sich auch nichts, wenn der Aufruf objekt- oder gar nachrichtenorientierter formuliert ist, wie z.B.

```
e = p.TueEtwas(a);
```

Mit der Syntax von Methodenaufrufen verbindet jeder Softwareentwickler einfach vor allem die 99,99% aller Fälle, in denen die Form tatsächlich ihre Versprechen erfüllt.

Im Falle der Übertragung einer Aufgabe an einen Service kann dieses Versprechen jedoch nicht erfüllt werden. Im Falle von Serviceaufrufen trügt der Schein der Form des direkten Methodenaufrufs. Immer! Der Grund dafür liegt im Willen der Serviceorientierung, Implementationsdetails von Services konsequent zu verbergen (Black-Box-Denken) und Services autonom, d.h. möglichst unabhängig von ihrer Umgebung zu machen, zu der auch ihre Konsumenten gehören.

Services bauen dazu eine Mauer auf und regeln den Zugang zu sich minutiös. Sie distanzieren sich also bewusst von anderem Code. Diese Distanz kann physisch sein, indem ein Service in einen eigenen Prozess oder gar auf einen eigenen Rechner ausgelagert wird. Sie kann aber auch nur »sozial« sein, indem die Ser-

vicenutzung z.B. speziellen Anforderungen an die Datenqualität unterliegt, dennoch aber physisch nah an ihren Konsumenten stattfindet.

Die Serviceorientierung macht also nicht nur keine Aussagen über die Holarchieebene, auf der Services zu realisieren sind. Sie macht auch keine Aussage darüber, mittels welcher Technologien Services kommunizieren sollen. Dass heute vor allem XML-Webservices zur Kommunikation eingesetzt werden, ist kein Ergebnis einer Forderung der Serviceorientierung. Services können auch über das Dateisystem oder den Stack kommunizieren.

Dennoch ist die Distanz – physisch oder »sozial« – zwischen Service-Holons immer größer als zwischen anderen Holons. Das ist der Preis von Autonomieforderung, lückenloser Kapselung und daraus folgender Notwendigkeit zu loser Kopplung. Für die Konsumenten eines Produzenten-Services hat das die Konsequenz, dass sie keine Annahmen über den Service jenseits der ihnen bekannten Kontrakte und Policies machen dürfen. Das wiederum bedeutet für die Versprechen der Kommunikationsform Methodenaufruf:

- Es ist ungewiss, ob die Ausführung der Methode verzögerungsfrei beginnt.
- Es ist ungewiss, ob der Produzent synchron, d.h. auf demselben Thread arbeitet.
- Es ist ungewiss, ob überhaupt ein Produzent immer verfügbar ist.
- Es ist ungewiss, ob die Kommunikation zwischen Produzent und Konsument sicher ist.
- Es ist ungewiss, ob die Kommunikationsstrecke ausfallsicher ist.

Ein direkter Methodenaufruf auf einem Produzenten-Service ist somit ein recht hohles Versprechen. Er steht dann für nichts mehr von dem, wofür seine Syntax ursprünglich geschaffen wurde. Und das ist nicht nur bei Serviceaufrufen so, sondern bei allen Aufrufen entfernter Funktionalität, d.h. wann immer eine Thread- oder Adressraumgrenze überschritten wird.

27.3 Form does not follow function, yet

Die syntaktische Form der Prozeduraufrufe wurde geschaffen für einen bestimmten Zweck, für eine bestimmte Funktionalität. Sie abstrahiert von den Feinheiten des aus einer Hochsprache erzeugten Maschinencodes, der Parameter und Rückgabewert über einen Stack oder Register zwischen Konsument-Methode und Produzent-Methode bewegt. Bei objektorientierten Sprachen kommt sogar noch eine Indirektion im Falle virtueller Methoden hinzu.

Die syntaktische Form der Prozeduraufrufe wurde damit für eine Inter-Holon-Kommunikation innerhalb desselben Adressraums auf demselben Thread geschaffen (Abb. 27–7).

RPC (sic!), Technologien für verteilte Objekte/Komponenten (z.B. COM+, CORBA) und nun auch XML-Webservices oder WCF (Windows Communication Foundation) – Microsofts Basistechnologie für die Kommunikation in serviceorientierten Softwaresystemen – haben jedoch die Inter-Holon-Kommunikation

ab Prozesseebene zum Thema. Obwohl die Modalitäten der Kommunikation also grundsätzlich anders sind als die, für die die syntaktische Form der Prozeduraufrufe ursprünglich entwickelt wurde, benutzen diese Technologien sie. Die Form entspricht damit nicht mehr der grundsätzlichen Funktion.

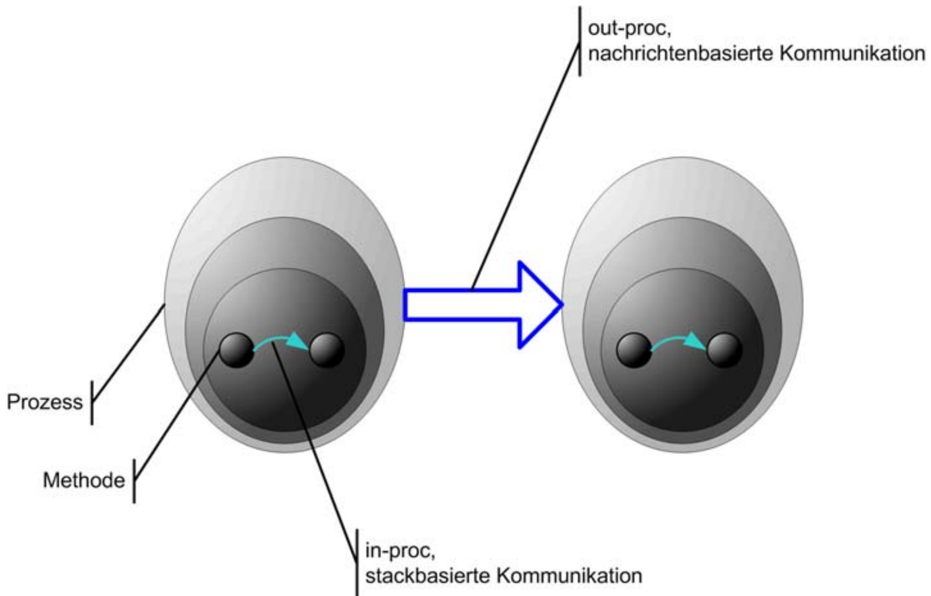

Abb. 27–7 *Die grundsätzliche Art der Kommunikation zwischen Holons wechselt beim Übergang von einem Adressraum in einen anderen.*

Dass das auf Dauer nicht ohne Folgen bleiben konnte, haben die 90er Jahre gezeigt, in denen Interprozesskommunikation zu einem Massenphänomen geworden ist. War RPC vorher die Domäne weniger Spezialisten, so haben in der Microsoft-Welt DDE, OLE Automation, MTS, COM+, Webservices, .NET Remoting und nun WCF schrittweise die Kommunikation mit entfernten Holons quasi trivialisiert. Das Motto schien zu lauten: Remote objects at your fingertips. Für C++ und Java verlief die Entwicklung vergleichbar. Die Folgen dieser technischen Trivialisierung von Kommunikation über größere Entfernungen waren jedoch nicht einfach nur einhellige Freude und Aufatmen, sondern anhaltende Probleme bei Performance und Skalierbarkeit. Beweis dafür sind die »Fallacies of Distributed Computing« [Fallacies], die Peter Deutsch und andere in den 90er Jahren formulieren mussten, um deutlich zu machen, dass Interprozesskommunikation fundamental anders funktioniert als Methodenaufrufe innerhalb eines Adressraums.

Eine Analogie für diesen Unterschied ist der Vergleich des Umgangs mit einem Fernsteuerauto und dem Mars Sojourner (Abb. 27–8): Ein Fernsteuerauto ist ständig im Blick des Steuernden, die Informationsübertragung von der Steuer-

einheit zum Auto ist quasi sofortig. Es kann auf Sicht gefahren werden. Der Mars Sojourner ist vielleicht auch im Kontrollzentrum der NASA sichtbar, er fährt allerdings mehrere Millionen Kilometer entfernt auf dem Mars herum. Das, was auf der Erde gesehen wird, ist also immer mehrere Minuten alt – und genauso lange bräuchten Steuerungsimpulse, um das Gefährt auf dem Mars zu erreichen. Die relativistische Physik lässt eine direkte Steuerung auf Sicht wie beim Spielzeugauto also nicht zu. Daraus folgt für die NASA, dass die Steuerung des Sojourners eine andere Form haben muss. Im Kontrollzentrum sitzt deshalb niemand mit einem Steuerknüppel in der Hand, um den Sojourner um einen Stein herum zu lenken.

Die Form der (drahtlosen) Steuerung eines Gerätes in der realen Welt ist somit abhängig von dessen Entfernung. Die Form der Steuerung folgt ihrer internen Funktion.

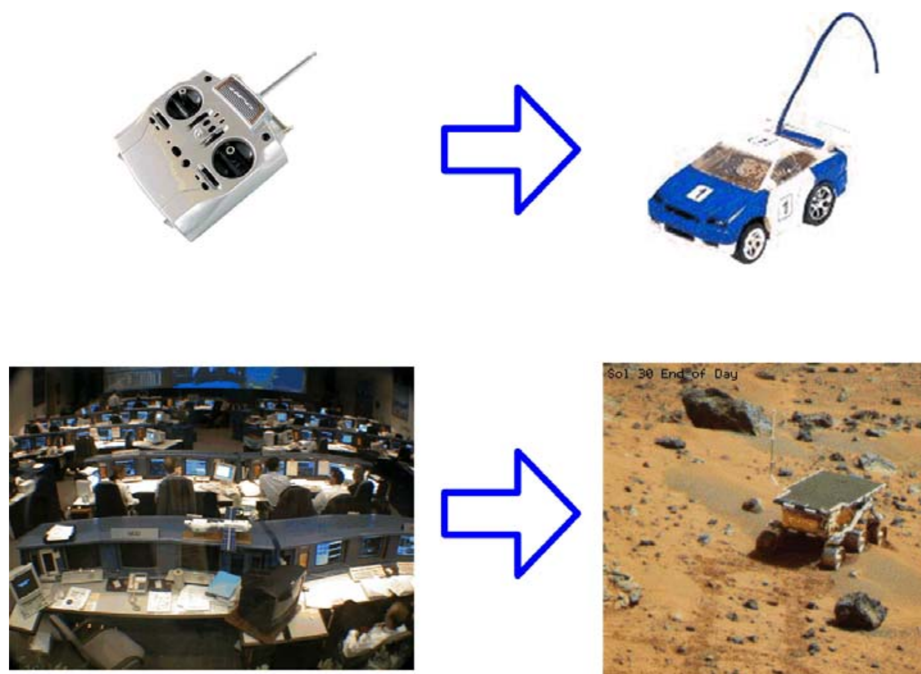

Abb. 27–8 *Die Art der Steuerung von Spielzeugauto und Mars Sojourner unterscheidet sich grundsätzlich aufgrund der unterschiedlichen Entfernungen.*

Auf die Welt der Kommunikation zwischen Software-Holons übertragen, muss die Erkenntnis lauten: Die Syntax des Methodenaufrufs ist die falsche Form für den Anstoß von Operationen in Services bzw. entfernten Holons jeder Art. Denn nicht nur in der realen Welt sollte der Gestaltungsleitsatz *form follows function* (FFF, [Wikipedia/FFF]) für Design und Architektur gelten.

Auch die Softwareentwicklung kann davon profitieren, dass sie die Form von Konstrukten ihrer Funktion anpasst. Es geht schlicht um das hohe Gut der Verständlichkeit von Code. Warum sollte nicht ein Blick auf ein Stück Code unmissverständlich klar machen, welche Funktionsweise dahintersteckt? In der realen Welt gelingt das ja auch, wie Abbildung 27–9 zeigt: Jeder erfasst sofort und intuitiv und verlässlich, wie diese Türen zu öffnen sind. Das ist gutes Design! Das ist hohe Verständlichkeit.

Abb. 27–9 *Wie diese Türen zu öffnen sind, ist intuitiv erfassbar und eindeutig, weil die Form des »Türgriffs« der Funktionsweise des Öffnungsmechanismus angepasst ist.*

Der Codeausschnitt

```
e = p.TueEtwas(a);
```

hingegen ist missverständlich. Ihm kann man nicht ansehen, wie die Kommunikation mit dem Produzent-Holon *p* funktioniert:

- Wie schnell läuft sie ab?
- Wie sicher ist sie?
- Ist der Produzent verfügbar?

Diese Art Unsicherheit und Missverständnis ist kontraproduktiv. Sie führt zu den wiederkehrenden Trugschlüssen über die Funktionsweise verteilter Systeme [Fallacies]. Sie behindert grundsätzlich die Entwicklung eines Verständnisses für serviceorientierte Softwaresysteme.

27.4 How form should follow function

Kommunikation ist in serviceorientierten Softwaresystemen keine Nebensache, sondern die Hauptsache. Es ist daher wünschenswert, der Hauptsache eine angemessene, unmissverständliche Form zu geben. Die Kommunikation zwischen physisch oder »sozial« entfernten Software-Holons sollte im Quellcode also nicht durch die Syntax eines Methodenaufrufs repräsentiert werden. Nur so wären Missverständnisse zu vermeiden. Derzeit gibt es jedoch keine Alternative zu Methodenaufrufen, wenn Operationen von einem Holon an ein anderes delegiert werden sollen.

Über alternative grundsätzliche Formen für den Anstoß entfernter Operationen ließe sich natürlich nachdenken, doch sie würden weitreichende Änderungen an den gängigen Hochsprachen erfordern. Pragmatischer im Hinblick auf eine Auflösung der Missverständlichkeit heutiger Kommunikationsnotation in verteilten Systemen scheint daher, die gegebene Syntax hinzunehmen – und stattdessen ihre Nutzung zu optimieren.

Was ließe sich also am Anstoß einer Operation in einem Produzent-Service wie

```
e = p.TueEtwas(a);
```

optimieren? Die Versprechen der Syntax müssen eingelöst werden! Das kann aber nur geschehen, wenn das aufzurufene Holon (*p*) »nahe« ist. Dies ist für einen Service *per definitionem* allerdings nie der Fall – und auch im Allgemeinen nicht für ein Holon auf einem anderen Thread oder in einem anderen Adressraum.

Solange die Form nicht verändert werden kann, liegt die Lösung deshalb darin, die Funktion zu verändern, damit sie wieder der Form folgt. Den Anstoß einer Operation in einem Produzent-Service muss also die Kommunikation mit einem nahen Software-Holon geben. Und da der Produzent nie »nah« ist, muss es zwangsläufig ein Drittes sein (Abb. 27–10).

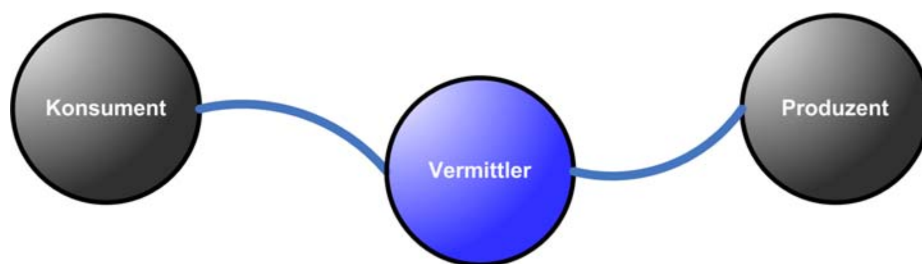

Abb. 27–10 *Services – oder allgemeiner: entfernte Software-Holons – sollten ausschließlich über einen Vermittler kommunizieren.*

Die Kommunikation zwischen Services – oder allgemeiner: entfernten Software-Holons – sollte somit immer über einen Vermittler laufen. Der Konsument spricht

mit dem Vermittler, der Vermittler leitet dies weiter an den Produzenten; der Produzent übergibt dem Vermittler ein Operationsresultat, der Vermittler leitet es zurück an den Produzenten.

Prinzipiell läuft die Kommunikation zwar auch heute so ab, denn eine Konsument-Methode kann nur vermittels eines Proxy-Stub-Paares eine entfernte Produzent-Methode z.B. als XML-Webservice aufrufen. Ziel der aktuellen Kommunikationstechnologien ist es jedoch, genau das zu verbergen und die Illusion zu vermitteln, Konsumenten würden direkt mit Produzenten verkehren. Ein Proxy-Stub-Paar ist daher kein Vermittler im geforderten Sinn, sondern nur ein übertünchtes Implementationsdetail der Kommunikationstechnologien. Ein Vermittler, der nicht als solcher zu erkennen ist, erfüllt nicht den Zweck, das Versprechen der Methodensyntax einzulösen.

Einen sichtbaren Vermittler zwischen Produzent und Konsument einzuführen ist allerdings auch nur ein erster Schritt in Richtung »ehrlicherer« Kommunikation in serviceorientierten Softwaresystemen. Es reicht nicht,

```
e = p.TueEtwas(a);
```

durch

```
e = vermittler.TueEtwas(a);
```

zu ersetzen. Der Vermittler wäre auch in diesem Fall nur ein Proxy und damit quasi unsichtbar. Der Vermittler muss sich vielmehr in der Semantik vom Produzenten unterscheiden. Nur dann ist er klar als solcher erkennbar. Große physische oder »soziale« Distanz zwischen Konsument und Produzent sollte sich also in einer semantischen Distanz im Code manifestieren.

Die Servicerorientierung formuliert diese Distanz in ihrem Grundsatz, Services sollten nicht Klassen an ihren Schnittstellen gemeinsam haben, sondern implementationsunabhängige Kontrakte. Zusammen mit der nachrichtenorientierten Kommunikation zwischen entfernten Software-Holons führen sie zu einem Impedance Mismatch zwischen den serviceinternen Datenmodellen und dem Datenmodell ihrer Relationen (Abb. 27–11). Auch diesen Unterschied kaschieren die aktuellen Kommunikationstechnologien, wenn sie den Produzentenaufruf direkt erscheinen lassen.

So erstrebenswert hohe Abstraktionsniveaus in der Softwareentwicklung sind, sie sollten sich nicht so weit von der Realität entfernen, dass Missverständnisse unausweichlich sind. Joel Spolskys *Law of Leaky Abstractions* (LOLA [Spolsky]) gilt auch für serviceorientierte Softwaresysteme.

Wie ein Vermittler zwischen Services intern funktioniert, ist selbstverständlich vor Produzenten und Konsument über ein Programmiermodell auf geeignetem Abstraktionsniveau zu verbergen. Dass jedoch ein Vermittler im Spiel ist und er eine Operation nicht direkt ausführt, sondern nur irgendwie vom Konsumenten an den Produzenten weiterleitet, sollte explizit sein. Nur so kann den erweiterten Grundsätzen der Serviceorientierung aus [Erl 2006] Rechnung getragen

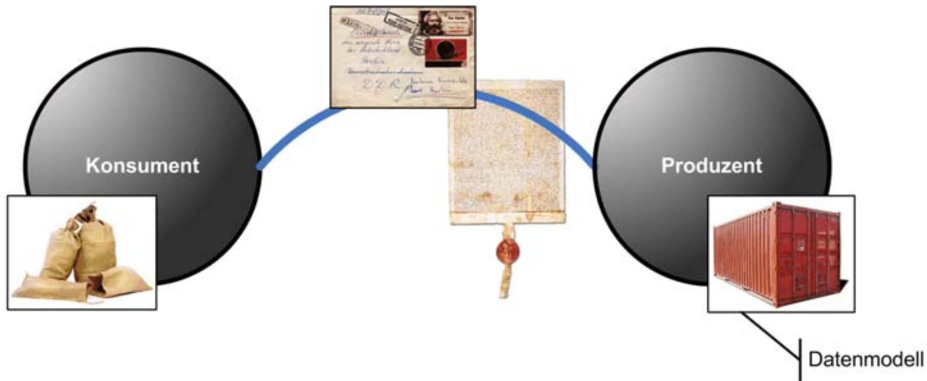

Abb. 27–11 *Konsument, Produzent und Kommunikationstechnologie implementieren einen Kontrakt u.U. sehr verschieden und führen damit zu Datenmodellen mit großem Impedance Mismatch.*

werden. Echte Entkopplung gibt es nur, wenn Services weder tatsächlich noch suggeriert direkt miteinander kommunizieren. Echte Entkopplung setzt einen deutlich sichtbaren, deutlich anders als Produzenten funktionierenden Vermittler voraus.

Wie lässt sich solche Andersartigkeit eines Vermittlers ausdrücken? Wodurch unterscheidet sich seine Semantik von der des eigentlich vom Konsumenten angepeilten Produzenten? Vermittler unterscheiden sich durch ihr Programmiermodell, d.h. durch ihre Funktionen von Produzenten. Wenn ein Produzent etwas tun soll, dann soll ein Vermittler etwas vermitteln. Ein Vermittler kann und darf also nicht die Operationen anbieten, die Konsumenten auf Produzenten erwarten.

Die allgemeine Form einer »Vermittlungsbemühung« hat daher sechs Bestandteile:

1. Vermittler
2. Operation des Vermittlers
3. Identifikation eines Produzenten
4. Anzustoßende Operation auf Produzent
5. Parameter für die Produzentenoperation
6. Vereinbarung zur Rückgabe eines Operationsresultats vom Produzenten zum Konsumenten

Besonders beachtenswert ist dabei:

- Vermittleroperation und Produzentenoperation unterscheiden sich deutlich; sie gehören unterschiedlichen semantischen Bereichen an.
- Wie Produzentenoperation und Parameter kodiert werden, ist für das Prinzip der Vermittlung unerheblich. Sie können in einem Parameter der Vermittleroperation zusammengefasst werden oder aus einer Parameterliste bestehen.
- Von entfernten Software-Holons und insbesondere Services ist nicht zu erwarten, dass ihre Operationen synchron zu ihren Konsumenten laufen.

Operationsresultate sollten daher nicht von der Vermittlungsoperation erwartet werden. Um sie zu empfangen, ist stattdessen eine explizite Vereinbarung zu treffen. Sie kann viele Formen annehmen, z.B. Rückruffunktion (Callback, Eventhandler) oder erneuter Vermittleraufruf mit Bezugnahme auf den Operationsanstoß im Produzenten (Korrelation).

Einige Beispiele für den Umgang mit Vermittlern mögen die Form der indirekten Kommunikation verdeutlichen.

Anstoß einer Operation mit expliziter Benennung, Parameterliste und ohne Resultat. Der Produzent wird an den Vermittler nur einmal gebunden:

```
Vermittler v;
…
v.Binden(produzentenidentifikation);
…
v.Vermittle("TueEtwas", a, b, c);
```

Operation und Parameter werden zusammen kodiert, der Produzent wird bei jeder Vermittlung spezifiziert. Auch hier wird kein Resultat erwartet:

```
Vermittler v;
…
TueDiesParameter patd;
…
v.Vermittle(patd, produzent1);
…
TueAnderesParameter pata;
…
v.Vermittle(pata, produzent2);
```

Wenn ein Konsument das Resultat eines Produzenten für den Fortgang seiner Arbeit braucht, sollte er den Vermittler bitten, darauf zu warten:

```
Vermittler v;
…
Korrelationsinfo k;
k = …;
…
v.Vermittle("TueEtwasMitResultat", …, produzent3, k);

Ergebnis e;
e = v.WarteAufAntwort(produzent3, k);
```

Die Korrelationsinformation kann auch vom Vermittler erzeugt werden und den Produzenten, auf den gewartet wird, enthalten:

```
Vermittler v;
…
Korrelationsinfo k;
k = v.Vermittle("TueEtwasMitResultat", …, produzent3);

Ergebnis e;
e = v.WarteAufAntwort(k);
```

Für den asynchronen Empfang von Resultaten bietet sich die Registrierung eines Callbacks beim Vermittler an:

```
Vermittler v;
…
Korrelationsinfo k;
k = …;
…
v.Vermittle("TueEtwasMitResultat", …, produzent4, k, callback);
```

Diese Form der Kommunikation verspricht nichts, was sie nicht halten kann! Der Vermittler hat selbstverständlich eine Repräsentation im Prozess des Konsumenten, so dass für Methodenaufrufe auf ihm Form und Funktion Hand in Hand gehen. Er übernimmt sofort, synchron, sicher und verlässlich die Aufgabe der Vermittlung. Dass die an den Produzenten zu übermittelnde Operation dann hinter den Kulissen später, asynchron, unsicher oder gar unzuverlässig abläuft, widerspricht nicht der Form. Das Angebot des Vermittlers ist ja *per definitionem* »Ich vermittle nur – und zwar prompt. Wann und wie die Arbeit erledigt wird, fällt nicht in meinen Zuständigkeitsbereich.« Die Form des Methodenaufrufs muss nur dieser prompten Vermittlung entsprechen, nicht der irgendwie gearteten Erledigung einer Aufgabe.

Das Abstraktionsniveau einer vermittelten Operation im Vergleich zu ihrem vermeintlich direkten Aufruf ist nur marginal niedriger. Die Feinheiten der Kommunikation zwischen Produzent und Konsument sind also weiterhin verborgen.

In Ermangelung einer ehrlicheren Form für direkte entfernte Aufrufe sind die Beauftragung eines Vermittlers und die Trennung von Operationsanstoß und Resultatsempfang heute der ehrlichste Weg zu einer Kooperation in verteilten Softwaresystemen.

27.5 Kooperation über gemeinsame Datenstrukturen

Welcher Art sollten Vermittler zwischen entfernten Software-Holons sein? Ist immer ein Enterprise Service Bus (ESB) einzusetzen? Reicht ein Warteschlangenserver? Vermittler zu benutzen, um Konsumenten von Produzenten zu entkoppeln, ist ja nicht neu. Sie allerdings konsequent zwischenzuschalten, wenn sich die Distanz zwischen kommunizierenden Software-Holons vergrößert, so dass das Versprechen des Methodenaufrufs hohl wird, das ist neu. Denn das betrifft nicht nur die Software- oder Servicearchitektur (SOA) ganzer Unternehmen, sondern jede verteilte Applikation, und bestehe sie aus nur zwei Threads oder AppDomains, geschweige denn mehreren Prozessen.

Die Illusion der direkten Kommunikation lauert überall. Trugschlüsse werden oft schneller gezogen, als man denkt.

Der hier vertretene Grundsatz, alle entfernte Kommunikation solle immer über einen klar erkennbaren Vermittler ablaufen, ist daher natürlich ganz allge-

mein zu verstehen und bezieht insofern ESB und Warteschlangen mit ein. Sie sind legitime Vermittler, deren Einsatz von den Architektur- und Kommunikationsbedürfnissen eines Szenarios abhängt – und natürlich auch vom Projektbudget. Entscheidend ist ausschließlich, dass sie sich im Konsumenten nicht als Produzenten ausgeben. Ihre eventuelle Kapselung sollte dort also nicht so weit getrieben werden, dass wieder die Illusion einer direkten, lokalen Operation entsteht.

ESB und Warteschlangenserver sind konkrete Vermittler mit konkreten Programmiermodellen. Ihre Semantik unterscheidet sich hinlänglich von denen entfernter Produzenten-Holons für geschäftliche Problemdomänen. Lassen sich von ihnen aber vielleicht auch allgemeine Hinweise für andere Vermittler ableiten? Wie sollten deren Programmiermodelle beschaffen sein, um Missverständnisse zu vermeiden?

Zweierlei lohnt sich, dafür in den Blick zu nehmen: das Verhältnis von Konsument und Produzent sowie die Natur der Kommunikation zwischen beiden.

Konsumenten und Produzenten, die sich nahe und eng gekoppelt sind, haben ein tiefes »Vertrauensverhältnis« zueinander und leben bewusst in hoher Abhängigkeit voneinander. Zwischen ihnen kann daher der »Kommandoton« direkter Methodenaufrufe herrschen. Ihr Verhältnis ist das einer Befehlshierarchie.

Je größer die physische oder »soziale« Distanz zwischen Software-Holons, desto größer aber auch deren Wille zur Autonomität. Den Services ist sie schon in den vier Grundsätzen ins Stammbuch geschrieben. Autonomität verträgt jedoch keinen Befehlston. Auch dies ist ein Grund, warum insbesondere Services nicht direkt von Konsumenten »herumkommandiert« werden sollten. Aus Sicht des Services widerspricht das seinem Willen zur Selbstbestimmung. Und aus Sicht der Konsumenten widerspricht das ihrer Unabhängigkeit, denn wer mit autonomen Holons kommuniziert, will z.B. nicht zu sehr auf eines festgelegt sein, wenn das aus seiner Autonomität heraus entscheidet, gerade nicht verfügbar zu sein.

Die Form enger Kopplung zwischen Software-Holons drückt sich also in Kommandoton und Befehlshierarchie aus; lose Kopplung und Autonomität hingegen verlangen Kooperation. Autonome bzw. entfernte Software-Holons haben zwar immer noch ein gemeinsames Ziel – die Erfüllung von Anwenderwünschen –, aber sie erreichen es auf andere Weise als monolithische, eng gekoppelte Softwaresysteme. In verteilten, lose gekoppelten Systemen wird das Ziel nicht durch Befehlskaskaden erreicht, sondern durch die Koordination der Leistungen von beteiligten Software-Holons.

Wo der Kommandoton instruktiv ist, funktioniert Kooperation konnotativ. Der Konsument bestimmt nicht direkt mit Instruktionen das Verhalten des Produzenten. Stattdessen stellt der Konsument seinen Wunsch »in den Raum«, und der Produzent lässt sich dadurch zu einem Verhalten »anregen« (Konnotation); der Produzent bestimmt also autonom sein Verhalten durch Betrachtung des Konsumentenwunsches.

Kooperation vermittels konnotativ angeregten Verhaltens braucht nun aber einen Raum, der Konsumenten wie Produzenten gemeinsam ist (Abb. 27–12). In

diesen Raum stellen Konsumenten ihre Wünsche, in diesem Raum »betrachten« Produzenten Wünsche.

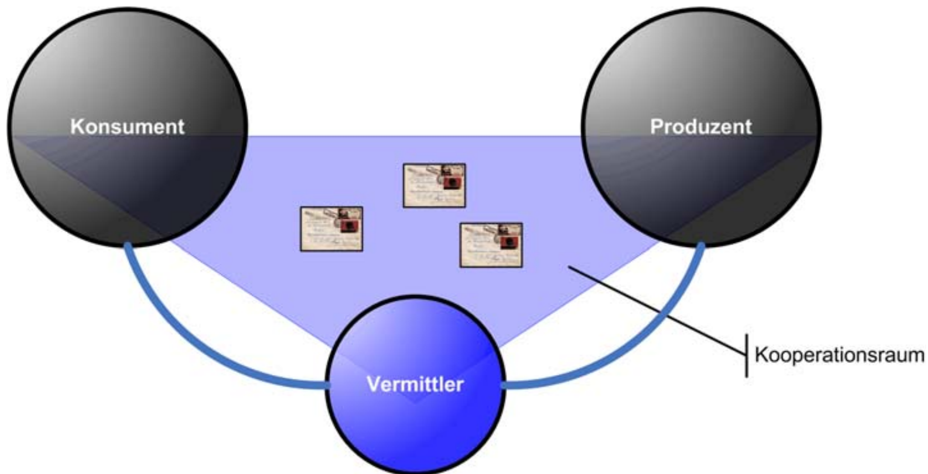

Abb. 27–12 *Konsumenten und Produzenten koordinieren ihre Kooperation über einen Kooperationsraum.*

Beispiele für solche Kooperationsräume im realen Leben sind der Posteingangskorb auf einem Schreibtisch oder die Bug-Liste in einer Issue-Tracking-Software oder auch das Fernsehprogramm. Sie entkoppeln Konsumenten/Auftraggeber und Produzenten/Auftragnehmer; sie machen den Produzenten unabhängig in seinen Entscheidungen, wie und wann er die Wünsche von Konsumenten erfüllt. Kooperationsräume sichern die Autonomität der Kooperationspartner.

In der Natur von Räumen liegt es nun, dass sie selbst keine Aufgaben erfüllen, sondern nur ihren Zugang regeln und einen Kontext herstellen. Räume sind keine Dienstleister, sondern Behältnisse. Und so steht ein Kooperationsraum für verteilte Software-Holons nicht für Funktionalität, sondern stellt eine Datenstruktur dar. Entfernte Holons koordinieren ihre Arbeit also über Konnotationen getragen von Kooperationsdatenstrukturen – oder kurz Kooperationsstrukturen bzw. Koordinationsstrukturen – statt durch Instruktionen.

Aufgespannt werden Kooperationsräume durch die Vermittler zwischen Software-Holons. Sie stellen die Kooperationsstrukturen dar, auf die verteilte Software-Holons zugreifen. Und da Datenstrukturen immer in einem Speicher existieren, können Kooperationsstrukturen als Datenstrukturen in einem gemeinsam genutzten Speicher (*Shared Memory)* angesehen werden. Überbrückt ein Vermittler dann die Grenzen verschiedener physischer Speicherbereiche, so spannt er sogar einen virtuellen Speicherbereich auf (*Virtual Shared Memory, VSM*), an dem alle Produzenten und Konsumenten teilhaben, die den Vermittler nutzen.

Daraus lässt sich ganz allgemein für das Programmiermodell von Vermittlern ableiten, dass sie ihre Natur als Datenstruktur hervorkehren müssen. Die Funk-

tionen von Vermittlern müssen deutlich machen, dass sie den Zugang zu einem Datenraum regeln. Sie beschreiben die Strukturen des Raumes und stehen nicht für Instruktionen gerichtet an Produzenten. Vermittleroperationen beziehen sich also immer auf den Vermittler und nie auf die eventuellen Empfänger des Vermittelten.

Eine Wartschlange ist in dieser Hinsicht der prototypische Vermittler zwischen entfernten Software-Holons: Sie implementiert eine Datenstruktur (FIFO), und ihre Operationen (Enqueue, Dequeue) machen Konsumenten deutlich, wie diese Datenstruktur aufgebaut ist. Sie definieren sogar ein asymmetrisches Verhältnis zwischen Konsument und Produzent und regeln den Zugang zur Datenstruktur unterschiedlich für beide Seiten.

Produzente/Auftragnehmer:

```
Vermittler v;
…
VermittlerQueue q;
q = v.CreateQueue("Queuename");
…
Arbeitspaket a;
a = q.Dequeue();
```

Konsument/Auftraggeber:

```
Vermittler v;
…
VermittlerQueue q;
q = v.GetQueue("Queuename");
…
Arbeitspaket a;
…
q.Enqueue(a);
```

Eine Warteschlange ist allerdings nur ein Beispiel für eine Kooperationsstruktur. Sie mag oft die geeignete Datenstruktur sein, um Auftraggeber und Auftragnehmer zu entkoppeln – aber es gibt sicherlich Gelegenheiten, in denen Kooperationen leichter über andere Datenstrukturen koordiniert werden können. Listen, Bäume, mehrdimensionale Felder, Hashtabellen ... sie und andere Datenstrukturen sollten als Vermittler gedacht werden können.

Wenn ein entferntes Software-Holon beispielsweise ein Resultat zurückliefern soll, dann ist eine Warteschlange nicht unbedingt geeignet für dessen Rücktransport. Einfacher gestaltet sich die Kooperation durch eine Kombination von Queue und Dictionary:

Produzente/Auftragnehmer:

```
Vermittler v;
…
VermittlerQueue q;
q = v.CreateQueue("Aufträge");
VermittlerDictionary d;
d = v.CreateDictionary("Resultate");
…
Arbeitspaket a;
a = q.Dequeue();
…
Resultat r;
…
d.Add(a.korrelationsId, r);
```

Konsument/Auftraggeber:

```
Vermittler v;
…
VermittlerQueue q;
q = v.GetQueue("Aufträge");
VermittlerDictionary d;
d = v.GetDictionary("Resultate");
…
Arbeitspaket a;
a.korrelationsId = …;
…
q.Enqueue(a);

Resultat r;
r = d[a.korrelationsId];
```

Wenn der Griff in das Dictionary für die Resultate – *d[a.korrelationsId]* – blockierend ist, wartet der Konsument synchron auf die Erledigung seines Auftrags durch einen Produzenten. Das ist kaum aufwendiger als ein vermeintlich direkter Aufruf mit viel Potenzial für Missverständnisse. Es ist aber sehr klar in der Darstellung des Verhältnisses von Konsument und Produzent. Die Autonomität beider Seiten wird gewahrt. Die entfernten Software-Holons werden deutlich entkoppelt, ohne das Abstraktionsniveau schmerzhaft zu senken. Missverständnisse haben hier keinen Raum.

Infrastruktur für Vermittler sollte also so ausgelegt sein, dass sie Kooperationsstrukturen ganz unterschiedlicher Art ermöglicht. Heutige Technologien wie XML-Webservices oder .NET Remoting, COM+/Enterprise Services sowie WCF in der Microsoft-Welt leisten das allerdings noch nicht. Im besten Fall erleichtern sie die Implementation einer FIFO-Datenstruktur. Ihr Abstraktionsniveau ist zu gering, sie kaschieren und vereinheitlichen nur die Details verschiedener low-level Kommunikations-APIs wie Sockets oder Warteschlangen. Solche Konsolidation hebt jedoch das Programmiermodell noch nicht auf das Abstraktionsniveau allgemeiner Vermittlungsdatenstrukturen.

In die richtige Richtung weisen eher Technologien wie Tuple Spaces (z.B. JavaSpaces [JavaSpaces] oder TSpaces [TSpaces]) oder VSM-Implementationen wie XtSpaces [XtSpaces].

27.6 Zusammenfassung

Ein vermeintlich direkter Kontakt zwischen entfernten Software-Holons und allemal autonomen Services widerspricht der heute einzig verfügbaren Form für solche Kontakte, dem Methodenaufruf. Er suggeriert Eigenschaften der Kommunikation, die sie in verteilten Szenarien unmöglich besitzen kann.

Darüber hinaus bedeutet direkter Kontakt immer Kommandoton zwischen entfernten Kommunikationspartnern. Der wiederum steht in keinem Verhältnis zu deren Wunsch nach Autonomität und Skalierbarkeit.

Sowohl aus Gründen der Verständlichkeit wie der angemessenen Repräsentation der Natur von und der Relationen zwischen entfernten Software-Holons verbietet sich ihr (vermeintlich) direkter Kontakt. Kommunikation zwischen ihnen sollte stattdessen immer konsequent über eine vermittelnde Instanz stattfinden.

Diese Indirektion stellt kein Performanceproblem dar, weil entfernte Kommunikation ohnehin um Zehnerpotenzen langsamer als In-Proc.-Kommunikation ist. Ein eventueller Effizienzverlust wird vor allem kompensiert durch eine Flexibilisierung der Organisation von Softwaresystemen. Indirektion entkoppelt.

Vermittler zwischen Kooperationspartnern sind selbst keine Dienstleister im Sinne der Problemdomäne des Softwaresystems, sondern Datenstrukturen. Kooperationspartner koordinieren sich selbstständig durch gemeinsamen Zugriff auf diese Kooperationsstrukturen bzw. Koordinationsstrukturen.

Als Implikation für die Softwaremodellierung ergibt sich: Die Distanz zwischen Software-Holons darf nicht nachträglich erhöht werden. Die Relationen zwischen Konsumenten und Produzenten sind von Anfang an als nah oder fern festzulegen. Die Kommunikation über große Distanzen erfolgt dann immer über einen expliziten Vermittler.

Transparenz in der Kommunikation soll ihre konkrete Implementation verbergen. Ein Methodenaufruf verbirgt Stackoperationen zur Laufzeit. Eine Koordinationsstruktur verbirgt, ob die entfernten Kooperationspartner per Named Pipes, TCP oder Dateisystem verbunden sind.

Die Transparenz in der Kommunikation darf hingegen nicht so weit gehen, dass sie die grundsätzliche Distanz zwischen Kommunikationspartnern verbirgt.

Gesucht sind also Kommunikationstechnologien, die eine Vielfalt von Kooperationsstrukturen ermöglichen. Gesucht ist Infrastruktur, mit der sich ein *Virtual Shared Memory* zwischen entfernten Kooperationspartnern aufspannen lässt. Denn das Motto der Zukunft für verteilte Softwaresysteme lautet: Kooperation statt Kommandoton!

Gernot Starke • Stefan Tilkov (Hrsg.)

SOA-Expertenwissen

Das Konzept der serviceorientierten Architektur (SOA) beeinflusst Geschäftsmodelle, Organisation und Informationstechnik von Unternehmen.

In diesem Buch zeigen viele renommierte Experten praxisnah alle wichtigen Facetten von SOA auf und erläutern dabei positive wie auch kontroverse Aspekte. Manager und IT-Architekten finden hier fundierte Entscheidungsgrundlagen aus geschäftlichen, organisatorischen sowie technischen Perspektiven.

Das Themenspektrum reicht von SOA-Grundlagen über betriebswirtschaftliche Aspekte, Prozess- und Methodenansätze, Governance, Architektur und Technik bis zum Betrieb von SOA-Infrastrukturen.

Praxisnähe und ganzheitliche Darstellung stehen im Vordergrund.

Aus dem Inhalt:

- SOA-Grundlagen
- Business-Aspekte
- Agilität und Wertschöpfung
- SOA-Prozesse und -Methoden
- Governance
- IT-Architektur und -Technik
- Betrieb
- Risiken und Kritik
- Praxisbeispiele

Die Entscheidungsgrundlage für SOA-Initiativen und -Projekte.

»SOA is a Lifestyle: Eine SOA-Initiative erfordert Änderungen an der Art und Weise, in der Projekte definiert, budgetiert, entwickelt, betrieben und kontrolliert werden. Sie erfordert einen bislang nicht gekannten Grad an Kollaboration zwischen IT- und Fachabteilungen.«

aus dem Vorwort von Anne Thomas Manes

Thema
- **Softwareentwicklung**
- **Unternehmens- und Softwarearchitektur**
- **Business-IT-Alignment**

Leser
- **Softwarearchitekten**
- **Projektleiter**
- **IT-Manager**

Website
- **www.soa-expertenwissen.de**

€ 59,00 (D)
€ 60,70 (A)

ISBN 978-3-89864-437-2

www.dpunkt.de